Igor Pro

Version 9

WaveMetrics

Updates

Please check our website at <http://www.wavemetrics.com/> or the Igor Pro Help menu for minor updates, which we make available for you to download whenever bugs are fixed.

If there are features that you would like to see in future versions of Igor or if you find bugs in the current version, please let us know. We’re committed to providing you with a product that does the job reliably and conveniently.

Notice

All brand and product names are trademarks or registered trademarks of their respective companies.

Manual Revision: October 12, 2023 (9.03)

WaveMetrics, Inc.
PO Box 2088
Lake Oswego, OR 97035
USA

Voice:	503-620-3001
FAX:	503-620-6754
Email:	support@wavemetrics.com, sales@wavemetrics.com
Web:	http://www.wavemetrics.com/

Table of Contents

Volume I — Getting Started

I-1 Introduction to Igor Pro I-1
I-2 Guided Tour of Igor Pro I-11

Volume II — User's Guide: Part 1

II-1 Getting Help II-1
II-2 The Command Window II-7
II-3 Experiments, Files and Folders II-15
II-4 Windows II-43
II-5 Waves II-61
II-6 Multidimensional Waves II-91
II-7 Numeric and String Variables II-101
II-8 Data Folders II-107
II-9 Importing and Exporting Data II-125
II-10 Igor HDF5 Guide II-183
II-11 Dialog Features II-227
II-12 Tables II-233
II-13 Graphs II-275
II-14 Category Plots II-355
II-15 Contour Plots II-365
II-16 Image Plots II-385
II-17 3D Graphics II-405
II-18 Page Layouts II-475

Volume III — User's Guide: Part 2

III-1 Notebooks III-1
III-2 Annotations III-33
III-3 Drawing III-61
III-4 Embedding and Subwindows III-79
III-5 Exporting Graphics (Macintosh) III-95
III-6 Exporting Graphics (Windows) III-101
III-7 Analysis III-107
III-8 Curve Fitting III-177
III-9 Signal Processing III-269
III-10 Analysis of Functions III-319
III-11 Image Processing III-351
III-12 Statistics III-383
III-13 Procedure Windows III-395
III-14 Controls and Control Panels III-413
III-15 Platform-Related Issues III-447

III-16 Text Encodings III-459
III-17 Miscellany III-495
III-18 Preferences III-515

Volume IV — Programming

IV-1 Working with Commands IV-1
IV-2 Programming Overview IV-23
IV-3 User-Defined Functions IV-29
IV-4 Macros IV-115
IV-5 User-Defined Menus IV-125
IV-6 Interacting with the User IV-143
IV-7 Programming Techniques IV-165
IV-8 Debugging IV-211
IV-9 Dependencies IV-229
IV-10 Advanced Topics IV-235

Volume V — Reference

V-1 Igor Reference V-1

Index

Volume I

Getting Started

Table of Contents

I-1 Introduction to Igor Pro I-1
I-2 Guided Tour of Igor Pro I-11

Chapter

I-1

Introduction to Igor Pro

Introduction to Igor Pro

Igor Pro is an integrated program for visualizing, analyzing, transforming and presenting experimental data.

Igor Pro's features include:

- Publication-quality graphics
- High-speed data display
- Ability to handle large data sets
- Curve-fitting, Fourier transforms, smoothing, statistics, and other data analysis
- Waveform arithmetic
- Image display and processing
- Combination graphical and command-line user interface
- Automation and data processing via a built-in programming environment
- Extensibility through modules written in the C and C++ languages

Some people use Igor simply to produce high-quality, finely-tuned scientific graphics. Others use Igor as an all-purpose workhorse to acquire, analyze and present experimental data using its built-in programming environment. We have tried to write the Igor program and this manual to fulfill the needs of the entire range of Igor users.

Igor 32-bit and 64-bit Versions

Igor is available in both 32-bit (Windows only) and 64-bit versions. When making a distinction between these versions, we sometimes refer to the 32-bit version as "IGOR32" and the 64-bit version as "IGOR64".

On Windows, both 32-bit and 64-bit applications are installed. The 64-bit version runs by default. You should run the 32-bit version only if compatibility with a 32-bit XOP (plug-in) is required.

On Macintosh, starting with Igor Pro 8.00, Igor is available only as a 64-bit application. If you need to run with a 32-bit XOP (plug-in) then you must run Igor Pro 7.

Igor Objects

The basic objects that all Igor users work with are:

- Waves
- Graphs
- Tables
- Page layouts

A collection of objects is called an "experiment" and is stored in an experiment file. When you open an experiment, Igor recreates the objects that comprise it.

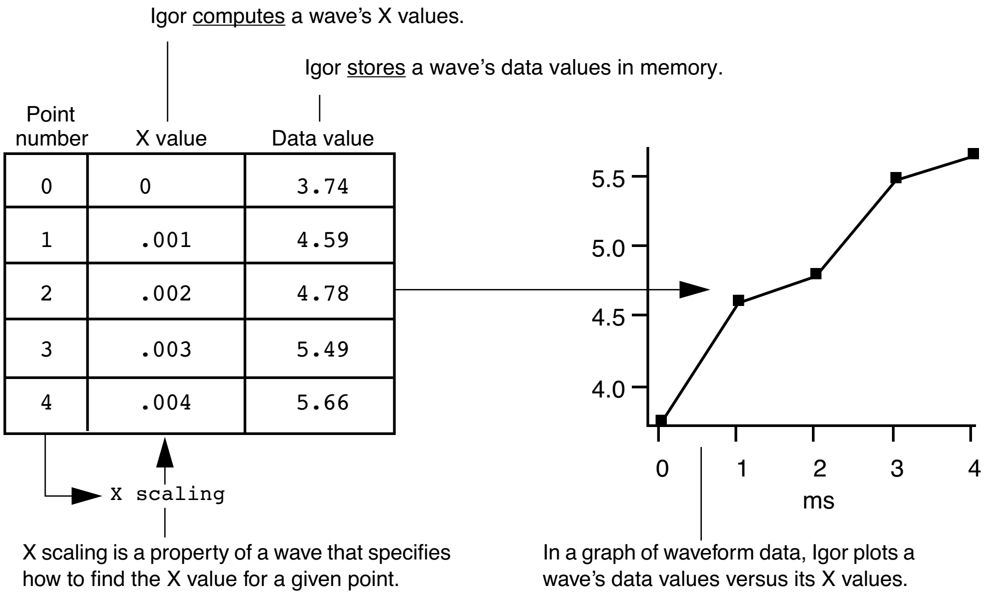
Waves — The Key Igor Concept

We use the term "wave" to describe the Igor object that contains an array of numbers. Wave is short for "waveform". The wave is the most important Igor concept.

Igor was originally designed to deal with waveform data. A waveform typically consists of hundreds to thousands of values measured at evenly-spaced intervals of time. Such data is usually acquired from a digital oscilloscope, scientific instrument or analog-to-digital converter card.

The distinguishing trait of a waveform is the *uniform spacing* of its values along an axis of time or other quantity. An Igor wave has an important property called "X scaling" that you set to specify the spacing of your data. Igor *stores* the Y component for each point of a wave in memory but it *computes* the X component based on the wave's X scaling.

In the following illustration, the wave consists of five data points numbered 0 through 4. The user has set the wave's X scaling such that its X values start at 0 and increment by 0.001 seconds per point. The graph displays the wave's stored data values versus its computed X values.


Igor computes a wave's X values.
Igor stores a wave's data values in memory.
X scaling
X scaling is a property of a wave that specifies how to find the X value for a given point.
5.5
5.0
4.5
4.0
0
1
2
3
4
ms
In a graph of waveform data, Igor plots a wave's data values versus its X values.

Point number	X value	Data value
0	0	3.74
1	.001	4.59
2	.002	4.78
3	.003	5.49
4	.004	5.66

Waves can have from one to four dimensions and can contain either numeric or text data.

Igor is also capable of dealing with data that does not fit the waveform metaphor. We call this XY data. Igor can treat two waves as an XY pair. In an XY pair, the data values of one wave supply the X component and the data values of another wave supply the Y component for each point in the pair.

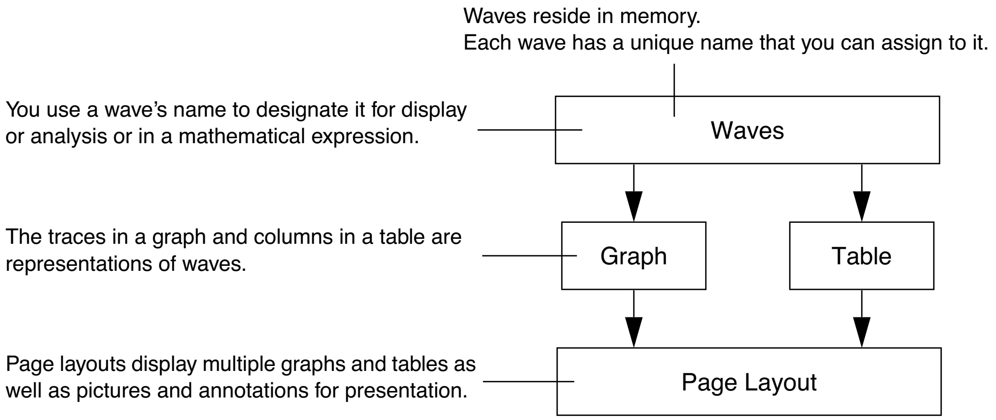
A few analysis operations, such as Fourier transforms, inherently work only on waveform data. They take a wave's X scaling into account.

Other operations work equally well on waveform or XY data. Igor can graph either type of data and its powerful curve fitting works on either type.

Most users create waves by loading data from a file. You can also create waves by typing in a table, evaluating a mathematical expression, acquiring from a data acquisition device, and accessing a database.

How Objects Relate

This illustration shows the relationships among Igor's basic objects. Waves are displayed in graphs and tables. Graphs and tables are displayed in page layouts. Although you can display a wave in a graph or table, a wave does not need to be displayed to exist.


Waves reside in memory.
Each wave has a unique name that you can assign to it.
You use a wave's name to designate it for display or analysis or in a mathematical expression.
Waves
The traces in a graph and columns in a table are representations of waves.
Graph
Table
Page layouts display multiple graphs and tables as well as pictures and annotations for presentation.
Page Layout

Each object has a name so that it can be referenced in an Igor command. You can explicitly set an object’s name or accept a default name created by Igor.

Graphs are used to visualize waves and to generate high-quality printouts for presentation. The traces in a graph are representations of waves. If you modify a wave, Igor automatically updates graphs. Igor labels the axes of a graph intelligently. Tick marks never run into one another and are always “nice” values no matter how you zoom in or pan around.

In addition to traces representing waveform or XY data, a graph can display an image or a contour plot generated from 2D data.

Tables are used to enter, inspect or modify wave data. A table in Igor is not the same as a spreadsheet in other graphing programs. A column in a table is a *representation* of the contents of a wave. The wave continues to exist even if you remove it from the table or close the table entirely.

Page layouts permit you to arrange multiple graphs and tables as well as pictures and annotations for presentation. If you modify a graph or table, either directly or indirectly by changing the contents of a wave, Igor automatically updates its representation in a layout.

Both graphs and layouts include drawing tools for adding lines, arrows, boxes, polygons and pictures to your presentations.

More Objects

Here are some additional objects that you may encounter:

- Numeric and string variables
- Data folders
- Notebooks
- Control panels
- 3D plots
- Procedures

A numeric variable stores a single number and a string variable stores a text string. Numeric and string variables are used for storing bits of data for Igor procedures.

A data folder can contain waves, numeric variables, string variables and other data folders. Data folders provide a way to keep a set of related data, such as all of the waves from a particular run of an experiment, together and separate from like-named data from other sets.

A notebook is like a text-editor or word-processor document. You can use a notebook to keep a log of results or to produce a report. Notebooks are also handy for viewing Igor technical notes or other text documentation.

A control panel is a window containing buttons, checkboxes and other controls and readouts. A control panel is created by an Igor user to provide a user interface for a set of procedures.

A 3D plot displays three–dimensional data as a surface, a scatter plot, or a path in space.

A procedure is a programmed routine that performs a task by calling Igor's built-in operations and functions and other procedures. Procedures range from very simple to very complex and powerful. You can run procedures written by WaveMetrics or by other Igor users. If you are a programmer or want to learn programming, you can learn to write your own Igor procedures to automate your work.

Igor’s Toolbox

Igor's toolbox includes a wide range of built-in routines. You can extend it with user-defined procedures written in Igor itself and with separately-compiled Igor extensions (plug-ins) that you obtain from WaveMetrics, from a colleague, from a third-party, or write yourself.

Built-In Routines

Each of Igor's built-in routines is categorized as a function or as an operation.

A built-in function is an Igor routine, such as sin, exp or ln, that directly returns a result. A built-in operation is a routine, such as Display, FFT or Integrate, that acts on an object and may create new objects but does not directly return a result.

A good way to get a sense of the scope of Igor's built-in routines is to scan the sections **Built-In Operations by Category** on page V-1 and **Built-In Functions by Category** on page V-7 in the reference volume of this manual.

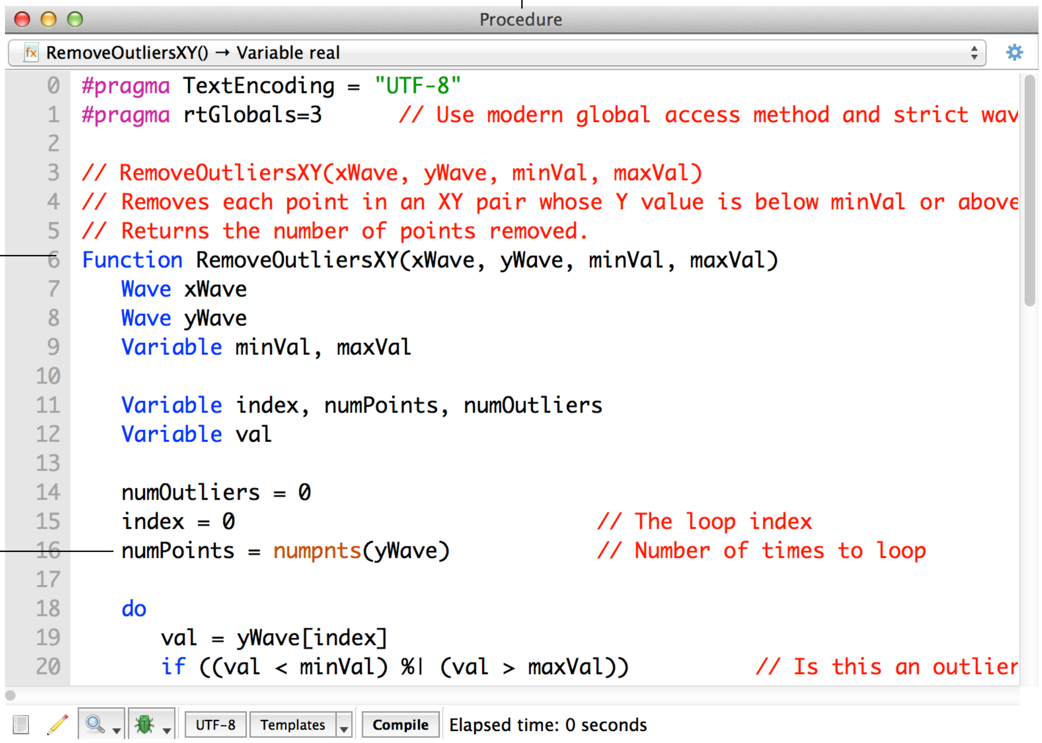
For getting reference information on a particular routine it is usually most convenient to choose Help→Command Help and use the Igor Help Browser.

User-Defined Procedures

A user-defined procedure is a routine written in Igor’s built-in programming language by entering text in a procedure window. It can call upon built-in or external functions and operations as well as other user-defined procedures to manipulate Igor objects. Sets of procedures are stored in procedure files.

You can create Igor procedures by entering text in a procedure window.

Each procedure has a name which you use to invoke it.



```
#pragma TextEncoding = "UTF-8"
#pragma rtGlobals=3      // Use modern global access method and strict wav

// RemoveOutliersXY(xWave, yWave, minVal, maxVal)
// Removes each point in an XY pair whose Y value is below minVal or above
// Returns the number of points removed.
Function RemoveOutliersXY(xWave, yWave, minVal, maxVal)
    Wave xWave
    Wave yWave
    Variable minVal, maxVal

    Variable index, numPoints, numOutliers
    Variable val

    numOutliers = 0
    index = 0                         // The loop index
    numPoints = numpnts(yWave)        // Number of times to loop

    do
        val = yWave[index]
        if ((val < minVal) %| (val > maxVal))         // Is this an outlier
```

Procedures can call operations, functions or other procedures. They can also perform waveform arithmetic.

Igor Extensions

An extension is a “plug-in” - a piece of external C or C++ code that adds functionality to Igor. For historical reasons, we use the term “XOP” to refer to an Igor extension. “XOP” is a contraction of “external operation”. The terms “XOP” and “Igor extension” are synonymous.

Originally XOPs were intended to allow adding operations only to Igor. Now XOPs can add much more, including functions, menus, dialogs, and windows, so “XOP” has the meaning “external module that extends Igor”.

To create an XOP, you must be a C or C++ programmer and you need the optional **Igor External Operations Toolkit**. See **Creating Igor Extensions** on page IV-208.

Although *creating* an extension is a job for a programmer, anyone can *use* an extension. The Igor installer automatically installs commonly used extensions in "Igor Pro Folder/Igor Extensions (64-bit)". These extensions are available for immediate use.

Less commonly used extensions are installed in "Igor Pro Folder/More Extensions (64-bit)". Available extensions are described in the “XOP Index” help file (choose Help→Help Windows→XOP Index.ihf). To activate an extension, see **Activating 64-bit Extensions** on page III-512.

Igor’s User Interface

Igor uses a combination of the familiar graphical user interface and a command-line interface. This approach gives Igor both ease-of-use and programmability.

The job of the user interface is to allow you to apply Igor’s operations and functions to objects that you create. You can do this in several ways:

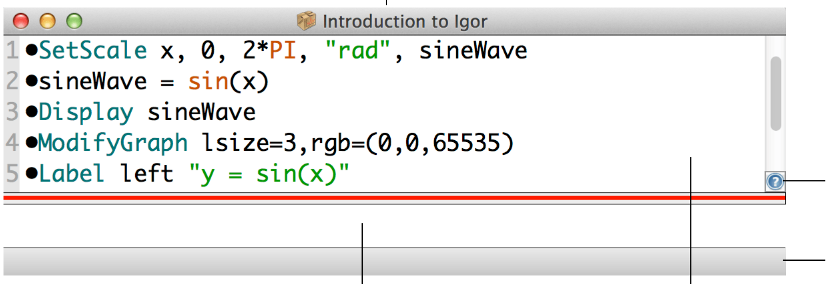
- Via menus and dialogs
- Using the main toolbar
- By typing Igor commands directly into the command line
- By writing Igor procedures

The Command Window

The command window is Igor’s control center. It appears at the bottom of the screen.

At the bottom of the command window is the command line. Above the red divider is the history area where executed commands are stored for review. Igor also uses the history area to report results of analyses like curve-fitting or waveform statistics.

The name of the current experiment appears as the title of the command window.



```
•SetScale x, 0, 2*PI, "rad", sineWave
•sineWave = sin(x)
•Display sineWave
•ModifyGraph lsize=3,rgb=(0,0,65535)
•Label left "y = sin(x)"
```

Help Browser button

Status area

You enter commands in the **command line**.

When Igor executes a command it transfers it to the **history area**.

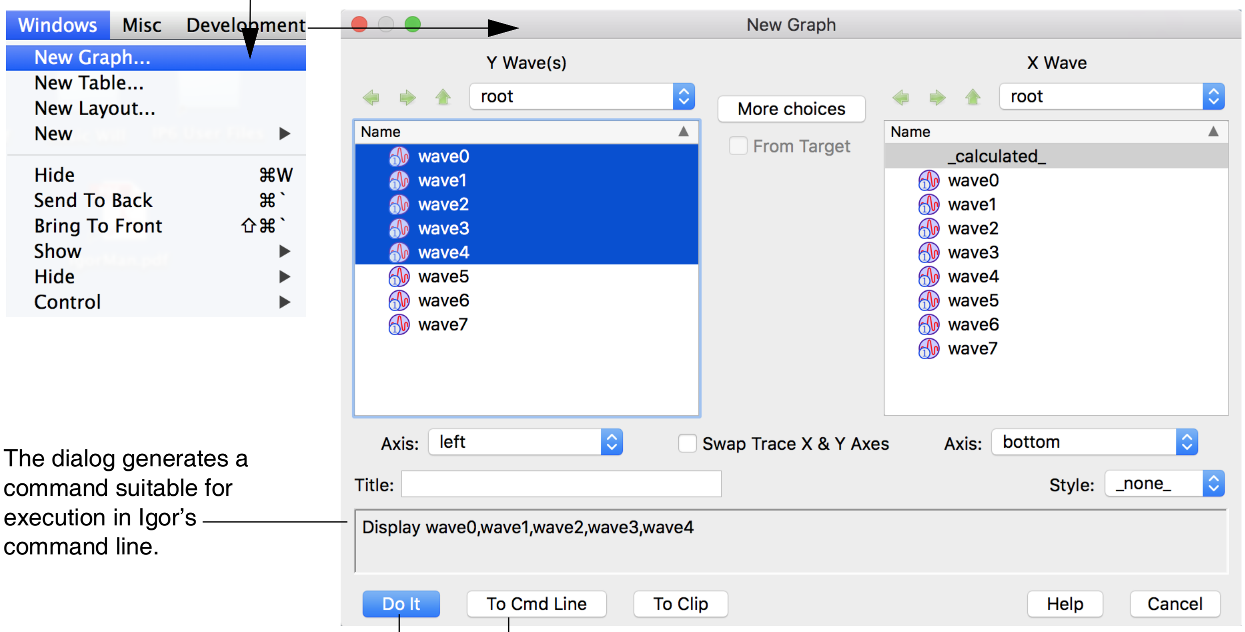
Menus, Dialogs and Commands

Menus and dialogs provide easy access to the most commonly-used Igor operations.

When you choose a menu item associated with an Igor operation, Igor presents a dialog. As you use the dialog, Igor generates a command and displays it in the **command box** near the bottom of the dialog. When you click the Do It button, Igor transfers the command to the command line where it is executed.

When you choose a menu item...

...Igor presents a dialog.



```
Display wave0,wave1,wave2,wave3,wave4
```

The dialog generates a command suitable for execution in Igor's command line.

Transfers the command to the command line and executes it.

Copies the command to the command line where you can edit it and then execute it.

As you get to know Igor, you will find that some commands are easier to invoke from a dialog and others are easier to enter directly in the command line.

There are some menus and dialogs that bypass the command line. Examples are the Save Experiment and Open Experiment items in the File menu.

The Igor Toolbar

The main toolbar provides quick access to frequently used features.

On Windows, the default location for the toolbar is just below the main menu bar. On Macintosh, it is at the top of the command window.

You can reposition the toolbar by clicking the dots at the left or top edge and dragging. You can either move it to a different edge of the outer frame window (*Windows*) or the command window (*Macintosh*), or you can drag it elsewhere to float the toolbar.

You can hide the toolbar by choosing Windows→Hide Toolbar and show it again by choosing Windows→Show Toolbar.

The visibility, position, and orientation of the toolbar is saved when you quit Igor.

Currently it is not possible to change which features are available on the toolbar.

Using Igor for Heavy-Duty Jobs

If you generate a lot of raw data or need to do custom technical computing, you will find it worthwhile to learn how to put Igor to heavy-duty use. It is possible to automate some or all of the steps involved in loading, processing, and presenting your data. To do this, you must learn how to write Igor procedures.

Igor includes a built-in programming environment that lets you manipulate waves, graphs, tables and all other Igor objects. You can invoke built-in operations and functions from your own procedures to build higher-level operations customized for your work.

Learning to write Igor procedures is easier than learning a lower-level language such as FORTRAN or C. The Igor programming environment is interactive so you can write and test small routines and then put them together piece-by-piece. You can deal with very high level objects such as waves and graphs but you also have fine control over your data. Nonetheless, it is still programming. To master it requires an effort on your part.

The Igor programming environment is described in detail in **Volume IV Programming**. You can get started by reading the first three chapters of that volume.

You can also learn about Igor programming by examining the WaveMetrics Procedures and example experiments that were installed on your hard disk.

Igor Documentation

Igor includes an extensive online help system and a comprehensive PDF manual.

The online help provides guided tours, tooltips, general usage information for all aspects of Igor, and reference information for Igor operations, functions and keywords.

The PDF manual contains the same information except for the tooltips.

The PDF manual, being in book format, is better organized for linear reading while the online help is usually preferred for reference information.

Tooltips

Igor displays tooltips when you hover the mouse over an icon or dialog item.

You can turn tooltips off using the Miscellaneous Settings dialog. Choose Misc→Miscellaneous Settings, click the Help icon on the left, and uncheck the Show Tooltips checkbox.

The Igor Help System

The Help menu provides access to Igor's help system, primarily through the Igor Help Browser.

- To display the Igor Help Browser, use the Help menu, click the question-mark icon in the command window, or press F1 (*Windows only*).
- Use the Igor Help Browser Help Topics tab to browse help topics.
- Use the Igor Help Browser Shortcuts tab to get a list of handy shortcuts and techniques.
- Use the Igor Help Browser Command Help tab to get reference information on Igor operations and functions. You can also right-click operation and function names in Igor windows to access the reference help.
- Use the Igor Help Browser Search tab to search Igor help, procedure and example files.

Most of the information displayed by the help browser comes from help files that are automatically loaded at launch time. The Help→Help Windows submenu provides direct access to these help files.

The Igor Manual

The Igor PDF manual resides in "Igor Pro Folder/Manual". You can access it by choosing Help→Manual.

The manual consists of five volumes and an index.

Volume I contains the Getting Started material, including the Guided Tour of Igor Pro.

Volumes II and III contain general background and usage information for all aspects of Igor other than programming.

Volume IV contains information for people learning to do Igor programming.

Volume V contains reference information for Igor operations, functions and keywords.

Hard copy of the manual is not available.

Learning Igor

To harness the power of Igor, you need to understand its fundamental concepts. Some of these concepts are familiar to you. However, Igor also incorporates a few concepts that will be new to you and may seem strange at first. The most important of these are *waves* and *experiments.*

In addition to this introduction, the primary resources for learning Igor are:

- The **Guided Tour of Igor Pro** in Chapter I-2

  The guided tour shows you how to perform basic Igor tasks step-by-step and reinforces your understanding of basic Igor concepts along the way.

  The guided tour provides an essential orientation to Igor and is highly-recommended for all Igor users.

- The Igor Pro PDF manual and online help files

  You can access the PDF manual through Igor's Help menu or by opening it directly from the Manual folder of the Igor Pro Folder where Igor is installed.

  You can access the help files through the Igor Help Browser (choose Help→Igor Help Browser) or directly through the Help→Help Windows submenu.

- The example experiments

  The example experiments illustrate a wide range of Igor features. They are stored in the Examples folder in the Igor Pro Folder. You can access them using the File→Example Experiments submenu or directly from the Examples folder of the Igor Pro Folder where Igor is installed.

You will best learn Igor through a combination of doing the guided tour, reading select parts of the manual (see suggestions following the guided tour), and working with your own data.

Videos of the guided tour are available at:

https://www.wavemetrics.com/igorpro/videotutorials.htm

Getting Hands-On Experience

This introduction has presented an overview of Igor, its main constituent parts, and its basic concepts. The next step is to get some hands-on experience and reinforce what you have learned by doing the **Guided Tour of Igor Pro** on page I-11.

Chapter

I-2

Guided Tour of Igor Pro

Overview

In this chapter we take a look at the main functions of Igor Pro by stepping through some typical operations. Our goal is to orient you so that you can comfortably read the rest of the manual or explore the program on your own. You will benefit most from this tour if you actually do the instructed operations on your computer as you read this chapter. Screen shots are provided to keep you synchronized with the tour.

Terminology

If you have read Chapter I-1, **Introduction to Igor Pro**, you already know these terms.

Experiment The entire collection of data, graphs and other windows, program text and other data that make up the current Igor environment or workspace.

Wave Short for waveform, this is basically a named array of data with optional extra information.

Name Because Igor contains a built-in programming and data transformation language, each object must have a unique name.

Command This is a line of text that performs some task. Igor is command-driven so that it can be easily programmed.

About the Tour

This tour consists of three sections: **Guided Tour 1 - General Tour** on page I-13, **Guided Tour 2 - Data Analysis** on page I-46, and **Guided Tour 3 - Histograms and Curve Fitting** on page I-53.

The General Tour is a rambling exploration intended to introduce you to the way things work in Igor and give you a general orientation.

The second and third tours guide you through Igor’s data analysis facilities including simple curve fitting.

When you’ve completed the first tour you may prefer to explore freely on your own before starting the second tour.

Guided Tour 1 - General Tour

In this exercise, we will generate data in three ways (typing, loading, and synthesizing) and we will generate graph, table, and page layout windows. We will jazz up a graph and a page layout with a little drawing and some text annotation. At the end, we will explore some of the more advanced features of Igor Pro.

Creating an Igor64 Alias or Shortcut

The 64-bit Igor Pro application is typically located at:

```
/Applications/Igor Pro 9 Folder/Igor64.app (Macintosh)
```

```
C:\Program Files\WaveMetrics\Igor Pro 9 Folder\IgorBinaries_x64\Igor.exe (Windows)
```

The ".app" and ".exe" extensions may be hidden on your system.

1. **Make an alias (Macintosh) or shortcut (Windows) for your Igor64 application file and put the alias or shortcut on your desktop. Name it Igor64.**

Launching Igor Pro

1. **Double-click your Igor64 alias or shortcut.**

   Igor starts up.

   On Windows, you an also launch Igor64 using the Start menu.

2. **Choose Misc→Preferences Off.**

   Turning preferences off ensures that the tour works the same for everyone.

Entering Data

1. **If a table window is showing, click it to bring it to the front.**

   When Igor starts up, it creates a new blank table unless this feature is turned off in the Miscellaneous Settings dialog. If the table is not showing, perform the following two steps:

2. **Choose the Windows→New Table menu item.**

   The New Table dialog appears.

3. **Click the Do It button.**

   A new blank table is created.

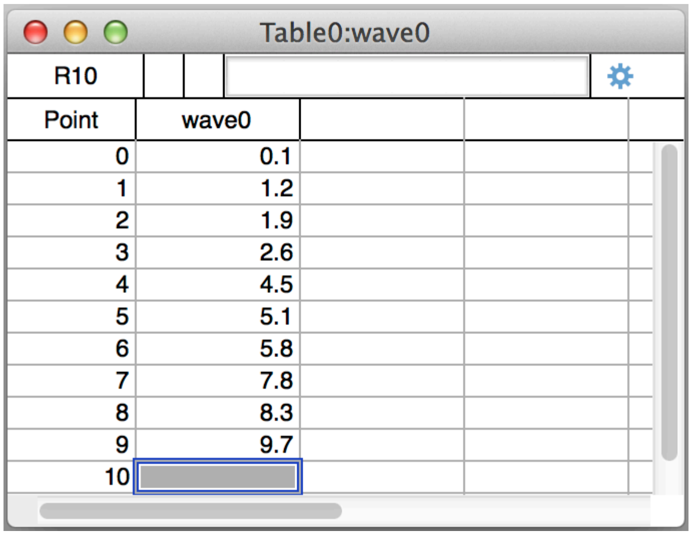
4. **Type "0.1" (without the quotes) and then press Return or Enter on your keyboard.**

   This creates a wave named "wave0" with 0.1 for the first point. Entering a value in the first row (point 0) of the first blank column automatically creates a new wave.

5. **Type the following numbers, pressing Return or Enter after each one:**

   1.2
   1.9
   2.6
   4.5
   5.1
   5.8
   7.8
   8.3
   9.7

   The table should look like this:



Point	wave0
0	0.1
1	1.2
2	1.9
3	2.6
4	4.5
5	5.1
6	5.8
7	7.8
8	8.3
9	9.7
10	

6. **Click in the first cell of the first blank column.**
7. **Enter the following numbers in the same way:**

   -0.12
   -0.08
   1.3
   1
   0.54
   0.47
   0.44
   0.2
   0.24
   0.13

8. **Choose Data→Rename.**
9. **Click “wave0” in the list and then click the arrow icon.**
10. **Replace “wave0” with “time”.**

    Notice that you can’t use the name “time” because it is the name of a built-in string function. We apologize for usurping such a common name.

11. **Change the name to “timeval”.**
12. **Click “wave1” in the list, click the arrow icon, and replace “wave1” with “yval”.**
13. **Click Do It.**

    The column headers in the table change to reflect the name changes.

Making a Graph

14. Choose the Windows→New Graph menu item.

The New Graph dialog appears. This dialog comes in a simple form that most people will use and a more complex form that you can use to create complex multi-axis graphs in one step.

15. If you see a button labeled Fewer Choices, click it.

16. In the Y Waves list, click “yval”.

17. In the X Wave list, click “timeval”.

18. Click Do It.

A simple graph is created.

Touching up a Graph

19. Position the cursor directly over the trace in the graph and double-click.

The Modify Trace Appearance dialog appears. You could also have chosen the corresponding menu item from the Graph menu.

Note: The Graph menu appears only when a graph is the target window. The *target* window is the window that menus and dialogs act on by default.

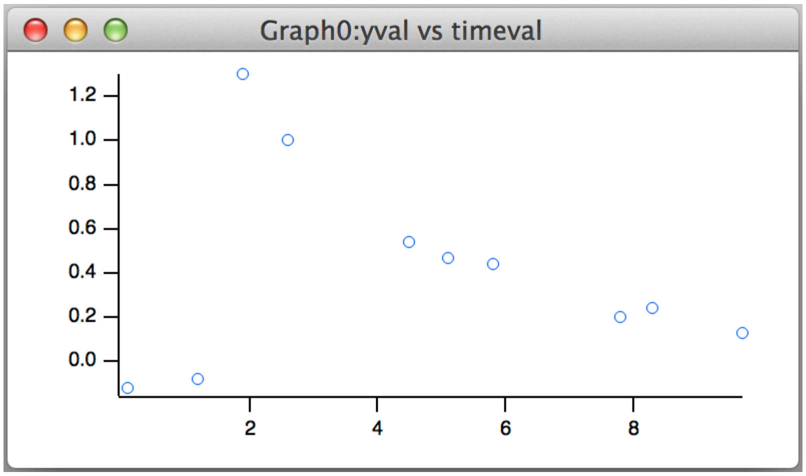
20. Choose Markers from the Mode pop-up menu.

21. Select the open circle from the Marker pop-up menu.

22. Set the marker color to blue.

23. Click Do It.

The graph should now look like this:



24. Position the cursor over the bottom axis line.

The cursor changes to this shape: ↕. This indicates the cursor is over the axis and also that you can offset the axis, and the corresponding plot area edge, to a new position.

25. Double-click directly on the axis.

The Modify Axis dialog appears. If another dialog appears, click Cancel and try again, making sure the ↕ cursor is showing when you double-click.

Note the Live Update checkbox in the top/right corner of the Modify Axis dialog. When it is checked, changes that you make in the dialog are immediately reflected in the graph. When it is unchecked, the changes appear only when you click Do It.

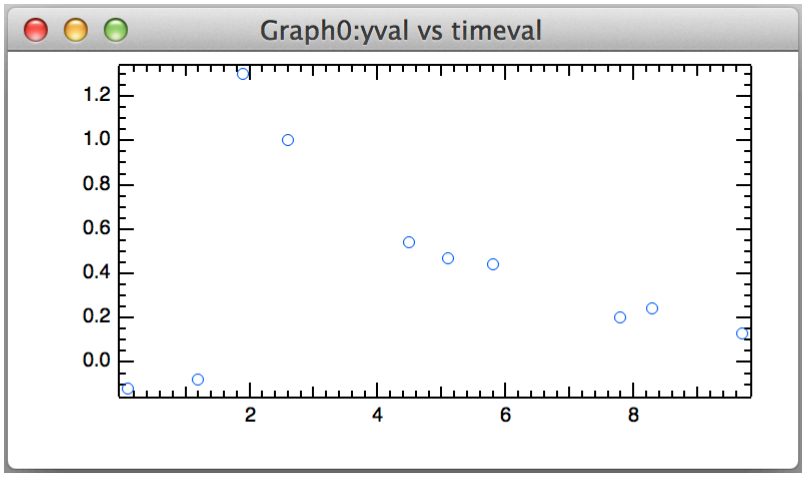
26. If it is not already showing, click the Axis tab.

27. Choose On from the Mirror Axis pop-up.

28. Click the Auto/Man Ticks tab.

29. **Click the Minor Ticks checkbox so it is checked.**
30. **Click the Ticks and Grids tab.**
31. **Choose Inside from the Location pop-up.**
32. **Choose the left axis from the Axis pop-up menu in the top-left corner of the dialog and then repeat steps 8 through 13.**
33. **Click Do It.**

    The graph should now look like this:



34. **Again double-click the bottom axis.**

    The Modify Axis dialog appears again.
35. **Click the Axis tab.**
36. **Uncheck the Standoff checkbox.**
37. **Choose the left axis from the Axis pop-up menu and repeat step 18.**
38. **Click Do It.**

    Notice that some of the markers now overlap the axes. The axis standoff setting offsets the axis so that markers and traces do not overlap it. You can use Igor's preferences to ensure that this and other settings default to your liking, as explained below.
39. **Double-click one of the tick mark labels (such as "6") on the bottom axis.**

    The Modify Axis dialog reappears, this time with the Axis Range tab showing. If another dialog or tab appears, cancel and try again, making sure to double click one of the tick mark labels on the bottom axis.
40. **Choose "Round to nice values" from the pop-up menu that initially reads "Use data limits".**
41. **Choose the left axis from the Axis pop-up menu and repeat step 22.**
42. **Click Do It.**

    Notice that the limits of the axes now fall on "nice" values.

Adding a Legend

1. **Choose the Graph→Add Annotation menu item.**

   The Add Annotation dialog appears.
2. **Click the Text tab if it is not already selected.**
3. **Choose Legend from the Annotation pop-up menu in the top-left corner of the dialog.**

   Igor inserts text to create a legend in the Annotation text entry area. The Preview area shows what the annotation will look like. The text `\s(yval)` generates the symbol for the yval wave. This is an "escape sequence" which creates special effects such as this.

4. **Change the second instance of “yval” to “Magnitude”.**

   The annotation annotation text should now be “\s(yval) Magnitude”.

5. **Click the Frame tab and choose Box from the Annotation Frame pop-up menu.**
6. **Choose Shadow from the Border pop-up menu.**
7. **Click the Position tab and choose Right Top from the Anchor pop-up menu.**

   Specifying an anchor point helps Igor keep the annotation in the best location as you make the graph window bigger or smaller.

8. **Click Do It.**

Adding a Tag

1. **Choose the Graph→Add Annotation menu item.**
2. **Choose Tag from the Annotation pop-up menu in the top-left corner of the dialog.**
3. **Click the Text tab, and in the annotation text entry area of the Text tab, type “When time is ”.**
4. **Choose Attach Point X Value from the Dynamic pop-up menu in the Insert area of the dialog.**

   Igor inserts the `\OX` escape code into the annotation text entry area.

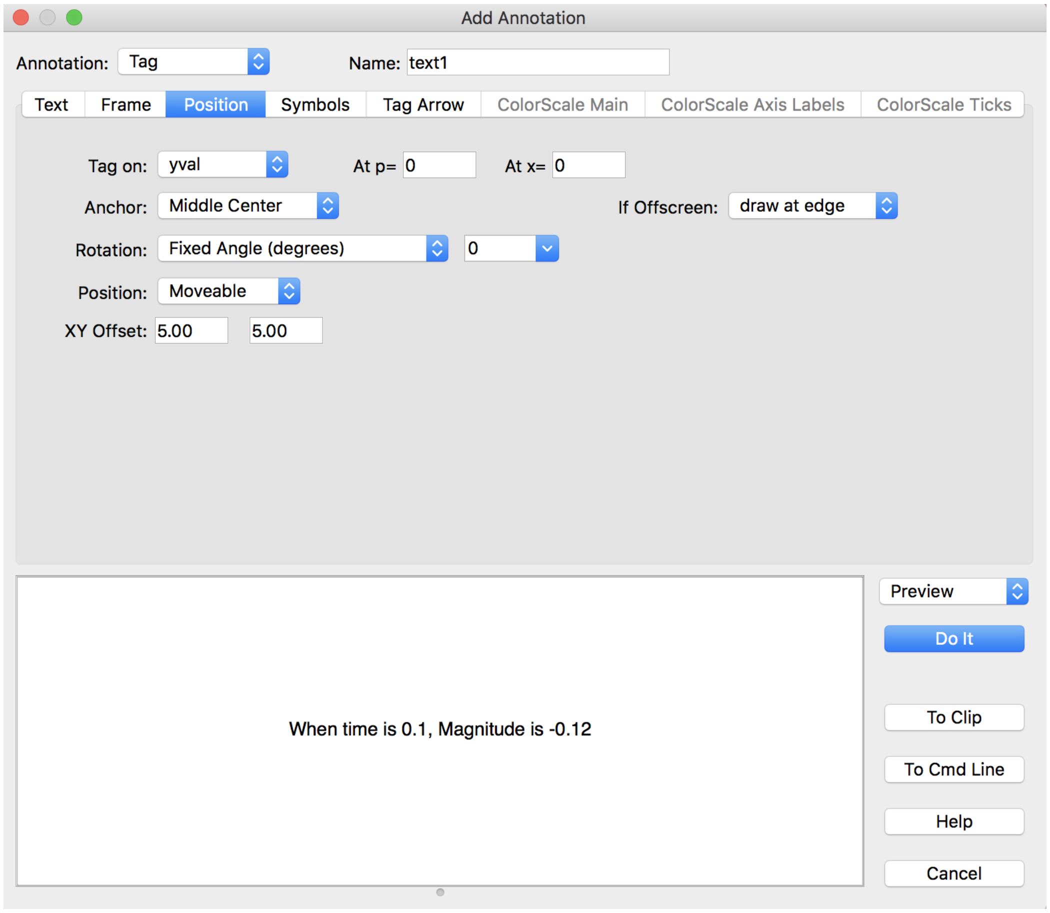
5. **In the annotation text entry area, add “, Magnitude is ”.**
6. **Choose Attach Point Y Value from the Dynamic pop-up menu.**

   Igor inserts the `\OY` escape code into the annotation text entry area.

7. **Click the Frame tab and choose None from the Annotation Frame pop-up menu.**
8. **Click the Tag Arrow tab and choose Arrow from the Connect Tag to Wave With pop-up menu.**

9. **Click the Position tab and choose “Middle center” from the Anchor pop-up menu.**

   The dialog should now look like this:



Annotation: Tag
Name: text1

Tag on: yval
At p= 0
At x= 0

Anchor: Middle Center
If Offscreen: draw at edge

Rotation: Fixed Angle (degrees) 0

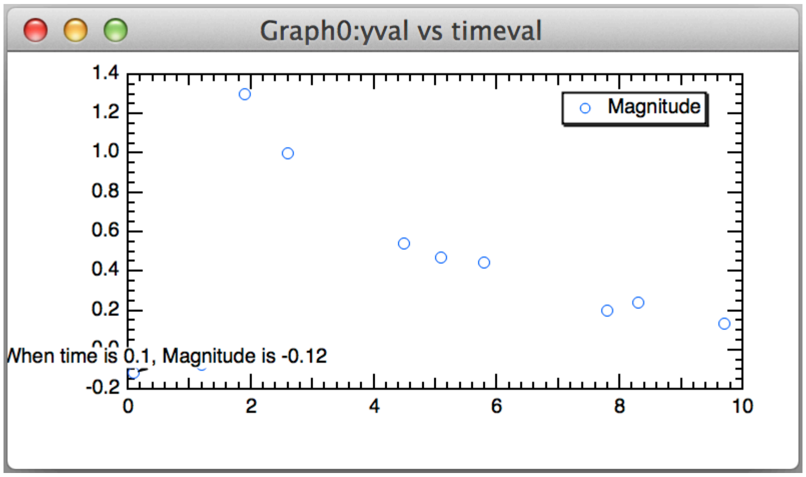
Position: Moveable

XY Offset: 5.00 5.00

When time is 0.1, Magnitude is -0.12

10. **Click Do It.**

    The graph should now look like this:



The tag is attached to the first point. An arrow is drawn from the center of the tag to the data point but you can’t see it because it is hidden by the tag text.

11. **Position the cursor over the text of the tag.**

    The cursor changes to a hand. This indicates you can reposition the tag relative to the data point it is attached to.

12. **Drag the tag up and to the right about 1 cm.**

    You can now see the arrow.

13. **With the cursor over the text of the tag, press Option (*Macintosh*) or Alt (*Windows*).**

    The cursor changes to this shape: ⊡ . (You may need to nudge the cursor slightly to make it change.)

14. **While pressing Option (*Macintosh*) or Alt (*Windows*), drag the box cursor to a different data point.**

    The tag jumps to the new data point and the text is updated to show the new X and Y values. Option-drag (*Macintosh*) or Alt-drag (*Windows*) the tag to different data points to see their X and Y values.

    Notice that the tip of the arrow touches the marker. This doesn't look good, so let's change it.

15. **Double-click the text part of the tag.**

    The Modify Annotation dialog appears.

16. **Click the Tag Arrow tab and change the Line/Arrow Standoff from "Auto" to "10".**

17. **Click the Change button.**

    The tip of the arrow now stops 10 points from the marker.

Using Preferences

If you have already set preferences to your liking and do not want to disturb them, you can skip this section.

1. **Choose Misc→Preferences On.**
2. **Click the graph window if it is not already active.**
3. **Choose the Graph→Capture Graph Prefs menu item.**

    The Capture Graph Preferences dialog appears.

4. **Click the checkboxes for XY plot axes and for XY plot wave styles.**
5. **Click Capture Preferences.**
6. **Choose Windows→New Graph.**
7. **Choose "yval" as the Y wave and "timeval" as the X wave.**
8. **Click Do It.**

    The new graph is created with a style similar to the model graph.

9. **Press Option (*Macintosh*) or Alt (*Windows*) while clicking the close button of the new graph.**

    The new graph is killed without presenting a dialog.

10. **Choose Graph→Capture Graph Prefs.**
11. **Click the checkboxes for XY plot axes and for XY plot wave styles.**
12. **Click Revert to Defaults.**
13. **Choose Misc→Preferences Off.**

    We turn preferences off during the guided tour to ensures that the tour works the same for everyone. This is not something you would do during normal work.

Making a Page Layout

1. **Choose the Windows→New Layout menu item.**

    The New Page Layout dialog appears. The names of all tables and graphs are shown in the list.

2. **In the Objects to Lay Out list, select Graph0.**
3. **Command-click (*Macintosh*) or Ctrl-click (*Windows*) on Table0.**

4. **Click Do It.**

   A page layout window appears with a Table0 object on top of a Graph0 object.

   The layout initially shows the page at 50% magnification but you may prefer to work at 100%. You can use the pop-up menu in the lower left corner of the window to change magnification.

5. **Click the Table0 object in the layout window.**

   The table object becomes selected, resize handles are drawn around the edges, and the cursor changes to a hand when over the table.

6. **Click in the middle of the table and drag it so you can see the right edge of the table.**

7. **Position the cursor over the handle (small black square) in the middle of the right edge of the table.**

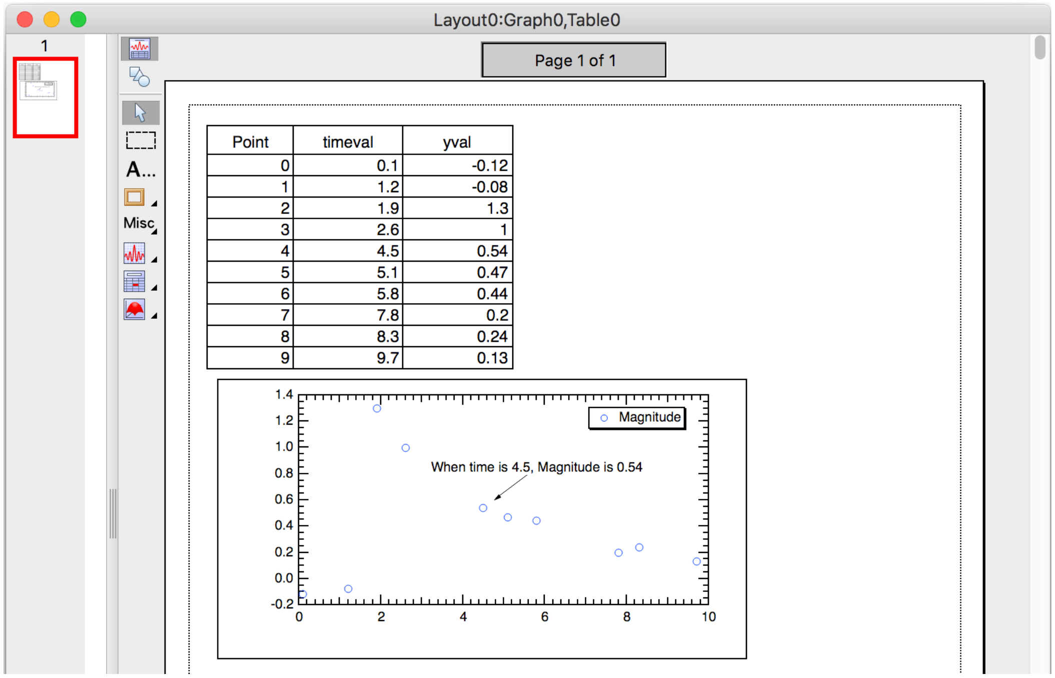
   The cursor changes to a two headed arrow indicating you can drag in the direction of the arrows.

8. **Drag the edge of the table to the left until it is close to the edge of the third column of numbers.**

   You need only get close — Igor snaps to the nearest grid line.

9. **In a similar fashion, adjust the bottom of the table to show all the data but without any blank rows.**

10. **Arrange the table and graph objects in the layout to approximately match this picture:**



Point	timeval	yval
0	0.1	-0.12
1	1.2	-0.08
2	1.9	1.3
3	2.6	1
4	4.5	0.54
5	5.1	0.47
6	5.8	0.44
7	7.8	0.2
8	8.3	0.24
9	9.7	0.13

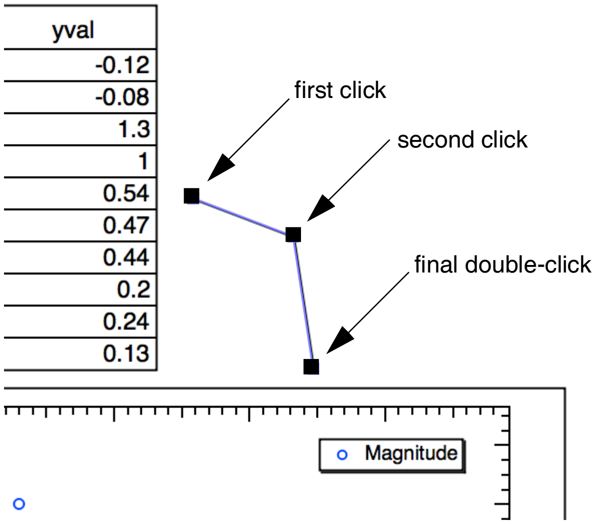
11. **Click this icon in the tool palette: .**

    This activates the drawing tools.

12. **Click this icon in the drawing tool palette: ⬠.**

    This is the polygon tool.

13. **Click once just to the right of the table, click again about 2 cm right and 1 cm down and finally double-click a bit to the right of the last click and just above the graph, as illustrated:**



The double-click exits the “draw polygon” mode and enters “edit polygon mode”. If you wish to touch up the defining vertices of the polygon, do so now by dragging the handles (the square boxes at the vertices).

14. **Click the Arrow tool in the palette.**

    This exits polygon edit mode.

15. **Click the polygon to select it.**

16. **Click the drawing environment icon ( ) and choose At End from the Line Arrow submenu.**

17. **Click this icon in the tool palette: .**

    This is the operate icon. The drawing tools are replaced by the normal layout tools.

    We are finished with the page layout for now.

18. **Choose Windows→Send To Back.**

Saving Your Work - Tour 1A

In this section, we will create a “Guided Tours” folder in which to save our work so far. Then we will save the current experiment in an Igor experiment file.

You can save Igor files anywhere. Here will will create the “Guided Tours” folder in the “Igor Pro User Files” folder. Igor creates “Igor Pro User Files” when it first starts. It is a good place to save Igor files.

1. **Choose Help→Show Igor Pro User Files.**

   Igor opens the "Igor Pro User Files" folder on the desktop.

   By default, this folder has the Igor Pro major version number in its name, for example, "Igor Pro 9 User Files", but it is generically called the "Igor Pro User Files" folder.

   Note where in the file system hierarchy this folder is located as you will need to know this in a subsequent step. The default locations are:

   Macintosh:
   `/Users/<user>/Documents/WaveMetrics/Igor Pro 9 User Files`

   Windows:
   `C:\Users\<user>\Documents\WaveMetrics\Igor Pro 9 User Files`

2. **Create a folder named “Guided Tours” in the “Igor Pro User Files” folder.**

3. **Activate Igor .**

4. **Choose File→Save Experiment As.**

   The save file dialog appears.

5. **Make sure that Packed Experiment File is selected as the file format.**

6. **Navigate to the “Guided Tours” folder.**

7. **Type “Tour 1A.pxp” in the name box.**

8. **Click Save.**

   The “Tour 1A.pxp” file contains all of your work in the current experiment, including waves that you created, graphs, tables and page layout windows.

   If you want to take a break, you can quit Igor Pro now.

Loading Data

Before loading data we will use a notebook window to look at the data file.

1. **If you are returning from a break, launch Igor and open your “Tour 1A.pxp” experiment file. Then turn preferences off using the Misc menu.**

   Opening the “Tour 1A.pxp” experiment file restores the Igor workspace to the state it was in when you saved the file. You can open the experiment file by using the Open Experiment item in the File menu. by double-clicking the experiment file, or by choosing File→Recent Experiments→Tour #1a.pxp.

2. **Choose the File→Open File→Notebook menu item.**

3. **Navigate to the folder “Igor Pro 9 Folder:Learning Aids:Sample Data” folder and open “Tutorial Data #1.txt.”**

   The Igor Pro 9 folder is typically installed in “/Applications” on Macintosh and in “C:/Program Files/WaveMetrics” on Windows.

   A notebook window showing the contents of the file appears. If desired, we could edit the data and then save it. For now we just observe that the file appears to be tab-delimited (tabs separate the columns) and contains names for the columns. Note that the name of the first column will conflict with the data we just entered and the other names have spaces in them.

4. **Click the close button or press Command-W (*Macintosh*) or Ctrl+W (*Windows*).**

   A dialog appears asking what you want to do with the notebook window.

5. **Click the Kill button.**

   The term “kill” means to “completely remove from the experiment”. The file will not be affected.

   Now we will actually load the data.

6. **Choose Data→Load Waves→Load Delimited Text.**

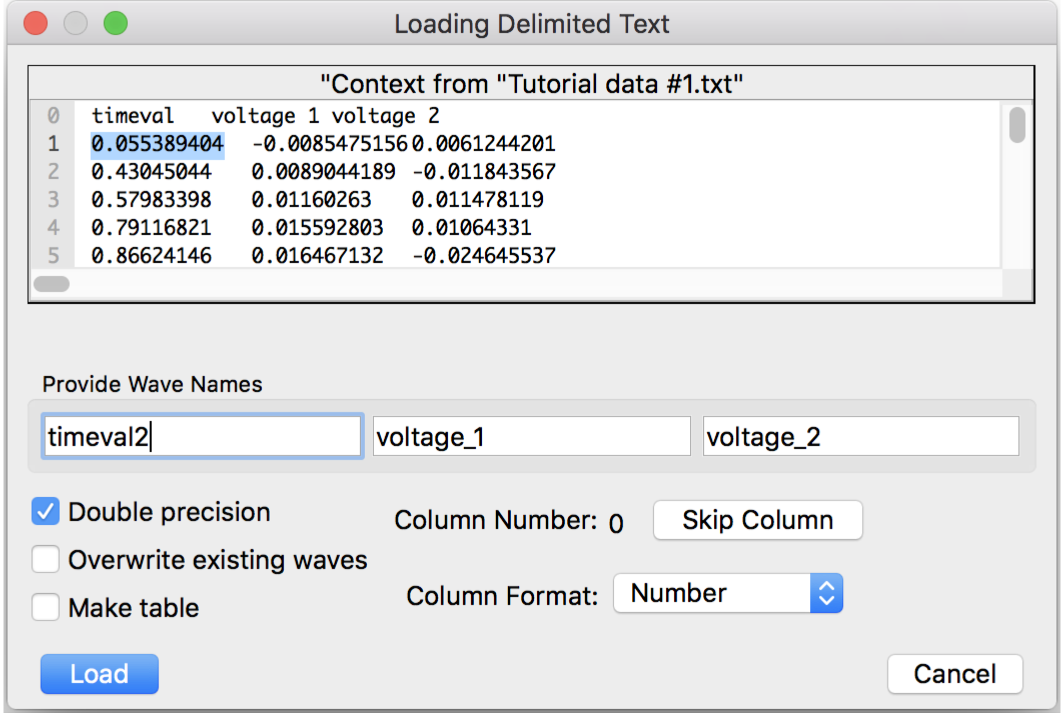
   An Open File dialog appears.

7. **Again choose “Tutorial Data #1.txt” and click Open.**

   The Loading Delimited Text dialog appears. The name “timeval” is highlighted and an error message is shown. Observe that the names of the other two columns were fixed by replacing the spaces with underscore characters.

8. **Change “timeval” to “timeval2”.**

   The dialog should now look like this:



"Context from "Tutorial data #1.txt"

```
0  timeval    voltage 1 voltage 2
1  0.055389404   -0.00854751560.0061244201
2  0.43045044   0.0089044189  -0.011843567
3  0.57983398   0.01160263    0.011478119
4  0.79116821   0.015592803   0.01064331
5  0.86624146   0.016467132   -0.024645537
```

Provide Wave Names

timeval2 | voltage_1 | voltage_2

☑ Double precision

□ Overwrite existing waves

□ Make table

Column Number: 0

Column Format: Number

9. **Click the Make Table box to check it and then click Load.**

   The data is loaded and a new table is created to show the data.

10. **Click the close button of the new table window.**

    A dialog is presented asking if you want to create a recreation macro.

11. **Click the No Save button.**

    The data we just loaded is still available in Igor. A table is just a way of viewing data and is not necessary for the data to exist.

The Load Delimited Text menu item that you used is a shortcut that uses default settings for loading delimited text. Later, when you load your own data files, choose Data→Load Waves→Load Waves so you can see all of the options.

Appending to a Graph

1. **If necessary, click in Graph0 to bring it to the front.**

   The Graph menu is available only when the target window is a graph.

2. **Choose the Graph→Append Traces to Graph menu item.**

   The Append Traces dialog appears. It is very similar to the New Graph dialog that you used to create the graph.

3. **In the Y Waves list, select voltage_1 and voltage_2.**
4. **In the X Wave list, select timeval2.**
5. **Click Do It.**

   Two additional traces are appended to the graph. Notice that they are also appended to the Legend.

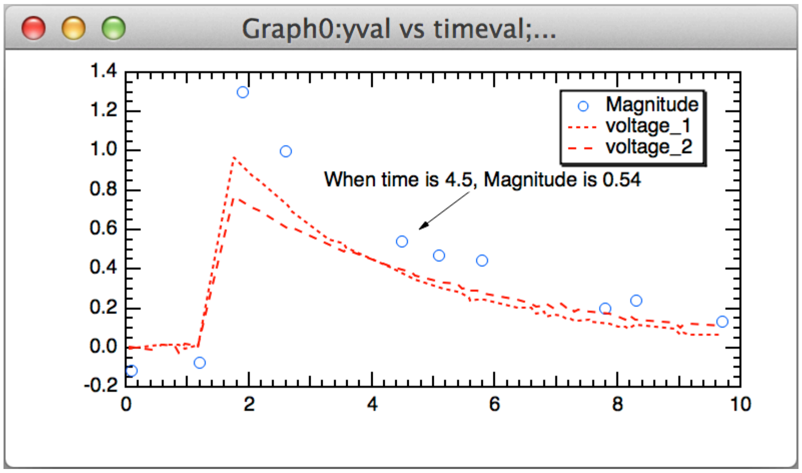
6. **Position the cursor over one of the traces in the graph and double-click.**

   The Modify Trace Appearance dialog appears with the trace you double-clicked already selected.

7. **If necessary, select voltage_1 in the list of traces.**

8. **Choose dashed line #2 from the Line Style pop-up menu.**
9. **Select voltage_2 in the list of traces.**
10. **Choose dashed line #3 from the Line Style pop-up menu.**
11. **Click Do It.**

    The graph should now look like this:



Offsetting a Trace

1. **Position the cursor directly over the voltage_2 trace.**

    The voltage_2 trace has the longer dash pattern.

2. **Click and hold the mouse button for about 1 second.**

    An XY readout appears in the lower-left corner of the graph and the trace will now move with the mouse.

3. **With the mouse button still down, press Shift while dragging the trace up about 1 cm and release.**

    The Shift key constrains movement to vertical or horizontal.

    You have added an offset to the trace. If desired, you could add a tag to the trace indicating that it has been offset and by how much. This trace offset does not affect the underlying wave data.

Unoffsetting a Trace

4. **Choose the Edit→Undo Trace Drag menu item.**

    You can undo many of the interactive operations on Igor windows.

5. **Choose Edit→Redo Trace Drag.**

    The following steps show how to remove an offset after it is no longer undoable.

6. **Double-click the voltage_2 trace.**

    The Modify Trace Appearance dialog appears with voltage_2 selected. (If voltage_2 is not selected, select it.) The Offset checkbox is checked.

7. **Click the Offset checkbox.**

    This turns offset off for the selected trace and the offset controls in the dialog are hidden.

8. **Click Do It.**

    The voltage_2 trace is returned to its original position.

Drawing in a Graph

1. **If necessary, click Graph0 to bring it to the front.**

2. **Choose the Graph→Show Tools menu item or press Command-T (*Macintosh*) or Ctrl+T (*Windows*).**

   A tool palette is added to the graph. The second icon from the top ( ) is selected indicating that the graph is in drawing mode as opposed to normal (or “operate”) mode.

3. **Click the top icon ( ) to go into normal mode.**

   Normal mode is for interacting with graph objects such as traces, axes and annotations. Drawing mode is for drawing lines, rectangles, polygons and so on.

4. **Click the second icon to return to drawing mode.**

5. **Click the drawing layer icon .**

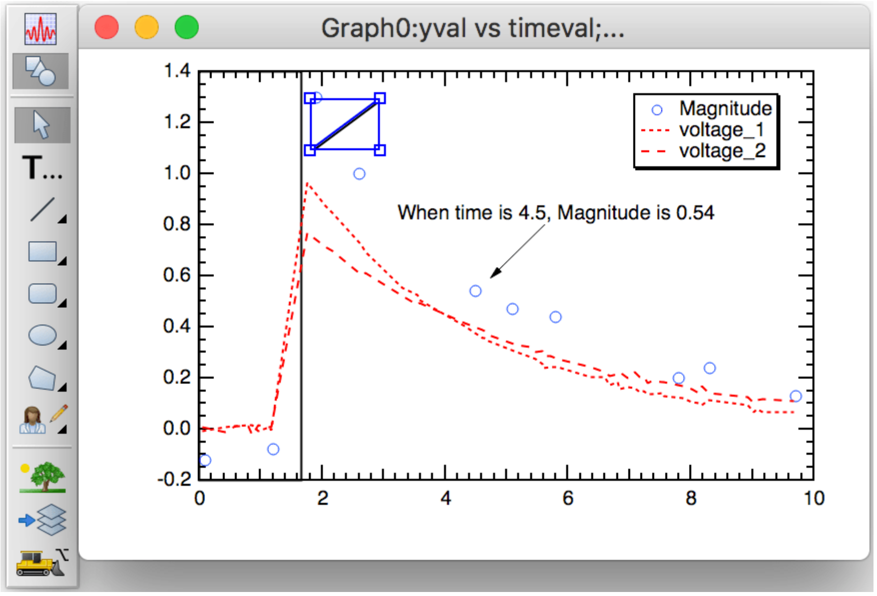
   A pop-up menu showing the available drawing layers and their relationship to the graph layers appears. The items in the menu are listed in back-to-front order.

6. **Choose UserBack from the menu.**

   We will be drawing behind the axes, traces and all other graph elements.

7. **Click the rectangle tool and drag out a rectangle starting at the upper-left corner of the plot area (y= 1.4, x=0 on the axes) and ending at the bottom of the plot area and about 1.5 cm in width (y= -0.2, x= 1.6).**

8. **Click the line tool and draw a diagonal line as shown, starting at the left, near the peak of the top trace, and ending at the right:**



9. **Click the drawing environment icon ( ) and choose At Start from the Line Arrow item.**

10. **Click the Text tool icon T... .**

11. **Click just to the right of the line you just drew.**

    The Create Text dialog appears.

12. **Type “Precharge”.**

13. **From the Anchor pop-up menu, choose Left Center.**

14. **Click Do It.**

15. **Click the graph's zoom button (*Macintosh*) or maximize button (*Windows*).**

To zoom the graph window on Macintosh, press the Option key while clicking the green button in the top/left corner of the window.

Notice how the rectangle and line expand with the graph. Their coordinates are measured relative to the plot area (rectangle enclosed by the axes).

16. **Click the graph's zoom button (*Macintosh*) or restore button (*Windows*).**

To restore the graph window on Macintosh, press the Option key while clicking the green button in the top/left corner of the window.

17. **Click the Arrow tool and then double-click the rectangle.**

The Modify Rectangle dialog appears showing the properties of the rectangle.

18. **Enter 0 in the Thickness box in the Line Properties section.**

This turns off the frame of the rectangle.

19. **Choose Solid from the Fill Mode pop-up menu.**

20. **Choose a light gray color from the Fore Color pop-up menu under the Fill Mode pop-up menu.**

21. **Click Do It.**

Observe that the rectangle forms a gray area behind the traces and axes.

22. **Again, double-click the rectangle.**

The Modify Rectangle dialog appears.

23. **From the X Coordinate pop-up menu, choose Axis Bottom.**

The X coordinates of the rectangle will be measured in terms of the bottom axis — as if they were data values.

24. **Press Tab until the X0 box is selected and type "0".**

25. **Tab to the X1 box and type "1.6".**

26. **Tab to the Y0 box and type "0".**

27. **Tab to Y1 and type "1".**

The X coordinates of the rectangle are now measured in terms of the bottom axis and the left side will be at zero while the right side will be at 1.6.

The Y coordinates are still measured relative to the plot area. Since we entered zero and one for the Y coordinates, the rectangle will span the entire height of the plot area.

28. **Click Do It.**

Notice the rectangle is nicely aligned with the axis and the plot area.

29. **Click the operate icon ( ) to exit drawing mode.**

30. **Press Option (*Macintosh*) or Alt (*Windows*), click in the middle of the plot area and drag about 2 cm to the right.**

The X axis range changes. Notice that the rectangle moved to align itself with the bottom axis.

31. **Choose Edit→Undo Modify.**

Making a Window Recreation Macro

1. **Click the graph's close button.**

Igor presents a dialog which asks if you want to save a window recreation macro. The graph's name is "Graph0" so Igor suggests "Graph0" as the macro name.

2. **Click Save.**

Igor generates a window recreation macro in the currently hidden procedure window. A window recreation macro contains the commands necessary to recreate a graph, table, page layout, control panel or 3D plot. You can invoke this macro to recreate the graph you just closed.

3. **Choose the Windows→Procedure Windows→Procedure Window menu item.**

   The procedure window is always present but is usually hidden to keep it out of the way. The window now contains the recreation macro for Graph0. You may need to scroll up to see the start of the macro. Because of the way it is declared:

   ```
   Window Graph0() : Graph
   ```

   this macro will be available from the Graph Macros submenu of the Windows main menu.

4. **Click the procedure window's close button.**

   This hides the procedure window. Most other windows display a dialog asking if you want to kill or hide the window, but the built-in procedure window and the help windows simply hide themselves.

Recreating the Graph

1. **Choose the Windows→Graph Macros→Graph0 menu item.**

   Igor executes the Graph0 macro which recreates a graph of the same name.

2. **Repeat step 1.**

   The Graph0 macro is executed again but this time Igor gave the new graph a slightly different name, Graph0_1, because a graph named Graph0 already existed.

3. **While pressing Option (*Macintosh*) or Alt (*Windows*), click the close button of Graph0_1.**

   The window is killed without presenting a dialog.

Using the Data Browser

The Data Browser lets you navigate through the data folder hierarchy and examine properties of waves and values of numeric and string variables.

1. **Choose the Data→Data Browser menu item.**

   The Data Browser appears.

2. **Make sure all of the checkboxes in the top-left corner of the Data Browser are checked.**

3. **Click the timeval wave icon to select it.**

   Note that the wave is displayed in the plot pane at the bottom of the Data Browser and the wave's properties are displayed just above in the info pane.

   If you don't see this, click the info icon (ⓘ) half-way down the left side fo the Data Browser window.

4. **Control-click (*Macintosh*) or right-click (*Windows*) on the timeval wave icon.**

   A contextual menu appears with a number of actions that you can perform on the selection.

5. **Press Escape to dismiss the contextual menu.**

   You can explore that and other Data Browser features later on your own.

6. **Click the Data Browser's close box to close it.**

Saving Your Work - Tour 1B

7. **Choose the File→Save Experiment As menu item.**

8. **Navigate to your "Guided Tours" folder.**

   This is the folder that you created under **Saving Your Work - Tour 1A** on page I-21.

9. **Change the name to "Tour 1B.pxp" and click Save.**

   If you want to take a break, you can quit from Igor now.

Using Igor Documentation

Now we will take a quick look at how to find information about Igor.

In addition to guided tours such as this one, Igor includes tooltips, general usage information, and reference information. The main guided tours, as well as the general and reference information, are available in both the online help files and in the Igor Pro PDF manual.

1. **Choose Misc→Miscellaneous Settings, click the Help icon on the left side, and verify that the Show Tooltips checkbox is checked.**

   If it is unchecked, check it..

2. **Click Save Settings to close the Miscellaneous Settings dialog.**

3. **Choose Data→Load Waves→Load Waves.**

   Igor displays the Load Waves dialog. This dialog provides an interface to the LoadWave operation which is how you load data into Igor from text data files.

4. **On Macintosh only, move the cursor over the Load Columns Into Matrix checkbox.**

   A tooltip appears in a yellow textbox. You can get a tip for most dialog items and icons this way.

5. **Click the Cancel button to quit the dialog.**

   Now let's see how to get reference help for a particular operation.

6. **Choose Help→Command Help.**

   The Igor Help Browser appears with the Command Help tab displayed.

   The information displayed in this tab comes from the Igor Reference help file - one of many help files that Igor automatically opens at launch. Open help files are directly accessible through Help→Help Windows but we will use the Igor Help Browser right now.

7. **If there is a Show All link above the left-hand list, click it.**

8. **Click any item in the list and then type "L".**

   Igor displays help for the first list item whose name starting with "L". We want the LoadWave operation.

9. **Press the down-arrow key until LoadWave is selected in the list.**

   Igor displays help for the LoadWave operation in the help area on the right.

   Another way to get reference help is to Control-click (Macintosh) or right-click (Windows) the name of an operation or function and choose the "Help For" menu item. This works in the command window and in procedure, notebook and help windows.

10. **In the Filter edit box just below the left-hand list, type “Matrix”.**

    The list now shows only operations, functions, and keywords whose names include “matrix”.

11. **Click Show All above the left-hand list.**

    The list shows all operations, functions, and keywords again.

12. **Click the Advanced Filtering control to reveal additional checkboxes and pop-up menus.**

    These controls provide other ways to filter what appears in the left-hand list.

    While we're in the Igor Help Browser, let's see what the other tabs are for.

13. **Click each of the Help Browser tabs and note their contents.**

    You can explore these tabs in more detail later.

    Next we will take a quick trip to the Igor Pro PDF manual. If you are doing this guided tour using the PDF manual, you may want to just read the following steps rather than do them to avoid losing your place.

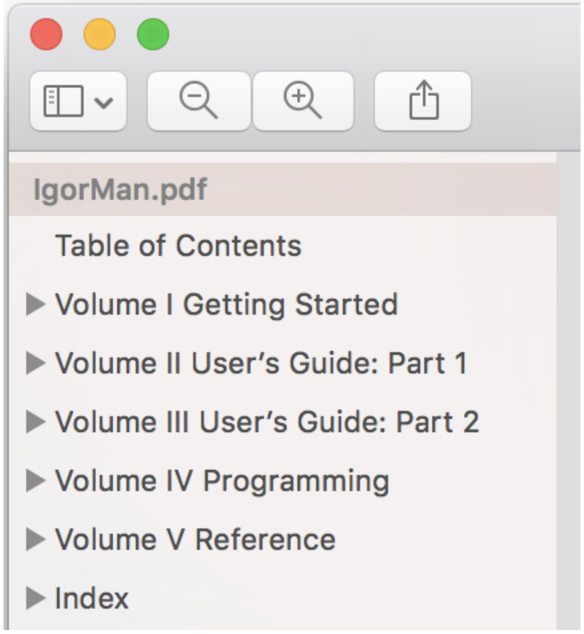
14. **Click the Manual tab and then click the Open Online Manual button.**

Igor opens the PDF manual in your PDF viewer - typically Adobe Reader or Apple's Preview program.

If you use Adobe Reader for viewing PDF files, you should have a Bookmarks pane on the left side of the PDF manual window. If not, choose View→Show/Hide→Navigation Panes→Bookmarks in Reader.

If you use Apple's Preview for PDF files, you should have a sidebar displaying the table of contents on one side of the main page. If not, choose View→Table of Contents in Preview.

Note in the Reader Bookmarks pane or the Preview table of contents that the PDF manual is organized into five volumes plus an index.



IgorMan.pdf

Table of Contents

▶ Volume I Getting Started

▶ Volume II User’s Guide: Part 1

▶ Volume III User’s Guide: Part 2

▶ Volume IV Programming

▶ Volume V Reference

▶ Index

15. **Use the bookmarks or table of contents to get a sense of what's in the manual.**

Expand the volume bookmarks to see the chapter names.

You may notice that the Igor PDF manual is rather large - over 2,000 pages at last count. You'll be happy to know that we don't expect you to read it cover-to-cover. Instead, read chapters as the need arises.

The information in the manual is also in the online help files. The manual, being in book format, is better organized for linear reading while the online help is usually preferred for accessing reference information.

In case you ever want to open it directly, you can find the PDF manual in "Igor Pro Folder/Manual".

That should give you an idea of where to look for information about Igor. Now let's get back to our hands-on exploration of Igor.

Graphically Editing Data

1. **If you quit Igor after the last save, open your “Tour 1B.pxp” experiment and turn preferences off.**
2. **Adjust the positions of the graph and table so you can see both.**

   Make sure you can see the columns of data in the table when the graph is the front window.
3. **If necessary, click the Graph0 window to bring it to the front.**
4. **Click the drawing mode icon ( ) to activate the drawing tools.**
5. **Move the cursor over the polygon icon ( ) and click and hold the mouse button.**

   A pop-up menu appears.

6. **Choose the Edit Wave menu item.**

7. **Click one of the open circles of the yval trace.**

   The trace is redrawn using lines and squares to show the location of the data points.

8. **Click the second square from the left and drag it 1 cm up and to the right.**

   Notice point 1 of yval and timeval changes in the table.

9. **Press Command-Z (*Macintosh*) or Ctrl+Z (*Windows*) or choose Edit→Undo.**

10. **Click midway between the first and second point and drag up 1 cm.**

    A new data point is added to the yval and timeval waves.

11. **Press Option (*Macintosh*) or Alt (*Windows*) and click the new data point.**

    The new data point is deleted.

    You could also have pressed Command-Z (*Macintosh*) or Ctrl+Z (*Windows*) to undo the insertion.

12. **Press Command (*Macintosh*) or Ctrl (*Windows*), click the line segment between the second and third point and drag a few cm to the right.**

    The line segment is moved and two points of yval and timeval are changed in the table.

13. **Press Command-Z (*Macintosh*) or Ctrl+Z (*Windows*) or choose Edit→Undo.**

14. **Click in the operate icon ( ) to exit drawing mode.**

15. **Choose File→Revert Experiment and answer *Yes* in the dialog.**

    This returns the experiment to the state it was in before we started editing the data.

16. **Choose File→New Experiment.**

    This clears the windows and data from the previously open experiment, creating a new, empty experiment.

Making a Category Plot (Optional)

Category plots show continuous numeric data plotted against non-numeric text categories.

1. **Choose the Windows→New Table menu item.**

2. **Click in the Do It button.**

   A new blank table is created.

3. **Type "Monday" (without the quotes) and then press Return or Enter.**

   A wave named "textWave0" was created with the text Monday as the value of the first point. Entering a non-numeric value in the first row of the first blank column automatically creates a new text wave.

4. **Type the following lines, pressing Enter after each one:**

   Tuesday
   Wednesday
   Thursday

5. **Click in the first cell of the next column and enter the following values:**

   10
   25
   3
   16

6. **Click in the first cell of the next column and enter the following values:**

   0
   12
   30
   17

7. **Choose Windows→New→Category Plot.**

   A dialog similar to the New Graph dialog appears. This dialog shows only text waves in the right-hand list.

8. **In the Y Waves list, select wave0 and wave1.**

9. **In the X Wave listm select textWave0.**

10. **Click Do It.**

    A category plot is created.

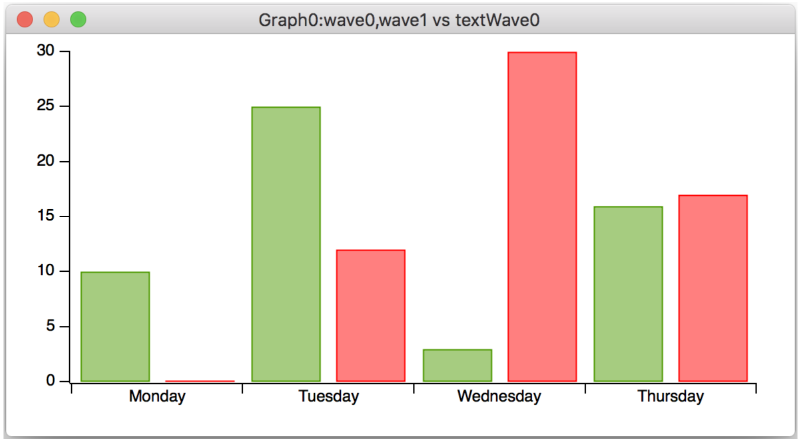
11. **Double-click one of the bars.**

    The Modify Trace Appearance dialog appears.

12. **Using the Color pop-up menu, set the color of the wave0 trace to green.**

13. **Click Do It.**

    The graph should now look like this:



14. **Choose File→Save Experiment As and save the current experiment as “Category Plots.pxp”.**

Category Plot Options (Optional)

This section explores various category-plot options. If you are not particularly interested in category plots, you can stop now, or at any point in the following steps, by skipping to the next section.

1. **Double-click one of the bars and, if necessary, select the wave0 in the list.**

2. **From the Grouping pop-up menu, choose Stack on Next.**

3. **Click Do It.**

   The left bar in each group is now stacked on top of the right bar.

4. **Choose the Graph→Reorder Traces menu item.**

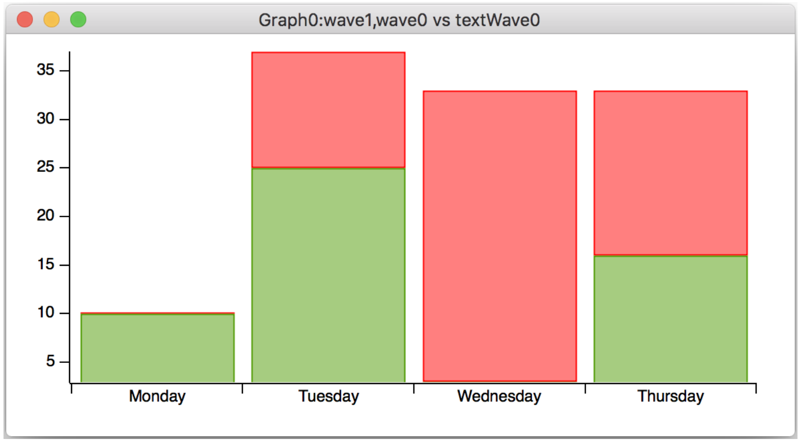
   The Reorder Traces dialog appears.

5. **Reverse the order of the items in the list by dragging the top item down and click Do It.**

   The bars are no longer stacked and the bars that used to be on the left are now on the right. The reason the bars are not stacked is that the trace that we set to Stack on Next mode is now last and there is no next trace.

6. **Using the Modify Trace Appearance dialog, set the wave1 trace to Stack on next. Click Do It.**

   The category plot graph should now look like this:



7. **Enter the following values in the next blank column in the table:**

   7
   10
   15
   9

   This creates a new wave named wave2.

8. **Click the graph to bring it to the front.**

9. **Choose Graph→Append to Graph→Category Plot.**

   The Append Category Traces dialog appears.

10. **In the Y Waves list, select wave2 and click Do It.**

    This adds a red bar underneath each green bar.

11. **Control-click (Macintosh) or right-click (Windows) one of the new red bars (underneath a green bar) to display the contextual pop-up menu and choose blue from the Color submenu.**

    The new bars are now blue.

12. **Using the Modify Trace Appearance dialog, change the grouping mode of the middle trace, wave0, to none.**

    We now have wave1 (red) stacked on wave0 (green) and wave0 not stacked on anything.

    Now the new wave2 bars (blue) are to the right of a group of two stacked bars. You can create any combination of stacked and side-by-side bars.

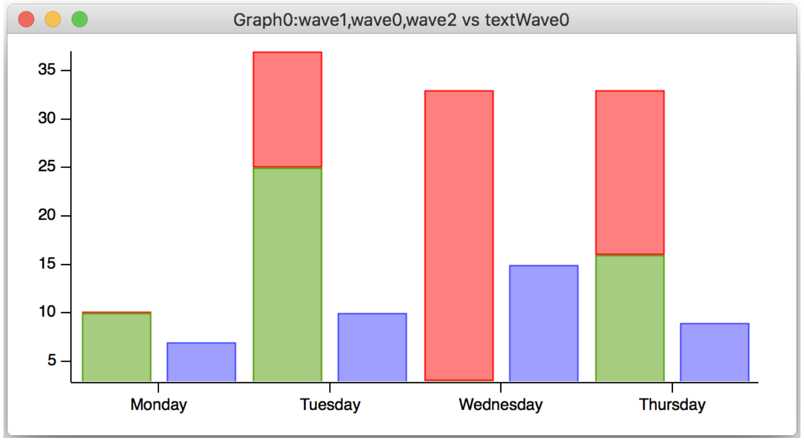
13. **Double-click the bottom axis.**

    The Modify Axis dialog appears with the bottom axis selected.

14. **Click the Auto/Man Ticks tab.**

15. **Click the Tick In Center checkbox and then click Do It.**

    Notice the new positions of the tick marks.



16. **Double-click the bottom axis again.**
17. **Click the Axis tab.**
18. **Change the value of Bar Gap to zero and then click Do It.**

    Notice that the bars within a group are now touching.

19. **Use the Modify Axis dialog to set the Category Gap to 50%.**

    The widths of the bars shrink to 50% of the category width.

20. **Choose Graph→Modify Graph.**
21. **Click the "Swap X & Y Axes" checkbox and then click Do It.**

    This is how you create a horizontal bar plot.

22. **Choose File→Save Experiment.**

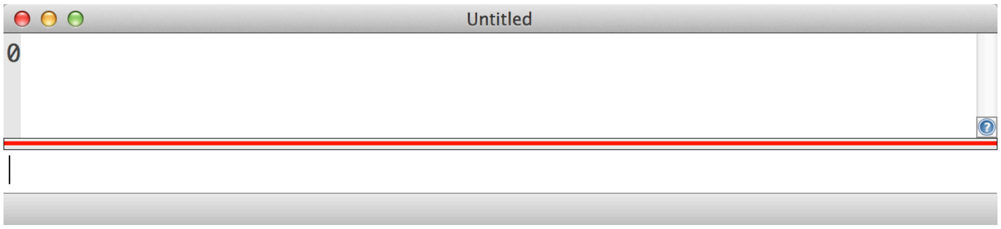
The Command Window

Parts of this tour make use of Igor's command line to enter mathematical formulae. Let's get some practice now.

1. **Choose File→New Experiment.**

   This clears any windows and data left over from previous experimentation.

   Your command window should look something like this:



   If you don't see the command window, choose Windows→Command Window.

   The command line is the space below the red separator whereas the space above the separator is called the history area.

2. **Click in the command line, type the following line and press Return or Enter.**

```
Print 2+2
```

   The Print command as well as the result are placed in the history area.

3. **Press the Up Arrow key.**

   The line containing the print command is selected, skipping over the result printout line.

4. **Press Return or Enter.**

   The selected line in the history is copied to the command line.

5. **Edit the command line so it matches the following and press Return or Enter.**

   ```
   Print "The result is ", 2+2
   ```

   The Print command takes a list of numeric or string expressions, evaluates them and prints the results into the history.

6. **Choose the Help→Igor Help Browser menu item.**

   The Igor Help Browser appears.

   You can also display the help browser by clicking the question-mark icon near the right edge of the command window and, on Windows, by pressing F1.

7. **Click the Command Help tab in the Igor Help Browser.**

8. **Click Advanced Filtering if necessary to reveal the advanced options.**

9. **If there is a Show All link above the left-hand list, click it.**

10. **Uncheck the Functions and Programming checkboxes and check the Operations checkbox.**

    A list of operations appears.

11. **In the pop-up menu next to the Operations checkbox, choose About Waves.**

12. **Select PlaySound in the list.**

    Tip: Click in the list to activate it and then type “p” to jump to PlaySound.

    The reference help for the PlaySound operation appears in the help area on the right.

13. **Click the help area on the right, scroll down to the Examples section, and select the first four lines of example text (starting with “Make”, ending with “PlaySound sineSound).**

14. **Choose the Edit→Copy menu to copy the selection.**

15. **Close the Igor Help Browser.**

16. **Choose Edit→Paste to paste the command into the command line.**

    All four lines are pasted into the command line area.

17. **Make the command window taller and then drag the red divider line up so you can see the commands in the command line.**

18. **Press Return or Enter to execute the commands.**

    The four lines are executed and a short tone plays.

19. **Click once on the last line in the history area, on “PlaySound sineSound”.**

    The entire command is selected just as if you pressed the arrow key.

20. **Press Return or Enter once to transfer the command to the command line and a second time to execute it.**

    The tone plays again as the line executes.

    We are finished with the “sineSound” wave that was created in this exercise so let’s kill the wave to prevent it from cluttering up our wave lists.

21. **Choose Data→Kill Waves.**

    The Kill Waves dialog appears.

22. **Select “sineSound” and click Do It.**

    The sineSound wave is removed from memory.

23. **Again click once on the history line “PlaySound sineSound”.**

24. **Press Return or Enter twice to re-execute the command.**

    An error dialog is presented because the sineSound wave no longer exists.

25. **Click OK to close the error dialog.**

26. **Choose Edit→Clear Cmd Buffer or press Command-K (*Macintosh*) or Ctrl+K (*Windows*).**

    When a command generates an error, it is left in the command line so you can edit and re-execute it. In this case we just wanted to clear the command line.

Synthesizing Data

In this section we will make waves and fill them with data using arithmetic expressions.

1. **Choose File→New Experiment.**

   This clears any windows and data left over from previous experimentation.

2. **Choose the Data→Make Waves menu item.**

   The Make Waves dialog appears.

3. **Type "spiralY" in the first box, press the tab key, and type "spiralX" in the second box.**
4. **Change Rows to 1000.**
5. **Click Do It.**

   Two 1000 point waves have been created. They are now part of the experiment but are not visible because we haven't displayed them in a table or graph.

6. **Choose Data→Change Wave Scaling.**

   The Change Wave Scaling dialog appears. We will use it to set the X scaling of the waves.

7. **If a button labeled More Options is showing, click it.**
8. **In the Waves list, click spiralY and then Command-click (*Macintosh*) or Ctrl-click (*Windows*) spiralX.**
9. **Choose Start and Right in the SetScale Mode pop-up menu.**
10. **Enter "0" for Start and "50" for Right.**
11. **Click Do It.**

    This executes a **SetScale** command specifying the X scaling of the spiralX and spiralY waves. X scaling is a property of a wave that maps a point number to an X value. In this case we are mapping point numbers 0 through 999 to X values 0 through 50.

12. **Type the following on the command line and then press Return or Enter:**

    ```
    spiralY = x*sin(x)
    ```

    This is a waveform assignment statement. It assigns a value to each point of the destination wave (spiralY). The value stored for a given point is the value of the right-hand expression at that point. The meaning of *x* in a waveform assignment statement is determined by the X scaling of the destination wave. In this case, *x* takes on values from 0 to 50 as Igor evaluates the right-hand expression for points 0 through 999.

13. **Execute this in the command line:**

    ```
    spiralX = x*cos(x)
    ```

    Now both spiralX and spiralY have their data values set.

Zooming and Panning

1. **Choose the Windows→New Graph menu item.**
2. **If necessary, uncheck the From Target checkbox.**
3. **In the Y Waves list, select "spiralY".**
4. **In the X Wave list, select "_calculated_".**
5. **Click Do It.**

   Igor creates a graph of spiralY's data values versus its X values.

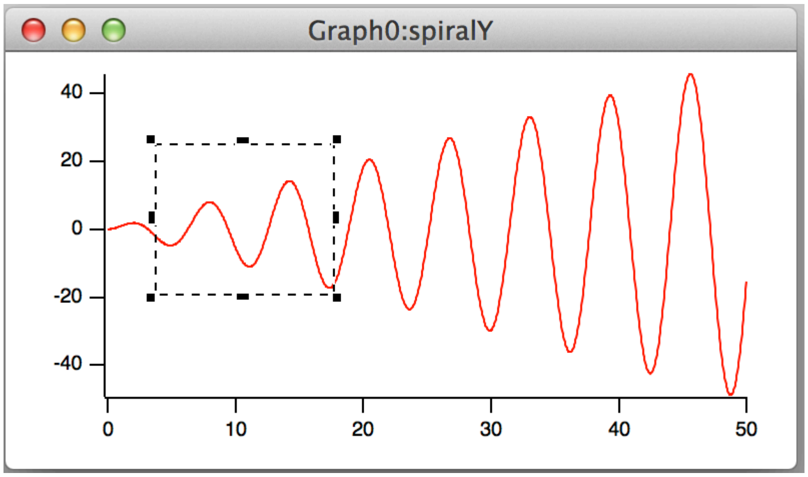
   Note that the X axis goes from 0 to 50. This is because the **SetScale** command we executed earlier set the X scaling property of spiralY which tells Igor how to compute an X value from a point number.

Choosing _calculated_ from the X Wave list graphs the spiralY data values versus these calculated X values.

6. **Position the cursor in the interior of the graph.**

   The cursor changes to a cross-hair shape.

7. **Click and drag down and to the right to create a marquee as shown:**



   You can resize the marquee with the handles (black squares). You can move the marquee by dragging the dashed edge of the marquee.

8. **Position the cursor inside the marquee.**

   The mouse pointer changes to this shape: ▬ , indicating that a pop-up menu is available.

9. **Click and choose Expand from the pop-up menu.**

   The axes are rescaled so that the area enclosed by the marquee fills the graph.

10. **Choose Edit→Undo Scale Change or press Command-Z (*Macintosh*) or Ctrl+Z (*Windows*).**

11. **Choose Edit→Redo Scale Change or press Command-Shift-Z (*Macintosh*) or Ctrl+Shift+Z (*Windows*).**

12. **Press Option (*Macintosh*) or Alt (*Windows*) and position the cursor in the middle of the graph.**

    The cursor changes to a hand shape. You may need to move the cursor slightly before it changes shape.

13. **With the hand cursor showing, drag about 2 cm to the left.**

14. **While pressing Option (*Macintosh*) or Alt (*Windows*), click the middle of the graph and gently fling it to the right.**

    The graph continues to pan until you click again to stop it.

15. **Click the plot area of the graph to stop panning.**

16. **Choose Graph→Autoscale Axes or press Command-A (*Macintosh*) or Ctrl+A (*Windows*).**

    Continue experimenting with zooming and panning as desired.

17. **Press Command-Option-W (*Macintosh*) or Ctrl+Alt+W (*Windows*).**

    The graph is killed. Option (*Macintosh*) or Alt (*Windows*) avoided the normal dialog asking whether to save the graph.

Making a Graph with Multiple Axes

1. **Choose the Windows→New Graph menu item.**

2. **If you see a button labeled More Choices, click it.**

   We will use the more complex form of the dialog to create a multiple-axis graph in one step.

3. **In the Y Waves list, select “spiralY”.**

4. **In the X Wave list, select “spiralX”.**

5. **Click Add.**

   The selections are inserted into the lower list in the center of the dialog.

6. **In the Y Waves list, again select “spiralY”.**

7. **In the X Wave list, select “_calculated_”.**

8. **Choose New from the Axis pop-up menu under the X Waves list.**

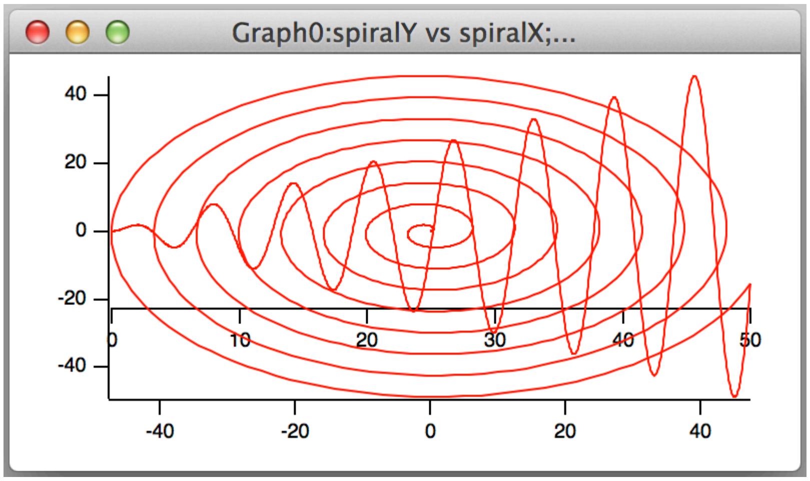
9. **Enter “B2” in the name box.**

10. **Click OK.**

    Note the command box at the bottom of the dialog. It contains two commands: a Display command corresponding to the initial selections that you added to the lower list and an AppendToGraph command corresponding to the current selections in the Y Waves and X Wave lists.

11. **Click Do It.**

    The following graph is created:



The interior axis is called a “free” axis because it can be moved relative to the plot rectangle. We will be moving it outside of the plot area but first we must make room by adjusting the plot area margins.

12. **Press Option (*Macintosh*) or Alt (*Windows*) and position the cursor over the bottom axis until the cursor changes to this shape: ╪.**

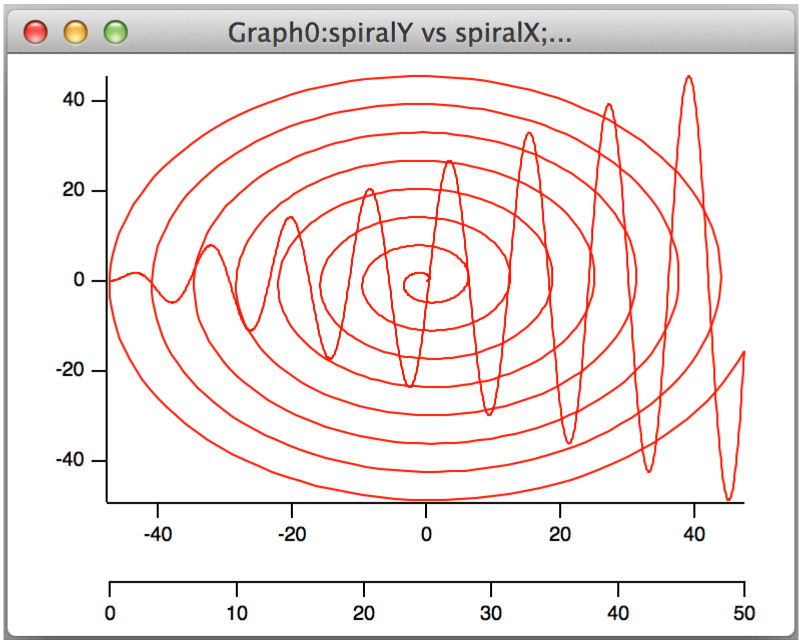
    This shape indicates you are over an edge of the plot area rectangle and that you can drag that edge to adjust the margin.

13. **Drag the margin up about 2 cm. Release the Option (*Macintosh*) or Alt (*Windows*).**

14. **Drag the interior axis down into the margin space you just created.**

15. **Resize the graph so the spiral is nearly circular.**

    The graph should now look like this:



Saving Your Work - Tour 1C

1. **Choose the File→Save Experiment As menu item.**
2. **Navigate to your "Guided Tours" folder.**

   This is the folder that you created under **Saving Your Work - Tour 1A** on page I-21.
3. **Type "Tour 1C.pxp" in the name box and click Save.**

   If you want to take a break, you can quit from Igor now.

Using Cursors

1. **If you are returning from a break, open your "Tour 1C.pxp" experiment and turn preferences off.**
2. **Click in the graph and choose the Graph→Show Info menu item.**

   A cursor info panel appears below the graph.
3. **Control-click (*Macintosh*) or right-click (*Windows*) in the name area for graph cursor A (the round one), where it says "A:".**
4. **Choose "spiralY" from the pop-up menu.**

   The A cursor is placed on point zero of spiralY.
5. **Repeat for cursor B but choose "spiralY#1" from the pop-up menu.**

   The wave spiralY is graphed twice. The #1 suffix is used to distinguish the second instance from the first. It is #1 rather than #2 because in Igor indices start from zero.
6. **Position the mouse pointer over the center of the cursor position control .**
7. **Click the blue slider and gently drag it to the right.**

   Both cursors move to increasing point numbers. They stop when one or both get to the end.

   You can also move the cursors using the left and right arrow keys on the keyboard or by clicking to the left or right of the blue slider.
8. **Practice moving the slider to the left and right.**

   Notice that the cursors move with increasing speed as the slider is displaced farther from the center.

9. **Click once on the dock for cursor A (the round black circle).**

   The circle turns white, indicating that cursor A is deselected.

10. **Move the slider to the left and right.**

    Notice that only cursor B moves.

11. **Click cursor B in the graph and drag it to another position on either trace.**

    You can also drag cursors from their docks to the graph.

12. **Click cursor A in the graph and drag it completely outside the graph.**

    The cursor is removed from the graph and returns to its dock.

13. **Choose Graph→Hide Info.**

14. **Click in the command window, type the following and press Return or Enter.**

    ```
    Print vcsr(B)
    ```

    The Y value at cursor B is printed into the history area. There are many functions available for obtaining information about cursors.

15. **Click in the graph and then drag cursor B off of the graph.**

Removing a Trace and Axis

16. **Choose the Graph→Remove from Graph menu item.**

    The Remove From Graph dialog appears with spiralY listed twice. When we created the graph we used spiralY twice, first versus spiralX to create the spiral and second versus calculated X values to show the sine wave.

17. **Click the second instance of spiralY (spiralY#1) and click Do It.**

    The sine wave and the lower axis are removed. An axis is removed when its last trace is removed.

18. **Drag the horizontal axis off the bottom of the window.**

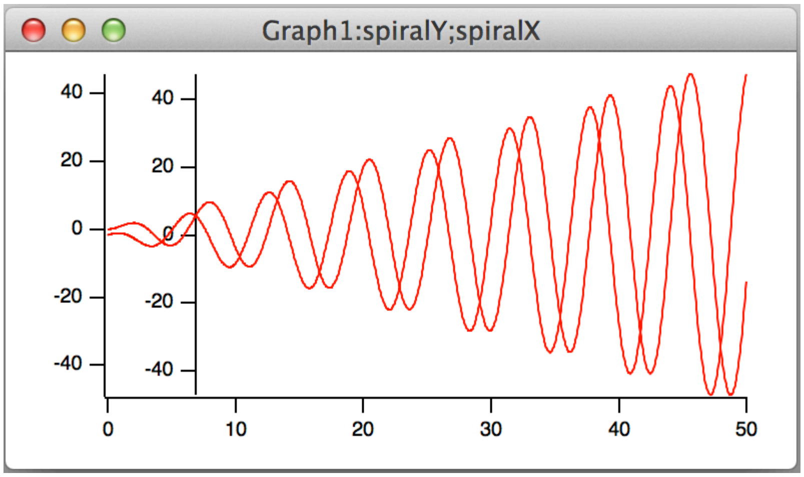
    This returns the margin setting to auto. We had set it to a fixed position when we option-dragged (*Macintosh*) or Alt-dragged (*Windows*) the margin in a previous step.

Creating a Graph with Stacked Axes

1. **Choose the Windows→New Graph menu item.**
2. **If you see a button labeled More Choices, click it.**
3. **In the Y Waves list, select “spiralY”.**
4. **In the X Wave list, select “_calculated_”.**
5. **Click Add.**
6. **In the Y Waves list, select “spiralX”.**
7. **In the X Wave list, select “_calculated_”.**
8. **Choose New from the Axis pop-up menu under the Y Waves list.**
9. **Enter “L2” in the name box.**
10. **Click OK.**

11. **Click Do It.**

    The following graph is created.



In the following steps we will stack the L2 axis on top of the left axis.

12. **Double-click the far left axis.**

    The Modify Axis dialog appears. If any other dialog appears, cancel and try again making sure the cursor is over the axis.

    The Left axis should be selected in the Axis pop-up menu in the upper-left corner of the dialog.

13. **Click the Axis tab.**

14. **Set the Left axis to draw between 0 and 45% of normal.**

15. **Choose L2 from the Axis pop-up menu.**

16. **Set the L2 axis to draw between 55 and 100% of normal.**

17. **In the Free Axis Position box, pop up the menu reading Distance from Margin and select Fraction of Plot Area.**

18. **Verify that the box labeled "% of Plot Area" is set to zero.**

    Steps 17 and 18 move the L2 axis so it is in line with the Left axis.

    Why don't we make this the default? Good question — positioning as percent of plot area was added in Igor Pro 6; the default behavior maintains backward compatibility.

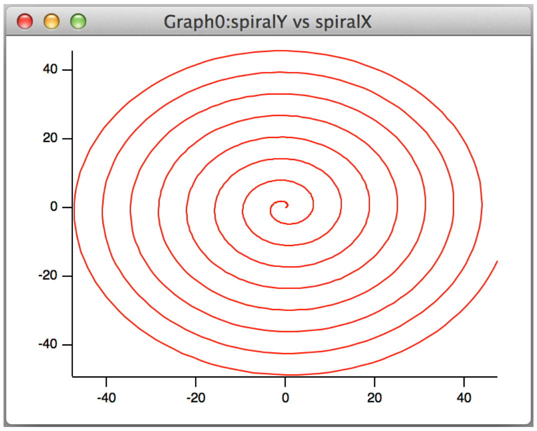
19. **Choose Bottom from the Axis pop-up menu.**

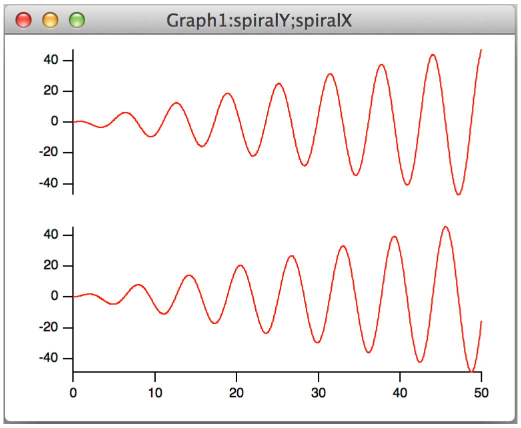
20. **Click the Axis Standoff checkbox to turn standoff off.**

21. **Click Do It.**

22. **Resize and reposition the Graph0 and Graph1 windows so they are side-by-side and roughly square.**

    The graphs should look like this:





Creating a Page Layout

1. **Choose the Windows→New Layout menu item and click Do It.**

   A new blank page layout window is created.

2. **Click in the graph icon ( ) and choose “Graph0”.**

   Graph0 is added to the layout.

3. **Again, click in the graph icon and choose “Graph1”.**

   Graph1 is added to the layout.

4. **Click the marquee icon ⬚.**

5. **Drag out a marquee that approximately fills the bottom half of the page.**

6. **Choose Layout→Arrange Objects.**

   The Arrange Objects dialog appears.

7. **Select both Graph0 and Graph1 and leave the Use Marquee checkbox checked.**

8. **Click Do It.**

   The two graphs are tiled inside the area defined by the marquee.

9. **Click in the page area outside the marquee to dismiss it.**

10. **Choose Windows→Send to Back.**

    This sends the page layout window behind all other windows.

Saving Your Work - Tour 1D

1. **Choose the File→Save Experiment As menu item.**

2. **Navigate to your “Guided Tours” folder.**

   This is the folder that you created under **Saving Your Work - Tour 1A** on page I-21.

3. **Type “Tour 1D.pxp” in the name box and click Save.**

   If you want to take a break, you can quit Igor Pro now.

Creating Controls

This section illustrates adding controls to an Igor graph — the type of thing a programmer might want to do. If you are not interested in programming, you can skip to the **End of the General Tour** on page I-45.

1. **If you are returning from a break, open your “Tour 1D.pxp” experiment and turn off preferences.**

2. **Click the graph with the spiral (Graph0) to bring it to the front.**

3. **Choose the Graph→Show Tools menu item or press Command-T (*Macintosh*) or Ctrl+T (*Windows*).**

   A tool palette is displayed to the left of the graph. The second icon is selected indicating that the graph is in the drawing as opposed to normal mode.

   The selector tool (arrow) is active. It is used to create, select, move and resize controls.

4. **Choose Graph→Add Control→Control Bar.**

   The Control Bar dialog appears.

5. **Enter a height of 30 points and click Do It.**

   This reserves a space at the top of the graph for controls.

6. **Click in the command line, type the following and press Return or Enter.**

   ```
   Variable ymult=1, xmult=1
   ```

   This creates two numeric variables and sets both to 1.0.

7. **Click Graph0 and then choose Graph→Add Control→Add Set Variable.**

   The SetVariable Control dialog appears.

   A SetVariable control provides a way to display and change the value of a variable.

8. **Choose ymult from the Value pop-up menu.**

9. **Enter 100 in the Width edit box.**

   This setting is back near the top of the scrolling list.

10. **Set the High Limit, Low Limit, and Increment values to 10, 0.1, and 0.1 respectively.**

    You may need to scroll down to find these settings.

11. **From the Font Size pop-up menu, choose 12.**

    You may need to scroll down to find this pop-up menu.

12. **Click Do It.**

    A SetVariable control attached to the variable ymult appears in the upper-left of the control bar.

13. **Double-click the ymult control.**

    The SetVariable Control dialog appears.

14. **Click the Duplicate button at the bottom of the dialog.**

15. **Choose xmult as the value.**

16. **Click Do It.**

    A second SetVariable control appears in the control bar. This one is attached to the xMult variable.

17. **Choose Graph→Add Control→Add Button.**

    The Button Control dialog appears.

18. **Tab to the Title edit box and enter “Update”.**

19. **Click the New button adjacent to Procedure.**

    The Control Procedure dialog appears in which you can create or edit the procedure to be called when control-related events occur. Such procedures are called “control action procedures”.

20. **Make sure the “Prefer structure-based procedures” checkbox is checked.**

21. **Edit the procedure text so it looks like this:**

    ```
    Function ButtonProc(ba) : ButtonControl
        STRUCT WMButtonAction& ba

        switch(ba.eventCode)
            case 2:                              // Mouse up
                WAVE spiralX
    ```

```
                NVAR xmult
                spiralX = x*cos(xmult*x)

                WAVE spiralY
                NVAR ymult
                spiralY = x*sin(ymult*x)

                break
        endswitch

        return 0
End
```

Proofread the function to make sure you entered it as shown above.

22. **Click the Save Procedure Now button.**

    The Control Procedure dialog disappears and the text you edited is inserted into the (currently hidden) procedure window.

23. **Click Do It.**

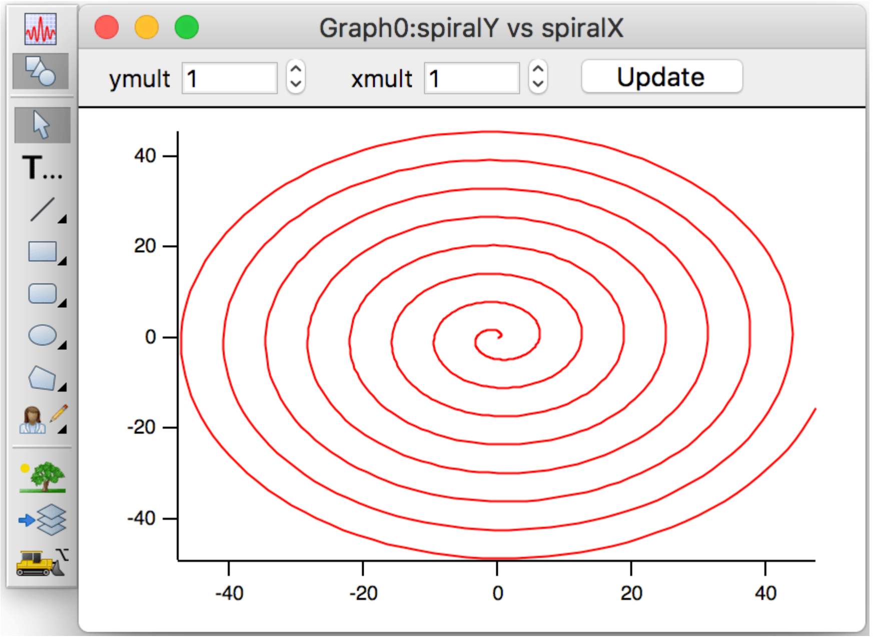
    A Button control is added to the control bar.

    The three controls are now functional but are not esthetically arranged.

24. **Use the Arrow tool to rearrange the three controls into a more pleasing arrangement. Expand the button so it doesn't crowd the text by dragging its handles.**

    After selecting a control with the arrow tool, you can drag it or use the arrow keys on the keyboard to finetune its position.

    The graph now looks like this:



25. **Click the top icon in the tool palette to enter "operate mode".**
26. **Choose Graph→Hide Tools or press Command-T (*Macintosh*) or Ctrl+T (*Windows*).**
27. **Click the up arrow in the ymult control.**

    The value changes to 1.1.

28. **Click the Update button.**

    The ButtonProc procedure that you created executes. The spiralY and spiralX waves are recalculated according to the expressions you entered in the procedure and the graphs are updated.

29. **Experiment with different ymult and xmult settings as desired.**

30. **Set both values back to 1 and click the Update button.**

    You can edit a value by typing in the SetVariable control and enter it by pressing Return or Enter.

Creating a Dependency

A dependency is a rule that relates the value of an Igor wave or variable to the values of other waves or variables. By setting up a dependency you can cause Igor to automatically update a wave when another wave or variable changes.

1. **Click the command window to bring it to the front.**
2. **Execute the following commands in the command line:**

    ```
    spiralY := x*sin(ymult*x)
    spiralX := x*cos(xmult*x)
    ```

    This is exactly what you entered before except here := is used in place of =. The := operator creates a dependency formula. In the first expression, the wave spiralY is made dependent on the variable ymult. If a new value is stored in ymult then the values in spiralY are automatically recalculated from the expression.
3. **Click Graph0 to bring it to the front.**
4. **Adjust the ymult and xmult controls but do not click the Update button.**

    When you change the value of ymult or xmult using the SetVariable control, Igor automatically executes the dependency formula. The spiralY or spiralX waves are recalculated and both graphs are updated.
5. **On the command line, execute this:**

    ```
    ymult := 3*xmult
    ```

    The ymult SetVariable control as well as the graphs are updated.
6. **Adjust the xmult value.**

    Again notice that ymult as well as the graphs are updated.
7. **Choose the Misc→Object Status menu item.**

    The Object Status dialog appears. You can use this dialog to examine dependencies.
8. **Click the the Current Object pop-up and choose spiralY from the Data Objects list.**

    The list on the right indicates that spiralY depends on the variable ymult.
9. **Double-click the ymult entry in the right-hand list.**

    ymult becomes the current object. The list on the right now indicates that ymult depends on xmult.
10. **Click the Delete Formula button.**

    Now ymult no longer depends on xmult.
11. **Click Done.**
12. **Adjust the xmult setting.**

    The ymult value is no longer automatically recalculated but the spiralY and spiralX waves still are.
13. **Click the Update button.**
14. **Adjust the xmult and ymult settings.**

    The spiralY and spiralX waves are no longer automatically recalculated. This is because the ButtonProc function called by the Update button does a normal assignment using = rather than := and that action removes the dependency formulae.

In real work, you should avoid using dependencies because they are hard to keep track of and debug. If a button can do the job then use it rather than the dependency.

Saving Your Work - Tour 1E

1. **Choose the File→Save Experiment As menu item.**

2. **Navigate to your “Guided Tours” folder.**

   This is the folder that you created under **Saving Your Work - Tour 1A** on page I-21.

3. **Type “Tour 1E.pxp” in the name box and click Save.**

End of the General Tour

This is the end of the general tour of Igor Pro.

If you want to take a break, you can quit from Igor Pro now.

Guided Tour 2 - Data Analysis

In this tour we will concentrate on the data analysis features of Igor Pro. We will generate synthetic data and then manipulate it using sorting and curve fitting.

Starting Guided Tour 2

1. **If Igor is already running, activate it and choose File→New Experiment.**

   In this case, skip to step 2.

2. **Double-click your Igor64 alias or shortcut.**

   Instructions for creating this alias or shortcut can be found under **Creating an Igor64 Alias or Shortcut** on page I-13.

   On Windows, you an also launch Igor64 using the Start menu.

3. **Choose Misc→Preferences Off.**

   Turning preferences off ensures that the tour works the same for everyone.

Creating Synthetic Data

We need something to analyze, so we generate some random X values and create some Y data using a math function.

1. **Type the following in the command line and then press Return or Enter:**

   ```
   SetRandomSeed 0.1
   ```

   This initializes the random number generator so you will get the same results as this guided tour.

2. **Type the following in the command line and then press Return or Enter:**

   ```
   Make/N=100 fakeX = enoise(5)+5,fakeY
   ```

   This generates two 100 point waves and fills fakeX with evenly distributed random values ranging from 0 to 10.

3. **Execute this in the same way:**

   ```
   fakeY = exp(-(fakeX-4)^2)+gnoise(0.1)
   ```

   This generates a Gaussian peak centered at 4.

4. **Choose the Windows→New Graph menu item.**
5. **If you see a button labeled Fewer Choices, click it.**
6. **In the Y Waves list, select “fakeY”.**
7. **In the X Wave list, select “fakeX”.**
8. **Click Do It.**

   The graph is a rat’s nest of lines because the X values are not sorted.

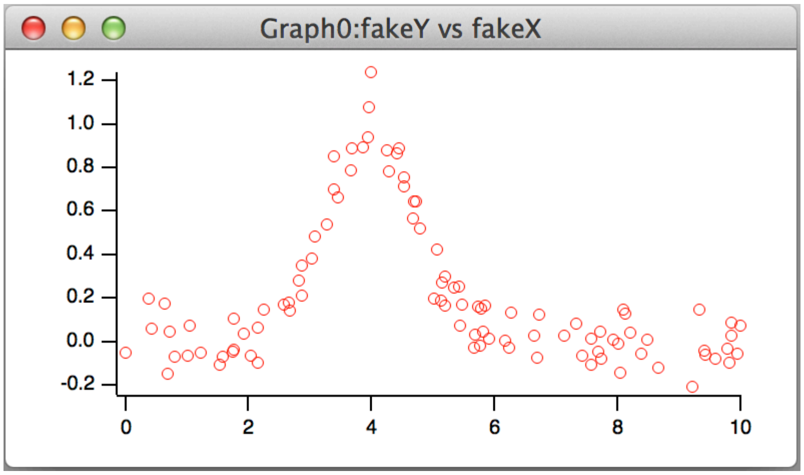
9. **Double-click the red trace.**

   The Modify Trace Appearance dialog appears.

10. **From the Mode pop-up choose Markers.**
11. **From the Markers pop-up menu choose the open circle.**

12. **Click Do It.**

    Now the graph makes sense.



Quick Curve Fit to a Gaussian

Our synthetic data was generated using a Gaussian function so let's try to extract the original parameters by fitting to a Gaussian of the form:

```
y = y0 + A*exp(-((x-x0)/width)^2)
```

Here y0, A, x0 and width are the parameters of the fit.

1. **Choose the Analysis→Quick Fit→gauss menu item.**

   Igor generated and executed a CurveFit command which you can see if you scroll up a bit in the history area of the command window. The CurveFit command performed the fit, appended a fit result trace to the graph, and reported results in the history area.

   At the bottom of the reported results we see the values found for the fit parameters. The amplitude parameter (A) should be 1.0 and the position parameter (x0) should be 4.0. We got 0.99222 ± 0.0299 for the amplitude and 3.9997 ± 0.023 for the position.

   Let's add this information to the graph.

2. **Choose Analysis→Quick Fit→Textbox Preferences.**

   The Curve Fit Textbox Preferences dialog appears.

   You can add a textbox containing curve fit results to your graph. The Curve Fit Textbox Preference dialog has a checkbox for each component of information that can be included in the textbox.

3. **Click the Display Curve Fit Info Textbox checkbox to check it and then click OK.**

   You have specified that you want an info textbox. This will affect future Quick Fit operations.

4. **Choose Analysis→Quick Fit→gauss again.**

   This time, Igor displays a textbox with the curve fit results. Once the textbox is made, it is just a textbox and you can double-click it and change it. But if you redo the fit, your changes will be lost unless you rename the textbox.

   That textbox is nice, but it's too big. Let's get rid of it.

   You could just double-click the textbox and click Delete in the Modify Annotation dialog. The next time you do a Quick Fit you would still get the textbox unless you turn the textbox feature off.

5. **Choose Analysis→Quick Fit→Textbox Preferences again. Click the Display Curve Fit Info Textbox checkbox to uncheck it. Click OK.**

6. **Choose Analysis→Quick Fit→gauss again.**

   The textbox is removed from the graph.

More Curve Fitting to a Gaussian

The Quick Fit menu provides easy access to curve fitting using the built-in fit functions, with a limited set of options, to fit data displayed in a graph. You may want more options. For that you must use the Curve Fitting dialog.

1. **Choose the Analysis→Curve Fitting menu item.**

   The Curve Fitting dialog appears.
2. **Click the Function and Data tab.**
3. **From the Function pop-up menu, choose gauss.**
4. **From the Y Data pop-up menu, choose fakeY.**
5. **From the X Data pop-up menu, choose fakeX.**
6. **Click the Data Options tab.**

   The Weighting and Data Mask pop-up menus should read "_none_".
7. **Click the Output Options tab.**

   The Destination pop-up menu should read "_auto_" and Residual should read "_none_".
8. **Click Do It.**

   During the fit a Curve Fit progress window appears. After a few passes the fit is finished and Igor waits for you to click OK in the progress window.
9. **Click OK.**

   The curve fit results are printed in the history. They are the same as in the previous section.

Sorting

In the next section we will do a curve fit to a subrange of the data. For this to work, the data must be sorted by X values.

1. **Double-click one of the open circle markers in the graph.**

   The Modify Traces Appearance dialog appears with fakeY selected. If fakeY is not selected, click it.
2. **From the Mode pop-up choose Lines between points and click Do It.**

   The fakeY trace reverts to a rat's nest of lines.
3. **Choose the Analysis→Sort menu item.**

   The Sorting dialog appears.
4. **If necessary choose Sort from the Operation pop-up menu.**
5. **Select "fakeX" in the Key Wave list and both "fakeX" and "fakeY" in the Waves to Sort list.**

   This will sort both fakeX and fakeY using fakeX as the sort key.
6. **Uncheck any checkboxes in the dialog that are checked, including the Display Output In checkbox.**
7. **Click Do It.**

   The rat's nest is untangled. Since we were using the lines between points mode just to show the results of the sort, we now switch back to open circles.
8. **Press Control and click (*Macintosh*) or right-click (*Windows*) on the fakeY trace (the jagged one).**

   A pop-up menu appears with the name of the trace at the top. If it is not "Browse fakeY" try again.
9. **Choose Markers from the Mode submenu.**

Fitting to a Subrange

Here we will again fit our data to a Gaussian but using a subset of the data. We will then extrapolate the fit outside of the initial range.

1. **Choose the Graph→Show Info menu item.**

   A cursor info panel is appended to the bottom of the graph.

   Two cursors are "docked" in the info panel, Cursor A and Cursor B.

2. **Place cursor A (the round one) on the fakeY trace.**

   One way to place the cursor is to drag it to the trace. Another way is to control-click (*Macintosh*) or right-click (*Windows*) on the name area which is just to the right of the cursor icon in the cursor info panel.

   Note that the cursor A icon in the dock is now black. This indicates that cursor A is selected, meaning that it will move if you use the arrow keys on the keyboard or the slider in the cursor info panel.

3. **Move cursor A to point #14.**

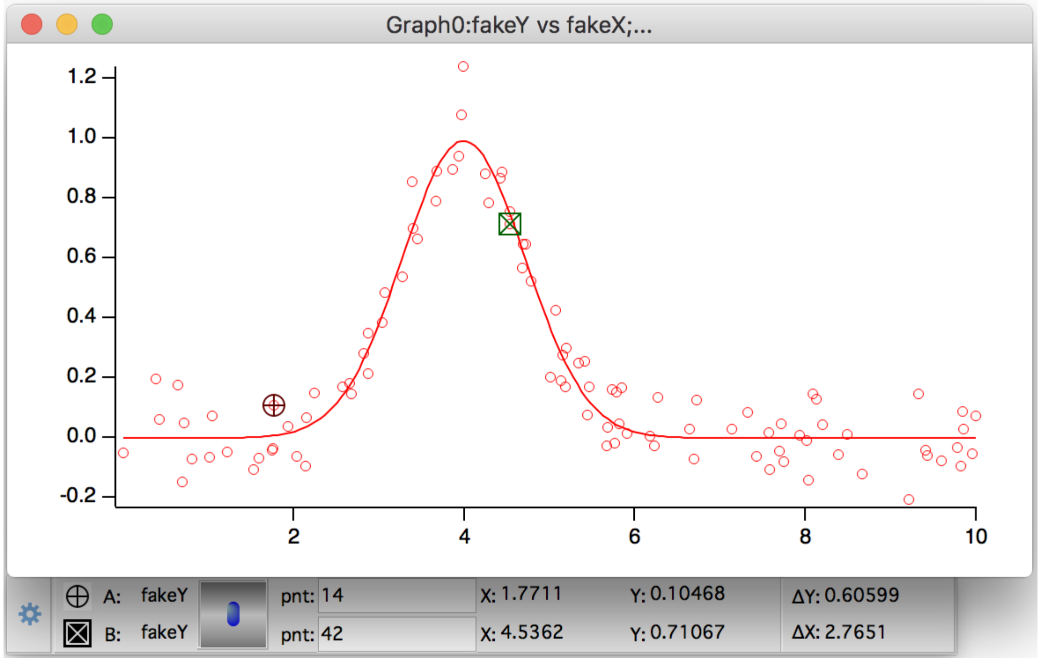
   To move the cursor one point at a time, use the arrow keys on the keyboard or click on either side of the slider in the cursor position control.

4. **Click the dock for cursor A in the cursor info panel to deselect it.**

   This is so you can adjust cursor B without affecting the position of cursor A.

5. **Place cursor B (the square one) on the fakeY trace and move it to point #42.**

   The graph should look like this:



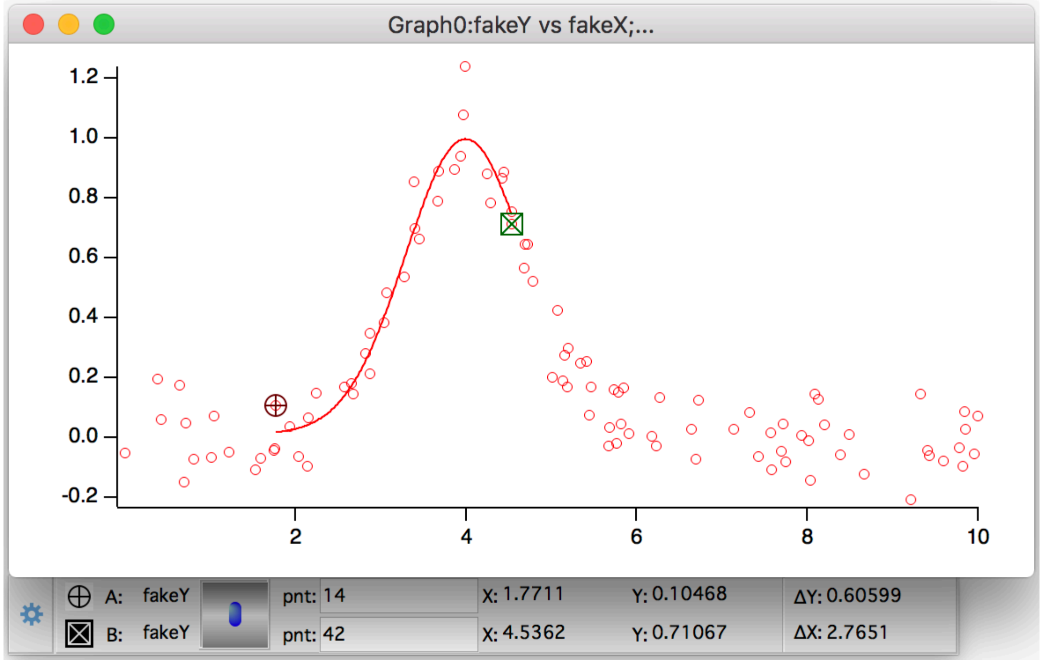
⊕ A: fakeY pnt: 14 X: 1.7711 Y: 0.10468 ΔY: 0.60599

☒ B: fakeY pnt: 42 X: 4.5362 Y: 0.71067 ΔX: 2.7651

6. **In the Analysis→Quick Fit menu make sure the Fit Between Cursors item is checked. If it is not, select it to check it.**

7. **Choose Analysis→Quick Fit→gauss.**

   Note that the fit curve is evaluated only over the subrange identified by the cursors.



A: fakeY pnt: 14 X: 1.7711 Y: 0.10468 ΔY: 0.60599

B: fakeY pnt: 42 X: 4.5362 Y: 0.71067 ΔX: 2.7651

We would like the fit trace to extend over the entire X range, while fitting only to the data between the cursors. This is one of the options available only in the Curve Fitting dialog.

8. **Choose Analysis→Curve Fitting and then click the Function and Data tab.**

   The curve fitting dialog appears and the settings should be as you left them. Check that the function type is gauss, the X data is fakeY, the X data is fakeX.

9. **Click the Data Options tab.**

10. **Click the Cursors button in the Range area.**

    This puts the text “pcsr(A)” and “pcsr(B)” in the range entry boxes.

    pcsr is a function that returns the wave point number at the cursor position.

11. **Select the Output Options tab and click the X Range Full Width of Graph checkbox to check it.**

12. **Click Do It.**

    The curve fit starts, does a few passes and waits for you to click OK.

13. **Click OK.**

    The fit was done using only the data between the cursors, but the fit trace extends over the entire X range.

    In the next section, we need the short version of the fit curve, so we will simply do the fit again:

14. **Choose Analysis→Quick Fit→gauss.**

Extrapolating a Fit After the Fit is Done

When you used the Quick Fit menu, and when you chose “_auto_” from the Destination pop-up menu in the Curve Fitting dialog, Igor created a wave named fit_fakeY to show the fit results. This is called the "fit destination wave." It is just an ordinary wave whose X scaling is set to the extent of the X values used in the fit.

In the preceding sections you learned how to make the curve fit operation extrapolate the fit curve beyond the subrange. Here we show you how to do this manually to illustrate some important wave concepts.

To extrapolate, we simply change the X scaling of fit_fakeY and re-execute the fit destination wave assignment statement which the CurveFit operation put in the history area.

1. **Choose the Data→Change Wave Scaling menu item.**
2. **If you see a button labeled More Options, click it.**
3. **From the SetScale Mode pop-up menu, choose Start and End.**
4. **Double-click "fit_fakeY" in the list.**

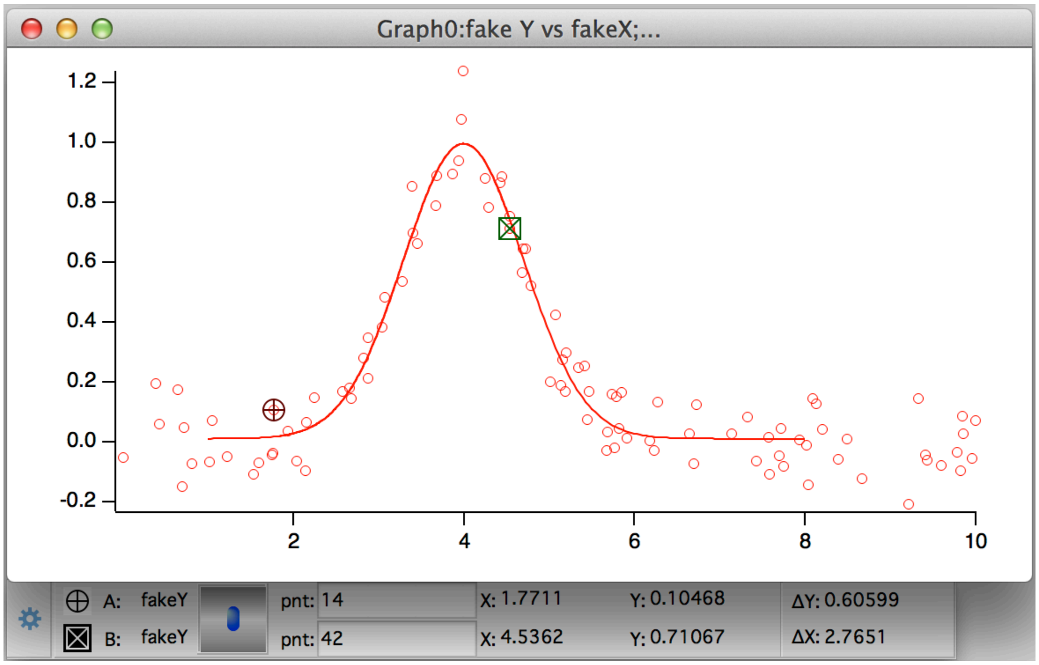
   This reads in the current X scaling values of fit_fakeY. The starting X value will be about 1.77 and the ending X will be about 4.53.
5. **Press Tab until the Start box is selected and enter 1.0.**
6. **Tab to the End box and type "8.0".**
7. **Click Do It**

   The fit_fakeY trace is stretched out and now runs between 1 and 8.

   Now we need to calculate new Y values for fit_fakeY using its new X values.
8. **In the history, find the line that starts "fit_fakeY=" and click it.**

   The entire line is selected. (The line in question is near the top of the curve fit report printed in the history.)
9. **Press Return or Enter once to copy the selection from the history to the command line and a second time to execute it.**

   The fit_fakeY wave now contains valid data between 1 and 8.



⊕ A: fakeY pnt: 14 X: 1.7711 Y: 0.10468 ΔY: 0.60599

☒ B: fakeY pnt: 42 X: 4.5362 Y: 0.71067 ΔX: 2.7651

Appending a Fit

The fit trace added automatically when Igor does a curve fit uses a wave named by adding "fit_" to the start of the Y data wave's name. If you do another fit to the same Y data, that fit curve will be overwritten. If you want to show the results of several fits to the same data, you will have to somehow protect the fit destination wave from being overwritten. This is done by simply renaming it.

1. **Choose the Data→Rename menu item.**
2. **Double-click the wave named fit_fakeY to move it into the list on the right.**
3. **Edit the name in the New Name box to change the name to "gaussFit_fakeY" and click Do It.**

4. **Position the A and B cursors to point numbers 35 and 61, respectively.**

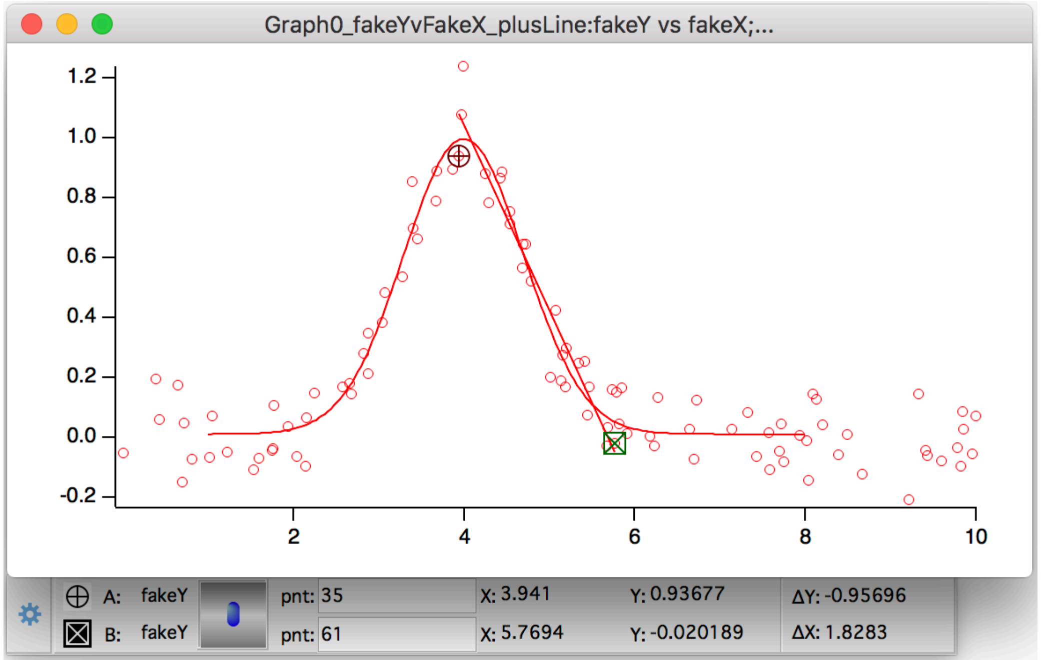
   A quick way to do this is to enter the numbers 35 and 61 in the edit boxes to the right of the cursor position control.

5. **Choose Analysis→Quick Fit→line.**

   Because there are two traces on the graph, Quick Fit doesn't know which one to fit and puts up the Which Data Set to Quick Fit dialog.

6. **Select fakeY from the list and click OK.**

   The line fit is appended to the graph:



⊕ A: fakeY pnt: 35 X: 3.941 Y: 0.93677 ΔY: -0.95696
☒ B: fakeY pnt: 61 X: 5.7694 Y: -0.020189 ΔX: 1.8283

This concludes Guided Tour 2.

Guided Tour 3 - Histograms and Curve Fitting

In this tour we will explore the Histogram operation and will perform a curve fit using weighting. The optional last portion creates a residuals plot and shows you how to create a useful procedure from commands in the history.

Starting Guided Tour 3

1. **If Igor is already running, activate it and choose File→New Experiment.**

   In this case, skip to step 2.

2. **Double-click your Igor64 alias or shortcut.**

   Instructions for creating this alias or shortcut can be found under **Creating an Igor64 Alias or Shortcut** on page I-13.

   On Windows, you an also launch Igor64 using the Start menu.

3. **Choose Misc→Preferences Off.**

   Turning preferences off ensures that the tour works the same for everyone.

Creating Synthetic Data

We need something to analyze, so let's generate some random values.

1. **Type the following in the command line and then press Return or Enter:**

   ```
   SetRandomSeed 0.1
   ```

   This initializes the random number generator so you will get the same results as this guided tour.

2. **Type the following in the command line and then press Return or Enter:**

   ```
   Make/N=10000 fakeY = enoise(1)
   ```

   This generates a 10,000 point wave filled with evenly distributed random values from -1 to 1.

Histogram of White Noise

Here we will generate a histogram of the evenly distributed "white" noise.

1. **Choose the Analysis→Histogram menu item.**

   The Histogram dialog appears.

2. **Select fakeY from the Source Wave list.**
3. **Verify that Auto is selected in the Output Wave menu.**
4. **Uncheck any checkboxes in the dialog that are checked, including the Display Output Wave checkbox.**
5. **Click the Auto-set Bin Range radio button.**
6. **Set the Number of Bins box to 100.**

   Note in the command box at the bottom of the dialog there are two commands:

   ```
   Make/N=100/O fakeY_Hist;DelayUpdate
   Histogram/B=1 fakeY,fakeY_Hist
   ```

   The first command makes a wave to receive the results, the second performs the analysis. The Histogram operation in the "Auto-set bin range" mode takes the number of bins from the output wave.

7. **Click the Do It button.**

   The histogram operation is performed.

   Now we will display the results.

8. **Choose Windows→New Graph.**
9. **Select fakeY_Hist in the Y Waves list and "_calculated_" in the X list.**

10. **Click the Do It button.**

    A graph is created showing the histogram results. Next we will touch it up a bit.

11. **Double-click the trace in the graph.**

    The Modify Trace Appearance dialog appears.

    "Left" is selected in the Axis pop-up menu in the top/left corner of the dialog indicating that changes made in the dialog will affect the left axis.

12. **Choose "Sticks to zero" from the Mode pop-up menu and click Do It.**

    The graph is redrawn using the new display mode.

13. **Double-click one of the tick mark labels (e.g., "100) of the left axis.**

    The Modify Axis dialog appears, showing the Axis Range tab.

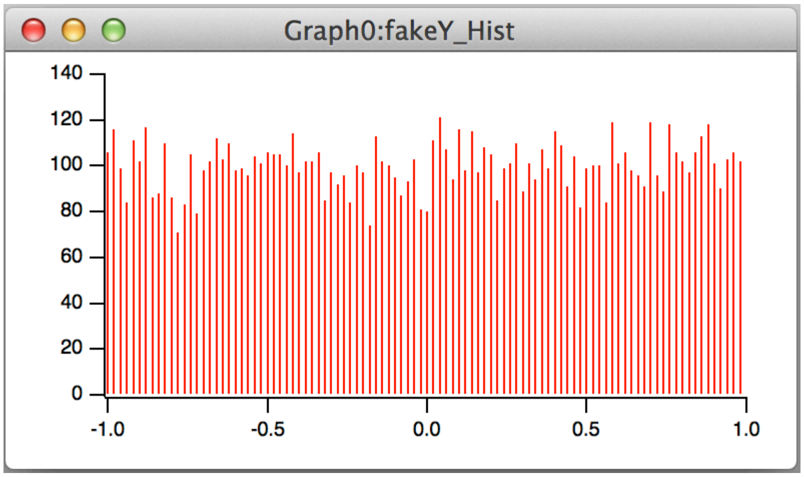
14. **From the two pop-up menus in the Autoscale Settings area, choose "Round to nice values" and "Autoscale from zero".**

15. **Choose Bottom from the Axis pop-up menu.**

16. **From the two pop-up menus in the Autoscale Settings area, choose "Round to nice values" and "Symmetric about zero".**

17. **Click the Do It button.**

    The graph should now look like this:



Saving Your Work - Tour 3A

1. **Choose the File→Save Experiment As menu item.**

2. **Navigate to your "Guided Tours" folder.**

   This is the folder that you created under **Saving Your Work - Tour 1A** on page I-21.

3. **Type "Tour 3A.pxp" in the name box and click Save.**

Histogram of Gaussian Noise

Now we'll do another histogram, this time with Gaussian noise.

1. **Type the following in the command line and then press Return or Enter:**

```
fakeY = gnoise(1)
```

2. **Choose the Analysis→Histogram menu item.**

   The dialog should still be correctly set up from the last time.

3. **Click the radio button labeled "Auto-set bins: 3.49*Sdev*N^-1/3".**

   The information text at the bottom of the Destination Bins box tells you that the histogram will have 48 bins.

   This is a method by Sturges for selecting a "good" number of bins for a histogram. See the **Histogram** operation on page V-349 for a reference.

4. **Click the Bin-Centered X Values checkbox to check it.**

   By default, the Histogram operation sets the X scaling of the output wave such that the X values are at the left edge of each bin, and the right edge is given by the next X value. This makes a nice bar plot.

   In the next section you will do a curve fit to the histogram. For curve fitting you need X values that represent the center of each bin.

5. **Click the Create Square Root(N) Wave checkbox to check it.**

   Counting data, such as a histogram, usually has Poisson-distributed values. The estimated mean of the Poisson distribution is simply the number of counts (N) and the estimated standard deviation is the square root of N.

   The curve fit will be biased if this is not taken into account. You will use this extra wave for weighting when you do the curve fit.

6. **Click the Do It button.**

   Note that the histogram output as shown in Graph0 has a Gaussian shape, as you would expect since the histogram input was noise with a Gaussian distribution.

7. **Choose Data→Data Browser.**

   The Data Browser shows you the waves and variables in your experiment. You should see three waves now: fakeY, fakeY_Hist, and W_SqrtN. FakeY_Hist contains the output of the Histogram operation and W_SqrtN is the wave created by the Histogram operation to receive the square root of N data.

8. **Close the Data Browser.**
9. **Click Graph0 and then double-click the trace to invoke the Modify Trace Appearance dialog.**
10. **Select Markers from the Mode menu, then select the open circle marker.**
11. **Click the Error Bars checkbox.**

    The Error Bars dialog appears.

12. **Select "+/- wave" from the Y Error Bars menu.**
13. **Pop up the Y+ menu and select W_SqrtN.**

    Note that the Y- menu is set to Same as Y+. You could now select another wave from the Y- menu if you needed asymmetric error bars.

14. **Click OK, then Do It.**

Saving Your Work - Tour 3B

1. **Choose the File→Save Experiment As menu item.**
2. **Navigate to your "Guided Tours" folder.**

   This is the folder that you created under **Saving Your Work - Tour 1A** on page I-21.

3. **Type "Tour 3B.pxp" in the name box and click Save.**

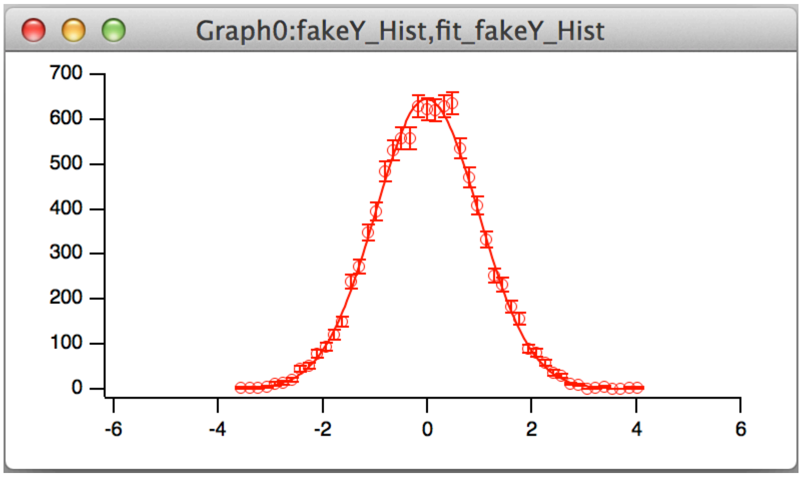
Curve Fit of Histogram Data

The previous section produces all the pieces required to fit a Gaussian to the histogram data, with proper weighting to account for the variance of Poisson-distributed data.

1. **Click the graph to make sure it is the target window.**

2. **In the Analysis→Quick Fit menu make sure the Weight from Error Bar Wave item is checked. If it is not, select it to check it.**
3. **Choose Analysis→Quick Fit→gauss.**

   The graph now looks like this:



   As shown in the history area, the fit results are:

```
Coefficient values ± one standard deviation
    y0     =-0.35284 ± 0.513
    A      =644.85 ± 7.99
    x0     =-0.0014111 ± 0.00997
    width  =1.406 ± 0.0118
```

   The original data was made with a standard deviation of 1. Why is the width 1.406? The way Igor defines its gauss fit function, width is sigma*$2^{1/2}$.

4. **Execute this command in the command line:**

```
Print 1.406/sqrt(2)
```

   The result, 0.994192, is pretty close to 1.0.

   It is often useful to plot the residuals from a fit to check for various kinds of problems. For that you need to use the Curve Fitting dialog.

5. **Choose Analysis→Curve Fitting.**
6. **Click the Function and Data tab and choose gauss from the Function menu.**
7. **Choose fakeY_Hist (not fakeY) from the Y Data menu.**
8. **Leave the X Data pop-up menu set to "_calculated_".**
9. **Click the Data Options tab. If there is text in the Start or End Range boxes, click the Clear button in the Range section.**
10. **Choose W_SqrtN from the Weighting pop-up menu.**
11. **Just under the Weighting pop-up menu there are two radio buttons. Click the top one which is labeled "Standard Dev".**
12. **Click the Output Options tab and choose "_auto_" from the Destination pop-up menu.**
13. **Set the Residual pop-up menu to "_auto trace_".**

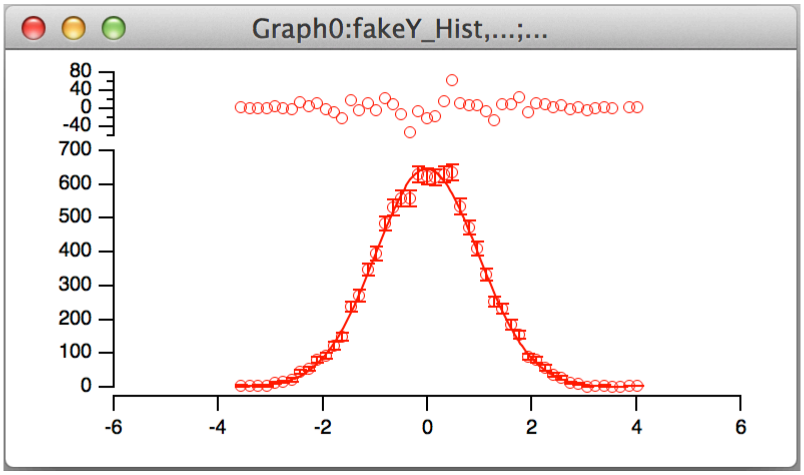
    Residuals will be calculated automatically and added to the curve fit in our graph.

14. **Uncheck the X Range Full Width of Graph checkbox.**

    This is appropriate only when we are fitting to a subset of the source wave.

15. **Click Do It.**

    The curve fit starts, does a few passes, and waits for you to click OK.



There is one small issue not addressed above. One of the bins contains zero; the square root of zero is, of course, zero. So the weighting wave contains a zero, which causes the curve fit to ignore that data point. It's not clear what is the best approach for fixing that problem. Some replace the zero with a one. These commands replace any zeroes in the weighting wave and re-do the fit:

```
W_SqrtN = W_SqrtN[p] == 0 ? 1 : W_SqrtN[p]
CurveFit/NTHR=0 gauss  fakeY_Hist /W=W_SqrtN /I=1 /D /R
```

This doesn't change the result very much, since there was just one zero in the histogram:

```
Coefficient values ± one standard deviation
    y0     =-0.40357 ± 0.464
    A      =644.76 ± 7.98
    x0     =-0.0014186 ± 0.00996
    width  =1.4065 ± 0.0115
```

Saving Your Work - Tour 3C

1. **Choose the File→Save Experiment As menu item.**
2. **Navigate to your "Guided Tours" folder.**

   This is the folder that you created under **Saving Your Work - Tour 1A** on page I-21.
3. **Type "Tour 3C.pxp" in the name box and click Save.**

Curve Fit Residuals (Optional)

The remaining guided tour sections are primarily of interest to people who want to use Igor programming to automate tasks. You may want to skip them and come back later. If so, jump ahead to **For Further Exploration** on page I-64.

In the next section, as an illustration of how the history area can be used as a source of commands to generate procedures, we will create a procedure that appends residuals to a graph. The preceding section illustrated that Igor is able to automatically display residuals from a curve fit, so the procedure that we write in the next section is not needed. Still, it demonstrates the process of creating a procedure. In preparation for writing the procedure, in this section we append the residuals manually.

If the curve fit to a Gaussian function went well and if the gnoise function truly produces noise with a Gaussian distribution, then a plot of the difference between the histogram data and the fitted function should not reveal any curvature.

1. **To remove the automatically generated residual from the Gaussian fit in the previous section, Control-click (*Macintosh*) or right-click (*Windows*) directly on the residual trace at the top of the graph and select Remove Res_fakeY_Hist from the pop-up menu.**

2. **Choose the Data→Duplicate Waves menu item.**

3. **Choose fakeY_Hist from the Template pop-up menu.**

4. **In the first Names box, enter "histResids".**

5. **Click Do It.**

   You now have a wave suitable for containing residuals.

6. **In the history area of the command window, find the line that reads:**

   ```
   fit_fakeY_Hist= W_coef[0]+W_coef[1]*exp(-((x-W_coef[2])/W_coef[3])^2)
   ```

   W_coef is a wave created by the CurveFit operation to contain the fit parameters. W_coef[0] is the y0 parameter, W_coef[1] is the A parameter, W_coef[2] is the x0 parameter and W_coef[3] is the width parameter.

   This line shows conceptually what the CurveFit operation did to set the data values of the fit destination wave.

7. **Click once on the line to select it and then press Return or Enter once.**

   The line is transferred to the command line.

8. **Edit the line to match the following:**

   ```
   histResids = fakeY_Hist -(W_coef[0]+W_coef[1]*exp(-((x-W_coef[2])/W_coef[3])^2))
   ```

   In other words, change `fit_fakeY_Hist` to `histResids`, click after the equals and type `fakeY_Hist - (` and then add a `)` to the end of the line.

   The expression inside the parentheses that you added represents the model value using the parameters determined by the fit. This command computes residuals by subtracting the model values from the data values on which the fit was performed.

   If the fit had used an X wave rather than calculated X values then it would have been necessary to substitute the name of the X wave for the "x" in the expression.

9. **Press Return or Enter.**

   This wave assignment statement calculates the difference between the measured data (the output of the Histogram operation) and the theoretical Gaussian (as determined by the CurveFit operation).

   Now we will append the residuals to the graph stacked above the current contents.

10. **Choose Graph→Append Traces to Graph.**

11. **Select histResids from the Y waves list and "_calculated_" from the X wave list.**

12. **Choose New from the Axis pop-up menu under the Y Waves list.**

13. **Enter "Lresid" in the Name box and click OK.**

14. **Click Do It.**

    The new trace and axis is added.

    Now we need to arrange the axes. We will do this by partioning the available space between the Left and Lresid axes.

15. **Double-click the far-left axis.**

    The Modify Axis dialog appears. If any other dialog appears, cancel and try again making sure the cursor is over the axis.

    If you have enough screen space you will be able to see the graph change as you change settings in the dialog. Make sure that the Live Update checkbox in the top/right corner of the dialog is selected.

16. **Click the Axis tab.**

    The Left axis should already be selected in the pop-up menu in the top-left corner of the dialog.

17. **Set the Left axis to draw between 0 and 70% of normal.**

18. **Choose Lresid from the Axis pop-up menu.**

19. **Set the Lresid axis to draw between 80 and 100% of normal.**

20. **Choose Fraction of Plot Area in the “Free axis position” menu.**

    The Lresid axis is a “free” axis. This moves it horizontally so it is in line with the Left axis.

21. **Choose Bottom from the Axis pop-up menu.**
22. **Click the Axis Standoff checkbox to turn standoff off.**

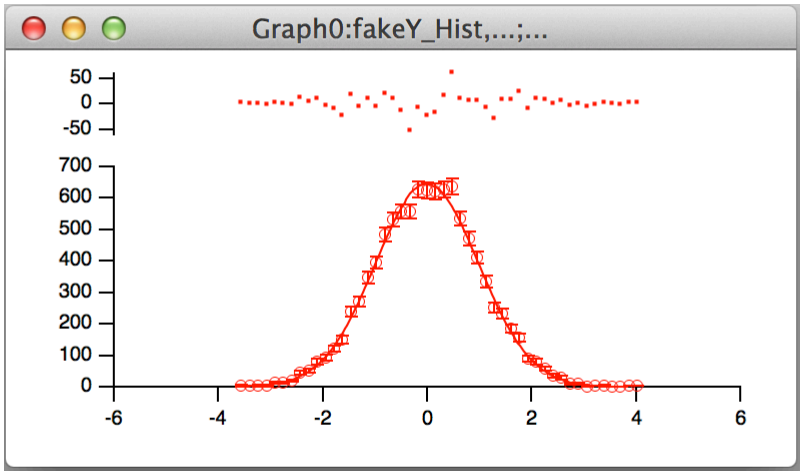
    Just a couple more touch-ups and we will be done. The ticking of the Lresid axis can be improved. The residual data should be in dots mode.

23. **Choose Lresid from the Axis pop-up menu again.**
24. **Click the Auto/Man Ticks tab.**
25. **Change the Approximately value to 2.**
26. **Click the Axis Range tab.**
27. **In the Autoscale Settings area, choose “Symmetric about zero” from the menu currently reading “Zero isn’t special”.**
28. **Click the Do It button.**
29. **Double-click the histResids trace.**

    The Modify Trace Appearance dialog appears with histResids already selected in the list.

30. **Choose Dots from the Mode pop-up menu**
31. **Set the line size to 2.00.**
32. **Click Do It.**

    The graph should now look like this:



Saving Your Work - Tour 3D

1. **Choose the File→Save Experiment As menu item.**
2. **Navigate to your “Guided Tours” folder.**

    This is the folder that you created under **Saving Your Work - Tour 1A** on page I-21.

3. **Type “Tour 3D.pxp” in the name box and click Save.**

Writing a Procedure (Optional)

In this section we will collect commands that were created as we appended the residuals to the graph. We will now use them to create a procedure that will append a plot of residuals to a graph.

1. **Make the command window tall enough to see the last 10 lines by dragging the top edge of the window upward.**

2. **Find the fifth line from the bottom that reads:**

```
•AppendToGraph/L=Lresid histResids
```

3. **Select this line and all the lines below it and copy them to the clipboard.**

4. **Choose the Windows→New→Procedure menu item.**

5. **Type "Append Residuals" (without the quotes) in the Document Name box and click OK.**

   A new procedure window appears. We could have used the always-present built-in procedure window, but we will save this new procedure window as a standalone file so it can be used in other experiments.

6. **Add a blank line to the window, type "Function AppendResiduals()", and press Return or Enter.**

7. **Paste the commands from the clipboard into the new window.**

8. **Type "End" and press Return or Enter.**

9. **Select the five lines that you pasted into the procedure window and then choose Edit→Adjust Indentation.**

   This removes the bullet characters copied from the history and prepends tabs to apply the normal indentation for procedures.

   Your procedure should now look like this:

```
Function AppendResiduals()
    AppendToGraph/L=Lresid histResids
    SetAxis/A/E=2 Lresid
    ModifyGraph nticks(Lresid)=2,standoff(bottom)=0,axisEnab(left)={0,0.7};DelayUpdate
    ModifyGraph axisEnab(Lresid)={0.8,1}, freePos(Lresid)=0;DelayUpdate
    ModifyGraph mode(histResids)=2,lsize(histResids)=2
End
```

10. **Delete the ";DelayUpdate" at the end of the two ModifyGraph commands.**

    DelayUpdate has no effect in a function.

    We now have a nearly functional procedure but with a major limitation — it only works if the residuals wave is named "histResids". In the following steps, we will change the function so that it can be used with any wave and also with an XY pair, rather than just with equally-spaced waveform data.

11. **Replace the first two lines of the function with the following:**

```
Function AppendResiduals(yWave, xWave)
    Wave yWave
    Wave xWave              // X wave or null if there is no X wave

    if (WaveExists(xWave))
       AppendToGraph/L=Lresid yWave vs xWave
    else
       AppendToGraph/L=Lresid yWave
    endif
    String traceName = NameOfWave(yWave)
```

12. **In the last ModifyGraph command in the function, change both "histResids" to "$traceName" (without the quotes).**

    The "$" operator converts the string expression that follows it into the name of an Igor object.

    Here is the completed procedure.

```
Function AppendResiduals(yWave,xWave)
    Wave yWave
    Wave/Z xWave            // X wave or null if there is no X wave

    if (WaveExists(xWave))
       AppendToGraph/L=Lresid yWave vs xWave
    else
       AppendToGraph/L=Lresid yWave
    endif
    String traceName = NameOfWave(yWave)
    SetAxis/A/E=2 Lresid
    ModifyGraph nticks(Lresid)=2,standoff(bottom)=0,axisEnab(left)={0,0.7}
    ModifyGraph axisEnab(Lresid)={0.8,1},freePos(Lresid)={0,kwFraction}
    ModifyGraph mode($traceName)=2,lsize($traceName)=2
End
```

Let's try it out.

13. **Click the Compile button at the bottom of the procedure window to compile the function.**

    If you get an error, edit the function text to match the listing above.

14. **Click the close button in the Append Residuals procedure window. A dialog asks if you want to kill or hide the window. Click Hide.**

    If you press Shift while clicking the close button, the window will be hidden without a dialog.

15. **Choose Windows→New Graph and click the Fewer Choices button if it is available.**

16. **Choose fakeY_Hist from the Y Waves list and _calculated_ from the X Wave list and click Do It.**

    A graph without residuals is created.

17. **In the command line, execute the following command:**

    ```
    AppendResiduals(histResids, $"")
    ```

    Because we have no X wave, we use $"" to pass NULL to AppendResiduals for the xWave parameter.

    The AppendResiduals function runs and displays the residuals in the graph, above the original histogram data.

    Next we will add a function that displays a dialog so we don't have to type wave names into the command line.

18. **Choose Windows→Procedure Windows→Append Residuals to show the Append Residuals procedure window.**

19. **Enter the following function below the AppendResiduals function.**

    ```
    Function AppendResidualsDialog()
        String yWaveNameStr, xWaveNameStr

        Prompt yWaveNameStr,"Residuals Data",popup WaveList("*",";","")
        Prompt xWaveNameStr,"X Data", popup "_calculated_;"+WaveList("*",";","")
        DoPrompt "Append Residuals", yWaveNameStr, xWaveNameStr
        if (V_flag != 0)
            return -1              // User canceled
        endif

        AppendResiduals($yWaveNameStr, $xWaveNameStr)

        return 0                   // Success
    End
    ```

    This function displays a dialog to get parameters from the user and then calls the AppendResiduals function.

    Let's try it out.

20. **Click the Compile button at the bottom of the procedure window to compile procedures.**

    If you get an error, edit the function text to match the listing above.

21. **Shift-click the close button to hide the procedure window. Then activate Graph1.**

22. **Control-click (*Macintosh*) or right-click (*Windows*) the residual trace at the top of the graph and select Remove histResids from the pop-up menu.**

    The Lresid axis is removed because the last wave graphed against it was removed.

23. **On the command line, execute this command:**

    ```
    AppendResidualsDialog()
    ```

    The AppendResidualsDialog function displays a dialog to let you choose parameters.

24. **Choose histResids from the Residuals Data pop-up menu.**

25. **Leave the X Wave pop-up set to "_calculated_".**

26. **Click Continue.**

    The graph should once again contain the residuals plotted on an axis above the main data.

    Next we will add a menu item to the Macros menu.

27. **Choose Windows→Procedure Windows→Append Residuals to show the Append Residuals procedure window.**

28. **Enter the following code before the AppendResiduals function:**

```
Menu "Macros"
    "Append Residuals...", AppendResidualsDialog()
End
```

29. **Click the Compile button.**

    Igor compiles procedures and adds the menu item to the Macros menu.

30. **Shift-click the close button to hide the procedure window. Then activate Graph1.**

31. **Control-click (*Macintosh*) or right-click (*Windows*) the residual trace at the top of the graph and select "Remove histResids" from the pop-up menu.**

32. **Click the Macros menu and choose the "Append Residuals" item**

    The procedure displays a dialog to let you choose parameters.

33. **Choose histResids from the Residuals Data pop-up menu.**

34. **Leave the X Wave pop-up menu set to "_calculated_".**

35. **Click the Continue button.**

    The graph should once again contain the residuals plotted on a new axis above the main data.

Saving a Procedure File (Optional)

Note: *If you are using the demo version of Igor Pro beyond the 30-day trial period, you cannot save a procedure file.*

Now that we have a working procedure, let's save it so it can be used in the future. We will save the file in the "Igor Pro User Files" folder - a folder created by Igor for you to store your Igor files.

More precisely, we will save the procedure file in the "User Procedures" subfolder of the Igor Pro User Files folder. You could save the file anywhere on your hard disk, but saving in the User Procedures subfolder makes it easier to access the file as we will see in the next section.

1. **Choose Help→Show Igor Pro User Files.**

   Igor opens the "Igor Pro User Files" folder on the desktop.

   By default, this folder has the Igor Pro major version number in its name, for example, "Igor Pro 9 User Files", but it is generically called the "Igor Pro User Files" folder.

   Note where in the file system hierarchy this folder is located as you will need to know this in a subsequent step. The default locations are:

   Macintosh:
   `/Users/<user>/Documents/WaveMetrics/Igor Pro 9 User Files`

   Windows:
   `C:\Users\<user>\Documents\WaveMetrics\Igor Pro 9 User Files`

   Note the "User Procedures" subfolder of the Igor Pro User Files folder. This is where we will save the procedure file.

2. **Back in Igor, choose Windows→Procedure Windows→Append Residuals to show the Append Residuals procedure window.**

3. **Choose File→Save Procedure As.**

4. **Enter the file name "Append Residuals.ipf".**

5. **Navigate to your User Procedures folder inside your Igor Pro User Files folder and click Save.**

   The Append Residuals procedure file is now saved in a standalone file.

6. **Click the close button on the Append Residuals procedure window.**

   Igor asks if you want to kill or hide the file.

7. **Click the Kill button.**

   This removes the file from the current experiment, but it still exists on disk and you can open it as needed.

   There are several ways to open the procedure file to use it in the future. One is to double-click it. Another is to choose the File→Open File→Procedure menu item. A third is to put a `#include` statement in the built-in procedure window, which is how we will open it in the next section.

Including a Procedure File (Optional)

The preferred way to open a procedure window that you intend to use from several different experiments is to use a #include statement. This section demonstrates how to do that.

Note: *If you are using the demo version of Igor Pro beyond the 30-day trial period, you did not create the Append Residuals.ipf file in the preceding section so you can't do this section.*

1. **In Igor, use the Windows→Procedure Windows→Procedure Window to open the built-in procedure window.**

2. **Near the top of the built-in procedure window, notice the line that says:**

   ```
   #pragma rtGlobals = 3
   ```

   This is technical stuff that you can ignore.

3. **Under the rtGlobals line, leave a blank line and then enter:**

   ```
   #include "Append Residuals"
   ```

4. **Click the Compile button at the bottom of the built-in procedure window.**

   Igor compiles the procedure window. When it sees the `#include` statement, it looks for the "Append Residuals.ipf" procedure file in the User Procedures folder and opens it. You don't see it because it was opened hidden.

5. **Use the Windows→Procedure Windows menu to verify that the Append Residuals procedure file is in fact open.**

   To remove the procedure file from the experiment, you would remove the `#include` statement from the built-in procedure window.

   #include is powerful because it allows procedure files to include other procedure files in a chain. Each procedure file automatically opens any other procedure files it needs.

   User Procedures is special because Igor searches it to satisfy #include statements.

   Another special folder is Igor Procedures. Any procedure file in Igor Procedures is automatically opened by Igor at launch time and left open till Igor quits. This is the place to put procedure files that you want to be open all of the time. It is the easiest way to make a procedure file available and is recommended for frequently-used files.

This concludes Guided Tour 3.

For Further Exploration

We developed the guided tours in this chapter to provide an overview of the basics of using Igor Pro and to give you some experience using features that you will likely need for your day-to-day work. Beyond these fundamentals, Igor includes a wide variety of features to facilitate much more advanced graphing and analysis of your data.

As you become more familiar with using Igor, you will want to further explore some of the additional learning and informational aids that we have included with Igor Pro.

- The Igor Pro manual is installed on your hard disk in PDF format. You can access it through the Igor Help Browser or open it directly from the Manual folder in the Igor Pro Folder.

  The material in the manual is the same as the material in the online help files but is organized in book format and is therefore better suited for linear reading. Unlike the help files, the PDF manual includes an index.

  The most important chapters at this point in your Igor learning curve are Chapter II-3, **Experiments, Files and Folders** and Chapter II-5, **Waves**. If you want to learn Igor programming, read Chapter IV-1, **Working with Commands**, Chapter IV-2, **Programming Overview**, and Chapter IV-3, **User-Defined Functions**.
- The Igor Help Browser provides online help, including reference material for all built-in operations, functions, and keywords, an extensive list of shortcuts, and the ability to search Igor help files, procedure files, examples and technical notes for key phrases. See **The Igor Help Browser** on page II-2 for more information.
- The Examples folder contains a wide variety of sample experiments illustrating many of Igor’s advanced graphing and programming facilities. You can access these most easily through the File→Example Experiments submenus.
- The Learning Aids folder contains additional guided tours and tutorials including a tutorial on image processing. You can access these through the File→Example Experiments→Tutorials submenus.
- A tutorial on 3D visualization can be found in the “3D Graphics” help window, accessible through the Help→Help Windows menu.
- The WaveMetrics Procedures folder contains a number of utility procedures that you may find useful for writing your own procedures and for your more advanced graphing requirements. For an overview of the WaveMetrics procedures and easy loading of the procedure files, choose Help→Help Windows→WM Procedures Index.
- The “More Extensions (64-bit)” folder contains a number of External Operations (XOPs), which add functionality not built into the Igor Pro application. Read the included help files to find out more about the individual XOPs and how to install them, or consult the External Operations Index in the XOP Index help file, which has brief description of each XOP.
- The Technical Notes folder contains additional information. Tech Note #000 contains an index to all of the other notes.
- The Igor Pro mailing list is an Internet mailing list where Igor users share ideas and help each other. See **Igor Mailing List** on page II-4 or select the Help→Support menu and then select Igor Mailing List from the Support Options list for information about the mailing list.
- **IgorExchange** is a user-to-user support web page and a repository for user-created Igor Pro projects. Choose Help→IgorExchange to visit it.

Volume II User's Guide: Part 1

Table of Contents

II-1 Getting Help II-1
II-2 The Command Window II-7
II-3 Experiments, Files and Folders II-15
II-4 Windows II-43
II-5 Waves II-61
II-6 Multidimensional Waves II-91
II-7 Numeric and String Variables II-101
II-8 Data Folders II-107
II-9 Importing and Exporting Data II-125
II-10 Igor HDF5 Guide II-183
II-11 Dialog Features II-227
II-12 Tables II-233
II-13 Graphs II-275
II-14 Category Plots II-355
II-15 Contour Plots II-365
II-16 Image Plots II-385
II-17 3D Graphics II-405
II-17 Page Layouts II-475

Chapter

II-1

Getting Help

Overview

There are a number of sources of information for learning about Igor Pro. The most important are:

- The Guided Tour
- The Igor help system
- The Igor Pro online manual
- The example experiments in the Examples folder
- The Igor mailing list
- The IgorExchange user-to-user support web page

The Guided Tour

The Igor guided tour is indispensible for learning Igor.

The time you spend doing the guided tour will be repaid many times over in productivity. To do the guided tour, choose Help→Getting Started.

The same material is available in the first two chapters of this manual: **Introduction to Igor Pro** on page I-1 and **Guided Tour of Igor Pro** on page I-11.

Online Manual

The Igor Pro installer installs the entire Igor Pro manual as an Adobe PDF file. The PDF manual contains the same information as the online help files but in the more familiar PDF format.

You can open the manual by choosing Help→Manual or by double-clicking the IgorMan.pdf file in "Igor Pro-Folder/Manual".

The manual is not available in hard-copy form.

Online Help

Igor's online help system consists of these components:

- The Igor help browser
- Igor help files
- Tooltips

The Igor Help Browser

The Igor Help Browser is designed to provide quick access to the most frequently-used Igor reference material and also to provide a starting point in searching for other kinds of information. You can display the Igor Help Browser by:

- Choosing Help→Igor Help Browser
- Pressing the help key (Macintosh only)
- Pressing the F1 key (Windows only)
- Clicking the question-mark icon in the lower/right corner of the command window

Igor Help Files

The Igor installer places help files primarily in "Igor Pro Folder/Igor Help Files" and in "Igor Pro Folder/More Help Files".

When Igor starts up, it automatically creates help windows by opening the Igor help files stored in "Igor Pro Folder/Igor Help Files" and in "Igor Pro User Files/Igor Help Files". You can display a help window using the Help Files tab of the Help Browser or by choosing it from the Help Windows submenu in the Help menu.

To see a list of topics in open help files, use the Igor Help Browser Help Topics tab.

To search all installed help files, whether open or not, use the Igor Help Browser Search Igor Files tab.

Hiding and Killing a Help Window

When you click the close button in a help window, Igor hides it.

Usually there is no reason to kill a help file, but if you want to kill one, you must press Option (*Macintosh*) or Alt (*Windows*) while clicking the close button.

Executing Commands from a Help Window

Help windows often show example Igor commands. To execute a command or a section of commands from a help window, select the command text and press Control-Enter or Control-Return. This sends the selected text to the command line and starts execution.

Tooltips

Igor provides tooltips for various icons, dialog items, and other visual features. You can turn these tips on or off using the Show Tooltips checkbox in the Miscellaneous Settings dialog. To display this setting, choose Misc→Miscellaneous Settings and select the Help category.

You can also use tooltips to get information about traces in graphs and columns in tables. Use Graph→Show Trace Info Tags and Table→Show Column Info Tags to turn these tips on and off.

Searching Igor Files

You can search Igor help files and procedure files using the Search Igor Files tab of the Igor Help Browser.

The search expression can consist of one or more (up to 8) terms. Terms are separated by the word “and”. Here are some examples:

interpolation	*One term*
spline interpolation	*One term*
spline and interpolation	*Two terms*
spline and interpolation and smoothing	*Three terms*

The second example finds the exact phrase “spline interpolation” while the third example finds sections that contain the words “spline” and “interpolation”, not necessarily one right after the other.

The only keyword supported in the search expression is “and”. Quotation marks in the search expression don’t mean anything special and should not be used.

If your search expression includes more than one term, a text box appears in which you can enter a number that defines what “and” means. For example, if you enter 10, this means that the secondary terms must appear within 10 paragraphs of the primary term to constitute a hit. A value of 0 means that the terms must appear in the same paragraph. In a plain text file, such as a procedure file, a paragraph is a single line of text. Blank lines count as one paragraph.

To speed up searches when entering multiple search terms, enter the least common term first. For example, searching help files for “hidden and axis and graph” is faster than searching for “graph and axis and hidden” because “hidden” is less common than “graph”.

Example Experiments

Igor Pro comes with a large collection of examples which demonstrate Igor features and techniques. You can access them using the File→Example Experiments submenu.

Igor Mailing List

The Igor mailing list is an Internet discussion list that provides a way for Igor Pro users to help one another and to share solutions and ideas. WaveMetrics also uses the list to post information on the latest Igor developments. For information about subscribing and other details about the mailing list, please visit this web page:

`http://www.wavemetrics.com/users/mailinglist.htm`

IgorExchange

IgorExchange is a user-to-user support and collaboration web site sponsored by WaveMetrics but run by and for Igor users. For information about IgorExchange, please visit this web page:

`http://www.igorexchange.com`

Updating Igor

WaveMetrics periodically releases free updates for Igor. Updates provide bug fixes and sometimes new features.

Igor Pro checks for available updates during startup by contacting one of our web sites. This update check is implemented to minimize any performance impact on starting Igor.

If an update is found, Igor presents a dialog in which you can choose to download the update, skip notifications about the current update, or be reminded later about the update. You can also choose to view the release notes for the update.

You can disable automatic update checking using the Updates section of the Miscellaneous Settings dialog. You can also enable checking for beta versions.

You can manually check for updates at any time by choosing Help→Updates for Igor Pro. This works regardless of whether automatic update checking is enabled.

If your Internet access requires a proxy server or other unusual configuration, Igor's update checking mechanism may fail. In those cases you can always go to http://www.wavemetrics.net to see if an update is available.

Technical Support

WaveMetrics provides technical support via email.

To send an email to WaveMetrics support, start by choosing Help→Contact Support. This creates an email containing information about your Igor installation, such as your Igor serial number and the Igor version, which we need to provide support.

In most cases, we need to reproduce your problem in order to solve it. It is best if you can provide a simplified example showing the problem.

For information on upgrades and other nontechnical information, send queries to:

`sales@wavemetrics.com`

World Wide Web

You will find our Web site at:

```
http://www.wavemetrics.com/
```

You can also choose Help→WaveMetrics Home Page.

Our Web site contains a page for searching our support database, and links to Igor-related FTP sites and to Igor users' Web pages. In addition, it contains a number of cool graphs. We are always grateful for new cool graphs. Contact us at sales@wavemetrics.com if you have a cool graph to share.

Help Shortcuts

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To activate the Igor Help Browser	Choose Help→Igor Help Browser or click the Igor Help Browser icon in the lower-right corner of the command window.	Press F1, choose Help→Igor Help Browser, or click the Igor Help Browser icon in the lower-right corner of the command window.
To get a contextual menu of commonly-used actions	Control-click the body of an Igor help window.	Right-click the body of an Igor help window.
To jump to a topic in an Igor help window	Click a blue underlined topic link in the help window.	Click a blue underlined topic link in the help window.
To execute commands in an Igor help window	Select the commands and press Control-Return or Control-Enter.	Select the commands and press Ctrl+Enter.
To kill a help window	Option-click the close button.	Press Alt and click the close button.
To insert a function or operation template in a procedure window or in the command line	Type or select the name of an Igor operation or function, Control-click it, and choose Insert Template.	Type or select the name of an Igor operation or function, right-click it, and choose Insert Template.
To get help for a function or operation from a procedure, notebook or help window or from the command line	Type or select the name of an Igor operation or function, Control-click it, and choose Help For <name> or choose Help→Help For <name>.	Type or select the name of an Igor operation or function, right-click it, and choose Help For <name> or choose Help→Help For <name>.
To revisit a previously-viewed location	Press Cmd-Option-Left Arrow (Go Back) or Cmd-Option-Right Arrow (Go Forward).	Press Alt+Left Arrow (Go Back) or Alt+Right Arrow (Go Forward).

Chapter

II-2

The Command Window

Overview

You can control Igor using menus and dialogs or using commands that you execute from the command window. Some actions, for example waveform assignments, are much easier to perform by entering a command. Commands are also convenient for trying variations on a theme — you can modify and reexecute a command very quickly. If you use Igor regularly, you may find yourself using commands more and more for those operations that you frequently perform.

In addition to executing commands in the Command window, you can also execute commands in a notebook, procedure or help window. These techniques are less commonly used than the command window. See **Notebooks as Worksheets** on page III-4 for more information.

This chapter describes the command window and general techniques and shortcuts. See Chapter IV-1, **Working with Commands**, for details on command usage and syntax.

The command window consists of a **command line** and a **history area.** When you enter commands in the command line and press Return (*Macintosh*) or Enter (*Windows and Macintosh*), the commands are executed. Then they are saved in the history area for you to review. If a command produces text output, that output is also saved in the history area. A bullet character is prepended to command lines in the history so that you can easily distinguish command lines from output lines.

The Command window includes a help button just below the History area scroll bar. Clicking the button displays the Help Browser window.

The total length of a command on the command line is limited to 2500 bytes. A command executed in the command line must fit in one line.

Command Window Example

Here is a quick example designed to illustrate the power of commands and some of the shortcuts that make working with commands easy.

1. **Choose New Experiment from the File menu.**
2. **Execute the following command by typing in the command line and then pressing Return or Enter.**

   ```
   Make/N=100 wave0; Display wave0
   ```

   This displays a graph.
3. **Press Command-J** (*Macintosh*) **or Ctrl+J** (*Windows*)**.**

   This activates the command window.
4. **Execute**

   ```
   SetScale x, 0, 2*PI, wave0; wave0 = sin(x)
   ```

   The graph shows the sine of x from 0 to 2π.

   Now we are going to see how to quickly retrieve, modify and reexecute a command.
5. **Press the Up Arrow key.**

   This selects the command that we just executed.
6. **Press Return or Enter.**

   This transfers the selection back into the command line.
7. **Change the “2” to “4”.**

   The command line should now contain:

   ```
   SetScale x, 0, 4*PI, wave0; wave0 = sin(x)
   ```
8. **Press Return or Enter to execute the modified command.**

   This shows the sine of x from 0 to 4π.
9. **While pressing Option** (*Macintosh*) **or Alt** (*Windows*)**, click the last command in the history.**

   This is another way to transfer a command from the history to the command line. The command line should now contain:

   ```
   SetScale x, 0, 4*PI, wave0; wave0 = sin(x)
   ```

10. **Press Command-K** (*Macintosh*) **or Ctrl+K** (*Windows*)**.**

    This “kills” the contents of the command line.

    Now let’s see how you can quickly reexecute a previously executed command.

11. **With Command and Option** (*Macintosh*) **or Ctrl and Alt** (*Windows*) **pressed, click the second-to-last command in the history.**

    This reexecutes the clicked command (the 2*PI command).

    Repeat this step a number of times, clicking the second-to-last command each time. This will alternate between the 2*PI command and the 4*PI command.

12. **Execute**

    ```
    WaveStats wave0
    ```

    Note that the WaveStats operation has printed its results in the history where you can review them. You can also copy a number from the history to paste into a notebook or an annotation.

There is a summary of all command window shortcuts at the end of this chapter.

The Command Buffer

The command line shows the contents of the **command buffer**.

Normally the command buffer is either empty or contains just one line of text. However you can copy multiple lines of text from any window and paste them in the command buffer and you can enter multiple lines by pressing Shift-Return or Shift-Enter to enter a line break. You can drag the red divider line up to show more lines or down to show fewer lines.

You can clear the contents of the command buffer by choosing the Clear Command Buffer item in the Edit menu or by pressing Command-K (*Macintosh*) or Ctrl+K (*Windows*).

When you invoke an operation from a typical Igor dialog, the dialog puts a command in the command buffer and executes it. The command is then transferred to the history as if you had entered the command manually.

If an error occurs during the execution of a command, Igor leaves it in the command buffer so you can edit and reexecute it. If you don’t want to fix the command, you should remove it from the command buffer by pressing Command-K (*Macintosh*) or Ctrl+K (*Windows*).

Because the command buffer usually contains nothing or one command, we usually think of it as a single line and use the term “command line”.

Command Window Title

The title of the command window is the name of the experiment that is currently loaded. When you first start Igor or if you choose New from the File menu, the title of the experiment and therefore of the command window is “Untitled”.

When you save the experiment to a file, Igor sets the name of the experiment to the file name minus the file extension. If the file name is “An Experiment.pxp”, the experiment name is “An Experiment”. Igor displays “An Experiment” as the command window title.

For use in procedures, the **IgorInfo**(1) function returns the name of the current experiment.

History Area

The history area is a repository for commands and results.

Text in the history area can not be edited but can be copied to the clipboard or to the command line. Copying text to the clipboard is done in the normal manner. To copy a command from the history to the command buffer, select the command in the history and press Return or Enter. An alternate method is to press Option (*Macintosh*) or Alt (*Windows*) and click in the history area.

To make it easy tocopy a command from the history to the command line, clicking a line in the history area selects the entire line. You can still select just part of a line by clicking and dragging.

Up Arrow and Down Arrow move the selection range in the history up or down one line selecting an entire line at a time. Since you normally want to select a line in the history to copy a command to the command line, Up Arrow and Down Arrow skip over non-command lines. Left Arrow and Right Arrow move the insertion point in the command line.

When you save an experiment, the contents of the history area are saved. The next time you load the experiment the history will be intact. Some people have the impression that Igor recreates an experiment by reexecuting the history. This is not correct. See **How Experiments Are Loaded** on page II-26 for details.

Limiting Command History

The contents of the history area can grow to be quite large over time. You can limit the number of lines of text retained in the history using the Limit Command History feature in the Command Window section of the Miscellaneous Settings dialog which is accessible through the Misc menu.

If you limit command history, when you save the experiment, Igor checks the number of history lines. If they exceed the limit, the oldest lines are deleted.

History Archive

When history lines are deleted through the Limit Command History feature, the History Archive feature allows you to tell Igor to write the deleted lines to a text file in the experiment's home folder.

To enable the History Archive feature for a given experiment, create a plain text file in the home folder of the experiment. The text file must be named

```
<Experiment Name> History Archive UTF-8.txt
```

where <Experiment Name> is the name of the current experiment. Now, when you save the experiment, Igor writes any deleted history lines to the history archive file.

Prior to Igor Pro 7 the history archive file was written in system text encoding and was named

```
<Experiment Name> History Archive.txt
```

If you used the history archive in Igor Pro 6 you need to create a new history archive file whose name includes “UTF-8”.

If the history archive file is open in another program, including Igor, the history archive feature may fail and history lines may not be written.

History Carbon Copy

This feature is expected to be of interest only in rare cases for advanced Igor programmers such as Bela Farago who requested it.

You can designate a notebook to be a "carbon copy" of the history area by creating a plain text or formatted notebook and setting its window name, via Windows->Window Control, to HistoryCarbonCopy. If the HistoryCarbonCopy notebook exists, Igor inserts history text in the notebook as well as in the history. However, if a command is initiated from the HistoryCarbonCopy notebook (see **Notebooks as Worksheets** on page III-4), Igor suspends sending history text to that notebook during the execution of the command.

If you rename the notebook to something other than HistoryCarbonCopy, Igor will cease sending history text to it. If you later rename it back to HistoryCarbonCopy, Igor will resume sending history text to it.

The history trimming feature accessed via the Miscellaneous Settings dialog does not apply to the HistoryCarbonCopy notebook. You must trim it yourself. Notebooks are limited to 16 million paragraphs.

When using a formatted notebook as the history carbon copy, you can control the formatting of commands and results by creating notebook rulers named Command and Result. When Igor sends text to the history carbon copy notebook, it always applies the Command ruler to commands. It applies the Result ruler to results if the current ruler is Normal, Command or Result. You must create the Command and Result rulers if you want Igor to use them when sending text to the history carbon copy.

This function creates a formatted history carbon copy notebook with the Command and Result rulers used automatically by Igor as well as an Error ruler which we will use for our custom error messages:

```
Function CreateHistoryCarbonCopy()
   NewNotebook /F=1 /N=HistoryCarbonCopy /W=(50,50,715,590)

   Notebook HistoryCarbonCopy backRGB=(0,0,0)// Set background to black

   Notebook HistoryCarbonCopy showRuler=0

   // Define ruler to govern commands.
   // Igor will automatically apply this to commands sent to history carbon copy.
   Notebook HistoryCarbonCopy newRuler=Command,
                        rulerDefaults={"Geneva",10,0,(65535,65535,0)}

   // Define ruler to govern results.
   // Igor will automatically apply this to results sent to history carbon copy.
   Notebook HistoryCarbonCopy newRuler=Result,
                        rulerDefaults={"Geneva",10,0,(0,65535,0)}

   // Define ruler to govern user-generated error messages.
   // We will apply this ruler to error messages that we send
   // to history carbon copy via Print commands.
   Notebook HistoryCarbonCopy newRuler=Error,
rulerDefaults={"Geneva",10,0,(65535,0,0)}
End
```

If the current ruler is not Normal, Command or Result, it is assumed to be a custom ruler that you want to use for special messages sent to the history using the Print operation. In this case, Igor does not apply the Result ruler but rather allows your custom ruler to remain in effect.

This function sends an error message to the history using the custom Error ruler in the history carbon copy notebook:

```
Function PrintErrorMessage(message)
   String message

   Notebook HistoryCarbonCopy, ruler=Error
   Print message

   // Set ruler back to Result so that Igor's automatic use of the Command
   // and Result rulers will take effect for subsequent commands.
   Notebook HistoryCarbonCopy, ruler=Result
End
```

XOP programmers can use the XOPNotice3 XOPSupport routine to control the color of text sent to the History Carbon Copy notebook.

Searching the Command Window

You can search the command line or the history by choosing Find from the Edit menu or by using the keyboard shortcuts as shown in the Edit menu. This displays the find bar.

Searching the command line is most often used to modify a previously executed command before reexecuting it. For example, you might want to replace each instance of a particular wave name with another wave name.

If the history area has keyboard focus, usually indicated by an active selection in the history, the find bar searches the history area. If the command window has keyboard focus, the find bar searches the command line. To be sure that Find will search the area that you want, you can click in that area before starting the search.

Command Window Magnification

You can magnify the size of text in the command line and history area.

To magnify the command line text, right-click in the command line and select an item from the Magnification submenu. To make the new magnification your default magnification for new experiments, right-click again and choose Set as Default for Command Line from the same submenu.

To magnify the history area text, right-click in the history area and select an item from the Magnification submenu. To make the new magnification your default magnification for new experiments, right-click again and choose Set as Default for History Area from the same submenu.

Command Window Formats

You can change the text format used for the command line. For example, you might prefer a different color. To do this, click in the command line and then choose Set Text Format from the Command Buffer submenu of the Misc menu. To set the text format for the history area, click in the history area and then choose Set Text Format from the History Area submenu of the Misc menu. To do this, the history area must have some text in it.

You can set other properties, such as background color, by choosing Document Settings from the Command Buffer or History Area submenus. The Document Settings dialog also sets the header and footer used when printing the history.

When you change the text format or document settings, you are changing the current experiment only. You may want to capture the new format and settings as a preference for new experiments. To do this, choose Capture Prefs from the Command Buffer and History Area submenus.

Getting Help from the Command Line

When working with the command line, you might need help in formulating a command. There are shortcuts that allow you to insert a template, view help, or find the definition of a user function. The command line also supports **Command Completion**.

To insert a template, type the name of the operation or function, Control-click (*Macintosh*) or right-click (*Windows*), and choose Insert Template.

To view help or to view the definition of a user function, type the name of the operation or function, Control-click (*Macintosh*) or right-click (*Windows*), and choose Help for <name>.

Command Window Shortcuts

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To activate the command window	Press Command-J.	Press Ctrl+J.
To clear the command buffer	Press Command-K.	Press Ctrl+K.
To get a contextual menu of commonly-used actions	Press Control and click the history area or command line	Right-click the history area or command line
To copy a line from the history to the command buffer	Click the line and press Return or Enter. Press Option and click the line.	Click the line and press Enter. Press Alt and click the line.
To reexecute a line from the history	Click the line and press Return or enter twice Press Command-Option and click the line.	Click the line and press Ctrl+Enter Press Ctrl+Alt and click the line.
To find a recently executed command in the history	Press the Up or Down Arrow keys.	Press the Up or Down Arrow keys.
To find text in the history	Click in the history area and press Command-F.	Click in the history area and press Ctrl+F.
To find text in the command line	Click in the command line and press Command-F.	Click in the command line and press Ctrl+F.
To insert a template	Type or select the name of an operation, Control-click, and choose Insert Template.	Type or select the name of an operation, right-click, and choose Insert Template.
To get help for a built-in or external operation or function	Type or select the name of an operation, Control-click, and choose Help for <name> or choose Help→Help For <name>.	Type or select the name of an operation, right-click, and choose Help for <name> or choose Help→Help For <name>.
To view the definition of a user-defined function	Type or select the name of an operation, Control-click, and choose Go to <name> or choose Edit→Help For <name>.	Type or select the name of an operation, right-click, and choose Go to <name> or choose Edit→Help For <name>.

Chapter

II-3

Experiments, Files and Folders

Experiments

An **experiment** is a collection of Igor objects, including waves, variables, graphs, tables, page layouts, notebooks, control panels and procedures. When you create or modify one of these objects you are modifying the **current experiment**.

You can save the current experiment by choosing File→Save Experiment. You can open an experiment by double-clicking its icon on the desktop or choosing File→Open Experiment.

Saving Experiments

There are two formats for saving an experiment on disk:

- As a packed experiment file. A packed experiment file has the extension .pxp.

  All waves, procedure windows, and notebooks are saved packed into the experiment file unless you explicitly save them separately.
- As an experiment file and an experiment folder (unpacked format). An unpacked experiment file has the extension .uxp.

  All waves, procedure windows, and notebooks are saved in separate files.

The packed format is recommended for most purposes. The unpacked format is useful for experiments that include very large numbers of waves (thousands or more).

Saving as a Packed Experiment File

In the packed experiment file, all of the data for the experiment is stored in one file. This saves space on disk and makes it easier to copy experiments from one disk to another. *For most work, we recommend that you use the packed experiment file format*.

The folder containing the packed experiment file is called the **home folder**.

To save a new experiment in the packed format, choose Save Experiment from the File menu.

Saving as an HDF5 Packed Experiment File

In Igor Pro 9 and later, you can save an Igor experiment in an HDF5 file. The main advantage is that the data is immediately accessible to a wide array of programs that support HDF5. The main disadvantage is that you will need Igor Pro 9 or later to open the file in Igor. Also HDF5 is considerably slower than PXP for experiments with very large numbers of waves.

To save an experiment in HDF5 packed experiment format, choose File→Save Experiment As. In the resulting Save File dialog, choose "HDF5 Packed Experiment Files (*.h5xp)" from the pop-up menu under the list of files. Then click Save.

If you want to make HDF5 packed experiment format your default format for saving new experiments, choose Misc→Miscellaneous Settings. In the resulting dialog, click the Experiment category on the left. Then choose "HDF5 Packed (.h5xp)" from the Default Experiment Format pop-up menu.

For normal use you don't need to know the details of how Igor stores data in HDF5 packed experiment files, but, if you are curious, you can get a sense by opening such a file using the HDF5 Browser (Data→Load Waves→New HDF5 Browser).

Programmers who want to read or write HDF5 packed experiment files from other programs can find technical details under **HDF5 Packed Experiment Files** on page II-223.

HDF5 Compression for Saving Experiment Files

If you are just starting with Igor, we recommend that you skip this section until you have more familiarity with it.

In Igor Pro 9 and later, you can save HDF5 packed experiment files in compressed form. This section provides a brief introduction to HDF5 compression as it relates to saving experiment files. For details on compression, see **HDF5 Compression** on page II-213.

Compression can reduce file size but also takes more time. Whether compression is worthwhile for you depends on the nature of your data and how you trade off file size versus time. If your experiments contain lots of relatively small waves with lots of noise, compression will save little disk space. If your experiments contain large waves with large sections containing one value (for example, an image that is mostly black), compression can save considerable disk space. The only way to tell if compression is worthwhile for you is to experiment with it.

To save an HDF5 packed experiment with compression via the File menu you need to enable HDF5 default compression and then provide three parameters via the Experiment section of the Miscellaneous Settings dialog. The parameters are:

- The minimum size a wave must be before compression is used
- A compression level from 0 (no compression) to 9 (max)
- Whether you want to enable shuffle (an optional process before compression)

Here are instructions for testing HDF5 compression with your experiment files. These instructions use the SaveExperiment operation via the TestHDF5ExperimentCompression function, not File->Save Experiment, and consequently are independent of the default compression settings in the Miscellaneous dialog.

1. Activate the "Test HDF5 Experiment Compression.ipf" procedure file as a global procedure file and restart Igor. The file is in WaveMetrics Procedures/File Input Output in your Igor Pro folder.
2. If you don't know how to do this, see **Activating WaveMetrics Procedure Files** on page II-33.
3. Open a typical experiment file.
4. Execute this:

   ```
   TestHDF5ExperimentCompression(10000, 2, 0)
   ```

   10000 is the minimum number of elements in a wave to be compressed. Smaller waves are saved uncompressed.

   2 is the zip compression level.

   0 means shuffle is off which is usually what you want.

The TestHDF5ExperimentCompression command saves copies of the current experiment in uncompressed form and in compressed form using the specified parameters and prints a message in the command window history area showing the effect of compression on the time to save the experiment and on the file size. The command deletes the copies so there is no junk left over.

Try different parameters for the minimum wave size and zip compression level to see how they affect save time and compression ratio. In most cases, higher zip compression levels provide small increases in compression and are not worth the time required.

Unless you find significant compression ratios, we recommend that you eschew compression. Disk space is abundant and time is precious.

By default, compression is disabled when you use the File menu to save an HDF5 packed experiment file. If you decide that you want to use compression, see **HDF5 Default Compression** on page II-214 to learn how to enable compression via the Miscellaneous Settings dialog.

Saving as an Unpacked Experiment File

In the unpacked format, an experiment is saved as an **experiment file** and an **experiment folder**. The file contains instructions that Igor uses to recreate the experiment while the folder contains files from which Igor loads data. The experiment folder is also called the **home folder**.

The main utility of this format is that it is faster for experiments that contain very large numbers of waves (thousands or more). However the unpacked format is more fragile and thus is not recommended for routine use.

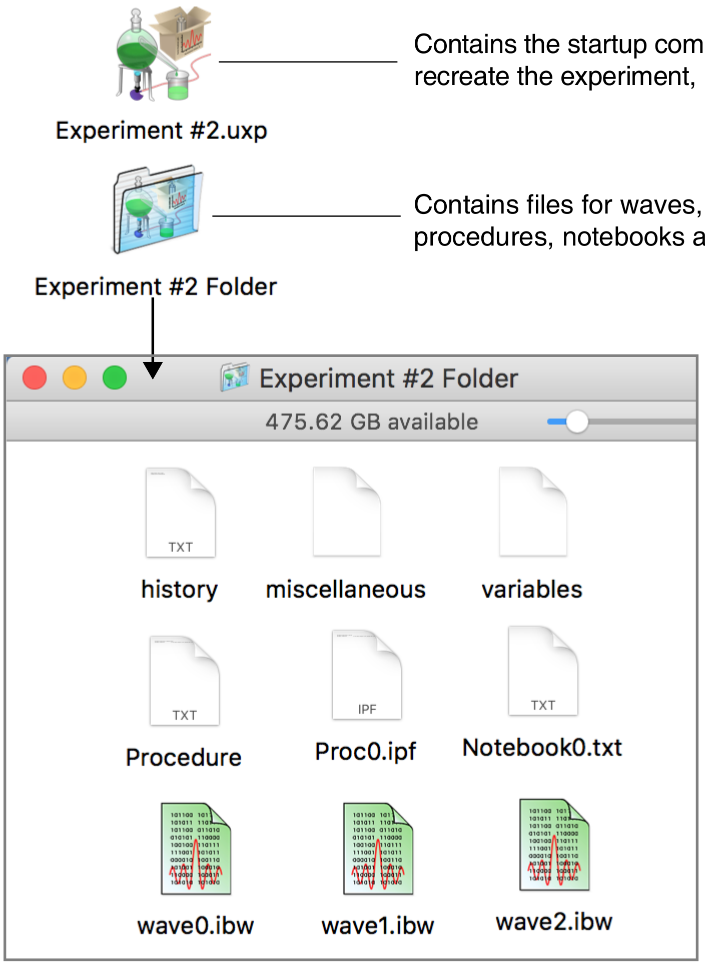
To save a new experiment in the unpacked format, choose Save Experiment from the File menu. At the bottom of the resulting Save File dialog, choose Unpacked Experiment Files from the popup menu. When you click Save, Igor writes the unpacked experiment file which as a ".uxp" extension.

Igor then automatically generates the experiment folder name by appending " Folder" or the Japanese equivalent, to the experiment file name. It then creates the unpacked experiment folder without further interaction. For example, if you enter "Test.uxp" as the unpacked experiment file name, Igor automatically uses "Test Folder", or the Japanese equivalent, as the unpacked experiment folder name.

If a folder named "Test Folder" already exists then Igor displays an alert asking if you want to reuse the folder for the unpacked experiment.

If the automatic generation of the unpacked experiment folder name causes a problem for you then you can save an experiment with the names of your choice using the **SaveExperiment** /F operation.

This illustration shows the icons used with an unpacked experiment and explains where things are stored.



Contains the startup commands that Igor executes to recreate the experiment, including all experiment windows.

Contains files for waves, variables, history, procedures, notebooks and other objects.

You normally have no need to deal with the files inside the experiment folder. Igor automatically writes them when you save an experiment and reads them when you open an experiment.

If the experiment includes data folders (see Chapter II-8, **Data Folders**) other than the root data folder, then Igor will create one subfolder in the experiment folder for each data folder in the experiment. The experiment shown in the illustration above contains no data folders other than root.

Note that there is one file for each wave. These are Igor Binary data files and store the wave data in a compact format. For the benefit of programmers, the Igor binary wave file format is documented in Igor Technical Note #003.

The “procedure” file holds the text in the experiment’s built-in procedure window. In this example, the experiment has an additional procedure window called Proc0 and a notebook.

The “variables” file stores the experiment’s numeric and string variables in a binary format.

The “miscellaneous” file stores pictures, page setup records, XOP settings, and other data.

The advantages of the unpacked experiment format are:

- Igor can save the experiment faster because it does not need to update files for waves, procedures or notebooks that have not changed.
- You can share files stored in one experiment with another experiment. However, sharing files can cause problems when you move an experiment to another disk. See **References to Files and Folders** on page II-24 for an explanation.

The disadvantages of the unpacked experiment format are:

- It takes more disk space, especially for experiments that have a lot of small waves.
- You need to keep the experiment file and folder together when you move the experiment to another disk.

Opening Experiments

You can open an experiment stored on disk by choosing Open Experiment from the File menu. You can first save your current experiment if it has been modified. Then Igor presents the Open File dialog.

When you select an experiment file and click the Open button, Igor loads the experiment, including all waves, variables, graphs, tables, page layouts, notebooks, procedures and other objects that constitute the experiment.

See **How Experiments Are Loaded** on page II-26 for details on how experiments are loaded.

Getting Information About the Current Experiment

You can see summary information about the current experiment by choosing File→Experiment Information. This displays the Experiment Information dialog.

The dialog shows when the current was last saved, whether it was modified since the last save, and other general information.

The dialog also shows whether the experiment uses long wave, variable, data folder, target window or symbolic path names. Experiments that use long names require Igor Pro 8.00 or later. See **Long Object Names** on page III-502 for details.

Merging Experiments

Normally Igor closes the currently opened experiment before opening a new one. But it is possible to merge the contents of an experiment file into the current experiment. This is useful, for example, if you want to create a page layout that contains graphs from two or more experiments. To do this, press Option (*Macintosh*) or Alt (*Windows*) and choose Merge Experiment from the File menu.

Note: *Merging experiments is an advanced feature that has some inherent problems and should be used judiciously*. If you are just learning to use Igor Pro, you should avoid merging experiments until you have become proficient. You may want to skim the rest of this section or skip it entirely. It assumes a high level of familiarity with Igor.

The first problem is that the merge operation creates a copy of data and other objects (e.g., graphs, procedure files, notebooks) stored in a packed experiment file. Whenever you create a copy there is a possibility that copies will diverge, creating confusion about which is the “real” data or object. One way to avoid this problem is to discard the merged experiment after it has served its purpose.

The second problem has to do with Igor’s use of names to reference all kinds of data, procedures and other objects. When you merge experiment B into experiment A, there is a possibility of name conflicts.

Igor prevents name conflicts for data (waves, numeric variables, string variables) by creating a new data folder to contain the data from experiment B. The new data folder is created inside the current data folder of the current experiment (experiment A in this case).

For other globally named objects, including graphs, tables, page layouts, control panels, notebooks, Gizmo plots, symbolic paths, page setups and pictures, Igor renames objects from experiment B if necessary to avoid a name conflict.

During the merge experiment operation, Igor looks for conflicts between target windows, between window recreation macros and between a target window and a recreation macro. If any such conflict is found, the window or window macro from experiment B is renamed.

Because page layouts reference graphs, tables, Gizmo plots, and pictures by name, renaming any of these objects may affect a page layout. The merge experiment operation handles this problem for page layouts that are open in experiment B. It does not handle the problem for page layout recreation macros in experiment B that have no corresponding open window.

If there are name conflicts in procedures other than window recreation macros, Igor will flag an error when it compiles procedures after finishing the merge experiment operation. You will have to manually resolve the name conflict by removing or renaming conflicting procedures.

Procedure windows have titles but do not have standard Igor names. The merge experiment operation makes no attempt to retitle procedure windows that have the same title.

The contents of the main procedure window from experiment B are appended to the contents of the main procedure window for experiment A.

During a normal experiment open operation, Igor executes experiment initialization commands. This is not done during an experiment merge.

Each experiment contains a default font setting that affects graphs and page layouts. When you do an experiment merge, the default font setting from experiment B is ignored, leaving the default font setting for experiment A intact. This may affect the appearance of graphs and layouts in experiment B.

The history from experiment B is not merged into experiment A. Instead, a message about the experiment merge process is added to the history area.

The system variables (K0…K19) from experiment B are ignored and not merged into experiment A.

Although the merge experiment operation handles the most common name conflict problems, there are a number problems that it can not handle. For example, a procedure, dependency formula or a control from experiment B that references data using a full path may not work as expected because the data from experiment B is loaded into a new data folder during the merge. Another example is a procedure that references a window, symbolic path or picture that is renamed by the merge operation because of a name conflict. There are undoubtedly many other situations where name conflicts could cause unexpected behavior.

Reverting an Experiment

If you choose Revert Experiment from the File menu, Igor asks if you’re sure that you want to discard changes to the current experiment. If you answer Yes, Igor reloads the current experiment from disk, restoring it to the state it was in when you last saved it.

New Experiments

If you choose New from the File menu, Igor first asks if you want to save the current experiment if it was modified since you last saved it. Then Igor creates a new, empty experiment. The new experiment has no experiment file until you save it.

By default, when you create a new experiment, Igor automatically creates a new, empty table. This is convenient if you generally start working by entering data manually. However, in Igor data can exist in memory without being displayed in a table. If you wish, you can turn automatic table creation off using the Experiment Settings category of the Miscellaneous Settings dialog (Misc menu).

Saving an Experiment as a Template

A template experiment provides a way to customize the initial contents of a new experiment. When you open a template experiment, Igor opens it normally but leaves it untitled and disassociates it from the template experiment file. This leaves you with a new experiment based on your prototype. When you save the untitled experiment, Igor creates a new experiment file.

Packed template experiments have ".pxt" as the file name extension instead of ".pxp". Unpacked template experiments have ".uxt" instead of ".uxp".

To make a template experiment, start by creating a prototype experiment with whatever waves, variables, procedures and other objects you would like in a new experiment. Then choose File→Save Experiment As, choose Packed Experiment Template or Unpacked Experiment Template from the file type pop-up menu, and save the template experiment.

You can convert an existing experiment file into a template file by changing the extension (".pxp" to ".pxt" or ".uxp" to ".uxt").

The Macintosh Finder’s file info window has a Stationery Pad checkbox. Checking it turns a file into a stationery pad. When you double-click a stationery pad file, Mac OS X creates a copy of the file and opens the copy. For most uses, the template technique is more convenient.

Browsing Experiments

You can see what data exists in the current experiment as well as experiments saved on disk using the Data Browser. To open the browser, choose Data→Data Browser. Then click the Browse Expt button. See **Data Folders** on page II-107 for details.

File Names and Paths

Igor supports file names up to 255 bytes long.

File paths can be up to 2000 byte long. Igor's 2000 byte limit applies even if the operating system supports longer paths.

As of this writing, on Macintosh, the operating system limits paths to 1026 bytes.

On Windows, prior to Windows 10 build 1607, released in August of 2016, the operating system limited paths to 260 bytes. Build 1607 raised this limit to 32767 characters. As of this writing, this feature is available only on systems that have "opted in" by setting the LongPathsEnabled registry setting. Using paths longer than 260 bytes may cause errors on some systems and with some software.

As described under **Path Separators** on page III-451, Igor accepts paths with either colons or backslashes on either platform.

The use of backslashes is complicated by the fact that Igor uses the backslash character as an escape character in literal strings. This is also described in detail under **Path Separators** on page III-451. The simplest solution to this problem is to use colon to separate path elements, even when you are running on Windows.

If you are writing procedures that need to extract sections of file paths or otherwise manipulate file paths, the **ParseFilePath** function may come in handy.

Symbolic Paths

A symbolic path is an Igor object that associates a short name with a folder on a disk drive. You can use this short name instead of a full path to specify a folder when you load, open or save a file. We recommend using symbolic paths instead of full paths when possible because they are more easily changed if your disk organization changes or if you want to access another location on disk.

Igor creates symbolic paths for the “Igor Pro Folder”, the “Igor Pro User Files” folder and for the current experiment’s home folder automatically. These symbolic paths exist in all experiments.

You can also create custom symbolic paths pointing to any folder on disk. These symbolic paths are part of the current experiment and cease to exist when you create a new experiment.

The New Symbolic Path Dialog

To access the New Symbolic Path dialog, choose New Path from the Misc menu. Use of this dialog is illustrated in the next section.

Symbolic Path Example

This example illustrate why you should use symbolic paths and how to use them. We assume that you have a folder full of text files containing data that you want to graph in Igor and that the organization of your hard disk is as follows:

```
hd or C:
   Users
      Jack
         Documents
            Data
               2016
                  May
                  June
                  July
```

Assume that we want to access files in the June folder.

To create a symbolic path for the folder:

1. Choose New Path from the Misc menu. This displays the New Symbolic Path dialog.
2. Enter Data in the Name field.
3. Click the Path button and locate the June folder.
4. Check the Overwrite checkbox.
5. Click Do It to create the symbolic path.

The NewPath command created by the dialog makes a symbolic path named Data which references:

```
hd:Users:Jack:Documents:Data:2016:June     (Macintosh)
C:\Users\Jack\Documents\Data\June          (Windows)
```

The command generated by the dialog uses Macintosh-style paths with colons separating components:

```
NewPath/O Data, "hd:Users:Jack:Documents:Data:2016:June"     // Macintosh

NewPath/O Data, "C:Users:Jack:Documents:Data:2016:June"      // Windows
```

The NewPath operation can also accept Windows-style paths with backslash characters, but this can cause problems and is not recommended. For details, see **Path Separators** on page III-451.

Once you have created it, you can select the Data path in dialogs where you need to choose a file. For example, in the Load Waves dialog, you can select Data from the Path list. You then click the File button and choose the file to be loaded. Igor generates a command like:

```
LoadWave /J /P=Data "Data1.txt"
LoadWave /J /P=Data "Data2.txt"
LoadWave /J /P=Data "Data3.txt"
```

These commands load data files from the June folder.

By using a symbolic path instead of the full path to the file to be loaded, you have isolated the location of the data files in one object - the symbolic path itself. This makes it easy to redirect commands or procedures that use the symbolic path. For example, you can re-execute the NewPath command replacing June with July:

```
NewPath/O Data, "hd:Users:Jack:Documents:Data:2016:July"    // Macintosh
```

```
NewPath/O Data, "C:Users:Jack:Documents:Data:2016:July"     // Windows
```

Now, if you re-execute the LoadWave commands, they will load data from July instead of June.

Isolating a specific location on disk in a symbolic path also simplifies life when you move from one user to another or from one machine to another. Instead of needing to change the full path in many commands, you can simply change the symbolic path.

Automatically Created Paths

Igor automatically creates a symbolic path named **Igor** which refers to the “Igor Pro 9 Folder”. The Igor symbolic path is useful only in rare cases when you want to access a file in the Igor Pro folder.

Igor also automatically creates a symbolic path named **IgorUserFiles** which refers to the Igor Pro User Files folder - see Special Folders for details. The IgorUserFiles symbolic path was added in Igor Pro 7.00.

Igor also automatically creates the **home** symbolic path. This path refers to the home folder for the current experiment. For unpacked experiments, this is the experiment folder. For packed experiments, this is the folder containing the experiment file. For new experiments that have never been saved, home is undefined.

Finally, Igor automatically creates a symbolic path if you do something that causes the current experiment to reference a file not stored as part of the experiment. This happens when you:

- Load an Igor binary wave file from another experiment into the current experiment
- Open a notebook file not stored with the current experiment
- Open a procedure file not stored with the current experiment

Creating these paths makes it easier for Igor to find the referenced files if they are renamed or moved. See **References to Files and Folders** on page II-24 for more information.

Symbolic Path Status Dialog

The Symbolic Path Status dialog shows you what paths exist in the current experiment. To invoke it, choose Path Status from the Misc menu.

The dialog also shows waves, notebook files, and procedure files referenced by the current experiment via a given symbolic path.

The Kill Paths Dialog

The Kill Symbolic Paths dialog removes from the current experiment symbolic paths that you no longer need. To invoke the dialog, choose Kill Paths from the Misc menu.

Killing a path does nothing to the folder referenced by the symbolic path. It just deletes the symbolic path name from Igor’s list of symbolic paths.

A symbolic path is in use — and Igor won’t let you kill it — if the experiment contains a wave, notebook window or procedure window linked to a file in the folder the symbolic path points to.

References to Files and Folders

An experiment can *reference* files that are not stored with the experiment. This happens when you load an Igor binary data file which is stored with a different experiment or is not stored with any experiment. It also happens when you open a notebook or procedure file that is not stored with the current experiment. We say the current experiment is *sharing* the wave, notebook or procedure file.

For example, imagine that you open an existing text file as a notebook and then save the experiment. The data for this notebook is in the text file somewhere on your hard disk. It is not stored in the experiment. What *is* stored in the experiment is a *reference* to that file. Specifically, the experiment file contains a command that will reopen the notebook file when you next reopen the experiment.

Note: When an experiment refers to a file that is not stored as part of the experiment, there is a potential problem. If you copy the experiment to an external drive to take it to another computer, for example, the experiment file on the external drive will contain a *reference* to a file on your hard disk. If you open the experiment on the other computer, Igor will ask you to find the referenced file. If you have forgotten to also copy the referenced file to the other computer, Igor will not be able to completely recreate the experiment.

For this reason, we recommend that you use references only when necessary and that you be aware of this potential problem.

If you transfer files between platforms file references can be particularly troublesome. See **Experiments and Paths** on page III-449.

Avoiding Shared Igor Binary Wave Files

When you load a wave from an Igor binary wave file stored in another experiment, you need to decide if you want to *share* the wave with the other experiment or *copy* it to the new experiment. Sharing creates a reference from the current experiment to the wave’s file and this reference can cause the problem noted above. Therefore, you should avoid sharing unless you want to access the same data from multiple experiments *and* you are willing to risk the problem noted above.

If you load the wave via the Load Igor Binary dialog, Igor will ask you if you want to share or copy. You can use the Miscellaneous Settings dialog to always share or always copy instead of asking you.

If you load the wave via the LoadWave operation, from the command line or from an Igor procedure, Igor will *not* ask what you want to do. You should normally use LoadWave’s /H flag, tells Igor to “copy the wave to home” and avoids sharing.

If you use the Data Browser to transfer waves from one experiment to another, Igor always copies the waves.

For further discussion, see **Home Versus Shared Waves** on page II-87 and **Home Versus Shared Text Files** on page II-56.

Adopting Files

Adoption is a way for you to copy a shared wave, notebook, or procedure file into the current experiment and break the connection to its original file. Adoption makes an experiment less fragile by being more self-contained. If you transfer it to another computer or send it to a colleague, all of the files needed to recreate the experiment will be stored in the experiment itself.

Adopting Waves

To adopt a wave file, select the wave in the Data Browser, right-click, and choose Adopt Wave. The Adopt Wave item is enabled only if the wave is shared. You can select multiple waves and adopt them in one step.

You can also adopt waves using File→Adopt All. See **Adopt All** below.

When you adopt a wave, Igor disconnects it from its original standalone file. The original file remains intact, but it is no longer referenced by the current experiment. The adoption is not final until you save the experiment.

If the current experiment is stored in packed form then, when you adopt a wave, it is saved in the packed experiment file. For an unpacked experiment, it is saved in the disk folder corresponding the waves data folder in the experiment folder.

Adopting Notebook and Procedure Files

To adopt a notebook or procedure file, choose Adopt Notebook or Adopt Procedure from the File menu. This item will be available only if the active window is a notebook or procedure file that is stored separate from the current experiment and the current experiment has been saved to disk.

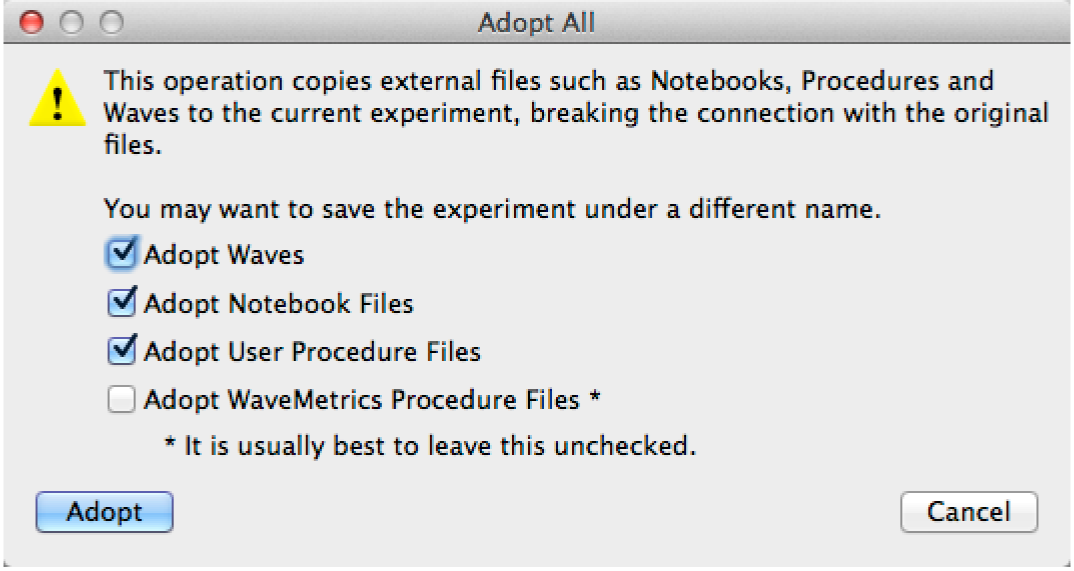
If the current experiment is stored in packed form then, when you adopt a file, Igor does a save-as to a temporary file. When you subsequently save the experiment, the contents of the temporary file are stored in the packed experiment file. Thus, the adoption is not finalized until you save the experiment.

If the current experiment is stored in unpacked form then, when you adopt a file, Igor does a save-as to the experiment's home folder. When you subsequently save the experiment, Igor updates the experiment's recreation procedures to open the new file in the home folder instead of the original file. Note that if you adopt a file in an unpacked experiment and then you do not save the experiment, the new file will still exist in the home folder but the experiment's recreation procedures will still refer to the original file. Thus, you should save the experiment after adopting a file.

To "unadopt" a procedure or notebook file, choose Save Procedure File As or Save Notebook As from the File menu.

Adopt All

You can adopt all referenced notebooks, procedure files and waves by pressing Shift and choosing File→Adopt All. This is useful when you want to create a self-contained packed experiment to send to someone else.



Adopt All

This operation copies external files such as Notebooks, Procedures and Waves to the current experiment, breaking the connection with the original files.

You may want to save the experiment under a different name.

☑ Adopt Waves

☑ Adopt Notebook Files

☑ Adopt User Procedure Files

□ Adopt WaveMetrics Procedure Files *

* It is usually best to leave this unchecked.

After clicking Adopt, choose File→Save Experiment As to save the packed experiment.

Saving All Standalone Files

Programmers may sometimes edit multiple procedure files while working on a given task. You can save all modified standalone procedure files in one step by choosing File→Save All Standalone Procedure Files. This saves standalone procedure files only, not packed procedure files including the built-in procedure window or new procedure windows that have not yet been saved to a file. The Save All Standalone Procedure Files menu item is available only when the active window is a procedure window.

Likewise you can save all modified notebook windows in one step by choosing File→Save All Standalone Notebook Files. This saves standalone notebook files only, not packed notebook files or new notebook windows that have not yet been saved to a file. The Save All Notebook Procedure Files menu item is available only when the active window is a notebook window.

How Experiments Are Loaded

It is not essential to know how Igor stores your experiment or how Igor recreates it. However, understanding this may help you avoid some pitfalls and increase your overall understanding of Igor.

Experiment Recreation Procedures

When you save an experiment, Igor creates procedures and commands, called “experiment recreation procedures” that Igor will execute the next time you open the experiment. These procedures are normally not visible to you. They are stored in the experiment file.

The experiment file of an unpacked experiment contains plain text, but its extension is not “.txt”, so you can’t open it with most word processors or editors.

As an example, let’s look at the experiment recreation procedures for a very simple unpacked experiment:

```
// Platform=Macintosh, IGORVersion=8.000, ...

// Creates the home symbolic path
NewPath home ":Unpacked Experiment Folder:"

// Reads the experiment variables from the "variables" file
ReadVariables

// Loads the experiment's waves
LoadWave/C /P=home "wave0.ibw"
LoadWave/C /P=home "wave1.ibw"
LoadWave/C /P=home "wave2.ibw"

DefaultFont "Helvetica"

MoveWindow/P 5,62,505,335         // Positions the procedure window
MoveWindow/C 2,791,1278,1018      // Positions the command window

Graph0()                          // Recreates the Graph0 window

// Graph recreation macro for Graph0
Window Graph0() : Graph
   PauseUpdate; Silent 1   // building window...
   Display /W=(35,44,430,252) wave0,wave1,wave2
EndMacro
```

When you open the experiment, Igor reads the experiment recreation procedures from the experiment file into the procedure window and executes them. The procedures recreate all of the objects and windows that constitute the experiment. Then the experiment recreation procedures are removed from the procedure window and your own procedures are loaded from the experiment’s procedure file into the procedure window.

For a packed experiment, the process is the same except that all of the data, including the experiment recreation procedures, is packed into the experiment file.

Experiment Initialization Commands

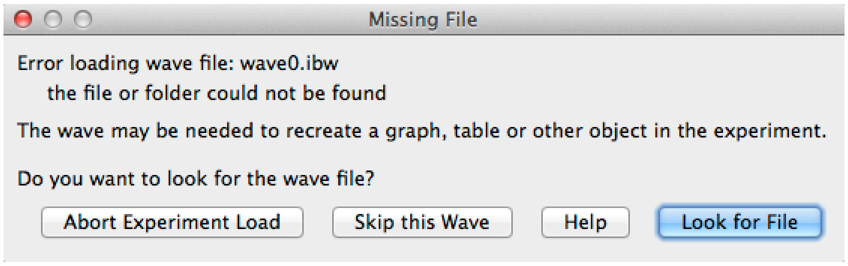
After executing the experiment recreation procedures and loading your procedures into the procedure window, Igor checks the contents of the procedure window. Any commands that precede the first macro, function or menu declaration are considered **initialization commands**. If you put any initialization commands in your procedure window then Igor executes them. This mechanism initializes an experiment when it is first loaded.

Savvy Igor programmers can also define a function that is executed whenever Igor opens any experiment. See **User-Defined Hook Functions** on page IV-280.

Errors During Experiment Load

It is possible for the experiment loading process to fail to run to a normal completion. This occurs most often when you move or rename a file or folder. It also happens if you move an experiment to a different computer and forget to also move referenced files or folders. See **References to Files and Folders** on page II-24 for details.

When a file is missing, Igor presents a dialog giving you several options:



Missing File

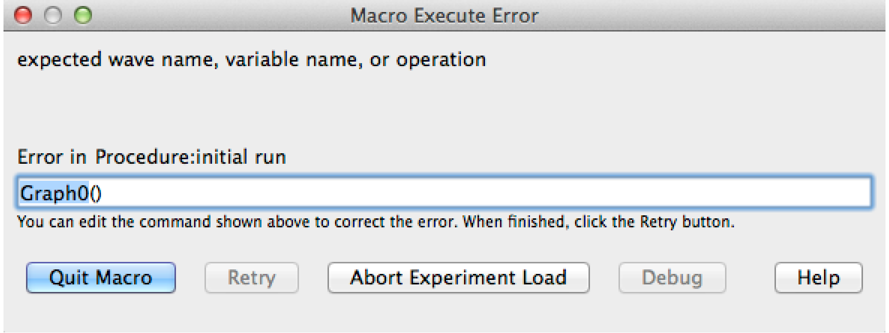
Error loading wave file: wave0.ibw
the file or folder could not be found

The wave may be needed to recreate a graph, table or other object in the experiment.

Do you want to look for the wave file?

If you elect to abort the experiment load, Igor will alert you that the experiment is in an inconsistent state. It displays some diagnostic information that might help you understand the problem and changes the experiment to Untitled. You should choose New Experiment or Open Experiment from the File menu to clear out the partially loaded experiment.

If you elect to skip loading a wave file, you may get another error later, when Igor tries to display the wave in a graph or table. In that case, you will see a dialog like this:



Macro Execute Error

expected wave name, variable name, or operation

Error in Procedure:initial run

Graph0()

You can edit the command shown above to correct the error. When finished, click the Retry button.

In this example, Igor is executing the Graph0 macro from the experiment recreation procedures in an attempt to recreate a graph. Since you elected to skip loading wave0, Igor can’t display it.

You have three options at this point, as explained in the following table.

Option	Effect
Quit Macro	Stops executing the current macro but continues experiment load. In this example, Graph0 would not be recreated. After the experiment load Igor displays diagnostic information.
Abort Experiment Load	Aborts the experiment load immediately and displays diagnostic information.
Retry	In this example, you could fix the macro by deleting “wave0,”. You would then click the Retry button. Igor would create Graph0 without wave0 and would continue the experiment load.

With the first two options, Igor leaves the experiment untitled so that you don’t inadvertently wipe out the original experiment file by doing a save.

How Igor Handles Missing Folders

When Igor saves an experiment, it stores commands in the experiment file that will recreate the experiment’s symbolic paths when you reopen the experiment. The commands look something like this:

```
NewPath home ":Test Exp Folder:"
NewPath/Z Data1 "::Data Folder #1:"
NewPath Data2 "::Data Folder #2:"
NewPath/Z Data3 "hd:Test Runs:Data Folder #3:"      // Macintosh
NewPath/Z Data3 "C:Test Runs:Data Folder #3:"       // Windows
```

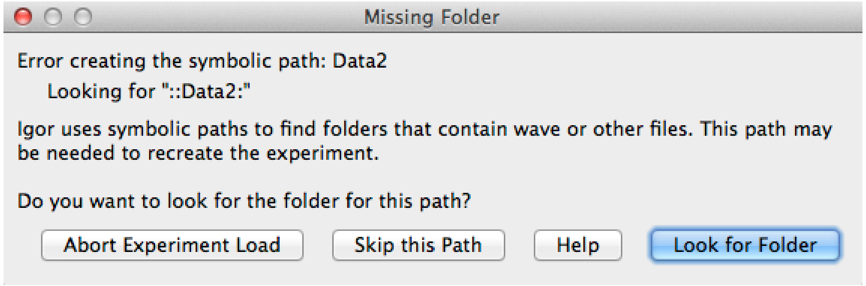
The location of the home folder is specified relative to the folder containing the experiment file. The locations of all other folders are specified relative to the experiment folder or, if they are on a different volume, using absolute paths. Using relative paths, where possible, ensures that no problems will arise if you move the experiment file and experiment folder *together* to another disk drive or another location on the same disk drive.

The /Z flags indicate that the experiment does not need to load any files from the Data1 and Data3 folders. In other words, the experiment has symbolic paths for these folders but no files need to be loaded from them to recreate the experiment.

When you reopen the experiment, Igor executes these NewPath commands. If you have moved or renamed folders, or if you have moved the experiment file, the NewPath operation will be unable to find the folder.

If the folder associated with the symbolic path is not needed to recreate the experiment, the NewPath command includes the /Z flag. In this case, Igor skips the creation of the symbolic path, generates no error, and just continues the load. The experiment will wind up without the missing symbolic path.

If the missing folder is needed to load waves, notebooks or procedure files then Igor asks if you want to look for the folder by displaying the Missing Folder dialog.



Error creating the symbolic path: Data2

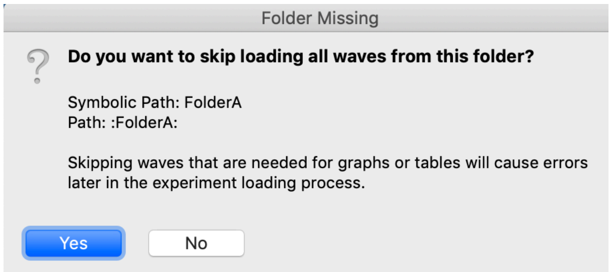
Looking for "::Data2:"

Igor uses symbolic paths to find folders that contain wave or other files. This path may be needed to recreate the experiment.

Do you want to look for the folder for this path?

If you click Look for Folder, Igor presents a Choose Folder dialog in which you can locate the missing folder.

If you click Skip This Path, Igor does not create the symbolic path. Igor then asks if you want to skip waves loaded using the symbolic path that is being skipped:



Folder Missing

Do you want to skip loading all waves from this folder?

Symbolic Path: FolderA
Path: :FolderA:

Skipping waves that are needed for graphs or tables will cause errors later in the experiment loading process.

If you click Yes, Igor skips all LoadWave commands using the symbolic path associated with the missing folder. This will cause errors later in the experiment loading process if the waves are needed for graphs, tables or other windows.

If you click No and the folder contains wave files referenced by the experiment, you will get Missing File dialogs later in the process giving you a chance to locate the wave files.

For example, if the experiment loads two waves using the Data2 path then the experiment's recreation commands would contain two lines like this:

```
LoadWave/C/P=Data2 "wave0.ibw"
LoadWave/C/P=Data2 "wave1.ibw"
```

If you elected to skip creating the Data2 path and clicked Yes when asked if you want to skip waves from that path, then Igor skips these LoadWave commands altogether.

If you elected to skip creating the Data2 path and clicked No when asked if you want to skip waves from that path, then each of these LoadWave commands presents the Missing File dialog.

If you were unable to find the Data2 folder then each of these LoadWave commands will present the Missing Wave File dialog.

If you are unable to find the wave file and if the wave is used in a graph or table, you will get more errors later in the experiment recreation process, when Igor tries to use the missing wave to recreate the graph or table.

How Experiments Are Saved

When you save an experiment for the first time, Igor just does a straight-forward save in which it creates a new file, writes to it, and closes it. However, when you resave a pre-existing experiment, which overwrites the previous version of the experiment file, Igor uses a "safe save" technique. This technique is designed to preserve your original file in the event of an error while writing the new version.

For purposes of illustration, we will assume that we are resaving an experiment file named "Experiment.pxp". The safe save proceeds as follows:

1. Rename the original file as "Experiment.pxp.T0.noindex". If an error occurs during this step, the save operation is stopped and Igor displays an error message.
2. Write the new version of the file as "Experiment.pxp".
3. If step 2 succeeds, delete the original file ("Experiment.pxp.T0.noindex").

   If step 2 fails, delete the new version of the file and rename the original version with the original name.

The ".noindex" suffix tells Apple's Spotlight program not to interfere with the save by opening the temporary file at an inopportune time.

The next three subsections are for use in troubleshooting file saving problems only. If you are not having a problem, you can skip them.

Experiment Save Errors

There are many reasons why an error may occur during the save of an experiment. For example, you may run out of disk space, the server volume you are saving to might be disconnected, or you may have a hardware failure, but these are uncommon.

The most common reason for a save error is that you cannot get write access to the file because:

1. The file is locked (Macintosh Finder) or marked read-only (Windows desktop).
2. You don't have permission to write to the folder containing the file.
3. You don't have permission to write to this specific file.
4. The file has been opened by another application. This could be a virus scanner, an anti-spyware program or an indexing program such as Apple's Spotlight.

Here are some troubleshooting techniques.

Macintosh File Troubleshooting

Open the file's Get Info window and verify that the file is not marked as locked. Also check the lock setting of the folder containing the file.

Next try doing a Save As to a folder for which you know you have write access, for example, to your home folder (e.g., "/Users/<user>" where <user> is your user name). If this works, the problem may be that you did not have sufficient permissions to write to the original folder or to the original file. Use the Finder Get Info window Sharing and Permissions section to make sure that you have read/write access for the file and folder.

If you are able to save a file to a new location but get an error when you try to resave the file, which overwrites the original file, then this may be an issue of another program opening the file at an inopportune time.

Windows File Troubleshooting

Open the file's Properties window and uncheck the read-only checkbox if it is checked. Do the same for the folder containing the file.

Next try doing a Save As to a folder for which you know you have write access, for example, to your Documents folder. If this works, the problem may be that you did not have sufficient permissions to write to the original folder or to the original file. This would happen, for example, if the folder was inside the Program Files folder and you are not running as an administrator.

If you think you should be able to write to the original file location, you will need to investigate permissions. You may want to enlist the help of a local expert as this can get complicated and works differently in different versions of Windows.

If you are able to save a file to a new location but get an error when you try to resave the file, which overwrites the original file, then this may be an issue of another program opening the file at an inopportune time. This typically happens in step 3 of the safe-save technique described above. Try disabling your antivirus software. For a technical explanation of this problem, see http://support.microsoft.com/kb/316609.

Special Folders

This section describes special folders that Igor automatically searches when looking for help files, Igor extensions (plug-ins that are also called XOPs) and procedure files.

Igor Pro Folder

The Igor Pro folder is the folder containing the Igor application on Macintosh and the IgorBinaries folder on Windows. By default, this folder has the Igor Pro major version number in its name, for example, “Igor Pro 9 Folder”, but it is generically called the “Igor Pro Folder”.

Igor looks inside the Igor Pro Folder for these special subfolders:

```
Igor Help Files
Igor Extensions
Igor Extensions (64-bit)
Igor Procedures
User Procedures
WaveMetrics Procedures
```

The Igor installer puts files in the special folders. Igor searches them when looking for help files, extensions and procedure files. *With very rare exceptions, you should not make any changes to the Igor Pro Folder.*

Igor Pro User Files

At launch time, Igor creates a special folder called the Igor Pro User Files folder. By default, this folder has the Igor Pro major version number in its name, for example, "Igor Pro 9 User Files", but it is generically called the "Igor Pro User Files" folder.

On Macintosh, the Igor Pro User Files folder is created by default in:

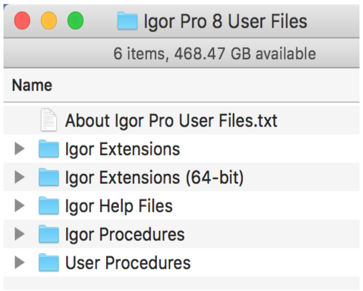
```
/Users/<user>/Documents/WaveMetrics
```

On Windows it is created by default in:

```
C:\Users\<user>\Documents\WaveMetrics
```

You can change the location of your Igor Pro User Files folder using Misc→Miscellaneous Settings but this should rarely be necessary.

The Igor Pro User Files folder looks like this:



Igor Pro 8 User Files

6 items, 468.47 GB available

Name

About Igor Pro User Files.txt
Igor Extensions
Igor Extensions (64-bit)
Igor Help Files
Igor Procedures
User Procedures

You can activate additional help files, extensions and procedure files that are part of the Igor Pro installation, that you create, or that you receive from third parties. To do this, add files, or aliases/shortcuts pointing to files, to the special subfolders within the Igor Pro User Files folder. See **Activating WaveMetrics Procedure Files** on page II-33 **Activating WaveMetrics XOPs** on page II-33 for details on activating additional WaveMetrics files.

You can display the Igor Pro User Files folder on the desktop by choosing Help→Show Igor Pro User Files. To display both the Igor Pro Folder and the Igor Pro User Files folder, which you typically want to do to activate a WaveMerics XOP or procedure file, press the shift key and choose Help→Show Igor Pro Folder and User Files.

Igor Help Files Folder

When Igor starts up, it opens any Igor help files in "Igor Pro Folder/Igor Help Files" and in "Igor Pro User Files/Igor Help Files". It treats any aliases, shortcuts and subfolders in "Igor Help Files" in the same way.

Standard WaveMetrics help files are pre-installed in "Igor Pro Folder/Igor Help Files".

If there is an additional help file that you want Igor to automatically open at launch time, put it or an alias/shortcut for it in "Igor Pro User Files/Igor Help Files".

Igor Extensions Folder

An Igor extension, also called an XOP (“external operation”), is a plug-in that adds functionality to Igor. WaveMetrics provides some extensions with Igor. Igor users and third parties can also create extensions. See **Igor Extensions** on page III-511 for details.

Igor extensions come in 32-bit and 64-bit versions. 32-bit extensions can run only with IGOR32 (the 32-bit version of Igor) and 64-bit extensions can run only with IGOR64 (the 64-bit version of Igor). As noted under **Igor 32-bit and 64-bit Versions** on page I-2, both IGOR32 and IGOR64 are installed on Windows but only IGOR64 is available on Macintosh.

When IGOR32 starts up, it searches "Igor Pro Folder/Igor Extensions" and "Igor Pro User Files/Igor Extensions" for 32-bit Igor extension files. These extensions are available for use in IGOR32. It treats any aliases, shortcuts and subfolders in "Igor Extensions" in the same way.

When IGOR64 starts up, it searches "Igor Pro Folder/Igor Extensions (64-bit)" and "Igor Pro User Files/Igor Extensions (64-bit)" for 64-bit Igor extension files. These extensions are available for use in IGOR64. It treats any aliases, shortcuts and subfolders in "Igor Extensions (64-bit)" in the same way.

Standard WaveMetrics extensions are pre-installed in "Igor Pro Folder/Igor Extensions" and "Igor Pro Folder/Igor Extensions (64-bit)".

Additional WaveMetrics extensions are described in the "XOP Index" help file, which you can access through the Help→Help Windows menu, and can be found in "Igor Pro Folder/More Extensions" and "Igor Pro Folder/More Extensions (64-bit)".

If there is an additional extension that you want to use, put it or an alias/shortcut pointing to it in "Igor Pro User Files/Igor Extensions" or "Igor Pro User Files/Igor Extensions (64-bit)".

Igor Procedures Folder

When Igor starts up, it automatically opens any procedure files in "Igor Pro Folder/Igor Procedures" and in "Igor Pro User Files/Igor Procedures". It treats any aliases, shortcuts and subfolders in "Igor Procedures" in the same way. Such procedure files are called "global" procedure files and are available for use from all experiments. See **Global Procedure Files** on page III-399 for details.

Standard WaveMetrics global procedure files are pre-installed in "Igor Pro Folder/Igor Procedures".

Additional WaveMetrics procedure files are described in the "WM Procedures Index" help file and can be found in "Igor Pro Folder/WaveMetrics Procedures". You may also create your own global procedure files or obtain them from third parties.

If there is an additional procedure file that you want Igor to automatically open at launch time, put it or an alias/shortcut pointing to it in "Igor Pro User Files/Igor Procedures".

User Procedures Folder

You can load a procedure file from another procedure file using a #include statement. This technique is used when one procedure file requires another. See **Including a Procedure File** on page III-401 for details.

When Igor encounters a #include statement, it searches for the included procedure file in "Igor Pro-Folder/User Procedures" and in "Igor Pro User Files/User Procedures". Any aliases, shortcuts and subfolders in "User Procedures" are treated the same way.

If there is an additional procedure file that you want to include from your procedure files, put it or an alias/shortcut pointing to it in "Igor Pro User Files/User Procedures".

WaveMetrics Procedures Folder

The "Igor Pro Folder/WaveMetrics Procedures" folder contains an assortment of procedure files created by WaveMetrics that may be of use to you. These files are described in the WM Procedures Index help file which you can access through the Help→Help Windows menu.

You can load a WaveMetrics procedure file from another procedure file using a #include statement. See **Including a Procedure File** on page III-401 for details.

There is no WaveMetrics Procedures folder in the Igor Pro User Files folder.

Activating Additional WaveMetrics Files

The following sections explain how to activate additional Igor Pro features.

Activating WaveMetrics Procedure Files

To activate a WaveMetrics-supplied procedure file that you want to be available in all experiments:

1. Press the shift key and choose Help→Show Igor Pro Folder and User Files. This displays the Igor Pro Folder and the Igor Pro User Files folder on the desktop.
2. Open the WaveMetrics Procedures folder in the Igor Pro Folder and identify the WaveMetrics procedure file that you want to activate.
3. Make an alias/shortcut for the file and put it in "Igor Pro User Files/Igor Procedures" folder. This causes Igor to automatically load the procedure file the next time Igor is launched.
4. Restart Igor.

The activated procedure file appears in the Windows→Procedure Windows submenu.

Activating WaveMetrics XOPs

To activate a WaveMetrics-supplied XOP (a plug-in also called an "extension") :

1. Press the shift key and choose Help→Show Igor Pro Folder and User Files. This displays the Igor Pro Folder and the Igor Pro User Files folder on the desktop.
2. Open the "More Extensions (64-bit)" folder in the Igor Pro Folder and identify the XOP that you want to activate. (If you are running IGOR32 on Windows, open the "More Extensions" folder.)
3. Make an alias/shortcut for the XOP file and put it in "Igor Pro User Files/Igor Extensions" folder. This causes Igor to automatically load the XOP the next time Igor is launched.
4. Restart Igor.

The activated XOP adds its operations and/or functions to the Igor Help browser and they are available for programmatic use. Some XOPs also add menu items.

Activating Other Files

You may create an Igor package or receive a package from a third party. You should store each package in its own folder in the Igor Pro User Files folder or elsewhere, at your discretion. You should not store such files in the Igor Pro Folder because it complicates backup and updating.

To activate files from the package, create aliases/shortcuts for the package files and put them in the appropriate subfolder of the Igor Pro User Files folder.

If you have a single procedure file or a single Igor extension that you want to activate, you may prefer to put it directly in the appropriate subfolder of the Igor Pro User Files folder.

Activating Files in a Multi-User Scenario

Our recommendation is that you activate files using the special subfolders in the Igor Pro User Files folder, not in the Igor Pro Folder. An exception to this is the multi-user scenario where multiple users are running the same copy of Igor from a server. In this case, if you want to activate a file for all users, put the file or an alias/shortcut for it in the appropriate subfolder of the Igor Pro Folder. Users will have to restart Igor for the change to take effect.

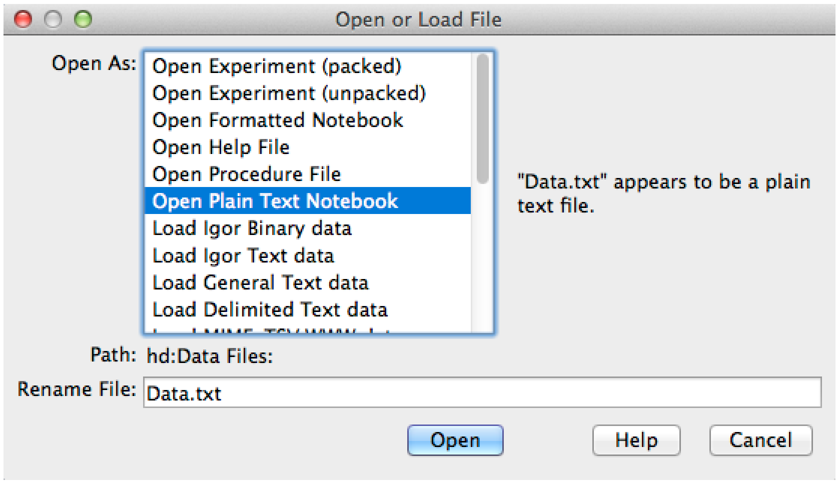
Igor File-Handling

Igor has many ways to open and load files. The following sections discuss some of the ways Igor deals with the various files it is asked to open.

The Open or Load File Dialog

When you open a file using menu item, such as File→Open File→Notebook, there is no question of how Igor should treat the file. This is not always the case when you drop a file onto the Igor icon or double-click a file on the desktop.

Often, Igor can determine how to open or load a file, and it will simply do that without asking the you about it. Sometimes Igor recognizes that a file (such as a plain text file or a formatted Igor notebook) can be appropriately opened several ways, and will ask you what to do by displaying the Open or Load File Dialog. The dialog presents a list of ways to open the file (usually into a window) or to load it as data.



"Data.txt" appears to be a plain text file.

Path: hd:Data Files:

Rename File: Data.txt

Tip: You can force this dialog to appear by holding down Shift when opening a file through the Recent Files or Recent Experiments menus, or when dropping a file onto the Igor icon.

This is especially useful for opening Igor help files as a notebook file for editing, or to open a notebook as a help file, causing Igor to compile it.

The list presents three kinds of methods for handling the file:

1. Open the file as a document window or an experiment.
2. Load the file as data without opening another dialog.
3. Load the file as data through the Load Waves Dialog or a File Loader Extension dialog.

If you choose one of the list items marked with an asterisk, Igor displays the selected dialog as if you had chosen the corresponding item from the Load Waves submenu of the Data menu.

Information about the file, or about how it was most recently opened, is displayed to the right of the list. The complete path to the file is shown below the list.

If you change the text in the Rename file edit box, Igor changes the file name before opening or loading the file.

Recent Files and Experiments

When you use a dialog to open or save an experiment or a file, Igor adds it to the Recent Experiments or Recent Files submenu in the File menu. When you choose an item from these submenus, Igor opens the experiment or file the same way in which you last opened or saved it.

For example, if you last opened a text file as an unformatted notebook, selecting the file from Recent Files will again open the file as an unformatted notebook. If you loaded it as a general text data file, Igor will load it as data again.

Igor does not remember all the details of how you originally load a data file, however. If you load a text data file with all sorts of fiddly tweaks about the format, Igor won't load it using the those same tweaks. To guarantee that Igor does load the data correctly, use the appropriate Load Data dialog.

Selecting an experiment or file while pressing Shift displays the Open or Load File dialog in which you can choose how Igor will open or load that file.

Desktop Drag and Drop

On Macintosh, you can drag and drop one or more files of almost any type onto the Igor icon in the dock or onto an alias for the Igor application on the desktop. On Windows, you must drag and drop files into an Igor window. One use for this feature is to load multiple data files at once.

If a dropped file was opened or loaded recently, it is listed in the Recent Files or Recent Experiments menu. In this case, Igor reopens or reloads it the same way.

If you press Shift while dropping a file on Igor, Igor displays the Open or Load File dialog in which you can choose how the file is to be handled.

Advanced programmers can customize Igor to handle specific types of files in different ways, such as automatically loading files with an XOP. See **User-Defined Hook Functions** on page IV-280.

IGOR64 Experiment Files

The 64 bit version of Igor can store waves larger than the 32-bit limit in packed experiment files but not in unpacked experiments. Huge wave storage uses a 64-bit capable wave record and is supported only for numeric waves.

The 32-bit versions of Igor, starting with version 6.20, supports 64-bit capable wave records. By default, IGOR32 attempts to read new these records and generates a lack of memory warning if they can not be loaded.

The rest of this section describes features for advanced users.

You can control some aspects of how Igor deals with 64-bit capable wave records using SetIgorOption. Remember that SetIgorOption settings last only until you quit Igor.

```
SetIgorOption UseNewDiskHeaderBytes = bytes
```

When bytes is non-zero, if the size of the wave data exceeds this value, the 64-bit capable wave record type is used. In IGOR32, the default value for bytes is 0 (off). In IGOR64, the default value for bytes is 100E6. If you want the 64-bit capable wave record type to be used only when absolutely necessary, use 2E9.

```
SetIgorOption UseNewDiskHeaderCompress = bytes
```

When bytes is non-zero, if the size of the wave data exceeds this value and if the 64-bit capable wave record type is used, the data is written compressed. The default value is 0 (off) because, although you can get substantial file size reduction for some kinds of data, there is also a substantial speed penalty when saving the experiment. In many cases, the compression actually results in a larger size than the original. If this occurs, Igor writes the original data instead of the compressed data.

```
SetIgorOption MaxBytesReadingWave = bytes
SetIgorOption BigWaveLoadMode = mode
```

These two options control how IGOR32 acts when loading experiments with the 64-bit capable wave record types.

mode can be 0 (default) for no special action. Igor attempts to read all the records.

If mode is 1, then any waves with data size exceeding bytes are silently skipped.

If mode is 2, then any waves with data size exceeding bytes are downgraded to a smaller size and only that portion of the data is loaded. Such partial waves are marked as locked and bit 1 of the lock flag is also set. In addition, the wave note contains the text "**PARTIAL LOAD**" and a warning dialog is presented after the experiment loads.

The default value for bytes is 500E6 in IGOR32.

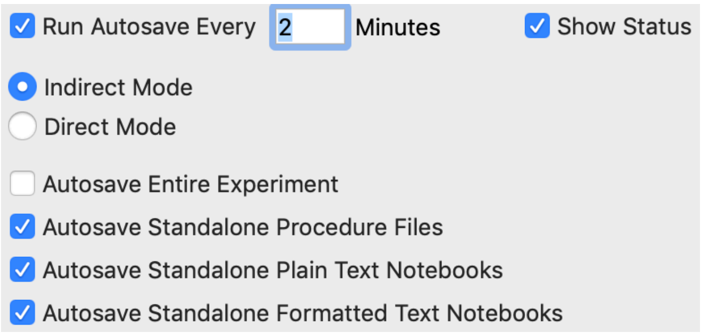
Autosave

Igor can autosave files while you work. This is of use in the event that Igor crashes or some other catastrophe occurs before you save.

If your habit is to save anytime you have made non-trivial changes that you want to keep then you don't need autosave. If your habit is to do a significant amount of work without saving, which is inherently risky, then autosave may prevent loss of work in the event of a crash.

Autosave is not foolproof and can cause confusing and unpredictable behavior if multiple users modify the same file or if you modify a file in multiple instance of Igor. For these and other reasons, you should consider autosave a last resort and always save when you have made progress and back up any work that would be painful to lose.

Autosave is turned off by default. You can turn it on and fine-tune how it works by choosing Misc→Miscellaneous Settings and clicking the Autosave icon in the lefthand list to display the Autosave pane of the Miscellaneous Settings dialog. The heart of it looks like this:



☑ Run Autosave Every 2 Minutes ☑ Show Status

● Indirect Mode
○ Direct Mode

□ Autosave Entire Experiment
☑ Autosave Standalone Procedure Files
☑ Autosave Standalone Plain Text Notebooks
☑ Autosave Standalone Formatted Text Notebooks

You turn autosave on by checking the Run Autosave checkbox. If turned on, autosave runs at the interval in minutes that you specify. Autosave runs when Igor is idling and not while procedures are running or if a modal dialog is displayed.

Alternatives to automatically autosaving include:

- Manually saving by choosing File→Save Experiment when you know the experiment is in good shape
- Running autosave manually when you know the experiment is in good shape by choosing File→Run Autosave Now
- Manually saving modified procedure files only using File→Save All Standalone Procedure Files

Igor can display information in its status bar showing the status of autosave. You can turn the autosave information on or off using the Show Status checkbox in the autosave pane of the Miscellaneous Settings dialog or by right-clicking the status bar. If autosave is enabled, the status bar information shows the time of the next autosave run. “Run” means “check if there are any files to be autosaved and autosave them if so.”

Modifying the same file in different instances of Igor while autosave is on will cause confusing and unpredictable behavior.

Igor programmers can query autosave settings using **IgorInfo**(13).

What Autosave Saves

Depending on the settings you choose, autosave can save the following types of files:

- Entire experiment files
- Standalone procedure files
- Standalone notebook files

In the sections below, we refer to the types of files you have chosen to autosave as "autosaveable files".

Autosave does not work on the following:

- Files that you have not yet saved to disk

  This includes experiment files, procedure files and notebook files that you have created but not yet saved to disk
- Unpacked experiment files (.uxp) in indirect mode
- Copies of procedure files #included into independent modules

  Such files should not be modified. See **Independent Modules and #include** on page IV-240 for details.

Autosave saves the following items only for packed experiments (.pxp and .h5xp) and **only if the Autosave Entire Experiment checkbox is checked**:

- The built-in procedure window
- Packed procedure files
- Packed notebook files
- Graphs, tables, page layouts, Gizmo plots, control panels and other windows
- Data folders and waves

Autosave Is Not A Substitute for Manual Saving or for Backing Up

The purpose of autosave is to increase the chance that you can recover work if Igor or your machine crashes or there is a power failure. Relying on autosave as a substitute for frequent saving backing up is courting disaster.

Depending on the autosave settings you choose, autosave may save at the wrong time (for example, just after you made a mistake) or may not save all of your work (see **What Autosave Saves** on page II-37).

A crash or other disaster may occur after you have made important changes but before autosave automatically runs.

You may realize that the work you saved today is wrong and you need to go back to yesterday's version of a file.

Your computer may fail catastrophically.

Your computer may be attacked by malware.

Your computer may be stolen.

Your computer may be damaged by fire, natural disaster, or some other mishap.

For these and other reasons, **you should always save when you have made progress and back up any work that would be painful to lose**. You should back up locally and also remotely for the case where your computer is destroyed or stolen.

Autosave Modes

You control the mode of operation of autosave using the Autosave pane of the Miscellaneous Settings dialog.

Autosave supports three modes. Each has its strengths and weaknesses.

- **Off**

  Autosave is off when the Run Autosave checkbox is unchecked.

  In this mode, Igor's autosave routine does not automatically run.

  Strengths

  Easy to understand.

  Takes no time.

  You are in complete control of when files are saved.

  Weaknesses

  You are in complete control of when files are saved.

  You are responsible for saving when you have made important changes.

- **Indirect Mode**

  Saves the current experiment (if it is packed, not if it is unpacked) and/or standalone procedure and standalone notebook files to .autosave files alongside the original files. For example, "Experiment.pxp" is autosaved to "Experiment.pxp.autosave" and "Proc0.ipf" is autosaved to "Proc0.ipf.autosave".

  Saving the current experiment is the same as choosing File→Save Experiment Copy.

  Saving a standalone procedure file is the same as choosing File→Procedure Copy.

  Saving a standalone notebook is the same as choosing File→Notebook Copy.

  You can choose whether to autosave the current experiment, standalone procedure files, and standalone notebooks.

  Indirect autosave of unpacked experiments is not supported.

  Strengths

  Does not risk overwriting your original files at the wrong time, for example just after you made a mistake.

  Weaknesses

  Saving the entire experiment can take a while for very large experiments.

  If you do not elect to save the entire experiment, many things are not autosaved, including the built-in procedure window, packed procedure files and notebooks, graphs, layouts and other windows, data folders and waves - see What Autosave Saves for details.

  Can interfere with sharing of files by multiple users or by multiple instances of Igor launched by a single user.

  Does not work with unpacked experiments.

  The .autosave files represent more clutter relative to direct mode.

  Recovering after a crash is more complicated compared to direct mode. Igor will ask if you want to use the .autosave file or the original file.

- **Direct Mode**

  Saves the current experiment and/or standalone procedure and standalone notebook files directly.

  Saving the current experiment is the same as choosing File→Save Experiment.

  Saving a standalone procedure file is the same as choosing File→Procedure.

  Saving a standalone notebook is the same as choosing File→Notebook.

  You can choose whether to autosave the current experiment, standalone procedure files, and standalone notebooks.

  Strengths

Simplicity.

Weaknesses

Saving the entire experiment can take a while for very large experiments.

Autosave may save a file at the wrong time, for example just after you made a mistake.

If you do not elect to save the entire experiment, many things are not autosaved, including the built-in procedure window, packed procedure files and notebooks, graphs, layouts and other windows, data folders and waves - see What Autosave Saves for details.

Can interfere with sharing of files by multiple users or by multiple instances of Igor launched by a single user.

Changes That Tee Up Autosaving the Entire Experiment

If you check the Autosave Entire Experiment checkbox and make a substantive change then Igor autosaves the entire experiment on the next autosave run.

Changes that are considered substantive include creating, killing or modifying waves and data folders, changing the content of a graph, table, layout, panel or Gizmo plot, and editing a procedure file or notebook.

Autosaving the entire experiment can be time-consuming. To prevent autosaving the entire experiment too often, some relatively minor changes do not tee up an experiment autosave. Among these are moving and resizing windows, changes to the history area that don't cause other objects to substantively change, and changes to global variables.

If the File→Run Autosave Now menu item is disabled, this indicates that no changes considered substantive were made since the last experiment save or last autosave.

Forcing Autosave to Run

You can force Igor to run its autosave routine when it is turned off or before it normally would run by choosing File→Run Autosave Now.

If the File→Run Autosave Now menu item is disabled, this means that there are no autosaveable files at present. This typically occurs because no autosaveable files were modified since the last time the autosave routine ran.

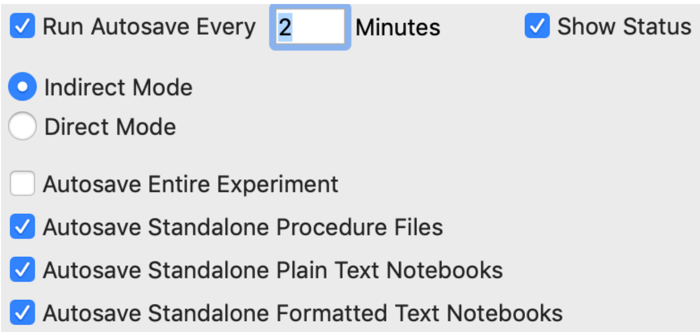
The Run Autosave Now menu command runs Igor's autosave procedure regardless of the state of the Run Autosave checkbox in the Autosave pane of the Miscellaneous Settings dialog. It does respect the other settings in that pane.

Saving Standalone Files Without Autosave

You can save all modified standalone procedure files or all modified standalone notebooks at once whether autosave is on or off. See **Saving All Standalone Files** on page II-26 for details.

Indirect Autosave Mode

The indirect autosave mode is more complicated than the direct mode. We will illustrate how indirect autosave works assuming the following settings in the Autosave pane of the Miscellaneous Settings dialog:



☑ Run Autosave Every 2 Minutes ☑ Show Status

● Indirect Mode
○ Direct Mode

□ Autosave Entire Experiment
☑ Autosave Standalone Procedure Files
☑ Autosave Standalone Plain Text Notebooks
☑ Autosave Standalone Formatted Text Notebooks

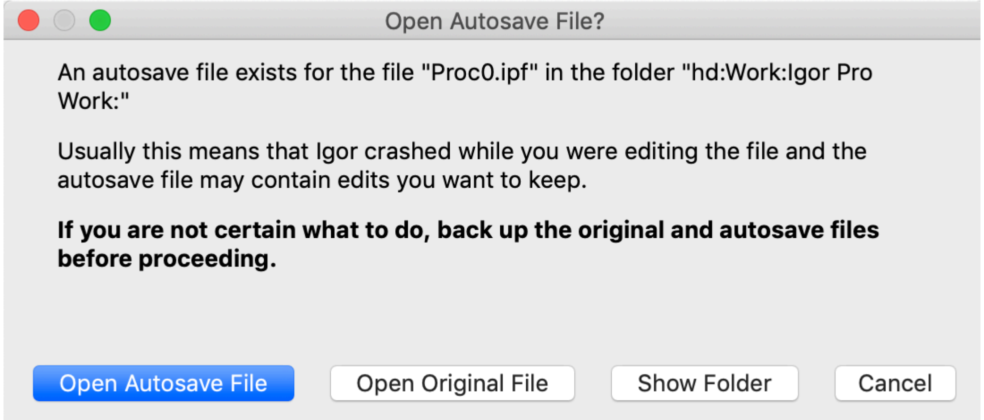
Now assume that you are editing a standalone procedure window that you previously saved to a file named "Proc0.ipf".

Every two minutes, Igor checks to see if there are autosaveable files that have been modified since the previous autosave. This check is performed when Igor is idling and not while procedures are running or if a modal dialog is displayed. If you have modified "Proc0.ipf" since the last autosave, Igor autosaves the file by writing its current contents to a file named "Proc0.ipf.autosave".

When you save "Proc0.ipf", close it without saving, or revert it, Igor deletes the corresponding "Proc0.ipf.autosave" file if it exists.

Recovering a File from a .autosave File

If Igor crashes or some other catastrophe occurs and Igor's indirect autosave routine executed before Igor was terminated, you will be left with the original file, "Proc0.ipf", and the .autosave file, "Proc0.ipf.autosave". If you attempt to open the original file, either manually or programmatically, Igor displays a dialog that looks like this:



Open Autosave File?

An autosave file exists for the file "Proc0.ipf" in the folder "hd:Work:Igor Pro Work:"

Usually this means that Igor crashed while you were editing the file and the autosave file may contain edits you want to keep.

If you are not certain what to do, back up the original and autosave files before proceeding.

NOTE: We recommend that you back up the original and autosave files before proceeding.

We recommend backing up in case you inadvertenly choose to open the wrong file.

If you click Cancel

The open operation is canceled and the original and autosave files are left unchanged.

If you click Open Original File

Igor opens the original file (e.g., "Proc0.ipf").

Igor then moves the autosave file (e.g., "Proc0.ipf.autosave") to the trash (Macintosh) or recycle bin (Windows). However, if the autosave file is open in Igor, it is not moved to the trash but may be overwritten by a subsequent autosave.

If you click Open AutoSave File

Igor renames the original file by appending a ".original" extension (e.g., "Proc0.ipf" is renamed as "Proc0.ipf.original").

Igor then renames the autosave file using the original file name (e.g., "Proc0.ipf.autosave" is renamed as "Proc0.ipf").

Igor then opens the autosave file which now has the original file name (e.g., "Proc0.ipf").

Igor then moves the original file, which now has the ".original" extension (e.g., "Proc0.ipf.original") to the trash (*Macintosh*) or recycle bin (*Windows*).

Indirect Autosave Mode Issues

Indirect mode can create confusing situations when files are shared by multiple users or by multiple instances of Igor launched by a single user.

For example, if a user is editing a shared procedure file with indirect autosave mode on, Igor creates a corresponding .autosave file. If another user directly or indirectly (e.g., via a #include statement) opens the original file, the existence of the .autosave file causes Igor to display the Open Autosave File dialog discussed in the preceding section. This will confuse the second user and potentially interfere with the first user's editing. A similar situation can occur with shared notebook and experiment files.

This kind of issue is mitigated by using a source code control system or by otherwise avoiding working directly on shared files or by turning autosave off.

Opening .autosave Files Explicitly

Normally you have no need to open an Igor .autosave file but you might want to do so to inspect it.

Igor opens procedure and notebook .autosave files for reading only. If you close a procedure or notebook file whose .autosave file you have also opened in Igor, Igor automatically closes the .autosave file before deleting it.

Igor does not allow you to open .autosave experiment files. If you want to open such a file, rename it without the .autosave extension before opening it in Igor.

Indirect Autosave and Unpacked Experiments

Indirect autosave of unpacked experiments is not supported.

If an unpacked experiment is open, autosaving of standalone procedure files and notebooks is still performed if enabled by the corresponding checkboxes in the Autosave pane of the Miscellaneous Settings dialog.

For background information on unpacked experiments, see **Saving as an Unpacked Experiment File** on page II-17.

Experimenting With Indirect Autosave Recovery

If you want to familiarize yourself with how indirect autosave works, you can use this function for experimentation. It writes files "Test Indirect Autosave.txt" and "Test Indirect Autosave.txt.autosave" to your Igor Pro User Files folder and then opens "Test Indirect Autosave.txt" as a notebook. Igor displays the "Open Autosave File?" dialog in which you can choose which file you want to open. The file not chosen is moved to the trash (*Macintosh*) or recycle bin (*Windows*).

```
Function TestOpenWithIndirectAutoSaveFilePresent()
   KillWindow/Z TestAutosaveNotebook

   String pathName = "IgorUserFiles"
   // Create original file
   String fileName = "Test Indirect Autosave.txt"
   Variable refNum
   Open/P=$pathName refNum as fileName
   fprintf refNum, "This is a test\r\n"
   Close refNum

   // Create the autosave file
   CopyFile /O /P=$pathName fileName as fileName + ".autosave"

   // Displays "Open Autosave File?" dialog
   OpenNotebook/P=$pathName/N=TestAutosaveNotebook fileName
   int err = GetRTError(1)       // Returns non-zero if user cancels
   Print err
End
```

Chapter

II-4

Windows

Windows

This section describes Igor's windows in general terms, the Windows menu, and window recreation macros.

Detailed information about each type of window can be found in these chapters:

Window Type	Chapter
Command window	Chapter II-2, **The Command Window** Chapter IV-1, **Working with Commands**
Procedure windows	Chapter III-13, **Procedure Windows**
Help windows	Chapter II-1, **Getting Help**
Graphs	Chapter II-13, **Graphs**
Tables	Chapter II-12, **Tables**
Layouts	Chapter II-18, **Page Layouts**
Notebooks	Chapter III-1, **Notebooks**
Control panels	Chapter III-14, **Controls and Control Panels**
Gizmo windows	Chapter II-17, **3D Graphics**
Camera windows	**NewCamera** on page V-678

The Command Window

When Igor first starts, the **command window** appears at the bottom of the screen.

Commands are automatically entered and executed in the command window's **command line** when you use the mouse to "point-and-click" your way through dialogs. You may optionally type the commands directly and press Return or Enter. Igor preserves a history of executed commands in the **history area**.

For more about the command window, see Chapter II-2, **The Command Window**, and Chapter IV-1, **Working with Commands**.

The Rest of the Windows

There are also a number of additional windows which are initially hidden:

- The main procedure window
- The Igor Help Browser
- Help windows for files in "Igor Pro Folder/Igor Help Files" and "Igor Pro User Files/Igor Help Files"

You can create additional windows for graphs, tables, page layouts, notebooks, control panels, Gizmos (3D plots), and auxiliary procedure windows, as well as more help windows.

The Target Window

Igor commands and menus operate on the **target window**. The target window is the top graph, table, page layout, notebook, control panel or XOP target window. The term "target" comes from the fact that these windows can be the target of command line operations such as ModifyGraph, ModifyTable and so on. The command window, procedure windows, help windows and dialogs can not be targets of command line operations and thus are not target windows.

Prior to version 4, Igor attempted to draw a special icon to indicate which window was the target. However, this special target icon is no longer drawn because of operating system conflicts.

The menu bar changes depending on the top window and the target window. For instance, if a graph is the target window the menu bar contains the Graph menu. However, you may type any command into the command line, including commands that do not apply to the target window. Igor will apply the command to the top window of the correct type.

Sometimes the top window isn't a target window, but it causes the menu bar to change. For example, if you activate a procedure window, the Procedure menu appears in the menu bar.

Window Names and Titles

Each graph, table, page layout, control panel, notebook, and Gizmo has a **title** and a **name**.

The title is what you see at the top of the window frame and in the Windows menu. Its purpose is to help you visually identify the window, and is usually descriptive of its contents or purpose.

The window *name* is not the same as the *title*. The purpose of the name is to allow you to refer to the window from a command, such as the DoWindow or AppendToGraph operations.

When you first create one of these windows, Igor gives it a name like Graph0, Table0, Layout0 or Panel0, and a title based on the name and window contents. You can change the window's title and name to something more descriptive using the Window Control dialog (Windows→Control submenu). Among other things, it renames and retitles the target window.

The Window Control dialog is also a good way to discover the name of the top window, since the window shows only the window title.

The command window, procedure windows, and help windows have *only* a title. The title is the name of the file in which they are stored. These windows do not have names because they can not be affected by command line operations.

Allowable Window Names

A window name is used for commands and therefore must follow the standard rules for naming Igor objects:

- The name must start with a letter.
- Additional characters can be alphanumeric or the underscore character.
- No other characters, including spaces, are allowed in standard Igor object names.
- No more than 255 bytes are allowed.
- The name must not conflict with other object names (you see a message if it does).

Prior to Igor Pro 8.00, window names were limited to 31 bytes. If you use long window names, your experiments will require Igor Pro 8.00 or later.

For more information, see **Object Names** on page III-501.

The Open File Submenu

The File menu contains the Open File submenu for opening an existing file as a notebook, Igor help window, or procedure window.

When you choose an item from the submenu, the Open File dialog appears for you to select a file.

The Windows Menu

You can use the Windows menu for making new windows, and for showing, arranging and closing (either hiding or "killing") windows. You can also execute "window recreation macros" that recreate windows that have been killed and "style macros" that modify an existing window's appearance.

Making a New Window

You can use the various items in the Windows menu and Windows→New submenu to create new windows. Most of these items invoke dialogs which produce commands that Igor executes to create the windows.

You can also create windows by typing these commands yourself directly in the command line. For example,

```
Display yData vs xData
```

creates a graph of the wave named yData on the Y axis, versus xData on the X axis.

You can create a new window by selecting the name of a window recreation macro from the Windows menu. See **Window Macros Submenus** on page II-48.

You can also create a window using the File→Open File submenu.

Activating Windows

To activate a window, click it, or choose an item from Windows menu or its submenus.

The Recent Windows submenu shows windows recently activated. This information is saved when you save an experiment to disk and restored when you later reopen the experiment.

By default, just the window’s title is displayed in the Windows menu. You can choose to display the title or the name for target windows using the Windows Menu Shows pop-up menu in the Miscellaneous section of the Miscellaneous Settings dialog.

Showing and Hiding Windows

All types of Igor windows can be hidden.

To hide a window, press Shift and choose Windows→Hide or use the keyboard shortcut Command-Shift-W (*Macintosh*) or Ctrl+Shift+W (*Windows*). You can also hide a window by pressing Shift and clicking the close button.

You can hide multiple windows at once using the Windows→Hide submenu. For example, to hide all graphs, choose Windows→Hide→All Graphs. If you press Shift while clicking the Windows menu, the sense of the menu items changes. For example, Hide→All Graphs changes to Hide→All Except Graphs.

The command window is not included in mass hides of any kind. If you want to hide it you must do so manually.

Similarly, you can show multiple windows at once using the Windows→Show submenu. For example, to show all graphs, choose Windows→Show→All Graphs. If you press Shift while clicking the Windows menu, the sense of the menu items changes. For example, Show→All Graphs changes to Show→All Except Graphs.

The Show All Except menu items do not show procedure windows and help files because there are so many of them that it would be counterproductive.

The Windows→Show→Recently Hidden Windows item shows windows recently hidden by a mass hide operation, such as Hide→All Graphs, or windows recently hidden manually (one-at-a-time using the close button or Command-Shift-W or Ctrl+Shift+W). In the case of manually hidden windows, “recently hidden” means within the last 30 seconds.

XOP windows do participate in Hide All XOP Windows and Show All XOP Windows only if XOP programmers specifically support these features.

Closing a Window

You can close a window by either choosing the Windows→Close menu item or by clicking in the window’s close button. Depending on the top window’s type, this will either kill or hide the window, possibly after a dialog asking for confirmation.

Killing Versus Hiding

“Killing” a window means the window is removed from the experiment. The memory used by the window is released and available for other purposes. The window’s title is removed from the Windows menu.

Killing a window that represents a file on disk does not delete the file. You can also kill a window with a **KillWindow** command.

“Hiding” a window simply means the window is made invisible, but is still part of the experiment and uses the same amount of memory. It can be made visible again by choosing its title from the Windows menu.

The command window and the built-in procedure window can be hidden but not killed. All other built-in windows can be hidden or killed.

When you create a window from a procedure, you can control what happens when the user clicks the close button using the `/K=<num>` flag in the command that creates the window.

You can hide a window programmatically using the `DoWindow/HIDE=1` operation.

Saving the Window Contents

Notebooks and procedure windows can be saved either in their own file, or in a packed experiment file with everything else. You can tell which is the case by choosing Notebook→Info or Procedure→Info. When you kill a notebook or a procedure window that contains unsaved information, a dialog will allow you to save it before killing the window.

Graph, table, control panel, page layout, and Gizmo windows are not saved as separate files, and are lost when you kill them unless you save a **window recreation macro** which you can execute to later recreate the window. Killing these windows and saving them as window recreation macros (stored in the built-in procedure window) frees up memory and reduces window clutter without losing any information. You can think of window recreation macros as “freeze-dried windows”.

Close Window Dialogs

When you close a graph, table, layout or control panel, or Gizmo window, Igor presents a Close dialog.

If you click the Save button Igor creates a window recreation macro in the main procedure window. It sets the macro’s subtype to Graph, Table, Layout, Panel, or Gizmo, so the name of the macro appears in the appropriate Macros submenu of the Windows menu. You can recreate the window using this menu.

If you don’t plan to use the window again, you should click the No Save button and no window recreation macro will be created.

If you have previously created a recreation macro for the window then the dialog will have a Replace button instead of a Save button. Clicking Replace replaces the old window recreation macro with a new one. If you know that you won’t need to recreate the window, you can delete the macro (see **Saving a Window as a Recreation Macro** on page II-47).

When you close a notebook or procedure window (other than the built-in procedure window), Igor presents a “hide or kill dialog”.

To hide a window, press Shift while clicking the close button.

To kill a graph, table, layout, control panel, or Gizmo window without the Close dialog, press Option (*Macintosh*) or Alt (*Windows*) while clicking the close button.

If you create a window programmatically using the Display, Edit, NewLayout, NewPanel, NewNotebook, or NewGizmo operation, you can modify the behavior of the close button using the /K flag.

Saving a Window as a Recreation Macro

When you close a window that can be saved as a recreation macro, Igor offers to create one by displaying the Close Window dialog. Igor stores the window recreation macro in the main procedure window of the current experiment. The macro uses much less memory than the window, and reduces window clutter. You can invoke the window recreation macro later to recreate the window. You can also create or update a window recreation macro using the Window Control dialog.

The window recreation macro contains all the necessary commands to reconstruct the window provided the underlying data is still present. For instance, a graph recreation macro contains commands to append waves to the graph, but does not contain any wave data. Similarly, a page layout recreation macro does not contain graphs or tables or the commands to create them. The macros refer to waves, graphs and tables in the current experiment by name.

Here is how you would use recreation macros to keep a graph handy, but out of your way:

The window recreation macro is evaluated in the context of the root data folder. This detail is of consequence only to programmers. See **Data Folders and Commands** on page II-111 for more information.

You can create or replace a window recreation macro without killing the window using **The Window Control Dialog** described on page II-49. The most common reason to replace a window recreation macro is to keep the macro consistent with the window that it creates.

When Igor displays the Close Window dialog, the proposed name of the window recreation macro is the same as the name of the window. You can save the window recreation macro under a different name, if you want, by entering the new name in the dialog. If you do this, Igor creates a new macro and leaves the original macro intact. You can run the new macro to create a new version of the window or you can run the old macro to recreate the old version. This way you can save several versions of a window, while displaying only the most recent one.

Window recreation macros stay in an experiment's procedure window indefinitely. If you know that you won't need to recreate a window for which a window recreation macro exists, you can delete the macro.

To locate a window recreation macro quickly:

- Activate any procedure window, press Option (*Macintosh*) or Alt (*Windows*), and choose the window recreation macro name from the appropriate macro submenu in the Windows menu.

To delete the macro, if you're sure you won't want it again, simply select all the text from the Macro declaration line to the End line. Press Delete to remove the selected text.

See **Saving and Recreating Graphs** on page II-350 for details specific to graphs.

Window Macros Submenus

The Windows menu has submenus containing graph, table, page layout, control panel, and Gizmo recreation macros. These menus also include graph, table, and page layout style macros.

Window recreation macros are created by the Close Window and Window Control dialogs, and by the DoWindow/R command. Style macros are created by the Window Control dialog and the DoWindow/R/S command.

Igor places macros into the appropriate macro submenu by examining the macro's subtype. The subtypes are Graph, Table, Layout, Panel, Gizmo, GraphStyle, TableStyle and LayoutStyle. See **Procedure Subtypes** on page IV-204 for details.

When you choose the name of a recreation macro from a macro submenu, the macro runs and recreates the window. Choosing a style macro runs the macro which changes the target window's appearance (its "style").

However, if a procedure window is the top window and you press Option (*Macintosh*) or Alt (*Windows*) and then choose the name of any macro, Igor displays that macro but does not execute it.

The Name of a Recreated Window

When you run a window recreation macro, Igor recreates the window with the same name as the macro that created it unless there is already a window by that name. In this case, Igor adds an underscore followed by a digit (e.g. _1) to the name of the newly created window to distinguish it from the preexisting window.

Changing a Window's Style From a Macro

When you run a style macro by invoking it from the Windows menu, from the command line or from another macro, Igor applies the commands in the macro to the top window. Usually these commands change the appearance of the window. For example, a graph style macro may change the color of graph traces or the axis tick marks.

Style macros are used most effectively with graph windows. For more information, see **Saving and Recreating Graphs** on page II-350 and **Graph Style Macros** on page II-350.

The Window Control Dialog

Choosing Control→Window Control displays the Window Control dialog which you can use to change the top window's title and name, and create or update its recreation and style macros. You can access this dialog quickly by pressing Command-Y (*Macintosh*) or Ctrl+Y (*Windows*).

You can also change the window's name. The window name is used to address the window from command line operations such as MoveWindow and also appears in the macro submenus of the Windows menu.

For more about names and titles, see **Window Names and Titles** on page II-45. Also see **Saving a Window as a Recreation Macro** on page II-47 for a discussion of window recreation macros, and see **Graph Style Macros** on page II-350 for details on style macros.

The Window Browser

The Window Browser allows you to find and manage windows of interest. Choose Windows→Window Browser to activate it.

The window list on the right side lists windows that meet the current filtering criteria. You can filter windows by name, by type, by visibility (visible or hidden), and based on whether they display a specific wave. Clicking the funnel icon at the bottom of the list toggles the display of advanced filtering controls. Clicking the gear icon displays the options menu in which you can set sorting and display options.

If a single window is selected in the window list, the controls on the left display information about the window. The buttons at the top of the Selection area control the types of information that are displayed. You can choose to display the following types of information:

- Window Information

  Includes the window name, window title, and window note. See **Window Names and Titles** on page II-45 for a discussion of the distinction between names and titles.

- Waves in Window

  Displays a list of waves used by the window. Right-click a wave icon to edit the wave in a table, view it in the Data Browser, or set the current data folder to the wave's data folder. You can also toggle display of full paths by right clicking anywhere in the list.

  You can select one or more waves in the list and drag them into an existing graph or table window to append them to the target window. See **Appending Traces by Drag and Drop** on page II-280 for details.

- Layout Objects

  Displays information about the pages in the selected page layout window and the objects on each page.

If multiple windows are selected, the Window Information control displays a list of window names.

The buttons above the window list allow you to show, hide, close the selected windows and to bring them to the front or send them to the back.

You can append selected graph, table, and Gizmo windows to a page layout by dragging them from the window list into a layout or by right clicking in the window list and choosing the appropriate item.

You can add menu items to the contextual menu in the window list. See **WindowBrowserWindowsPopup Menu** on page IV-140 for details.

Arranging Windows

You can tile or stack windows by choosing the appropriate items from the Control submenu in the Windows menu.

You can customize the behavior of the Tile and Stack items using the Tile or Stack Windows dialog.

You can also move windows around using the MoveWindow, StackWindows, and TileWindows commands.

The Tile or Stack Windows Dialog

The Tile or Stack Windows dialog is useful for tiling a few windows or even for setting the size and position of a single window.

Select individual windows from the Windows to Arrange list, and entire classes of windows with the checkboxes.

If you want subsequent selections of the Tile (or Stack) menu item to stack the same types of windows with the same rows, columns, grout, tiling area, etc., check the "Capture as pref" checkbox. Windows selected in the Windows to Arrange menu aren't remembered by the preferences; only the window type checkboxes are remembered. There are separate settings and preferences for Stack and for Tile.

Although the TileWindows and StackWindows operations can tile and stack panels, panels don't show up here because they don't resize very well.

The Window Tiling Area subdialog specifies the area where tiling *and stacking* take place.

You can specify the tiling area in one of four ways:

- By entering screen positions in units of points
- By dragging the pictorial representation of the tiling area
- By clicking the "Use default area" button to use the default tiling area
- By positioning any non-dialog window before you enter the dialog, and clicking the "Use top window" button

Window Position and Size Management

There are four items in the Control submenu of the Windows menu that help you manage the position and size of windows.

Move to Preferred Position

Moves the active window to the position and size determined by preferences. For each type of window, you can set the preferred position and size using the Capture Prefs dialog (e.g., Capture Graph Prefs for graphs).

Move to Full Size Position

Moves and sizes the active window to display as much of the content as practical. On Macintosh, this is the same as clicking the zoom button. On Windows, the size is limited to the size of the frame window.

Retrieve Window

Moves the active window and sizes it if necessary so that all of the window is visible.

Retrieve All Windows

Moves all windows and sizes them if necessary so that all of each window is within the screen on Macintosh or within the frame on Windows. This is often useful when you open an experiment that was created on a system with a larger screen or Windows frame than yours.

Send to Back — Bring to Front

The Send to Back item in the Windows menu sends the top window to the bottom of the desktop, behind all other windows. This function can also be accessed by pressing Command-' (*Macintosh*) or Ctrl+E (*Windows*). After sending a window behind, you can bring it to the front by choosing Bring to Front or by pressing Command-Shift-' (*Macintosh*) or Ctrl+Shift+E (*Windows*). You can also press these keys repeatedly to cycle through all windows.

You can also send a window to the back with the `DoWindow/B` command and bring it to the front with the `DoWindow/F` command.

Text Windows

Igor Pro displays text in procedure, notebook, and Igor help windows as well as in the command and history areas of the command window. This section discusses behavior common to all of these windows.

Executing Commands

You can execute commands selected in a notebook, procedure or help window by pressing Control-Enter or Control-Return. You can also execute selected commands by Control-clicking (*Macintosh*) or right-clicking (*Windows*) and choosing Execute Selection.

For more on this, see **Notebooks as Worksheets** on page III-4.

Text Window Navigation

The term "keyboard navigation" refers to selection and scrolling actions that Igor performs in response to the arrow keys and to the Home, End, Page Up, and Page Down keys. Macintosh and Windows have different conventions for these actions in windows containing text. You can use either Macintosh or Windows conventions on either platform.

By default, Igor uses Macintosh conventions on Macintosh and Windows conventions on Windows. You can change this using the Keyboard Navigation menu in the Misc Settings section of the Miscellaneous Settings dialog. If you use Macintosh conventions on Windows, use Ctrl in place of the Macintosh Command key. If you use Windows conventions on Macintosh, use Command in place of the Windows Ctrl key.

Macintosh Text Window Navigation

Key	No Modifier	Option	Command
Left Arrow	Move selection left one character	Move selection left one word *	Move selection to start of line
Right Arrow	Move selection right one character	Move selection right one word *	Move selection to end of line
Up Arrow	Move selection up one line	Move selection up one paragraph *	Move selection up one screen
Down Arrow	Move selection down one line	Move selection down one paragraph *	Move selection down one screen
Home	Scroll to start of document	Scroll to start of document	Not used
End	Scroll to end of document	Scroll to end of document	Not used
Page Up	Scroll up one screen	Scroll up one screen	Not used
Page Down	Scroll down one screen	Scroll down one screen	Not used

* If you choose Macintosh keyboard navigation on Windows, these Alt-Arrow key events are not supported. Alt-Left-Arrow is the keyboard shortcut for Edit→Go Back and Alt-Right-Arrow is the keyboard

shortcut for Edit→Go Forward. Alt-Up-Arrow nudges the selection up and Alt-Down-Arrow nudges the selection down.

Windows Text Window Navigation

Key	No Modifier	Ctrl
Left Arrow	Move selection left one character	Move selection left one word
Right Arrow	Move selection right one character	Move selection right one word
Up Arrow	Move selection up one line	Move selection up one paragraph
Down Arrow	Move selection down one line	Move selection down one paragraph
Home	Move selection to start of line	Move selection to start of document
End	Move selection to end of line	Move selection to end of document
Page Up	Scroll up 1 screen	Scroll up 1 screen
Page Down	Scroll down 1 screen	Scroll down 1 screen

Text Window Go Back and Go Forward

Certain actions add entries to a go-back stack that Igor maintains in memory. These actions include opening a text window, doing a find, clicking a help link, right-clicking and choosing the Help For and Go To menu items, and using the navigation bar.

You can return to locations in text windows that you recently visited using the Go Back and Go Forward buttons. These appear in the navigation bar of procedure and help windows and in the status area of notebook windows. The Go Back button looks like a "less-than" symbol and the Go Forward button looks like a "greater-than" symbol. You can also use the Edit menu Go Back and Go Forward items or their keyboard equivalents:

	Macintosh	Windows
Go Back	Cmd-Option-Left-Arrow	Alt+Left-Arrow
Go Forward	Cmd-Option-Right-Arrow	Alt+Right-Arrow

If your mouse has Go Back and Go Forward buttons, you can use them also.

Between the Go Back and Go Forward buttons is a button which when clicked displays the Go Back pop-up menu. You can return to a specific recently-visited location by choosing an item from a submenu in the Go Back pop-up menu.

Igor can keep the go back locations for all text windows in one stack or it can keep separate stacks for procedure windows, notebook windows and help windows. You control this using the "Separate stacks for procedure, notebook, help windows" menu item in the Go Back pop-up menu. When that item is checked, Igor maintains separate stacks. When it is unchecked, Igor maintains one stack for all three types of windows. Which you choose is a matter of taste. The advantage of keeping multiple stacks is that it most of the time you are interested in windows of the same type. The advantage of keeping just one stack is that it allows you to go back to a procedure file after right-clicking for help for an Igor operation or function.

Igor keeps a maximum of 25 locations in each stack and discards older locations when necessary.

Finding Text in the Active Window

You can find text active windowby choosing Edit→Find or by pressing Command-F (*Macintosh*) or Ctrl+F (*Windows*). This displays the Find bar.

On Macintosh, you can search for the next occurrence of a string by selecting the string and choosing Edit→Use Selection For Find or pressing Command-E to enter the selected text as the find string and Command-G to find it.

On Windows, you can search for the next occurrence of a string by selecting the string and choosing Edit→Find Selection or pressing Ctrl+H to enter the selected text as the find string and find it.

Find and Replace

You can find and replace text by choosing Edit→Replace Text or by pressing Command-R (*Macintosh*) or Ctrl+R (*Windows*). This displays the Find and Replace bar.

Here is another method for finding and replacing text:

1. Move the selection to the top of the active window.
2. Use Edit→Find to find the first instance of the target string.
3. Manually change the first instance, then copy the new text to the Clipboard.
4. Press Command-G (*Macintosh*) or Ctrl+G (*Windows*) to find the next occurrence.
5. Press Command-V (*Macintosh*) or Ctrl+V (*Windows*) to paste.
6. Repeat steps 4 and 5 until done.

Finding Text in Multiple Windows

You can find text in multiple windows by choosing Edit→Find Text in Multiple Windows. This displays the Find Text in Multiple Windows window which allows you to search all help windows, all procedure windows, and all notebooks.

You can also use the Igor Help Browser to search in multiple files, including files that are not open in your current experiment. See **The Igor Help Browser** on page II-2 for details.

Text Magnification

You can magnify the text in any window to make it bigger or smaller to suit your taste.

In help windows, procedure windows, plain text notebooks, and formatted text notebooks, you can use the magnifying glass icon in the bottom-left corner of the window. You can also use the Magnification submenu in the contextual menu for the window. To display the contextual menu, Control-click (*Macintosh*) or right-click (*Windows*) in the body of the window.

You can also set the magnification for the command line, history area, and the debugger. These areas do not display the magnifying glass icon so you must use the contextual menu.

You may notice some anomalies when you use text magnification. For example, in a formatted text notebook, text may wrap at a different point in the paragraph and may change in relation to tab stops. This happens because fonts are not available in fractional sizes and because the actual width of text does not scale linearly with font size.

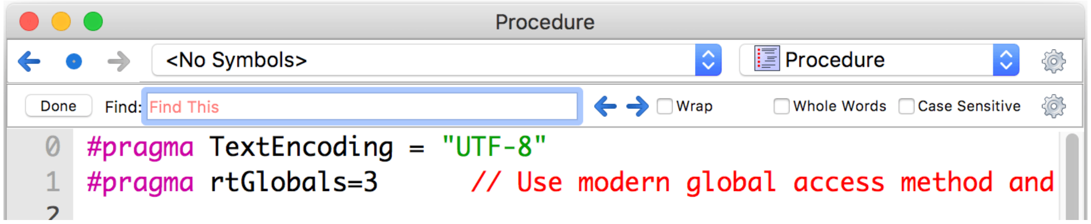
You can set the default magnification for each type of text area by choosing a magnification from the Magnification popup menu and then choosing Set As Default from the same popup menu. Any text areas whose magnification is set to Default will use the newly specified default magnification. For example, if you want text in all help files to appear larger, open any help file, choose a larger magnification, 125% for example, and then choose Set As Default For Help Files. All help files whose current magnification is set to Default will be updated to use the new default.

The default magnification for the command line and history area controls the magnification that will be used the next time you launch Igor Pro.

The magnification setting is saved in formatted notebooks and help files only. If you change the magnification setting for one of these files and then save and close the file, the magnification setting will be restored when you reopen the file. For all other types of text areas, including procedure windows and plain text notebooks, the magnification setting is not stored in the file. If you close and reopen such a file, it will reopen using the default magnification for that type of text area.

The Find Bar

The find bar is available in help, procedure and notebook windows, and other places where you might need to search for text. To display it, choose Edit→Find or press Command-F (Macintosh) or Ctrl+F (Windows). The Find Bar appears at the top of the host window:



```
#pragma TextEncoding = "UTF-8"
#pragma rtGlobals=3       // Use modern global access method and
```

Enter text in the Find edit box and press the right arrow icon to find the next occurrence of the text. Press the left arrow icon to find the preceding occurrence.

You can also find the next occurrence by pressing Command-G (Macintosh) or Ctrl+G (Windows). You can also find the preceding occurrence by pressing Command-Shift-G (Macintosh) or Ctrl+Shift+G (Windows).

You can set the text to be found by selecting it in the active window and pressing Command-E (Macintosh) or Ctrl+H (Windows).

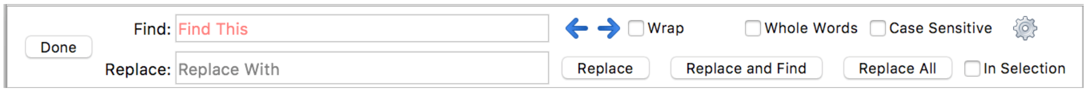
By default, when you use these keyboard equivalents, if the find bar is not showing, Igor displays it. If you prefer that Igor not display the find bar when you use these equivalents, click the gear icon and check the Show Find Bar on Edit→Find Only checkbox.

You can set the size of the find bar using the gear icon.

To hide the find bar, click the Done button or press the Esc key.

Search and Replace

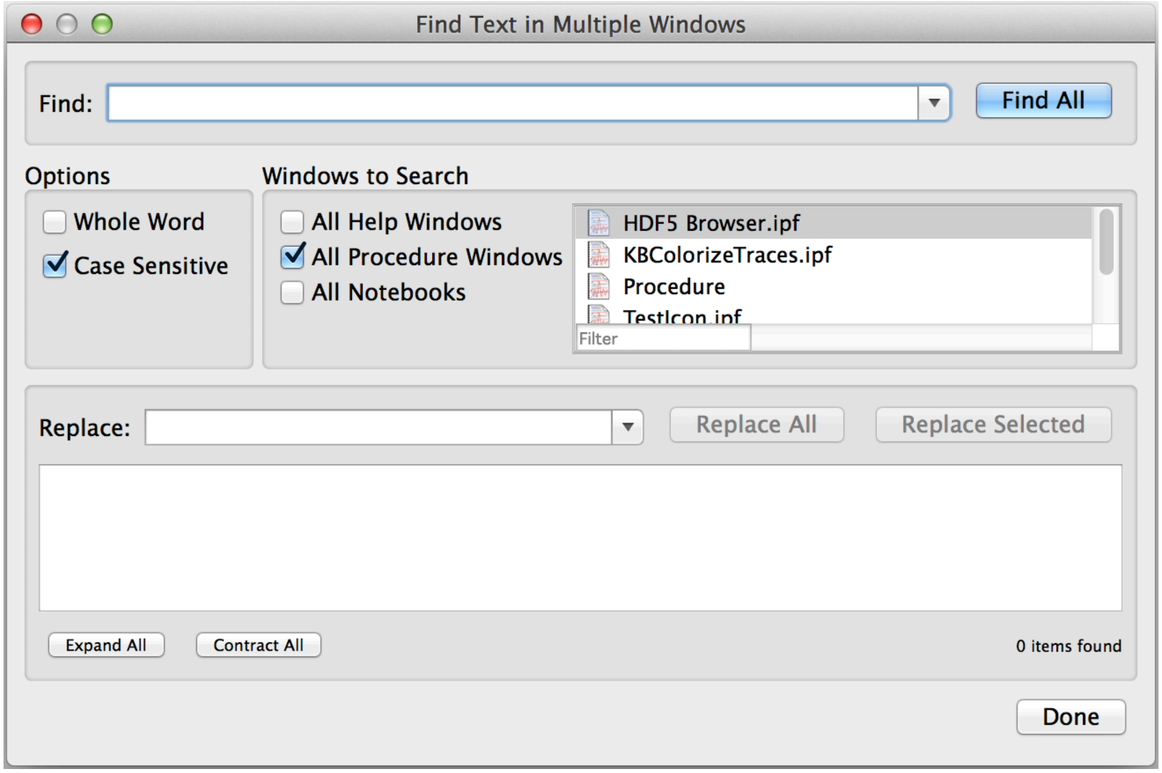
If the window you are working on contains editable text, you can choose Edit→Replace or press Command-R (Macintosh) or Ctrl+R (Windows) to display the replace text portion of the find bar:



While replacing text is undoable, the potential for unintended and wide-ranging consequences is such that we recommend saving the file before doing a mass replace using Replace All so you can revert-to-saved if necessary.

The Find Text in Multiple Windows Dialog

You can perform a Find on multiple help, procedure and notebook windows at one time by choosing Edit→Find in Multiple Windows or by pressing Command-Shift-F (Macintosh) or Ctrl+Shift+F (Windows). This invokes the Find Text in Multiple Windows dialog:



☐ Whole Word
☑ Case Sensitive

☐ All Help Windows
☑ All Procedure Windows
☐ All Notebooks

HDF5 Browser.ipf
KBColorizeTraces.ipf
Procedure
TestIcon.ipf

0 items found

Enter text to be found in the Find box. Use the checkboxes to select the types of windows to search. To narrow the list of files to search, you can enter a filter string in the Filter box at the bottom of the window list. The list then shows only list items whose names contain the filter string.

When you click the Find All button, Igor commences searching all the text in the listed windows. During the search, Igor displays a small progress dialog showing how far through the window list the search has gotten. You can stop the search by clicking the Cancel button.

For each window in which the text is found, an entry appears in the panel at the bottom of the dialog. A number in parentheses indicates how many instances of the search text were found in that window.

Each of these window entries can be opened using the disclosure control at the left end of the item. Then a snippet of text around the search string is shown, with the search string highlighted. The number at the left end of the found text snippet is the line number within the searched window.

Double-clicking a text snippet item takes you to the found text in the window containing the text.

Search and Replace in Multiple Windows

After a search is finished, you can use the found text items to replace all instances of the text that were found in editable windows. Enter the replacement text in the Replace edit box.

Clicking the Replace All button replaces all the found instances with the replacement text.

NOTE: Replace All in multiple windows cannot be undone. Use it with great care.

If you make a mistake, choose File→Revert Experiment, File→Revert Notebook, or File→Procedure and make sure you have reverted all affected windows. This is the only way to recover.

You can limit the replacement by selecting a subset of the found text items and clicking the Replace Selected button. Click a found text item to select it. Shift-click another item to select all items between that item and the one previously clicked. To select non-contiguous items, Command-click (Macintosh) or Ctrl-click (Windows). You can also click and drag over multiple items. It is still a good idea to save before replacing.

Home Versus Shared Text Files

A text file that is stored in a packed experiment file, or in the experiment folder for an unpacked experiment, is a "home" text file. Otherwise it is a "shared" text file.

Home text files are typically intended for use by their owning experiment only. Shared text files are typically intended for use by by multiple experiments.

When you create a text file in a packed experiment, it saved by default in the packed experiment file and is a home text file. It becomes a shared text file only if you explicitly save it to a standalone file.

When you create a text file in an unpacked experiment, it saved by default in the experiment folder and is a home text file. It becomes a shared text file only if you explicitly save it to a standalone file outside of its experiment folder.

When you save a packed experiment as unpacked, home text files are stored in the experiment folder.

When you save an unpacked experiment as packed, home text files are saved in the packed experiment file.

You can use the File Information dialog, which you access by choosing Procedure→Info or Notebook→Info, to determine if a text file is shared. For shared text files, the dialog says "This file is stored separate from the experiment file" for packed and unpacked experiments. For home text, the dialog says "This file is stored in packed form in the experiment file" for packed experiments and "This file is in the experiment folder" for unpacked experiments.

You can convert a shared text file to a home text file by adopting it. See **Adopting Notebook and Procedure Files** on page II-25 for details.

Using an External Text Editor

Igor supports the use of external editors for editing Igor procedure files and plain text notebooks. This allows you to use your favorite text editor rather than Igor for editing plain text files if you prefer. This is mostly intended for use by advanced programmers who are accustomed to using external editors in other programming environments.

Prior to Igor7, Igor kept plain text files open as long as the corresponding procedure or notebook window was open in Igor. This interfered with the use of external editors.

Now a procedure or plain text notebook window opens its file just long enough to read the text into memory and then close it. If you modify the text in Igor and do a save, Igor reopens the file, writes the modified text to it, and closes the file.

If you modify the file using an external text editor instead of Igor, Igor notices the change, reopens the file, reloads the modified text into memory, and closes the file.

Supporting external editors creates issues that Igor must deal with:

- If you modify the file in an external editor, the text is now out of sync with the text in Igor's window. In this case, Igor notices that the file has been changed and either reloads the text into memory or notifies you, depending on your external editor miscellaneous settings.
- If you save modifications to the file in an external editor and also edit the document in Igor, the text in Igor is in conflict with the external file. In this case Igor informs you of the conflict and lets you choose which version to keep.
- If you move the disk file to new location, delete the file or rename it, Igor's information about the

file is no longer valid. In this case Igor notices that the file has disappeared and gives you options for dealing with it.

The following sections explain how Igor deals with these issues in more detail.

A Plain Text File Has Been Modified Externally

Once per second Igor checks every procedure and plain text notebook window to see if its file has been modified externally. If so then it either automatically reloads the file into memory or notifies you about the external modification and allows you to reload the file at your leisure. The factory default behavior is to automatically reload externally modified files and to check for modified files only when Igor is the active application.

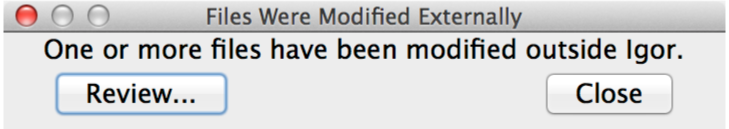
You can control Igor's behavior using the Miscellaneous Settings Dialog. Choose Misc→Miscellaneous Settings, click Text Editing in the lefthand pane, and click the External Editor tab.

If you select Ask Before Reloading, two things happen when the file is modified externally:

- Igor displays a **Reload** button in the status area at the bottom of the document window:

  The Reload button will be visible only if the window is visible.

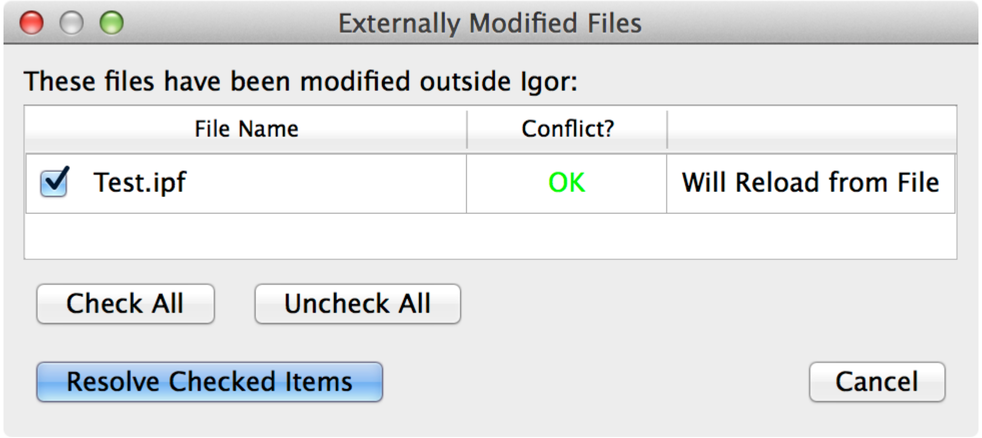
- Igor displays a small floating notification window:

  
  Files Were Modified Externally

  One or more files have been modified outside Igor.

Clicking the **Reload** button causes Igor to reload the file's text.

Clicking the **Review** button in the notification window displays a dialog in which you can review all the files that are currently modified externally:


Externally Modified Files

These files have been modified outside Igor:

File Name	Conflict?	
☑ Test.ipf	OK	Will Reload from File

Click **Resolve Checked Items** to reload the file into memory.

A Plain Text File Has Been Modified Externally and In Igor

When a file has been modified both in an external editor and by editing in an Igor window, we say it is "in conflict". Igor never automatically reloads a file that is in conflict.

When a file is in conflict, Igor displays a **Resolve Conflict** button in the status area in the Igor document window. Clicking that button brings up a dialog giving four choices to deal with the conflict:

- Reload External Changes

  Loads the contents of the file into memory, discarding changes made in Igor.

- Adopt Document

  Severs the tie between the document and the file, keeping the Igor changes and discarding the changes to the file. To learn more about adopting a file, see **Adopting Notebook and Procedure Files** on page II-25.

- Close the Window

  Closes the procedure or notebook window in Igor, discarding changes made in Igor.

- Save Igor Changes

  Writes changes made in Igor to the file. This resolves the conflict in Igor and creates a conflict in the external editor. Different external editors have different approaches to such conflicts.

Files that are in conflict also cause the notification window to appear. Clicking the Review button in the notification window displays a dialog which gives you the same options for resolving the conflict.

Editing a File with External Modifications

If a file has been modifed externally and not re-loaded in Igor, typing in the Igor window creates a conflict even if there wasn't one previously. This may be undesirable, so when Igor detects this situation, it displays a dialog, similar to the Resolve Conflict dialog, asking you what should be done. The choices are:

- Allow Typing

This choice tells Igor that you want to be allowed to modify the document in Igor. You are putting off resolving the conflict for later. You will have to decide at some point to resolve the conflict in one of the ways described above.

- Reload External Changes

The external modifications will be loaded into the Igor window. Modifications in Igor's copy are discarded.

- Adopt Document

Severs the tie between the document and the file. To learn more about adopting a file, see Adopting Notebook and Procedure Files.

You also have the option of clicking the Cancel button. In that case, the situation is not resolved, and any further attempt to type in the window will cause the dialog to be displayed again.

A Plain Text File Is Missing

If the file associated with an Igor plain text document window is moved, renamed or deleted, Igor will note that the file is missing. In this case, Igor displays a **File Missing** button in the status area in the Igor document window. Clicking that button displays a File Missing dialog giving four choices to deal with the conflict:

- Adopt Document

  Severs the tie between the document and the file. To learn more about adopting a file, see Adopting Notebook and Procedure Files.

- Close the Window

  Closes the procedure or notebook window in Igor.

- Find Missing File

  Displays an Open File dialog allowing you to locate the moved or renamed file.

- Save As

  Displays a Save As dialog allowing you to save the document to a new file.

Missing or Modified Text Files Dialog

A plain text file (procedure file or plain text notebook) may be missing because you deleted it, renamed it, or moved it, or it was deleted, renamed or moved by a program.

A plain text file may have been modified by another program if you edited it in an external editor, for example.

If you try to do a save while a plain text file is missing or modified, Igor displays the Missing or Modified file dialog which asks if you want to cancel the save or continue it. Usually you should cancel the save and use the Files Were Modified Externally window, which appears in the top/right corner of the screen, to review and address the situation.

If you elect to continue the save, it is likely that an error will occur.

To prevent an unattended procedure from hanging, the check for missing or modified files is not done if the save is invoked from a procedure.

Window Shortcuts

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To close a window	Click the close button or press Command-W.	Click the close button or press Ctrl+W.
To kill a window with no dialog	Press Option and click the close button or press Command-Option-W.	Press Alt and click the close button or press Ctrl+Alt+W.
To hide a window	Press Shift and click the close button or press Command-Shift-W.	Press Shift and click the close button or press Ctrl+Shift+W.
To invoke the Window Control dialog	Press Command-Y.	Press Ctrl+Y.
To send the top window behind all others	Press Control-Command-E.	Press Ctrl+E.
To bring the bottom window on top of all others	Press Shift-Control-Command-E.	Press Ctrl+Shift+E.
To send all windows that are completely visible behind all others	Press Command-Option-E.	Press Ctrl+Alt+E.
To activate a recently activated window	Press Command, click the main menu bar, and select from the Recent menu.	Press Ctrl, click the main menu bar, and select from the Recent menu.
To move a window to its preferred size and position	Click the zoom button.	Press Alt and click the maximize button.
To move a window to its full-size position	Press Shift-Option and click the zoom button.	Press Shift +Alt and click the maximize button.
To activate the command window	Press Command-J.	Press Ctrl+J.
To clear the command buffer	Press Command-K.	Press Ctrl+K.
To open the built-in procedure window	Press Command-M.	Press Ctrl+M.
To cycle through all procedure windows	Press Command-Option-M.	Press Ctrl+Alt+M.
To open the Igor help browser	Choose Help→Igor Help Browser.	Press F1.
To find a phrase in a text window	Press Command-F.	Press Ctrl+F.
To find the same phrase again	Press Command-G. Press Command-Shift-G to search backward.	Press Ctrl+G. Press Ctrl+Shift+G to search backward.
To find the selected phrase	Press Command-E and Command-G. Press Command-E and Command-Shift-G to search backward. This shortcut can be changed through the Miscellaneous Settings dialog.	Press Ctrl+H. Press Ctrl+Shift+H to search backward.

Chapter II-5

Waves

Overview

We use the term “wave” to describe the Igor object that contains an array of numbers. Wave is short for “waveform”. The main purpose of Igor is to store, analyze, transform, and display waves.

Chapter I-1, **Introduction to Igor Pro**, presents some fundamental ideas about waves. Chapter I-2, **Guided Tour of Igor Pro**, is designed to make you comfortable with these ideas. In this chapter, we assume that you have been introduced to them.

This chapter focuses on one-dimensional numeric waves. Waves can have up to four dimensions and can store text data. Multidimensional waves are covered in Chapter II-6, **Multidimensional Waves**. Text waves are discussed in this chapter.

The primary tools for dealing with waves are Igor’s built-in operations and functions and its waveform assignment capability. The built-in operations and functions are described in detail in Chapter V-1, **Igor Reference**.

This chapter covers:

- waves in general
- operations for making, killing and managing waves
- setting and examining wave properties
- waveform assignment

and other topics.

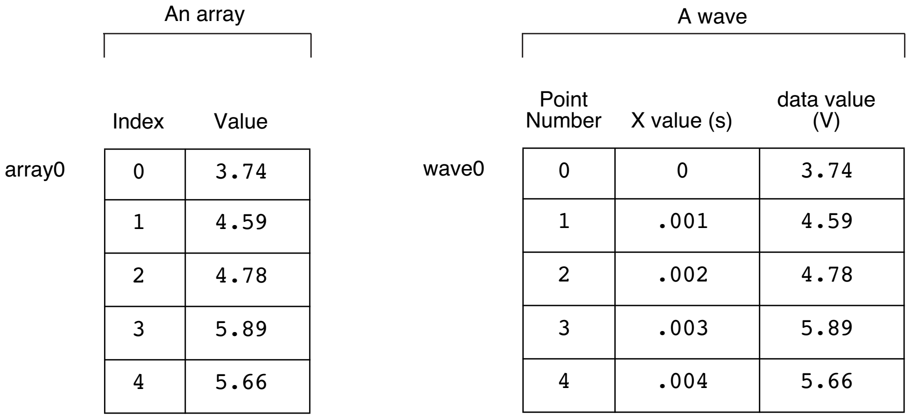
Waveform Model of Data

A wave consists of a number of components and properties. The most important are:

- the wave name
- the X scaling property
- X units
- an array of data values
- data units

The waveform model of data is based on the premise that there is a straight-line mapping from a point number index to an X value or, stated another way, that the data is uniformly spaced in the X dimension. This is the case for data acquired from many types of scientific and engineering instruments and for mathematically synthesized data. If your data is *not* uniformly spaced, you can use two waves to form an XY pair. See **XY Model of Data** on page II-63.

A wave is similar to an array in a standard programming language like FORTRAN or C.



Index	Value
0	3.74
1	4.59
2	4.78
3	5.89
4	5.66

Point Number	X value (s)	data value (V)
0	0	3.74
1	.001	4.59
2	.002	4.78
3	.003	5.89
4	.004	5.66

An array in a standard language has a **name** (array0 in this case) and a number of **values**. We can reference a particular value using an **index**.

A wave also has a **name** (wave0 in this case) and **data values**. It differs from the array in that it has *two* indices. The first is called the **point number** and is identical to an array index or row number. The second is called the **X value** and is in the natural X units of the data (e.g., seconds, meters). Like point numbers, X values are not stored in memory but rather are *computed*.

The X value is related to the point number by the wave's X scaling, which is a property of the wave that you can set. The X scaling of a wave specifies how to compute an X value for a given point number using the formula:

```
x[p] = x0 + p*dx
```

where x[p] is the X value for point p. The two numbers x0 and dx constitute the wave's X scaling property. x0 is the starting X value. dx is the difference in X value from one point to the next. X values are uniformly spaced along the data's X dimension.

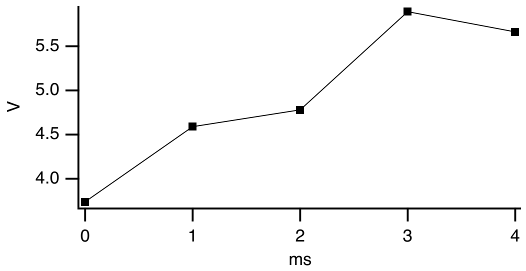
The **SetScale** operation (see page V-853) sets a wave's X scaling. You can use the Change Wave Scaling dialog to generate SetScale commands.

Why does Igor use this model for representing data? We chose this model because it provides all of the information that needed to properly display, analyze and transform waveform data.

By setting your data's X scaling, and X and data units in addition to its data values, you can make a proper graph in one step. You can execute the command

```
Display wave0
```

to produce a graph like this:



If your data is uniformly spaced on the X axis, it is *critical* that you understand and use X scaling.

The X scaling information is essential for operations such as integration, differentiation and Fourier transforms and for functions such as the **area** function (see page V-40). It also simplifies waveform assignment by allowing you to reference a single value or range of values using natural units.

Igor waves can have up to four dimensions. We call these dimensions X, Y, Z and T. X scaling extends to dimension scaling. For each dimension, there is a starting index value (x0, y0, z0, t0) and a delta index value (dx, dy, dz, dt). See Chapter II-6, **Multidimensional Waves**, for more about multidimensional waves.

XY Model of Data

If your data is not uniformly spaced along its X dimension then it can not be represented with a single wave. You need to use two waves as an XY pair.

In an XY pair, the data values of one wave provide X values and the data values of the other wave provide Y values. The X scaling of both waves is irrelevant so we leave it in its default state in which the x0 and dx components are 0 and 1. This gives us

```
x[p] = 0 + 1·p
```

This says that a given point's X value is the same as its point number. We call this "point scaling". Here is some sample data that has point scaling.

	X wave		
	Point Number	X value ()	data value (V)
xWave	0	0	0.0
	1	1	.0013
	2	2	.0021
	3	3	.0029
	4	4	.0042

	Y wave		
	Point Number	X value ()	data value (V)
yWave	0	0	3.74
	1	1	4.59
	2	2	4.78
	3	3	5.89
	4	4	5.66

The X values serve no purpose in the XY model. Therefore, we change our thinking and look at an XY pair this way.

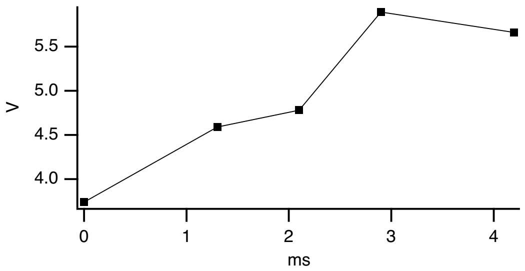
	X wave	
	Point Number	Value (s)
xWave	0	0.0
	1	.0013
	2	.0021
	3	.0029
	4	.0042

	Y wave	
	Point Number	Value (V)
yWave	0	3.74
	1	4.59
	2	4.78
	3	5.89
	4	5.66

We can execute

```
Display yWave vs xWave
```

and it produces a graph like this.



Some operations, such as Fast Fourier Transforms and convolution, require equally spaced data. In these cases, it may be desirable for you to create a uniformly spaced version of your data by interpolation. See **Converting XY Data to a Waveform** on page III-109.

Some people who have uniformly spaced data still use the XY model because it is what they are accustomed to. **This is a mistake**. If your data is uniformly spaced, it will be well worth your while to learn and use the waveform model. It greatly simplifies graphing and analysis and makes it easier to write Igor procedures.

Making Waves

You can make waves by:

- Loading data from a file
- Typing or pasting in a table
- Using the **Make** operation (via a dialog or directly from the command line)
- Using the **Duplicate** operation (via a dialog or directly from the command line)

Most people start by loading data from a file. Igor can load data from text files. In this case, Igor makes a wave for each column of text in the file. Igor can also load data from binary files or application-specific files created by other programs. For information on loading data from files, see **Importing Data** on page II-126.

You can enter data manually into a table. This is recommended only if you have a small amount of data. See **Using a Table to Create New Waves** on page II-239.

To synthesize data with a mathematical expression, you would start by making a wave using the **Make** operation (see page V-526). This operation is also often used inside an Igor procedure to make waves for temporary use.

The **Duplicate** operation (see page V-185) is an important and handy tool. Many built-in operations transform data in place. Thus, if you want to keep your original data as well as the transformed copy of it, use Duplicate to make a clone of the original.

Wave Names

All waves in Igor have names so that you can reference them from commands. You also use a wave's name to select it from a list or pop-up menu in Igor dialogs or to reference it in a waveform assignment statement.

You need to choose wave names when you use the **Make**, **Duplicate** or **Rename** operations via dialogs, directly from the command line, and when you use the Data Browser.

All names in Igor are case insensitive; wave0 and WAVE0 refer to the same wave.

The rules for the kind of characters that you can use to make a wave name fall into two categories: standard and liberal. Both standard and liberal names are limited to 255 bytes in length.

Prior to Igor Pro 8.00, wave names were limited to 31 bytes. If you use long wave names, your wave and experiment files will require Igor Pro 8.00 or later.

Standard names must start with an alphabetic character (A - Z or a-z) and may contain ASCII alphabetic and numeric characters and the underscore character only. Other characters, including spaces, dashes and periods and non-ASCII characters are not allowed. We put this restriction on standard names so that Igor can identify them unambiguously in commands, including waveform assignment statements.

Liberal names, on the other hand, can contain any character except control characters (such as tab or carriage return) and the following four characters:

```
   "  '  :  ;
```

Standard names can be used without quotation in commands and expressions but liberal names must be quoted. For example:

```
Make wave0; wave0 = p          // wave0 is a standard name
Make 'wave 0'; 'wave 0' = p    // 'wave 0' is a liberal name
```

Igor can not unambiguously identify liberal names in commands unless they are quoted. For example, in

```
wave0 = miles/hour
```

miles/hour could be a single wave or it could be the quotient of two waves.

To make them unambiguous, you must enclose liberal names in single straight quotes whenever they are used in commands or waveform arithmetic expressions. For example:

```
wave0 = 'miles/hour'
Display 'run 98', 'run 99'
```

NOTE: Writing procedures that work with liberal names requires extra effort and testing on the part of Igor programmers (See **Programming with Liberal Names** on page IV-168). We recommend that you avoid using liberal names until you understand the potential problems and how to solve them.

See **Object Names** on page III-501 for a discussion of object names in general.

Number of Dimensions

Waves can consist of one to four dimensions. You determine this when you make a wave. You can change it using the **Redimension** operation (see page V-788). See Chapter II-6, **Multidimensional Waves** for details.

Wave Data Types

Each wave has data type that determines the kind of data that it stores.You set a wave's data type when you create it. You can change it using the Data Browser, the **Redimension** operation (see page V-788) or the Redimension dialog.

There are three classes of wave data types:

- Numeric data types
- Text
- References (wave referencesand data folder references)

Each numeric data type can be either real or complex. Text and reference data types can not be complex.

Reference data types are used in programming only.

You can programmatically determine the data type of a wave using the **WaveType** function.

Numeric Wave Data Types

This table shows the numeric precisions available in Igor.

Precision	Range	Bytes per Point
Double-precision floating point	10^{-324} to 10^{+307} (~15 decimal digits)	8
Single-precision floating point	10^{-45} to 10^{+38} (~7 decimal digits)	4
Signed 64-bit integer	-2^63 to 2^63 - 1	8
Signed 32-bit integer	-2,147,483,647 to 2,147,483,648	4
Signed 16-bit integer	-32,768 to 32,767	2
Signed 8-bit integer	-128 to 127	1
Unsigned 64-bit integer	0 to 2^64 - 1	8
Unsigned 32-bit integer	0 to 4,294,967,295	4
Unsigned 16-bit integer	0 to 65,535	2
Unsigned 8-bit integer	0 to 255	1

The 64-bit integer types were added in Igor Pro 7.00.

For most work, single precision waves are appropriate.

Single precision waves take up half the memory and disk space of double precision. With the exception of the FFT and some special purpose operations, Igor uses double precision for calculations regardless of the numeric precision of the source wave. However, the narrower dynamic range and smaller precision of single precision is not appropriate for all data. If you are not familiar with numeric errors due to limited range and precision, it is safer to use double precision for analysis.

Integer waves are intended for data acquisition purposes and are not intended for use in analysis. See **Integer Waves** on page II-85 for details.

Default Wave Properties

When you create a wave using the **Make** operation (see page V-526) operation with no optional flags, it has the following default properties.

Property	**Default**
Number of points	128
Data type	Real, single-precision floating point
X scaling	x0=0, dx=1 (point scaling)
X units	Blank
Data units	Blank

These are the key wave properties. For a comprehensive list of properties, see **Wave Properties** on page II-88.

If you make a wave by loading it from a file or by typing in a table, it has the same default properties except for the number of points.

However you make waves, if they represent waveforms as opposed to XY pairs, you should use the Change Wave Scaling dialog to set their X scaling and units.

Make Operation

Most of the time you will probably make waves by loading data from a file (see **Importing Data** on page II-126), by entering it in a table (see **Using a Table to Create New Waves** on page II-239), or by duplicating existing waves (see **Duplicate Operation** on page II-70).

The **Make** operation is used for making new waves. See the **Make** operation (see page V-526) for additional details.

Here are some reasons to use Make:

- To make waves to play around with.
- For plotting mathematical functions.
- To hold the output of analysis operations.
- To hold miscellaneous data, such as the parameters used in a curve fit or temporary results within an Igor procedure.

The Make Waves dialog provides an interface to the **Make** operation. To use it, choose Make Waves from the Data menu.

Waves have a definite number of points. Unlike a spreadsheet program which automatically ignores blank cells at the end of a column, there is no such thing as an “unused point” in Igor. You can change the number of points in a wave using the Redimension Waves dialog or the **Redimension** operation (see page V-788).

The “Overwrite existing waves” option is useful when you don’t know or care if there is a wave with the same name as the one you are about to make.

Make Operation Examples

Make coefs for use in curve fitting:

```
Make/O coefs = {1.5, 2e-3, .01}
```

Make a wave for plotting a math function:

```
Make/O/N=200 test; SetScale x 0, 2*PI, test; test = sin(x)
```

Make a 2D wave for image or contour plotting:

```
Make/O/N=(20,20) w2D; w2D = (p-10)*(q-10)
```

Make a text wave for a category plot:

```
Make/O/T quarters = {"Q1", "Q2", "Q3", "Q4"}
```

It is often useful to make a clone of an existing wave. Don’t use Make for this. Instead use the **Duplicate** operation (see page V-185).

Make/O does not preserve the contents of a wave and in fact will leave garbage in the wave if you change the number of points, numeric precision or numeric type. Therefore, after doing a Make/O you should not assume anything about the wave’s contents. If you know that a wave exists, you can use the **Redimension** operation instead of Make. Redimension does preserve the wave’s contents.

Waves and the Miscellaneous Settings Dialog

The state of the Type popup menu in the Make Waves dialog, the precision of waves created by typing in a table, and the way Igor Binary waves are loaded (whether they are copied or shared) are preset with the Miscellaneous Settings dialog using the Data Loading Settings category; see **Miscellaneous Settings** on page III-500.

Changing Dimension and Data Scaling

When you make a 1D wave, it has default X scaling, X units and data units. You should use the **SetScale** operation (see page V-853) to change these properties.

The Change Wave Scaling dialog provides an interface to the **SetScale** operation. To use it, choose Change Wave Scaling from the Data menu.

Scaled dimension indices can represent ordinary numbers, dates, times or date&time values. In the most common case, they represent ordinary numbers and you can leave the Units Type pop-up menu in the Set X Properties section of the dialog on its default value: Numeric.

If your data is waveform data, you should enter the appropriate Start and Delta X values. If your data is XY data, you should enter 0 for Start and 1 for Delta. This results in the default “point scaling” in which the X value for a point is the same as the point number.

Normally you should leave the Set X Properties and Set Data Properties checkboxes selected. Deselect one of them if you want the dialog to generate commands to set only X or only Data properties. When working with multidimensional data, the X of Set X Properties can be changed to Y, Z or T via the pop-up menu. See Chapter II-6, **Multidimensional Waves**.

If you want to observe the properties of a particular wave, double-click it in the list or select the wave and then click the From Wave button. This sets all of the dialog items according to that wave’s properties.

Igor uses the dimension and data Units to automatically label axes in graphs. Igor can handle units consisting of 49 bytes or less. Typically, units should be short, standard abbreviations such as “m”, “s”, or “g”. If your data has more complex units, you can enter the complex units or you may prefer to leave the units blank.

Advanced Dimension and Data Scaling

If you click More Options, Igor displays some additional items in the dialog. They give you two additional ways to specify X scaling and allow you to set the wave’s “data full scale” values. These options are usually not needed but for completeness are described in this section.

In spite of the fact that there is only one way of calculating X values, there are three ways you can specify the x0 and dx values. The SetScale Mode pop-up menu changes the meaning of the scaling entries above. The simplest way is to simply specify x0 and dx directly. This is the Start and Delta mode in the dialog and is the only way of setting the scaling unless you click the More Options button. As an example, if you have data that was acquired by a digitizer that was set to sample at 1 MHz starting 150 µs after t=0, you would enter 150E-6 for Start and 1E-6 for Delta.

The other two ways of specifying X scaling are to set the starting and ending X values are and to calculate dx from the number of points. In the Start and End mode you specify the X value of the last data point. Using the Start and Right mode you specify the X at the end of the last interval. For example, assuming our digitizer (above) created a 100 point wave, we would enter 150E-6 as Start for either mode. If we selected the Start and End mode we would enter 249E-6 for End (150E-6 + 99*1E-6). If we selected Start and Right we would enter 250E-6 for Right.

The min and max entries allow you to set a property of a wave called its “data full scale”. This property doesn’t serve a critical purpose. Igor does not use it for any computation or graphing purposes. It is merely a way for you to document the conditions under which the wave data was acquired. For example, if your data comes from a digital oscilloscope and was acquired on the ±10v range, you could enter -10 for min and +10 for max. When you make waves, both of these will initially be set to zero. If your data has a meaningful data full scale, you can set them appropriately. Otherwise, leave them zero.

The data units, on the other hand *are* used for graphing purposes, just like the dimension units.

Date, Time, and Date&Time Units

The units “dat” are special, specifying that the scaled dimension indices or data values of a wave contain date, time, or date&time information.

If you have waveform data then set the X units of your waveform to "dat".

If you have XY data then set the data units of your X wave to "dat". In this case your X wave must be double-precision floating point in order to have enough precision to represent dates accurately.

For example, if you have a waveform that contains some quantity measured once per day, you would set the X units for the wave to “dat”, set the starting X value to the date on which the first measurement was done, and set the Delta X value to one day. Choosing Date from the Units Type pop-up menu sets the X units to “dat”. You can enter the starting value as a date rather than as a number of seconds since 1/1/1904, which is how Igor represents dates internally. When Igor graphs the waveform, it will notice that the X units are “dat” and will display dates on the X axis.

If instead of a waveform, you have an XY pair, you would set the data units of the X wave to “dat”, by choosing Date from the Units Type pop-up menu in the Set Data Properties section of the dialog. When you graph the XY pair, Igor will notice that the X wave contains dates and will display dates on the X axis.

The Units Type pop-up menus do not correspond directly to any property of a wave. That is, a wave doesn’t have a units type property. Instead, these menus merely identify what kind of values you are dealing with so that the dialog can display the values in the appropriate format.

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-85.

For information on dates and times in tables, see **Date/Time Formats** on page II-256.

For information on dates and times in graphs, see **Date/Time Axes** on page II-315.

Duplicate Operation

Duplicate is a handy and frequently-used operation. It can make new waves that are exact clones of existing waves. It can also clone a section of a wave and thus provides an easy way to break a big wave up into smaller waves.

Here are some reasons to use Duplicate:

- To hold the results of a transformation (e.g. integration, differentiation, FFT) while preserving the original data.
- To hold the “destination” of a curve fit.
- For holding temporary results within an Igor procedure.
- To extract a section of a wave.

The Duplicate Waves dialog provides an interface to the **Duplicate** operation (see page V-185). To use it, choose Duplicate Waves from the Data menu.

The cursors button is used in conjunction with a graph. You can make a graph of your template wave. Then put the cursors on the section of the template that you want to extract. Choose Duplicate Waves from the Data menu and click the cursors button. Then click Do It. This clones the section of the template wave identified by the cursors.

People sometimes make the mistake of using the **Make** operation when they should be using Duplicate. For example, the destination wave in a curve fit must have the same number of points, numeric type and numeric precision as the source wave. Duplicating the source wave insures that this will be true.

Duplicate Operation Examples

Clone a wave and then transform the clone:

```
Duplicate/O wave0, wave0_d1; Differentiate wave0_d1
```

Use Duplicate to inherit the properties of the template wave:

```
Make/N=200 wave0; SetScale x 0, 2*PI, wave0; wave0 = sin(x)
Duplicate wave0, wave1; wave1 = cos(x)
```

Make a destination wave for a curve fit:

```
Duplicate/O data1, data1_fit
CurveFit gauss data1 /D=data1_fit
```

Compare the first half of a wave to the second:

```
Duplicate/O/R=[0,99] data1, data1_1
Duplicate/O/R=[100,199] data1, data1_2
Display data1_1, data1_2
```

We often use the /O flag (overwrite) with Duplicate because we don’t know or care if a wave already exists with the new wave name.

Killing Waves

The **KillWaves** operation (see page V-471) removes waves from the current experiment. This releases the memory used by the waves. Waves that you no longer need clutter up lists and pop-up menus in dialogs. By killing them, you reduce this clutter.

Here are some situations in which you would use KillWaves:

- You are finished examining data that you loaded from a file.
- You are finished using a wave that you created for experimentation.

- You no longer need a wave that you created for temporary use in an Igor procedure.

The Kill Waves dialog provides an interface to the **KillWaves** operation. To use it, choose Kill Waves from the Data menu.

Igor will not let you kill waves that are used in graphs, tables or user-defined functions so they do not appear in the list.

Note: Igor can not tell if a wave is referenced from a macro. Thus, Igor will let you kill a wave that is referenced from a macro but not used in any other way. The most common case of this is when you close a graph and save it as a recreation macro. Waves that were used in the graph are now used only in the macro and Igor will let you kill them. If you execute the graph recreation macro, it will be unable to recreate the graph.

KillWaves can delete the Igor binary wave file from which a wave was loaded, called the “source file”. This is normally not necessary because the wave you are killing either has never been saved to disk or was saved as part of a packed experiment file and therefore was not loaded from a standalone file.

The “Kill all waves not in use” option is intended for those situations where you have created an Igor experiment that contains procedures which load, graph and process a batch of waves. After you have processed one batch of waves, you can kill all graphs and tables and then kill all waves in the experiment in preparation for loading the next batch. This affects only those waves in the current data folder; waves in any other data folders will not be killed.

KillWaves Operation Examples

Here are some simple examples using KillWaves.

```
// Kills all target windows and all waves.
// Does not kill nontarget windows (procedure and help windows).
Function KillEverything()
   String windowName

   do
      windowName = WinName(0, 1+2+4+16+64)// Get next target window
      if (CmpStr(windowName, "") == 0) // If name is ""
         break                        // we are done so break loop
      endif
      KillWindow $windowName          // Kill this target window
   while (1)

   KillWaves/A                        // Kill all waves
End

// This illustrates killing a wave used temporarily in a procedure
Function Median(w)                 // Returns median value of wave w
   Wave w

   Variable result

   Duplicate/O w, temp             // Make a clone of wave
   Sort temp, temp                 // Sort clone
   result = temp[numpnts(temp)/2]

   KillWaves temp                  // Kill clone

   return result
End
```

For more examples, see the “Kill Waves” procedure file in the “WaveMetrics Procedures” folder.

Browsing Waves

The Data Browser (Data menu) lets you see what waves (as well as strings and variables) exist at any given time. It also lets you see what data folders exist and set the current data folder. The Data Browser is described in detail in Chapter II-8, **Data Folders**.

Igor Pro 6 had a Browse Waves dialog which you accessed via the Data→Browse Waves menu item. Because it provided the same functionality as the Data Browser, the dialog and menu item were removed in Igor Pro 7.00.

Renaming Waves

You can rename a wave using:

- The Data Browser
- The Rename dialog (Data menu)
- The **Rename** operation from the command line

The **Rename** operation (see page V-796) renames waves as well as other objects.

Here are some reasons for renaming waves:

- You have loaded a bunch of waves from a file and Igor auto-named the waves.
- You have decided on a naming convention for waves and you want to make existing waves follow the convention.
- You are about to load a set of waves whose names will be the same as existing waves and you want to get the existing waves out of the way but still keep them in memory. (You could also achieve this by moving them to a new data folder.)

To use the **Rename** operation, choose Rename from the Data menu. This brings up the Rename Objects dialog.

Redimensioning Waves

The Redimension operation can change the following properties of a wave:

- The number of dimensions in the wave
- The number of elements in each dimension
- The numeric precision (e.g., single to double)
- The numeric type (e.g., real to complex)

The Redimension Waves dialog provides an interface to the **Redimension** operation (see page V-788). To use it, choose Redimension Waves from the Data menu.

When Redimension adds new elements to a wave, it sets them to zero for a numeric wave and to blank for a text wave.

The following commands illustrate two ways of changing the numeric precision of a wave. Redimension preserves the contents of the wave whereas Make does not.

```
Make/N=5 wave0=x
Edit wave0
Redimension/D wave0        // This preserves the contents of wave0
Make/O/D/N=5 wave0         // This does not
```

See **Vector (Waveform) to Matrix Conversion** on page II-98 for information on converting a 1D wave into a 2D wave while retaining the data (i.e., reshaping).

You cannot change a wave from numeric to text or vice versa. The following examples illustrate how you can make a text copy of a numeric wave and a numeric copy of a text wave:

```
Make/N=10 numWave = p
Make/T/N=(numpnts(numWave)) textWave = num2str(numWave)
Make/N=(numpnts(textWave)) numWave2 = str2num(textWave)
```

However, you can lose precision because num2str prints with only 6 digits of precision.

Inserting Points

There are two ways to insert new points in a wave. You can do this by:

- Using the **InsertPoints** operation
- Typing or pasting in a table

This section deals with the **InsertPoints** operation (see page V-443). For information on typing or pasting in a table, see Chapter II-12, **Tables**.

Using the **InsertPoints** operation, you can insert new data points at the start, in the middle or at the end of a 1D wave. You can also insert new elements in multidimensional waves. For example, you can insert new columns in a 2D matrix wave. The inserted values will be 0 for a numeric wave and "" for a text wave.

The Insert Points dialog provides an interface to the **InsertPoints** operation. To use it, choose Insert Points from the Data menu.

If the value that you enter for first point is greater than the number of elements in the selected dimension of a selected wave, the new points are added at the end of the dimension. InsertPoints can change the dimensionality of a wave. For example, if you insert a column in a 1D wave, you end up with at 2D wave.

If the top window is a table at the time that you select Insert Points, Igor will preset the dialog items based on the selection in the table.

Deleting Points

There are two ways to delete points from a wave. You can do this by:

- Using the **DeletePoints** operation
- Doing a cut in a table

This section deals with the **DeletePoints** operation (see page V-157). For information on cutting in a table, see Chapter II-12, **Tables**.

Using the **DeletePoints** operation, you can delete data points from the start, middle or end of a 1D wave. You can also delete elements from multidimensional waves. For example, you can delete columns from a 2D matrix wave.

The Delete Points dialog provides an interface to the **DeletePoints** operation. To use it, choose Delete Points from the Data menu.

If the value that you enter for first point is greater than the number of elements in the selected dimension of a selected wave, DeletePoints will do nothing to that wave. If the number of elements is too large, DeletePoints will delete from the specified first element to the end of the dimension.

Except for the case of removing all elements, which leaves the wave as 1D, DeletePoints does not change the dimensionality of a wave. Use **Redimension** for that.

If the top window is a table at the time that you choose Delete Points, Igor will preset the dialog items based on the selection in the table.

Waveform Arithmetic and Assignments

Waveform arithmetic is a very flexible and powerful part of Igor's analysis capability. You can write assignment statements that work on an entire wave or on a subset of a wave, much as you would write an assignment to a single variable in a standard programming language.

In a wave assignment statement, a wave appears on the left side and a mathematical expression appears on the right side. Here are some examples.

```
wave0 = sin(x)
wave0 = log(wave1/wave2)
wave0[0,99] = wave1[100 + p]
```

A wave on the left side is called the **destination wave**. A wave on the right side is called a **source wave**.

When Igor executes a wave assignment statement, it evaluates the expression on the right-hand side one time for each point in the destination wave. The result of each evaluation is stored in the corresponding point in the destination wave.

During execution, the symbol *p* has a value equal to the number of the point in the destination wave which is being set and the symbol *x* has a value equal to the X value at that point. The X value for a given point is determined by the number of the point and the X scaling for the wave. To see this, execute the following commands, one-at-a-time:

```
Make/N=5 wave0; SetScale/P x 0, .1, wave0; Edit wave0.xy
wave0 = p
wave0 = x
```

The first assignment statement sets the value of each point of wave0 to the point number. The second assignment statement sets the value of each point of wave0 to the X value for that point.

A source wave returns its data value at the point being evaluated. In the example

```
wave0 = log(wave1/wave2)
```

Igor evaluates the right-hand expression once for each point in wave0. During each evaluation of the expression, wave1 and wave2 return their data values at the point being evaluated. Consequently, these commands are equivalent:

```
wave0 = log(wave1/wave2)

wave0 = log(wave1[p]/wave2[p])
```

Understanding Wave Assignments

To understand wave assignments, it is critical to understand the meaning of p and x. To this end, it may be helpful to think of what happens inside Igor. A wave assignment statement causes a loop to run inside Igor.

p is the loop index of the internal loop and runs over the range of points being set in the destination wave. In the preceding example, p starts from 0 and is incremented each time through the internal loop, up to N-1, where N is the number of points in wave0. For each value of p, the right-hand expression is evaluated and the result is stored in the corresponding point of wave0.

When you see `wave1` or `wave1[p]` on the right-hand side of a waveform assignment statement, this returns the value of wave1 at point p, which is the point in the destination wave (wave0) that Igor is setting during the current iteration of the internal loop.

x is similar to p except that, instead of being the loop index of the internal Igor loop, it is calculated from the loop index using the X scaling of the destination wave:

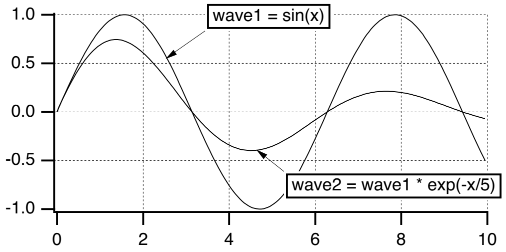
```
x = x0 + p*dx
```

Wave assignment is an important Igor feature and it will be worth your time to understand it. You may need to reread this material and do some experimentation on the command line. If you are new to Igor, you may prefer to return to this topic after you have gained experience.

Wave Assignment Example

This command sequence illustrates some of the ideas explained above.

```
Make/N=200 wave1, wave2                // Two waves, 200 points each
SetScale/P x, 0, .05, wave1, wave2     // Set X values from 0 to 10
Display wave1, wave2                   // Create a graph of waves
wave1 = sin(x)                         // Assign values to wave1
wave2 = wave1 * exp(-x/5)              // Assign values to wave2
```



Since wave1 has 200 points, the wave assignment statement `wave1=sin(x)` evaluates `sin(x)` 200 times, once for each point in wave1. The first point of wave1 is point number 0 and the last point of wave1 is point number 199. The symbol p, not used in this example, goes from 0 to 199. The symbol x steps through the 200 X values for wave1 which start from 0 and step by .05, as specified by the SetScale command. The result of each evaluation is stored in the corresponding point in wave1, making wave1 about 1.5 cycles of a sine wave.

Since wave2 also has 200 points, the wave assignment statement `wave2=wave1*exp(-x/5)` evaluates `wave1*exp(-x/5)` 200 times, once for each point in wave2. In this assignment, the right-hand expression contains a wave, wave1. As Igor executes the assignment, p goes from 0 to 199. Each of the 200 times the right side is evaluated, wave1 returns its data value for the corresponding point. The result of each evaluation is stored in the corresponding point in wave2 making wave2 about 1.5 cycles of a damped sine wave.

The effect of a wave assignment statement is to set the data values of the destination wave. Igor does not remember the functional relationship implied by the assignment. In this example, if you changed wave1, wave2 would not change automatically. If you wanted wave2 to have the same functional relationship to wave1 as it had before you changed wave1, you would have to reexecute the `wave2=wave1*exp(-x/5)` assignment.

There is a special kind of wave assignment statement that *does* establish a functional relationship. It should be used sparingly. See **Wave Dependency Formulas** on page II-84 for details.

More Wave Assignment Features

Just as the symbol p returns the current element number in the rows dimension, the symbols q, r and s return the current element number in the columns, layers and chunks dimensions of multidimensional waves. The symbol x in the rows dimension has analogs y, z and t in the columns, layers and chunks dimensions. See Chapter II-6, **Multidimensional Waves**, for details.

You can use multiple processors to execute a waveform assignment statement that takes a long time. See **Automatic Parallel Processing with MultiThread** on page IV-323 for details.

The right-hand expression is evaluated in the context of the data folder containing the destination wave. See **Data Folders and Assignment Statements** on page II-111 for details.

Usually source waves have the same number of points and X scaling as the destination wave. In some cases, it is useful to write a wave assignment statement where this is not true. This is discussed under **Mismatched Waves** on page II-83.

Indexing and Subranges

Igor provides two ways to refer to a specific point or range of points in a 1D wave: X value indexing and point number indexing. Consider the following examples.

```
wave0[54] = 92          // Sets wave0 at point 54 to 92
wave0(54) = 92          // Sets wave0 at X=54 to 92
wave0[1,10] = 92        // Sets wave0 from point 1 to point 10 to 92
wave0(1,10) = 92        // Sets wave0 from X=1 to X=10 to 92
```

Brackets tell Igor that you are indexing the wave using point numbers. The number or numbers inside the brackets are interpreted as point numbers of the indexed wave.

Parentheses tell Igor that you are indexing the wave using X values. The number or numbers inside the parentheses are interpreted as X values of the indexed wave. When you use an X value as an index, Igor first finds the point number corresponding to that X value based on the indexed wave's X scaling, and then uses that point number as the point index.

If the wave has point scaling then these two methods have identical effects. However, if you set the X scaling of the wave to other than point scaling then these commands behave differently. In both cases the range is inclusive.

You can specify not only a range but also a point number increment. For example:

```
wave0[0,98;2] = 1       // Sets even numbered points in wave0 to 1
wave0[1,99;2] = -1      // Sets odd numbered points in wave0 to -1
```

The number after the semicolon is the increment. Igor begins at the starting point number and goes up to and including the ending point number, skipping by the increment. At each resulting point number, it evaluates the right-hand side of the wave assignment statement and sets the destination point accordingly.

Increments can also be used when you specify a range in terms of X value but the increment is always in terms of point number. For example:

```
wave0(0,100;5) = PI     // Sets wave0 at specified X values to PI
```

Here Igor starts from the point number corresponding to x = 0 and goes up to and including the point number that corresponds to x = 100. The point number is incremented by 5 at each iteration.

You can take some shortcuts in specifying the range of a destination wave. The subrange start and end values can both be omitted. When the start is omitted, point number zero is used. When the end is omitted, the last point of the wave is used. You can also use a * character or INF to specify the last point. An omitted increment value defaults to a single point.

Here are some examples that illustrate these shortcuts:

```
wave0[ ,50] = 13           // Sets wave0 from point 0 to point 50

wave0[51,] = 27            // Sets wave0 from point 51 to last point

wave0[51,*] = 27           // Sets wave0 from point 51 to last point

wave0[51,INF] = 27         // Sets wave0 from point 51 to last point

wave0[ , ;2] = 18.7        // Sets every even point of wave0

wave0[1,*;2] = 100         // Sets every odd point of wave0
```

A subrange of a destination wave may consist of a single point or a range of points but a subrange of a source wave must consist of a single point. In other words the wave assignment statement:

```
wave1(4,5) = wave2(5,6)    // Illegal!
```

is not legal. In this assignment, x ranges from 4 to 5. You can get the desired effect using:

```
wave1(4,5) = wave2(x+1)    // OK!
```

By virtue of the range specified on the left hand side, x goes from 4 to 5. Therefore, x+1 goes from 5 to 6 and the right-hand expression returns the values of wave2 from 5 to 6.

Indexing with an index wave

You can set specific elements of the destination wave using another wave to provide index values using this syntax:

```
destWave[indexWave] = <expression>
```

This feature was added in Igor Pro 8.00.

When the destination wave is one dimensional, the index wave must contain a list of valid point numbers. For example:

```
Make/O/N=10 destWave = 0
Make/O indexWave = {2,5,8}
destWave[indexWave] = p; Print destWave
   destWave[0]= {0,0,2,0,0,5,0,0,8,0}
```

When the destination wave is multidimensional, the index wave can be two dimensional with a valid row index in the first column and a valid column index in the second column. The next example sets all elements of the destination wave to zero except for elements (3,2), (5,4), and (7,6) which it sets to 999.

```
Make/O/N=(10,10) destWave = 0
Make/O/N=(3,2) indexWave
indexWave[0][0] = {3,5,7}           // Store row indices in column 0
indexWave[0][1] = {2,4,6}           // Store column indices in column 1
Edit indexWave
destWave[indexWave] = 999
Edit destWave
```

When the destination wave is multidimensional, it is legal for the index wave to be 1D containing linear point numbers as if the destination wave were itself 1D. The assignment statement is evaluated as if the destination wave were 1D so q, r, s, y, z and t return zero on the righthand side. This example uses a 1D index wave to set the same elements as the preceding example:

```
Make/O/N=(10,10) destWave = 0
Make/O indexWave = {23,45,67}
destWave[indexWave] = 888
```

In addition to using an index wave on the left hand side, you can also use a value wave on the right with this syntax:

```
destWave[indexWave] = {valueWave}
```

The value wave is a 1D vector of values. It should contain the same number of values as there are index values and should have the same type (numeric, string, etc.) as the destination wave. Here is an example:

```
Make/O/N=10 destWave = 0
Make/O indexWave = {2,5,8}
Make/O valueWave = {777,776,775}
destWave[indexWave] = {valueWave}
```

When you use an index wave, a list of individual values is not supported:

```
destWave[indexWave] = {777,776,775}    // Error - value wave expected in braces
```

In the next example we treat a 2D destination wave as 1D by providing a 1D index wave:

```
Make/O/N=(10,10) destWave = 0
Make/O indexWave = {23,45,67}
Make/O valueWave = {666,665,664}
destWave[indexWave] = {valueWave}
```

In the next example we treat a 2D the destination wave as 2D by providing a 2D index wave:

```
Make/O/N=(10,10) destWave = 0
Make/O/N=(3,2) indexWave
indexWave[0][0] = {3,5,7}          // Store row indices in column 0
indexWave[0][1] = {2,4,6}          // Store column indices in column 1
Make/O valueWave = {555,554,553}
destWave[indexWave] = {valueWave}
```

Interpolation in Wave Assignments

If you specify a fractional point number or an X value that falls between two data points, Igor will return a linearly interpolated data value. For example, wave1[1.75] returns the value of wave1 three-quarters of the way from the data value of point 1 to the data value of point 2. This interpolation is done only for one-dimensional waves. See **Multidimensional Wave Assignment** on page II-96, for information on assignments with multidimensional data.

This is a powerful feature. Imagine that you have an evenly spaced calibration curve, called calibration, and you want to find the calibration values at a specific set of X coordinates as stored in a wave called xData. If you have set the X scaling of the calibration wave, you can do the following:

```
Duplicate xData, yData
yData = calibration(xData)
```

This uses the interpolation feature of Igor's wave assignment statement to find a linearly-interpolated value in the calibration wave for each X coordinate in the xData wave.

Lists of Values

You can assign values to a wave or to a subrange of a wave using a list of values in curly braces. You can also add rows and columns.

1D Lists of Values

This section shows how to set elements of a 1D wave and add rows using lists of values. As shown below a 1D list of values consists of a lists of row values in curly braces.

```
Make/O/N=5 wave0 = NaN        // Make 5 point wave and display in table
Edit/W=(5,45,450,350) wave0

wave0 = {0, 1, 2}             // Redimension wave0 to 3 rows and set Y values

Make/O/N=5 wave0 = NaN        // Restore to 5 points

wave0[1,3]= {1, 2, 3}         // Set points 1 through 3

wave0 = NaN
wave0[1]= {1, 2, 3}           // Set points 1 through 3 (same as previous)

wave0 = NaN
wave0[0,4;2]= {0, 2, 4}       // Set points 0, 2, and 4 using a step of 2

// Extend wave0 from 5 rows to 8 rows and set new Y values.
// Adds rows because the index, 5, is equal to the number of wave rows.
wave0 = NaN
wave0[5]= {5, 6, 7}

Make/O/N=5 wave0 = NaN        // Restore to 5 points
```

```
// Add a column changing wave0 from 1D to 2D. Adds a column
// because the column index, 1, is equal to the number of wave columns.
wave0[][1] = {10,11,12,13,14}
```

2D Lists of Values

This section shows how to set elements of a 2D wave and add rows and columns using lists of values. As shown below a 2D list of values consists of a a nested list of lists in curly braces. Each inner list defines the row values for one column.

```
Make/O/N=(4,3) w2D = NaN      // Make wave with 4 rows and 3 columns
Edit/W=(225,45,850,350) w2D

// Redimension w2D and set elements.
// Set column 0 to {1,2,3}, column 1 to {10,11,12}.
w2D = {{0,1,2}, {10,11,12}}

Make/O/N=(4,3) w2D = NaN      // Restore to 4 rows by 3 column

// Set column 0 to {0,1,2,3}, column 1 to {10,11,12,13}.
// Here [] can be read as "all rows" so this means set
// all rows of columns 0 through 1.
w2D[][0,1] = {{0,1,2,3}, {10,11,12,13}}

// Extend w2D from 4 rows to 5 rows and set new Y values.
// Adds a row because the row index, 4, is equal to the number of wave rows.
// Here [] can be read as "all columns" so this means set row 4, all columns.
w2D[4][] = {{4},{14},{24}}

// Extend w2D from 5 rows to 7 rows and set new Y values.
w2D[5][] = {{5,6},{15,16},{25,26}}

Make/O/N=(4,3) w2D = NaN      // Restore to 4 rows by 3 column

// Extend w2D from 3 columns to 4 columns and set new Y values.
// Adds a column because the column index, 3, is equal
// to the number of wave columns.
w2D[][3] = {{30,31,32,33}}

// Extend w2D from 4 columns to 6 columns and set new Y values.
w2D[][4] = {{40,41,42,43},{50,51,52,53}}
```

Wave Initialization

From Igor’s command line or in a procedure, you can make a wave and initialize it with a single command, as illustrated in the following examples:

```
Make wave0=sin(p/8)         // wave0 has default number of points
Make coeffs={1,2,3}         // coeffs has just three points
```

Example: Normalizing Waves

When comparing the shape of multiple waves you may want to normalize them so that the share a common range. For example:

```
// Create some sample data
Make waveA = 3*sin(x/8)
Make waveB = 2*sin(pi/16 + x/8)

// Display the waves
Display waveA, waveB
ModifyGraph rgb(waveB)=(0,0,65535)
```

```
// Normalize the waves
Variable aMin = WaveMin(waveA)
Variable bMin = WaveMin(waveB)
waveA -= aMin
waveB -= bMin
Variable aMax = WaveMax(waveA)
Variable bMax = WaveMax(waveB)
waveA /= aMax
waveB /= bMax
```

Note the use of the temporary variables aMin and bMin. They are needed for two reasons. First, if we wrote `waveA -= WaveMin(waveA)`, then **WaveMin** would be called once for each point in waveA, which would be a waste of time. Worse than that, the minimum value in waveA would change during the course of the waveform assignment statement, giving incorrect results.

There are sometimes faster ways to do waveform arithmetic. For large waves, the **FastOp** and **MatrixOp** operations provide increased speed:

```
waveA -= aMin                    // FastOp does not support wave-variable
FastOp waveA = (1/aMax) * waveA

MatrixOp/O waveA = waveA - aMin
MatrixOp/O waveA = waveA / aMax
```

Example: Converting XY Data to Waveform Data

There are some times when it is desirable to convert XY data to uniformly spaced waveform data. For example, the Fast Fourier Transform requires uniformly spaced data. If you have measured XY data in the time domain, you would need to do this conversion before doing an FFT on it.

We can make some sample XY data as follows:

```
Make/N=1024 xWave, yWave
xWave = 2*PI*x/1024 + gnoise(.001)
yWave = sin(xwave)
```

xWave has values from 0 to 2π with a bit of noise in them. Our data is not uniformly spaced in the x dimension but it is monotonic — always increasing, in this case. If it were not monotonic we could sort the XY pair.

We can create a waveform representing our XY data as follows:

```
Duplicate ywave, wave0
SetScale x 0, 2*PI, wave0
wave0 = interp(x, xwave, ywave)
```

The SetScale command sets the scaling of wave0 so that its X values run from 0 to 2π. Its data values are generated by picking a value off the curve represented by ywave versus xwave at each of these X values using linear interpolation.

See **Converting XY Data to a Waveform** on page III-109 for a discussion of other interpolation techniques.

Example: Concatenating Waves

Concatenating waves can be done much more easily using the **Concatenate** operation (see page V-82). This simple example serves mainly to illustrate a use of wave assignment statements.

Suppose we have three waves of 100 points each: wave1, wave2 and wave3. We want to create a fourth wave, wave4, which is the concatenation of the three original waves. Here is the sequence of commands to do this.

```
Make/N=300 wave4
wave4[0,99] = wave1[p]              // Set first third of wave4
wave4[100,199] = wave2[p-100]       // Set second third of wave4
wave4[200,299] = wave3[p-200]       // Set last third of wave4
```

In this example, we use a subrange of wave4 as the destination of our wave assignment statements. The right-hand expressions index the appropriate values of wave1, wave2 and wave3. Remember that p ranges over the points being evaluated in the destination. So, p ranges from 0 to 99 in the first assignment, from 100 to 199 in the second assignment and from 200 to 299 in the third assignment. In each of the assignments, the wave on the right-hand side has only 100 points, from point 0 to point 99. Therefore we must offset p on the right-hand side to pick out the 100 values of the source wave.

Example: Decomposing Waves

Suppose we have a 300 point wave, wave4, that we want to decompose into three waves of 100 points each: wave1, wave2 and wave3. Here is the sequence of commands to do this.

```
Make/N=100 wave1,wave2,wave3
wave1 = wave4[p]            // Get first third of wave4
wave2 = wave4[p+100]        // Get second third of wave4
wave3 = wave4[p+200]        // Get last third of wave4
```

In this example, we use a subrange of wave4 as the source of our data. We index the desired segment of wave4 using point number indexing. Since wave1, wave2 and wave3 each have 100 points, p ranges from 0 to 99. In the first assignment, we access points 0 to 99 of wave4. In the second assignment, we access points 100 to 199 of wave4. In the third assignment, we access points 200 to 299 of wave4.

You could also use the **Duplicate** operation (see page V-185) to make a wave from a section of another wave.

Example: Complex Wave Calculations

Igor includes a number of built-in functions for manipulating complex numbers and complex waves. These are illustrated in the following example.

We make a time domain waveform and do an FFT on it to generate a complex wave. The example function shows how to pick out the real and imaginary part of the complex wave, how to find the sum of squares and how to convert from rectangular to polar representation. For more information on frequency domain processing, see **Fourier Transforms** on page III-270.

```
Function ComplexWaveCalculations()
   // Make a time domain waveform
   Make/O/N=1024 wave0
   SetScale x 0, 1, "s", wave0            // Goes from 0 to 1 second
   wave0 = sin(2*PI*x) + sin(6*PI*x)/3 + sin(10*PI*x)/5 + sin(14*PI*x)/7
   Display wave0 as "Time Domain"

   // Do the FFT
   FFT/DEST=cwave0 wave0                  // cwave0 is complex, 513 points
   cwave0 /= 512;cwave0[0] /= 2           // Normalize amplitude
   Display cwave0 as "Frequency Domain";SetAxis bottom, 0, 25

   // Calculate magnitude and phase
   Make/O/N=513 mag0, phase0, power0      // These are real waves
   CopyScales cwave0, mag0, phase0, power0
   mag0 = real(r2polar(cwave0))
   phase0 = imag(r2polar(cwave0))
   phase0 *= 180/PI                       // Convert to Degrees
   Display mag0 as "Magnitude and Phase"; AppendToGraph/R phase0
   SetAxis bottom, 0, 25
   Label left, "Magnitude";Label right, "Phase"

   // Calculate power spectrum
   power0 = magsqr(cwave0)
   Display power0 as "Power Spectrum";SetAxis bottom, 0, 25
End
```

Example: Comparison Operators and Wave Synthesis

The comparison operators ==, >=, >, <= and < can be useful in synthesizing waves. Imagine that you want to set a wave so that its data values equal -π for x<0 and +π for x>=0. The following wave assignment statement accomplishes this:

```
wave1 = -pi*(x<0) + pi*(x>=0)
```

This works because the conditional statements return 1 when the condition is true and 0 when it is false, and then the multiplication proceeds.

You can also make such assignments using the conditional operator:

```
wave0 = (x>0) ? pi : -pi
```

A series of impulses can be made using the **mod** function and ==. This wave equation will assign 5 to every tenth point starting with point 0, and 0 to all the other points:

```
wave1 = (mod(p,10)==0) * 5
```

Example: Wave Assignment and Indexing Using Labels

Dimension labels can be used to refer to wave values by a meaningful name. Thus, for example, you can create a wave to store coefficient values and directly refer to these values by the name of the coefficient (e.g., coef[%Friction]) instead of a potentially confusing and less meaningful numeric index (e.g., coef[1]). You can also view the wave values and labels in a table.

You create wave dimension labels using the **SetDimLabel** operation (see page V-838); for details see **Dimension Labels** on page II-93. Dimension labels may be up to 255 bytes in length; if you use liberal names, such as those containing spaces, make certain to enclose these names within single quotation marks.

Prior to Igor Pro 8.00, dimension labels were limited to 31 bytes. If you use long dimension labels, your wave files and experiments will require Igor Pro 8.00 or later.

In this example we create a wave and use the **FindPeak** operation (see page V-247) to get peak parameters of the wave. Next we create an output parameter wave with appropriate labels and then assign the FindPeak results to the output wave using the labels.

```
// Make a wave and get peak parameters
Make test=sin(x/30)
FindPeak/Q test

// Create a wave with appropriate row labels
Make/N=6 PeakResult
SetDimLabel 0,0,'Peak Found', PeakResult
SetDimLabel 0,1,PeakLoc, PeakResult
SetDimLabel 0,2,PeakVal, PeakResult
SetDimLabel 0,3,LeadingEdgePos, PeakResult
SetDimLabel 0,4,TrailingEdgePos, PeakResult
SetDimLabel 0,5,'Peak Width', PeakResult

// Fill PeakResult wave with FindPeak output variables
PeakResult[%'Peak Found']   =V_flag
PeakResult[%PeakLoc]        =V_PeakLoc
PeakResult[%PeakVal]        =V_PeakVal
PeakResult[%LeadingEdgePos] =V_LeadingEdgeLoc
PeakResult[%TrailingEdgePos]=V_TrailingEdgeLoc
PeakResult[%'Peak Width']   =V_PeakWidth

// Display the PeakResult values and labels in a table
Edit PeakResult.ld
```

In addition to the method illustrated above, you can also create and edit dimension labels by displaying the wave in a table and showing the dimension labels with the data. See **Showing Dimension Labels** on page II-235 for further details on using tables with labels.

Mismatched Waves

For most applications you will not need to mix waves of different lengths. In fact, doing this is more often the result of a mistake than it is intentional. However, if your application requires mixing you will need to know how Igor handles this.

Let's consider the case of assigning the value of one wave to another with a command such as

```
wave1 = wave2
```

In this assignment, there is no explicit indexing, so Igor evaluates the expression as if you had written:

```
wave1 = wave2[p]
```

If wave2 has more points than wave1, the extra points have no effect on the assignment since p ranges from 0 to n-1, where n is the number of points in wave1.

If wave2 has fewer points than wave1 then Igor will try to evaluate wave2[p] for values of p greater than the length of wave2. In this case, it returns the value of the last point in wave2 during the wave assignment statement but also raises an "index out of range" error. Previous versions of Igor did not raise this error.

It may be that you actually want the values in wave1 to span the values in wave2 by interpolating between values in wave2. To get Igor to do this, you must explicitly index the appropriate X values on the right side. For instance, if you have two waves of different lengths, you can do this:

```
big = small[p*(numpnts(small)-1)/(numpnts(big)-1)]
```

NaNs, INFs and Missing Values

The data value of a point in a floating point numeric wave is normally a finite number but can also be a NaN or an INF. NaN means "not a number". An expression returns the value NaN when it makes no sense mathematically. For example, `log(-1)` returns the value NaN. You can also set a point to NaN, using a table or a wave assignment statement, to represent a missing value. An expression returns the value INF when it makes sense mathematically but has no finite value. `log(0)` returns the value -INF.

The IEEE floating point standard defines the representation and behavior of NaN values. There is no way to represent a NaN in an integer wave. If you attempt to store NaN in an integer wave, you will store a garbage value.

Comparison operators do not work with NaN parameters because, by definition, NaN compared to anything, even another NaN, is false. Use **numtype** to test if a value is NaN.

Igor ignores NaNs and INFs in curve fit and wave statistics operations. NaNs and INFs have no effect on the scaling of a graph. When plotting, Igor handles NaNs and INFs properly, as missing and infinite values respectively.

Igor does *not* ignore NaNs and INFs in many other operations, especially those that are DSP related such as FFT. In general, any operation that numerically combines all or most of the data points from a wave will give meaningless results if one or more points is a NaN or INF. Notable examples include the **area** and **mean** functions and the **Integrate** and **FFT** operations. Some operations that only mix a few points such as Smooth and Differentiate will "contaminate" only those points in the vicinity of the NaN or INF. You can use the Interpolate operation (Analysis menu) to create a NaN-free version of a wave.

If you get NaNs from functions such as **area** or **mean** or operations such as **Convolve** or any other functions or operations that sum points in waves, it indicates that some of the points in the wave are NaN. If you get NaNs from curve-fitting results, it indicates that Igor's curve fitting has failed. See **Curve Fitting Troubleshooting** on page III-266 for troubleshooting tips.

See **Dealing with Missing Values** on page III-112 for techniques for dealing with NaNs.

Don’t Use the Destination Wave as a Source Wave

You may get unexpected results if the destination of a wave assignment statement also appears in the right-hand expression. Consider these examples:

```
wave1 -= wave1(5)
wave1 -= vcsr(A)            // where cursor A is on wave1
```

Each of these examples is an attempt to subtract the value of wave1 at a particular point from every point in wave1. This will not work as expected because the value of wave1 at that particular point is altered during the assignment. At some point in the assignment, wave1(5) or vcsr(A) will return 0 since the value at that point in wave1 will have been subtracted from itself.

You can get the desired result by using a variable to store the value of wave1 at the particular point.

```
Variable tmp
tmp = wave1(5); wave1 -= tmp
tmp = vcsr(A); wave1 -= tmp
```

Wave Dependency Formulas

You can cause a wave assignment statement to “stick” to the wave by substituting “:=” for “=” in the statement. This causes the wave to become dependent upon the objects referenced in the expression. For example:

```
Variable/G gAngularFrequency = 5
wave1 := sin(gAngularFrequency*x)    // Note ":="
Display wave1
```

If you now execute “gAngularFrequency = 8” you will see the wave automatically update. Similarly if you change the wave’s X scaling using the **SetScale** operation (see page V-853), the wave will be automatically recalculated for the new range of X values.

Dependencies should be used sparingly if at all. Overuse creates a web of interactions that are difficult to understand and difficult to debug. It is better to explicitly update the target when necessary.

See Chapter IV-9, **Dependencies**, for further discussion.

Using the Wave Note

One of the properties of a wave is the **wave note**. This is just some plain text that Igor stores with each wave. The note is empty when you create a wave. There is no limit on its length.

You can inspect and edit a wave note using the Data Browser. You can set or get the contents of a wave note from an Igor procedure using the **Note** operation (see page V-694) or the **note** function (see page V-694).

Originally we thought of the wave note as a place for an experimenter to store informal comments about a wave and it is fine for that purpose. However, over time both we and many Igor users have found that the wave note is also a handy place to store additional, user-defined properties of a wave in a structured way. These additional properties are editable using the Data Browser but they can also be used and manipulated by procedures.

To do this, you store keyword-value pairs in the wave note. For example, a note might look like this:

```
CELLTYPE:rat hippocampal neuron
PATTERN:1VN21
TREATMENT:PLACEBO
```

You could then write Igor functions to set the CELLTYPE, PATTERN and TREATMENT properties of a wave. You can retrieve such properties using the **StringByKey** function.

Integer Waves

Igor provides support for integer waves primarily to aid in data acquisition projects. They allow people who are interfacing with hardware to write/read directly into integer waves. This allows for slightly quicker live display and also saves the programmer from having to convert integers to floating point.

Integer waves are also appropriate for storing images.

Aside from memory considerations there is no other reason to use integer waves. You might expect that wave assignment statements would evaluate more quickly when an integer wave is the destination. This is not the case, however, because Igor still uses floating point for the assignment and only converts to integer for storage.

Date/Time Waves

Dates are represented in Igor date format - as the number of seconds since midnight, January 1, 1904. Dates before that are represented by negative values. Igor supports dates from the year -32768 to the year 32767.

As of Igor7, Igor uses the Gregorian calendar convention in which there is no year 0. The year before year 1 is year -1.

A date can not be accurately stored in the data values of a single precision or integer wave. Make sure to use double precision to store dates and times.

You must set the data units of a wave containing date or date/time data to "dat". This tells Igor that the wave contains date/time data and affects the default display of axes in graphs and columns in tables.

The following example illustrates the use of date/time data. First we create some date/time data and view it in a table:

```
Make/D/N=10/O testDate = date2secs(2011,4,p+1)
SetScale d 0, 0, "dat", testDate // Tell Igor this wave stores date/time data
```

We used `SetScale d` to set the data units of the wave to "dat".

Next we view the wave in a table:

```
Edit testDate
ModifyTable width(testDate)=120  // Make column wider
```

The data is displayed as dates but it is stored as numbers - specifically the number of seconds since January 1, 1904. We can see this by changing the column format:

```
ModifyTable format=1             // Display as integer
```

Now we return to date format:

```
ModifyTable format(testDate)=6   // Display as date again
```

Next we create some time data. This wave will not store dates and therefore does not need to be double-precision:

```
Make/N=10/O testTime = 3600*p    // Data is stored in seconds
AppendToTable testTime
```

Now we create a date/time wave by adding the time data to the date data. Since this wave will store dates it must be double-precision and must have "dat" data units. We accomplish this by using the Duplicate operation to duplicate the original date wave:

```
Duplicate/O testDate, testDateTime
AppendToTable testDateTime
ModifyTable width(testDateTime)=120    // Make wider
```

Igor displays the date/time wave in date format because it has "dat" units and all of the time values are 00:00:00. We now change the column format to date/time:

```
ModifyTable format(testDateTime)=8     // Set column to date/time format
```

Finally, we add the time:

```
testDateTime = testDateTime + testTime // Add time to date
```

To check the data type of your waves, use the info pane in the Data Browser. The data type shown for date/time waves should be “Double Float 64 bit”. If not, use Data→Redimension Waves to redimension as double-precision.

So far we have looked at storing dates in the data of a wave. Typically such a date wave is used to supply the X wave of an XY pair. If your data is waveform in nature, you would store date data in the X values of a wave treated as a waveform. For example:

```
Make/N=100 wave0 = sin(p/8)
SetScale/P x date2secs(2011,4,1), 60*60*24, "dat", wave0
Display wave0
Edit wave0.id; ModifyTable format(wave0.x)=6
```

Here the SetScale command is used to set the X scaling and units of the wave, not the data units as before. In this case, the wave does not need to be double-precision because Igor always calculates X values using double-precision regardless of the wave's data type.

Text Waves

Text waves are just like numeric waves except they contain text rather than numbers. Like numeric waves, text waves can have one to four dimensions.

To create a text wave:

- Type anything but a number into the first unused cell of a table.
- Import data from a delimited text file that contains nonnumeric columns.
- Use the **Make** operation with the /T flag.

You can use the Make Waves dialog to generate text waves by choosing Text from the Type pop-up menu. Most often you will create text waves by entering text in a table. See **Using a Table to Create New Waves** on page II-239 for more information.

You can store anything in an element of a text wave. There is no length limit. You can edit text waves in a table or assign values to the elements of a text wave using a wave assignment statement.

You can use text waves in category plots, to automatically label individual data points in a graph (use markers mode and choose a text wave via the marker pop-up menu) and for storing notes in a table. Programmers may find that text waves are handy for storing a collection of diverse data, such as inputs to or outputs from a complex Igor procedure.

Here is how you can create and initialize text waves on the command line:

```
Make/T textWave= {"First element","Second and last element"}
```

To see the text wave, create a table:

```
Edit textWave
```

Now you can try some wave assignment statements and see the result in the table:

```
textWave[2] = {"Third element"}      // Create new row
textWave += "*"                      // Append asterisk to each point
textWave = "*" + textWave            // Prepend asterisk to each point
```

Text Wave Text Encodings

This section is of concern only if you store non-ASCII data in a text wave.

Igor interprets the bytes stored in a text wave in light of the wave's text content text encoding setting. This setting will be UTF-8 for waves created in Igor7 or later and, typically, MacRoman or Windows-1252 for waves created in previous versions.

Igor7 and later use Unicode, in the form of UTF-8, as the working text encoding. When you access the cont ents of a text wave element or store text in a text wave element, Igor does whatever text encoding conversion is required. For example, assume that macRomanTextWave is a wave created in Igor6 on Macintosh. Then:

```
String str = macRomanTextWave[0]       // Igor converts from MacRoman to UTF-8
str = "Area = 2πr"                     // π is a non-ASCII character
macRomanTextWave[0] = str              // Igor converts from UTF-8 to MacRoman
Make/O/T utf8TextWave                  // New waves use UTF-8 text encoding
utf8TextWave = macRomanTextWave        // Igor converts from MacRoman to UTF-8
macRomanTextWave = utf8TextWave        // Igor converts from UTF-8 to MacRoman
```

For further discussion, see **Text Encodings** on page III-459 and **Wave Text Encodings** on page III-472.

Using Text Waves to Store Binary Data

While a numeric wave stores a fixed number of bytes in each element, a text wave has the ability to store a different number of bytes in each point. This makes text waves handy for storing a variable number of variable-length blobs. This is something that an advanced Igor programmer might do in a sophisticated package of procedures.

If you do this, you need to mark the text wave as containing binary data. Otherwise Igor will try to interpret it as text, leading to errors. For details on this issue, see **Text Waves Containing Binary Data** on page III-475.

Wave Text Encoding Properties

Igor Pro 7 uses Unicode internally. Older versions of Igor used non-Unicode text encodings such as MacRoman, Windows-1252 and Shift JIS.

Igor Pro 7 must convert from the old text encodings to Unicode when opening old files. It is not always possible to get this conversion right. You may get incorrect characters or receive errors when opening files containing non-ASCII text.

For a discussion of these issues, see **Text Encodings** on page III-459 and **Wave Text Encodings** on page III-472.

Home Versus Shared Waves

A wave that is stored in a packed experiment file, or in the disk folder corresponding to its data folder for an unpacked experiment, is a "home" wave. Otherwise it is a "shared" wave.

Home waves are typically intended for use by their owning experiment only. Shared waves are typically intended for use by by multiple experiments.

When you create a wave in a packed experiment, it saved by default in the packed experiment file and is a home wave. It becomes a shared wave only if you explicitly save it to a standalone file.

When you create a wave in an unpacked experiment, it is saved by default in the disk folder corresponding to its data folder and is a home wave. For waves in root, the disk folder is the experiment folder. For waves in other data folders, the disk folder is a subfolder of the experiment folder. The wave becomes a shared wave only if you explicitly save it to a standalone file outside of its default disk folder.

When you save a packed experiment as unpacked, home waves are stored in their default disk folder in the experiment folder.

When you save an unpacked experiment as packed, home waves are saved in the packed experiment file.

You can use the Data Browser to determine if a wave is shared. For shared waves, the Data Browser info pane shows the path and file name of the wave file on disk. For home waves this information is omitted.

You can convert a shared wave to a home wave by adopting it. See **Adopting Files** on page II-24 for details.

Wave Properties

Here is a complete list of the properties that Igor stores for each wave.

Property	Comment
Name	Used to reference the wave from commands and dialogs. 1 to 255 bytes. Standard names start with a letter. May contain letters, numbers or underscores. Prior to Igor Pro 8.00, wave names were limited to 31 bytes. If you use long wave names, your wave files and experiments will require Igor Pro 8.00 or later. Liberal names may contain almost any character but must be enclosed in single quotes. See **Wave Names** on page II-65. The name is assigned when you create a wave. You can use the **Rename** operation (see page V-796) to change it.
Data type	A numeric, text or reference data type. See **Wave Data Types** on page II-66. Set when you create a wave. Use the **Redimension** operation (see page V-788) to change it.
Length	Number of data points in the wave. Also, size of each dimension for multidimensional waves. Set when you create a wave. Use the **Redimension** operation (see page V-788) to change it.
X scaling (x0 and dx)	Used to compute X values from point numbers. Also Y, Z and T scaling for multidimensional waves. The X value for point p is computed as X = x0 + p*dx. Set by **SetScale** operation (see page V-853).
X units	Used to auto-label axes. Also Y, Z and T units for multidimensional waves. Set by **SetScale** operation (see page V-853).
Data units	Used to auto-label axes. Set by **SetScale** operation (see page V-853).
Data full scale	For documentation purposes only. Not used. Set by **SetScale** operation (see page V-853).
Note	Holds arbitrary text related to wave. Set by **Note** operation (see page V-694) or via the Data Browser. Readable via **note** function (see page V-694).
Dimension labels	Holds a label up to 255 bytes in length for each dimension index and for each dimension. See **Dimension Labels** on page II-93. Prior to Igor Pro 8.00, dimension labels were limited to 31 bytes. If you use long dimension labels, your wave files and experiments will require Igor Pro 8.00 or later.

Property	Comment
Dependency formula	Holds right-hand expression if wave is dependent. Set when you execute a dependency assignment using := or the **SetFormula** operation (see page V-847). Cleared when you do an assignment using plain =.
Creation date/time	Date & time when wave was created.
Modification date/time	Date & time when wave was last modified.
Lock	Wave lock state. A locked wave can not be modified. Set by **SetWaveLock** operation (see page V-858).
Source folder	Identifies folder containing wave’s source file, if any.
File name	Name of wave’s source file, if any.
Text encodings	See **Wave Text Encodings** on page III-472.

Chapter

II-6

Multidimensional Waves

Overview

Chapter II-5, **Waves**, concentrated on one-dimensional waves consisting of a number of rows. In Chapter II-5, **Waves**, the rows were referred to as “points” and the symbol p stood for row number, which was called “point number”. Scaled row numbers were called X values and were represented by the symbol x.

This chapter now extends the concepts from Chapter II-5, **Waves**, to waves of up to four dimensions by adding the column, layer and chunk dimensions. The symbols q, r and s stand for column, layer and chunk numbers. Scaled column, layer and chunk numbers are called Y, Z and T values and are represented by the symbols y, z and t.

We call a two-dimensional wave a “matrix”; it consists of rows (the first dimension) and columns (the second dimension). After two dimensions the terminology becomes a bit arbitrary. We call the next two dimensions “layers” and “chunks”.

Here is a summary of the terminology:

Dimension Number	0	1	2	3
Dimension Name	row	column	layer	chunk
Dimension Index	p	q	r	s
Scaled Dimension Index	x	y	z	t

Each element of a 1D wave has one index, the row index, and one data value.

Each element of a 2D wave has two indices, the row index and the column index, and one data value.

Each element of a 3D wave has three indices (row, column, layer) and one data value.

Each element of a 4D wave has four indices (row, column, layer, chunk) and one data value.

Creating Multidimensional Waves

Multidimensional waves can be created using the Make operation:

```
Make/N=(numRows,numColumns,numLayers,numChunks) waveName
```

When making an N-dimensional wave, you provide N values to the /N flag. For example:

```
// Make a 1D wave with 20 rows (20 points total)
Make/N=20 wave1

// Make a matrix (2D) wave with 20 rows and 3 columns (60 elements total)
Make/N=(20,3) wave2
```

The Redimension operation’s /N flag works the same way.

```
// Change both wave1 and wave2 so they have 10 rows and 4 columns
Redimension/N=(10,4) wave1, wave2
```

The operations InsertPoints and DeletePoints take a flag (/M=*dimensionNumber*) to specify the dimension into which elements are inserted. For example:

```
InsertPoints/M=1 2,5,wave2    //M=1 means column dimension
```

This command inserts 5 new columns in front of column number 2. If the /M=1 had been omitted or if /M=0 had been used then 5 new rows would have been inserted in front of row number 2.

You can also create multidimensional waves using the Make operation with a list of data values. For example:

```
// Create a 1D wave consisting of a single column of 3 rows
Make wave1 = {1,2,3}
```

```
// Creates a 2D wave consisting of 3 rows and 2 columns
Make wave2 = {{1,2,3},{4,5,6}}
```

See **Lists of Values** on page II-78 for further discussion of the use of curly braces.

The Duplicate operation can create an exact copy of a multidimensional wave or, using the /R flag, extract a subrange. Here is the syntax of the /R flag:

```
Duplicate/R=[startRow,endRow][startCol,endCol] and so on...
```

You can use the character * for any end field to specify the last element in the given dimension or you can just omit the end field. You can also specify just [] to include all of a given dimension. If the source wave has more dimensions than you specify in the /R flag, then all of those dimensions are copied. For example:

```
// Make a 3D wave to play with
Make/N=(5,4,3) wave3A = p + 10*q + 100*r

// Duplicate rows 1 through 2, columns 2 through the end, and all layers
Duplicate/R=[1,2][2,*] wave3A, wave3B     // 2 rows, 2 columns, 3 layers

// Create a 3D wave consisting of all rows of column 2, layer 0
Duplicate/R=[][2,2][0,0] wave3A, wave3C   // 5 rows, 1 column, 1 layer
```

Igor considers wave3C to be 3 dimensional wave and not 1 dimensional, even though it consists of just one column of data, because the number of columns and layers are greater than zero. This is a subtle distinction and can cause confusion. For example, you may think you have extracted a 1D wave from a 3D object but you will find that wave3C will not show up in dialogs where 1D waves are required.

You can turn the 3D wave wave3C into a 1D wave using the following command:

```
Redimension/N=(-1,0) wave3C
```

The -1 value does not to change the number of rows whereas the 0 value for the number of columns indicates that there are no dimensions past rows (in other words, no columns, layers or chunks).

Programmer Notes

For historical reasons, you can treat the symbols x and p like global variables, meaning that you can store into them as well as retrieve their values by referencing them. But this serves no purpose and is not recommended.

Unlike x and p, y, z, t, q, r and s act like functions and you can’t store into them.

Here are some functions and operations that are useful in programming with multidimensional waves:

```
DimOffset, DimDelta, DimSize
FindDimLabel, SetDimLabel, GetDimLabel
```

Dimension Labels

A dimension label is a name associated with a dimension (rows, columns, layers or chunks) or with a specific dimension index (row number, column number, layer number or chunk number).

Dimension labels are primarily an aid to the Igor procedure programmer when dealing with waves in which certain elements have distinct purposes. Dimension labels can be set when loading from a file, and can be displayed, created or edited in a table (see **Showing Dimension Labels** on page II-235).

You can give names to individual dimension indices in multidimensional or 1D waves. For example, if you have a 3 column wave, you can give column 0 the name “red”, column 1 the name “green” and column 2 the name “blue”. You can use the names in wave assignments in place of literal numbers. To do so, you use the % symbol in front of the name. For example:

```
wave2D[][%red] = wave2D[p][%green]     //Set red column equal to green column
```

To create a label for a given index of a given dimension, use the SetDimLabel operation.

For example:

```
SetDimLabel 1, 0, red, wave2D
```

1 is the dimension number (columns), 0 is the dimension index (column 0) and red is the label.

The function GetDimLabel returns a string containing the name associated with a given dimension and index. For example:

```
Print GetDimLabel(wave2D,1,0)
```

prints “red” into the history area.

The FindDimLabel function returns the index value associated with the given label. It returns the special value -2 if the label is not found. This function is useful in user-defined functions so that you can use a numeric index instead of a dimension label when accessing a wave in a loop. Accessing wave data using a numeric index is much faster than using a dimension label.

In addition to setting the name for individual dimension index values, you can set the name for an entire dimension by using an index value of -1. For example:

```
SetDimLabel 1, -1, ColorComponents, wave2D
```

This sets the label for the columns dimension to “ColorComponents”. This label appears in a table if you display dimension labels.

You can copy dimension labels from one dimension to another or from one wave to another using **CopyDimLabels**.

Dimension labels can contain up to 255 bytes and may contain spaces and other normally illegal characters if you surround the name in single quotes or if you use the $ operator to convert a string expression to a name. For example:

```
wave[%'a name with spaces']
wave[%$"a name with spaces"]
```

Dimension labels have the same characteristics as object names. See **Object Names** on page III-501 for a discussion of object names in general.

Long Dimension Labels

Prior to Igor8, wave dimension labels were limited to 31 bytes. If you create dimension labels of length 31 bytes or fewer, waves saved by Igor8 or later are saved in a format that is compatible with Igor7 or before. If you create a dimension label longer than 31 bytes, that wave is saved in a format that is incompatible with Igor7 and before.

If you attempt to save an Igor binary wave file or an experiment file that has waves with long dimension labels, Igor displays a warning dialog telling you that the experiment will require Igor Pro 8.00 or later. The warning dialog is presented only when you save an Igor binary wave file or experiment interactively, not if you save it programmatically using SaveExperiment. You can suppress the dialog by clicking the "Do not show this message again" checkbox.

Graphing Multidimensional Waves

You can easily view two-dimensional waves as images and as contour plots using Igor’s built-in operations. See Chapter II-15, **Contour Plots**, and Chapter II-16, **Image Plots**, for further information about these types of graphs. You can also create waterfall plots where each column in the matrix wave corresponds to a separate trace in the waterfall plot. For more details, see **Waterfall Plots** on page II-326.

Additional facilities for displaying multi-dimensional waves in Igor Pro are provided by the Gizmo extension, which create surface plots, slices through volumes and many other 3D plots. To get started with Gizmo, see **3D Graphics** on page II-405.

It is possible to graph a subset of a wave, including graphing rows or columns from a multidimensional wave as traces. See **Subrange Display** on page II-321 for details.

Analysis on Multidimensional Waves

Igor Pro includes the following capabilities for analysis of multidimensional data:

- Multidimensional waveform arithmetic
- Matrix math operations
- Image processing
- Multidimensional Fast Fourier Transform
- The **MatrixOp** operation

There are many analysis operations for 1D data that we have not yet extended to support multiple dimensions. Multidimensional waves do not appear in dialogs for these operations. If you invoke them on multidimensional waves from the command line or from an Igor procedure, Igor treats the multidimensional waves as if they were 1D. For example, the Smooth operation treats a 2D wave consisting of n rows and m columns as if it were a 1D wave with n*m rows. In some cases the operation will be useful. In other cases, it will make no sense.

Multidimensional Wave Indexing

You can use multidimensional waves in wave expressions and assignment statements just as you do with 1D waves (see **Indexing and Subranges** on page II-76). To specify a particular element of a 4D wave, use the syntax:

`wave[`*`rowIndex`*`][`*`columnIndex`*`][`*`layerIndex`*`][`*`chunkIndex`*`]`

Similarly, to specify an element of a 4D wave using *scaled* dimension indices, use the syntax:

`wave(`*`xIndex`*`)(`*`yIndex`*`)(`*`zIndex`*`)(`*`tIndex`*`)`

To index a 3D wave, omit the chunk index. To index a 2D wave, omit the layer and chunk indices.

rowIndex is the number, starting from zero, of the row of interest. It is an unscaled index. *xIndex* is simply the row index, offset and scaled by the wave's X scaling property, which you set using the SetScale operation (Change Wave Scaling in Data menu).

Using scaled indices you can access the wave's data using its natural units. You can use unscaled or scaled indices, whichever is more convenient. column/Y, layer/Z and chunk/T indices are analogous to row/X indices.

Using bracket notation tells Igor that the index you are supplying is an unscaled dimension index. Using parenthesis notation tells Igor that you are supplying a scaled dimension index. You can even mix the bracket notation with parenthesis notation.

Here are some examples:

```
Make/N=(5,4,3) wave3D = p + 10*q + 100*r
SetScale/I x, 0, 1, "", wave3D
SetScale/I y, -1, 1, "", wave3D
SetScale/I z, 10, 20, "", wave3D
Print wave3D[0][1][2]
Print wave3D(0.5)[2](15)
```

The first Print command prints 210, the value in row 0, column 1 and layer 2. The second Print command prints 122, the value in row 2 (where x=0.5), column 2 and layer 1 (where z=15).

Since wave3D has three dimensions, we do not, and must not, specify a chunk index.

There is one important difference between wave access using 1D waves versus multidimensional waves. For 1D waves alone, Igor performs linear interpolation when the specified index value, whether scaled or unscaled, falls between two points. For multidimensional waves, Igor returns the value of the element whose indices are closest to the specified indices.

When a multidimensional wave is the destination of a wave assignment statement, you can specify a subrange for each dimension. You can specify an entire dimension by using []. For example:

```
wave3D[2][][1,2] = 3
```

This sets row 2 of all columns and layers 1 and 2 to the value 3.

Note that indexing of the form [] (entire dimension) or [1,2] (range of a dimension) can be used on the left-hand side only. This is because the indexing on the left side determines which elements of the destination are to be set whereas indexing on the right side identifies a particular element in the source which is to contribute to a particular value in the destination.

Multidimensional Wave Assignment

As with one-dimensional waves, you can assign a value to a multidimensional wave using a wave assignment statement. For example:

```
Make/O/N=(3,3) wave0_2D, wave1_2D, wave2_2D
wave1_2D = 1.0; wave2_2D = 2.0
wave0_2D = wave1_2D / wave2_2D
```

The last command sets all elements of wave0_2D equal to the quotient of the corresponding elements of wave1_2D and wave2_2D.

Important: Wave assignments as shown in the above example where waves on the right-hand side do not include explicit indexing are defined only when all waves involved have the same dimensionality. The result of the following assignment is undefined and may produce surprising results.

```
Make/O/N=(3,3) wave33
Make/O/N=(2,2) wave22
wave33 = wave22
```

Whenever waves of mismatched dimensionality are used you should specify explicit indexing as described next.

In a wave assignment, Igor evaluates the right-hand side one time for each element specified by the left-hand side. During this evaluation, the symbols p, q, r and s take on the value of the row, column, layer and chunk, respectively, of the element in the destination for which a value is being calculated. For example:

```
Make/O/N=(5,4,3) wave3D = 0
Make/O/N=(5,4) wave2D = 999
wave3D[][][0] = wave2D[p][q]
```

This stores the contents of wave2D in layer 0 of wave3D. In this case, the destination (wave3D) has three dimensions, so p, q and r are defined and s is undefined. The following discussion explains this assignment and presents a way of thinking about wave assignments in general.

The left-hand side of the assignment specifies that Igor is to store a value into all rows (the first []) and all columns (the second []) of layer zero (the [0]) of wave3D. For each of these elements, Igor will evaluate the right-hand side. During the evaluation, the symbol p will return the row number of the element in wave3D that Igor is about to set and the symbol q will return the column number. The symbol r will have the value 0 during the entire process. Thus, the expression wave2D[p][q] will return a value from wave2D at the corresponding row and column in wave3D.

As the preceding example shows, wave assignments provide a way of transferring data between waves. With the proper indexing, you can build a 2D wave from multiple 1D waves or a 3D wave from multiple

2D waves. Conversely, you can extract a layer of a 3D wave into a 2D wave or extract a column from a 2D wave into a 1D wave. Here are some examples that illustrate these operations.

```
// Build a 2D wave from multiple 1D waves (waveforms)
Make/O/N=5, wave0=p, wave1=p+1, wave2=p+2    // 1D waveforms
Make/O/N=(5,3) wave0_2D
wave0_2D[][0] = wave0[p]      // Store into all rows, column 0
wave0_2D[][1] = wave1[p]      // Store into all rows, column 1
wave0_2D[][2] = wave2[p]      // Store into all rows, column 2

// Build a 3D wave from multiple 2D waves
Duplicate/O wave0_2D, wave1_2D; wave1_2D *= -1
Make/O/N=(5,3,2) wave0_3D
wave0_3D[][][0]= wave0_2D[p][q]  // Store into all rows/cols, layer 0
wave0_3D[][][1]= wave1_2D[p][q]  // Store into all rows/cols, layer 1

// Extract a layer of a 3D wave into a 2D wave
wave0_2D = wave0_3D[p][q][0]     // Extract layer 0 into 2D wave

// Extract a column of a 2D wave into a 1D wave
wave0 = wave0_2D[p][0]           // Extract column 0 into 1D wave
```

To understand assignments like these, first figure out, by looking at the indexing on the left-hand side, which elements of the destination wave are going to be set. (If there is no indexing on the left then all elements are going to be set.) Then think about the range of values that p, q, r and s will take on as Igor evaluates the right-hand side to get a value for each destination element. Finally, think about how these values, used as indices on the right-hand side, select the desired source element.

To create such an assignment, first determine the indexing needed on the left-hand side to set the elements of the destination that you want to set. Then think about the values that p, q, r and s will take on. Then use p, q, r and s as indices to select a source element to be used when computing a particular destination element.

Here are some more examples:

```
// Extract a row of a 2D wave into a 1D wave
Make/O/N=3 row1
row1 = wave0_2D[1][p]         // Extract row 1 of the 2D wave
```

In this example, the *row* index (p) for the destination is used to select the source *column* while the source row is always 1.

```
// Extract a horizontal slice of a 3D wave into a 2D wave
Make/O/N=(2,3) slice_R2       // Slice consisting of all of row 2
slice_R2 = wave0_3D[2][q][p]  // Extract row 2, all columns/layers
```

In this example, the row data for slice_R2 comes from the layers of wave0_3D because the p symbol (row index) is used to select the layer in the source. The column data for slice_R2 comes from the columns of wave0_3D because the q symbol (column index) is used to select the column in the source. All data comes from row 2 in the source because the row index is fixed at 2.

You can store into a range of elements in a particular dimension by using a range index on the left-hand side. As an example, here are some commands that shift the horizontal slices of wave0_3D.

```
Duplicate/O wave0_3D, tmp_wave0_3D
wave0_3D[0][][] = tmp_wave0_3D[4][q][r]
wave0_3D[1,4][][] = tmp_wave0_3D[p-1][q][r]
KillWaves tmp_wave0_3D
```

The first assignment transfers the slice consisting of all elements in row 4 to row zero. The second assignment transfers slice n-1 to slice n. To understand this, realize that as p goes from 1 to 4, p-1 indexes into the preceding row of the source.

Vector (Waveform) to Matrix Conversion

Occasionally you will may need to convert between a vector form of data and a matrix form of the same data values. For example, you may have a vector of 16 data values stored in a waveform named sixteenVals that you want to treat as a matrix of 8 rows and 2 columns.

Though the Redimension operation normally doesn't move data from one dimension to another, in the special case of converting to or from a 1D wave Redimension will leave the data in place while changing the dimensionality of the wave. You can use the command:

```
Make/O/N=16 sixteenVals          // 1D
Redimension/N=(8,2) sixteenVals  // Now 2D, no data lost
```

to accomplish the conversion. When redimensioning from a 1D wave, columns are filled first, then layers, followed by chunks. Redimensioning from a multidimensional wave to a 1D wave doesn't lose data, either.

Matrix to Matrix Conversion

To convert a matrix from one matrix form to another, don't directly redimension it to the desired form. For instance, if you have a 6x6 matrix wave, and you would like it to be 3x12, you might try:

```
Make/O/N=(6,6) thirtySixVals        // 2D
Redimension/N=(3,12) thirtySixVals  // This loses the last three rows
```

But Igor will first shrink the number of rows to 3, discarding the data for the last three rows, and then add 6 columns of zeroes.

The simplest way to work around this is to convert the matrix to a 1D vector, and then convert it to the new matrix form:

```
Make/O/N=(6,6) thirtySixVals        // 2D
Redimension/N=36 thirtySixVals      // 1D vector preserves the data
Redimension/N=(3,12) thirtySixVals  // Data preserved
```

Multidimensional Decimation

You can create reduced-size representations of 2D and 3D waves.

The ImageInterpolate operation with the pixelate keyword can create a reduced-size representation of a 2D wave, or of the layers of a 3D wave, by averaging blocks of rows and columns. Use ImageInterpolate/PXSZ={*nx*,*ny*} pixelate where *nx* and *ny* are the number of rows and columns respectively to average.

The ImageInterpolate operation with the pixelate keyword can also create a reduced-size representation of a 3D wave by averaging 3D subsections. Use ImageInterpolate/PXSZ={*nx*,*ny*,*nz*} pixelate where *nx*, *ny*, and *nz* are the number of rows, columns, and layers respectively to average.

The ReduceMatrixSize function, provided by the "Reduce Matrix Size.ipf" WaveMetrics procedure file which was added in Igor Pro 9.00, creates a reduced-size representation of a 2D wave, or of the layers of a 3D wave, by sampling rows and columns or by averaging blocks of rows and columns. It uses a wave assignment for sampling and ImageInterpolate pixelate for averaging and provides a convenient wrapper for both techniques. To use ReduceMatrixSize, activate it as a global procedure file or #include it using:

```
#include <Reduce Matrix Size>
```

See **Decimation** on page III-135 for a discussion of decimating 1D waves.

Multidimensional Fourier Transform

Igor's FFT and IFFT routines are mixed-radix and multidimensional. Mixed-radix means you do not need a power of two number of data points (or dimension size).

There is only one restriction on the dimensions of a wave: when performing a forward FFT on real data, the number of rows must be even. Note, however, that if a given dimension size is a prime number or contains a large prime in its factorization, the speed will be reduced to that of a normal Discrete Fourier Transform (i.e., the number of operations will be on the order of N^2 rather than N*log(N)).

For more information about the FFT, see **Fourier Transforms** on page III-270 and the **FFT** operation on page V-222.

Treating Multidimensional Waves as 1D

Sometimes it is useful to treat a multidimensional wave as if it were 1D. For example, if you want to know the number of NaNs in a 2D wave, you can pass the wave to **WaveStats**, even though WaveStats treats its input as 1D.

In other cases, you need to understand the layout of data in memory in order to treat a multidimensional wave as 1D.

A 2D wave consists of some number of columns. In memory, the data is laid out column-by-column. This is called "column-major order". In column-major order, consecutive elements of a given column are contiguous in memory.

For example, execute:

```
Make/N=(2,2) mat = p + 2*q
Edit mat
```

The wave mat consists of two columns. The first column contains the values 0 and 1. The second column contains the values 2 and 3.

You can pass this wave to an operation or function that is not multidimensional-aware and it will treat the wave as if it were one column containing 0, 1, 2, 3. For an example:

```
Print WaveMax(mat)       // Prints 3
```

Here is an example of using the knowledge of how a multidimensional wave is laid out in memory:

```
Function DemoMDAs1D()
   // Make a 2D wave
   Make/O/N=(5,3) mat = p + 10*q
   Variable numRows = DimSize(mat,0)

   // Find the sum of each column
   Variable numColumns = DimSize(mat,1)
   Make/O/N=(numColumns) Sums
   Sums = sum(mat, p*numRows, (p+1)*numRows-1)
   Edit mat, Sums
End
```

The statement

```
Sums = sum(mat, p* numRows, (p+1)* numRows-1)
```

passes the 2D wave mat to the **sum** function which is not multidimensional-aware. Because sum is not multidimensional-aware, it requires that we formulate the startX and endX parameters treating mat as if it were a 1D wave with the data arranged in column-major order.

You can also treat 3D and 4D waves as 1D. In a 3D wave, the data for layer n+1 follows the data for layer n. In a 4D wave, the data for chunk n+1 follows the data for chunk n.

Chapter

II-7

Numeric and String Variables

Overview

This chapter discusses the properties and uses of global numeric and string variables. For the fine points of programming with global variables, see **Accessing Global Variables and Waves** on page IV-65.

Numeric variables are double-precision floating point and can be real or complex. String variables can hold an arbitrary number of bytes. Igor stores all global variables when you save an experiment and restores them when you reopen the experiment.

Numeric variables or numeric expressions containing numeric variables can be used in any place where literal numbers are appropriate including as operands in assignment statements and as parameters to operations, functions, and macros.

When using numeric variables in operation flag parameters, you need parentheses. See **Reference Syntax Guide** on page V-15.

String variables or string expressions can be used in any place where strings are appropriate. String variables can also be used as parameters where Igor expects to find the name of an object such as a wave, variable, graph, table or page layout. For details on this see **Converting a String into a Reference Using $** on page IV-62.

In Igor7 or later, Igor assumes that the contents of string variables are encoded as UTF-8. If you store non-ASCII text in string variables created by Igor6 or before, you need to convert it for use in Igor7 or later. See **String Variables and Text Encodings** on page II-106 for details.

Creating Global Variables

There are 20 built-in numeric variables (K0 … K19), called system variables, that exist all the time. Igor uses these mainly to return results from the CurveFit operation. We recommend that you refrain from using system variables for other purposes.

All other variables are user variables. User variables can be created in one of two ways:

- Automatically in the course of certain operations
- Explicitly by the user, via the Variable/G and String/G operations

When you create a variable directly from the command line using the Variable or String operation, it is always global and you can omit the /G flag. You need /G in Igor procedures to make variables global. The /G flag has a secondary effect — it permits you to overwrite existing global variables.

Uses For Global Variables

Global variables have two properties that make them useful: globalness and persistence. Since they are global, they can be accessed from any procedure. Since they are persistent, you can use them to store settings over time.

Using globals for their globalness creates non-explicit dependencies between procedures. This makes it difficult to understand and debug them. Using a global variable to pass information from one procedure to another when you could use a parameter is bad programming and should be avoided except under rare circumstances. Consequently, you should use global variables when you need persistence.

A legitimate use of a global variable for its globalness is when you have a value that rarely changes and needs to be accessed by many procedures.

Variable Names

Variable names consist of 1 to 255 bytes. Only ASCII characters are allowed. The first character must be alphabetic. The remaining characters can be alphabetic, numeric or the underscore character. Names in Igor are case insensitive.

Prior to Igor Pro 8.00, variable names were limited to 31 bytes. If you use long variable names, your experiments will require Igor Pro 8.00 or later.

Variable names must be standard names, not liberal names. See **Object Names** on page III-501 for details.

Variable names must not conflict with the names of other Igor objects, functions or operations.

You can rename a variable using the Rename operation, or the Rename Objects dialog via the Misc menu.

System Variables

System variables are built in to Igor. They are mainly provided for compatibility with older versions of Igor and are not recommended for general use. You can see a list of system variables and their values by choosing the Object Status item in the Misc menu.

There are 20 system variables named K0,K1...K19 and one named veclen. The K variables are used by the curve fitting operations.

System variables can be used only from the normal "main thread", not from premptive theads. For more about preemptive threads, see **ThreadSafe Functions and Multitasking** on page IV-329 and **Thread Data Environment** on page IV-330.

The `veclen` variable is present for compatibility reasons. In previous versions of Igor, it contained the default number of points for waves created by the Make operation. This is no longer the case. Make will always create waves with 128 points unless you explicitly specify otherwise using the /N=(<number of points>) flag.

Although the CurveFit operation stores results in the K variables, it does so only for compatibility reasons and it also creates user variables and waves to store the same results.

However, the CurveFit operation does use system variables for the purpose of setting up initial parameter guesses if you specify manual guess mode. You can also use a wave for this purpose if you use the kwCWave keyword. See the **CurveFit** operation on page V-124.

It is best to not rely on system variables unless necessary. Since Igor writes to them at various times, they may change when you don’t expect it.

The Data Browser does not display system variables.

System variables are stored on disk as single precision values so that they can be read by older versions of Igor. Thus, you should store values that you want to keep indefinitely in your own global variables.

User Variables

You can create your own global variables by using the `Variable/G` (see **Numeric Variables** on page II-104) and `String/G` operations (see **String Variables** on page II-105). Variables that you create are called “user variables” whether they be numeric or string. You can browse the global user variables by choosing the Object Status item in the Misc menu. You can also use the Data Browser window (Data menu) to view your variables.

Global user variables are mainly used to contain persistent settings used by your procedures.

Special User Variables

In the course of some operations, Igor automatically creates special user variables. For example, the CurveFit operation creates the user variable `V_chisq` and others to store various results generated by the curve fit. The names of these variables always start with the characters “`V_`” for numeric variables or “S_” for string variables. The meaning of these variables is documented along with the operations that generate them in Chapter V-1, **Igor Reference**.

In addition, Igor sometimes checks for `V_` variables that you can create to modify the default operation of certain routines. For example, if you create a variable with the name V_FitOptions, Igor will use that to control the CurveFit, FuncFit and FuncFitMD operations. The use of these variables is documented along with the operations that they affect.

When an Igor operation creates V_ and S_ variables, they are global if the operation was executed from the command line and local if the operation was executed in a procedure. See **Accessing Variables Used by Igor Operations** on page IV-123 for details.

Numeric Variables

You create numeric user variables using the `Variable` operation from the command line or in a procedure. The syntax for the Variable operation is:

```
Variable [flags] varName [=numExpr] [,varName [=numExpr]]...
```

There are three optional flags:

/C	Specifies complex variable.
/D	Obsolete. Used in previous versions to specify double-precision. Now all variables are double-precision.
/G	Specifies variable is to be global and overwrites any existing variable.

The variable is initialized when it is created if you supply the initial value with a numeric expression using `=numExpr`. If you create a numeric variable and specify no initializer, it is initialized to zero.

You can create more than one variable at a time by separating the names and optional initializers for multiple variables with a comma.

When used in a procedure, the new variable is local to that procedure unless the /G flag is used. When used on the command line, the new variable is always global.

Here is an example of a variable creation with initialization:

```
Variable v1=1.1, v2=2.2, v3=3.3*sin(v2)/exp(v1)
```

Since the /C flag was not specified, the data type of v1, v2 and v3 is double-precision real.

Since the /G flag was not specified, these variables would be global if you invoked the `Variable` operation directly from the command line or local if you invoked it in a procedure.

`Variable/G varname` can be invoked whether or not a variable of the specified name already exists. If it does exist as a variable, its contents are not altered by the operation unless the operation includes an initial value for the variable.

To assign a value to a complex variable, use the `cmplx()` function:

```
Variable/C cv1 = cmplx(1,2)
```

You can kill (delete) a global user variable using the Data Browser or the `KillVariables` operation. The syntax is:

```
KillVariables [flags] [variableName [,variableName]...]
```

There are two optional *flags*:

/A	Kills all global variables in the current data folder. If you use /A, omit variableName.
/Z	Doesn't generate an error if a global variable to be killed does not exist.

For example, to kill global variable `cv1` without worrying about whether it was previously defined, use the command:

```
KillVariables/Z cv1
```

Killing a global variable reduces clutter and saves a bit of memory. You can not kill a system variable or local variable.

To kill all global variables in the current data folder, use KillVariables/A/Z.

String Variables

You create user string variables by calling the `String` operation from the command line or in a procedure. The syntax is:

```
String [/G] strName [=strExpr] [,strName [=strExpr]... ]
```

The optional /G flag specifies that the string is to be global, and it overwrites any existing string variable.

The string variable is initialized when it is created if you supply the initial value with a string expression using =*strExpr*. If you create a string variable and specify no initializer it is initialized to the empty string ("").

When you call String from the command line or from a macro, the string variable is initialized to the specified initial value or to the empty string ("") if you provide no initial value.

When you declare a local string variable in a user-defined function, it is null (has no value) until you assign a value to it, via either the initial value or a subsequent assignment statement. Igor generates an error if you use a null local string variable in a user-defined function.

When you call String in a procedure, the new string is local to that procedure unless you include the /G flag. When you call String from the command line, the new string is always global.

You can create more than one string variable at a time by separating the names and optional initializers for multiple string variables with a comma.

Here is an example of variable creation with initialization:

```
String str1 = "This is string 1", str2 = "This is string 2"
```

Since /G was not used, these strings would be global if you invoked String directly from the command line or local if you invoked it in a procedure.

`String/G strName` can be invoked whether or not a variable of the given name already exists. If it does exist as a string, its contents are not altered by the operation unless the operation includes an initial value for the string.

You can kill (delete) a global string using the Data Browser or the `KillStrings` operation. The syntax is:

```
KillStrings [flags] [stringName [,stringName ]...]
```

There are two optional *flags*:

A	Kills all global strings in the current data folder. If you use /A, omit stringName.
/Z	Doesn't generate an error if a global string to be killed does not exist.

For example, to kill global string `myGlobalString` without worrying about whether it was previously defined, use the command:

```
KillStrings/Z myGlobalString
```

Killing a string reduces clutter and saves a bit of memory. You can not kill a local string.

To kill all global strings in the current data folder, use KillStrings/A/Z.

Local and Parameter Variables in Procedures

You can create variables in procedures as parameters or local variables. These variables exist only while the procedure is running. They can not be accessed from outside the procedure and do not retain their values from one invocation of the procedure to the next. See **Local Versus Global Variables** on page IV-61 for more information.

String Variables and Text Encodings

Igor uses Unicode internally. Prior to Igor7, Igor used non-Unicode text encodings such as MacRoman, Windows-1252 and Shift JIS.

If you have string variables containing non-ASCII text in old experiments, they will be misinterpreted. For a discussion of this issues, see **String Variable Text Encodings** on page III-478.

Chapter

II-8

Data Folders

Overview

Using data folders, you can store your data within an experiment in a hierarchical manner. Hierarchical storage is useful when you have multiple sets of similar data. By storing each set in its own data folder, you can organize your data in a meaningful way and also avoid name conflicts.

Data folders contain four kinds of **data objects**:

- Waves
- Numeric variables
- String variables
- Other data folders

Igor's data folders are very similar to a computer's hierarchical disk file system except they reside wholly in memory and not on disk. This similarity can help you understand the concept of data folders but you should take care not to confuse them with the computer's folders and files.

Data folders are particularly useful when you conduct several runs of an experiment. You can store the data for each run in a separate data folder. The data folders can have names like "run1", "run2", etc., but the names of the waves and variables in each data folder can be the same as in the others. In other words, the information about which run the data objects belong to is encoded in the data folder name, allowing the data objects themselves to have the same names for all runs. You can write procedures that use the same wave and variable names regardless of which run they are working on.

Data folders are very handy for programmers who need to store private data between one invocation of their procedures and the next. This use of data folders is discussed under **Managing Package Data** on page IV-249.

The Data Browser window allows you to examine the data folder hierarchy. You display it by choosing Data→Data Browser.

One data folder is designated as the "current" data folder. Commands that do not explicitly target a particular data folder operate on the current data folder. You can see and set the current data folder using the Data Browser or using the **GetDataFolder** and **GetDataFolderDFR** functions and the **SetDataFolder** operation.

You can use the Data Browser not only to see the hierarchy and set the current data folder but also to:

- Create new data folders.
- Move, duplicate, rename and delete objects.
- Browse other Igor experiment files and load data from them into memory.
- Save a copy of data in the current experiment to an experiment file or folder on disk.
- See and edit the contents of variables, strings, and waves in the info pane by selecting an object
- See a simple plot of 1D or 2D waves by selecting one wave at a time in the main list while the plot pane is visible.
- See a simple plot of a wave while browsing other Igor experiments.
- See variable, string and wave contents by double-clicking their icons.
- See a simple histogram or wave statistics for one wave at a time.

A similar browser is used for wave selection in dialogs. For details see **Dialog Wave Browser** on page II-228.

Before using data folders, be sure to read **Using Data Folders** on page II-112.

Programmers should read **Programming with Data Folders** on page IV-169.

Data Folder Syntax

Data folders are named objects like other Igor objects such as waves and variables. Data folder names follow the same rules as wave names. See **Liberal Object Names** on page III-501.

Igor Pro's data folders use the colon character (:) to separate components of a path to an object.

A data folder named "root" always exists and contains all other data folders.

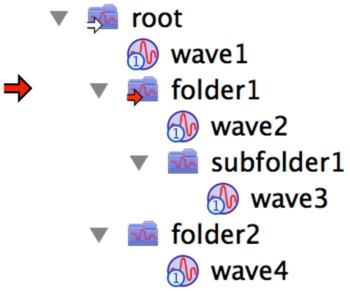
A given object can be specified in a command using:

- A full path
- A partial path
- Just the object name

The object name alone can only be used when the current data folder contains the object.

A full path starts with "root" and does not depend on the current data folder. A partial path starts with ":" and is relative to the current data folder.

Assume the data folder structure shown below, where the arrow indicates that folder1 is the current data folder.



Each of the following commands creates a graph of one of the waves in this hierarchy:

```
Display wave2
Display :subfolder1:wave3
Display root:folder1:subfolder1:wave3
Display ::folder2:wave4
```

The last example illustrates the rule that you can use multiple colons to walk back up the hierarchy: from folder1 (the current data folder), up one level, then down to folder2 and wave4. Here is another valid, though silly, example:

```
Display root:folder1:subfolder1:::folder2:wave4
```

Occasionally you need to specify a data folder itself rather than an object in a data folder. In that case, just leave off the object name. The path specification should therefore have a trailing colon. However, Igor will generally understand what you mean if you omit the trailing colon.

If you need to specify the current data folder, you can use just a single colon. For example:

```
KillDataFolder :
```

kills the current data folder, and all its contents, and then sets the current data folder to the parent of the current. Non-programmers might prefer to use the Data Browser to delete data folders.

Recall that the $ operator converts a string expression into a single name. Since data folders are named, the following is valid:

```
String df1 = "folder1", df2="subfolder1"
Display root:$(df1):$(df2):wave3
```

This is a silly example but the technique would be useful if df1 and df2 were parameters to a procedure.

Note that parentheses must be used in this type of statement. That is a result of the precedence of $ relative to :.

When used at the beginning of a path, the $ operator works in a special way and must be used on the entire path:

```
String path1 = "root:folder1:subfolder1:wave3"
Display $path1
```

When liberal names are used within a path, they must be in single quotes. For example:

```
Display root:folder1:'subfolder 1':'wave 3'
String path1 = "root:folder1:'subfolder 1':'wave 3'"
Display $path1
```

However, when a simple name is passed in a string, single quotes must not be used:

```
Make 'wave 1'
String name
name = "'wave 1'"       // Wrong.
name = "wave 1"         // Correct.
Display $name
```

Data Folder Operations and Functions

Most people will use the Data Browser to create, view and manipulate data folders. The following operations will be mainly used by programmers, who should read **Programming with Data Folders** on page IV-169.

NewDataFolder *path*
SetDataFolder *path*
KillDataFolder *path*
DuplicateDataFolder *srcPath*, *destPath*
MoveDataFolder *srcPath*, *destPath*
MoveString *srcPath, destPath*
MoveVariable *srcPath*, *destPath*
MoveWave *wave*, *destPath* [*newname*]
RenameDataFolder *path*, *newName*
Dir

The following are functions that are used with data folders.

GetDataFolder(*mode* [, *dfr*])
CountObjects(*pathStr*,*type*)
GetIndexedObjName(*pathStr*,*type*,*index*)
GetWavesDataFolder(*wave*,*mode*)
DataFolderExists(*pathStr*)
DataFolderDir(*mode*)

Data Folders Reference Functions

Programmers can utilize data folder references in place of paths. Data folder references are lightweight objects that refer directly to a data folder whereas a path, consisting of a sequence of names, has to be looked up in order to find the actual target folder.

Here are functions that work with data folder references:

GetDataFolder(*mode* [, *dfr*])
GetDataFolderDFR()
GetIndexedObjNameDFR(*dfr*, *type*, *index*)
GetWavesDataFolderDFR(*wave*)
CountObjectsDFR(*dfr*, *type*)
DataFolderRefChanges(*dfr*, *changeType*)
DataFolderRefStatus(*dfr*)

```
DataFolderRefsEqual(dfr1, dfr2)
NewFreeDataFolder()
```

For information on programming with data folder references, see **Data Folder References** on page IV-78.

Data Folders and Commands

Igor normally evaluates commands in the context of the current data folder. This means that, unless qualified with a path to a particular data folder, object names refer to objects in the current data folder. For example:

```
Function Test()
   Make wave1
   Variable/G myGlobalVariable
End
```

This function creates wave1 and myGlobalVariable in the current data folder. Likewise executing:

```
WaveStats wave1
```

operates on wave1 in the current data folder.

Data Folders and User-Defined Functions

To access global variables and waves from a user-defined function, you must first create an NVAR, SVAR or WAVE reference. These references are local objects that point to global objects. See **Accessing Global Variables and Waves** on page IV-65 for details.

Data Folders and Window Recreation Macros

Window recreation macros begin with the Window keyword. They are used to recreate graphs, tables, and other Igor windows and are explained under **Saving a Window as a Recreation Macro** on page II-47.

Window recreation macros are evaluated in the context of the root data folder. In effect, Igor sets the current data folder to root when the window macro starts and restores it when the window macro ends.

Macros that begin with the Macro or Proc keywords evaluate their commands in the context of the *current* data folder.

Evaluating window recreation macros this way ensures that a window is recreated correctly regardless of the current data folder, and provides some compatibility with window macros created with prior versions of Igor Pro which didn’t have data folders.

This means that object names within window recreation macros that don’t explicitly contain a data folder path refer to objects in the root data folder.

Data Folders and Assignment Statements

Wave and variable assignment statements are evaluated in the context of the data folder containing the wave or variable on the left-hand side of the statement:

```
root:subfolder:wave0 = wave1 + var1
```

is a shorter way of writing the equivalent:

```
root:subfolder:wave0 = root:subfolder:wave1 + root:subfolder:var1
```

This rule also applies to dependency formulae which use := instead of = as the assignment operator.

Data Folders and Controls

ValDisplay controls evaluate their value expression in the context of the root data folder.

SetVariable controls remember the data folder in which the controlled global variable exists, and continue to function properly when the current data folder is different from the controlled variable.

See Chapter III-14, **Controls and Control Panels**, for details about controls.

Data Folders and Traces

You cannot tell by looking at a trace in a graph which data folder it resides in.

The easiest way to find out what data folder a trace's wave resides in is to use the trace info help. Choose Graph→Show Trace Info Tags and then hover the mouse over the trace to get trace info.

Another method is to use the Modify Trace Appearance dialog. Double-click the trace to display the dialog. The Traces list shows the full data folder path for each trace.

Finally, you can create and examine the graph window recreation macro. See **Saving a Window as a Recreation Macro** on page II-47 for details.

Using Data Folders

You can use data folders for many purposes. Here are two common uses of data folders.

Hiding Waves, Strings, and Variables

Sophisticated Igor procedures may need a large number of global variables, strings and waves that aren't intended to be directly accessed by the user. The programmer who creates these procedures should keep all such items within data folders they create with unique names designed not to conflict with other data folder names.

Users of these procedures should leave the current data folder set to the data folder where their raw data and final results are kept, so that the procedure's globals and waves won't clutter up the dialog lists.

Programmers creating procedures should read **Managing Package Data** on page IV-249.

Separating Similar Data

One situation that arises during repeated testing is needing to keep the data from each test run separate from the others. Often the data from each run is very similar to the other runs, and may even have the same name. Without data folders you would need to choose new names after the first run.

By making one data folder for each test run, you can put all of the related data for one run into each folder. The data can use identical names, because other identically named data is in different data folders.

Using data folders also keeps the data from various runs from being accidently combined, since only the data in the current data folder shows up in the various dialogs or can be used in a command without a data folder name.

The WaveMetrics-supplied "Multipeak Fitting" example experiment's procedures work this way: they create data folders to hold separate peak curve fit runs and global state information.

Using Data Folders Example

This example will use data folders to:

- Load data from two test runs into separate data folders
- Create graphs showing each test run by itself
- Create a graph comparing the two test runs

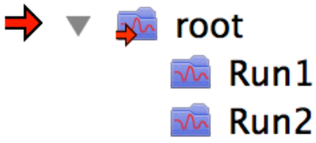
First we'll use the Data Browser to create a data folder for each test run.

Open the Data Browser, and set the current data folder to root by right-clicking the root icon and choosing Set as Current Data Folder.

Click the root data folder, and click the New Data Folder button. Enter "Run1" for the new data folder's name and click OK.

Click New Data Folder again. Enter “Run2” for the new data folder’s name and click OK.

The Data Browser window should look like this:



Now let’s load sample data into each data folder, starting with Run1.

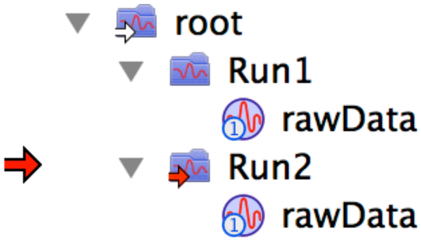
Set the current data folder to Run1, then choose Data→Load Waves→Load Delimited. Select the CSTATIN.ASH file from the Sample Data subfolder of the Learning Aids folder, and click Open. In the resulting Loading Delimited Text dialog, name the loaded wave “rawData” and click Load. We will pretend this data is the result of Run 1. Type “Display rawData” on the command line to graph the data.

Set the current data folder to Run2, and repeat the wave loading steps, selecting the CSTATIN.ASV file instead. In the resulting Loading Delimited Text dialog, name the loaded wave “rawData”. We will pretend this data is the result of Run 2. Repeat the “Display rawData” command to make a graph of this data.

Notice that we used the same name for the loaded data. No conflict exists because the other rawData wave is in another data folder.

In the Data Browser, set the current data folder to root.

In the Data Browser, uncheck the Variables and Strings checkboxes in the Display section. Open the Run1 and Run2 icons by clicking the disclosure icons next to them. At this point, the Data Browser should look something like this:



You can easily make a graph displaying both rawData waves to compare them better. Choose Windows-New Graph to display the New Graph dialog. Use the dialog wave browser controls (see **Dialog Wave Browser** on page II-228) to select both `root:Run1:rawData` and `root:Run2:rawData`. Click Do It.

You can change the current data folder to anything you want and the graphs will continue to display the same data. Graphs remember which data folder the waves belong to, and so do graph recreation macros. This is often what you want, but not always.

Suppose you have many test runs in your experiment, each safely tucked away in its own data folder, and you want to “visit” each test run by looking at the data using a single graph which displays data from the test run’s data folder only. When you visit another test run, you want the graph to display data from that other data folder only.

Additionally, suppose you want the graph characteristics to be the same (the same axis labels, annotations, line styles and colors, etc.). You could:

- Create a graph for the first test run
- Kill the window, and save the graph window macro.
- Edit the recreation macro to reference data in another data folder.
- Run the edited recreation macro.

The recreated graph will have the same appearance, but use the data from the other data folder. The editing usually involves changing a command like:

```
SetDataFolder root:Run1:
```

to:

```
SetDataFolder root:Run2:
```

If the graph displays waves from more than one data folder, you may need to edit commands like:

```
Display rawData,::Run1:rawData
```

as well.

However, there is another way that doesn't require you to edit recreation macros: use the ReplaceWave operation to replace waves in the graph with waves from the other folder.

- Set the current data folder to the one containing the waves you want to view
- Activate the desired graph
- Execute in the command line:

```
ReplaceWave allinCDF
```

This replaces all the waves in the graph with identically named waves from the current data folder, if they exist. See the **ReplaceWave** for details. You can choose Graph→Replace Wave to display a dialog in which you can generate ReplaceWave commands interactively.

Though we have only one wave per data folder, we can try it out:

- Set the current data folder to Run1.
- Select the graph showing data from Run2 only (CSTATIN.ASV).
- Execute in the command line:

```
ReplaceWave allinCDF
```

The graph is updated to show the rawData wave from Run1.

For another Data Folder example, choose File→Example Experiments→Tutorials→Data Folder Tutorial.

The Data Browser

The Data Browser lets you navigate through the data folder hierarchy, examine properties of waves and values of numeric and string variables, load data objects from other Igor experiments, and save a copy of data from the current experiment to an experiment file or folder on disk.

To open the browser choose Data Browser from the Data menu.

The user interface of the Data Browser is similar to that of the computer desktop. The basic Igor data objects (variables, strings, waves and data folders) are represented by icons and arranged in the main list based on their hierarchy in the current experiment. The browser also sports several buttons that provide you with additional functionality.

The main components of the Data Browser window are:

- The main list which displays icons representing data folders, waves, and variables
- The Display checkboxes which control the types of objects displayed in the main list
- The buttons for manipulating the data hierarchy
- The info pane which displays information about the selected item
- The plot pane which displays a graphical representation of the selected wave

The Main List

The main list occupies most of the Data Browser when it is first invoked.

At the top of the data tree is the root data folder which by default appears expanded. By double-clicking a data folder icon you can change the display so that the tree is displayed with your selection as the top data folder instead of root. You can use the pop-up menu above the main list to change the current top data folder to another data folder in the hierarchy.

Following the top data folder are all the data objects that it contains. Objects are grouped by type and by default are listed in creation order. You can change the sort order using the gear icon. You can filter the list using the filter textbox.

You can select data objects with the mouse. You can select multiple contiguous data objects by shift-clicking. You can select multiple discontiguous data objects by Command-clicking (*Macintosh*) or Ctrl-clicking (*Windows*).

Note: Objects remain selected even when they are hidden inside collapsed data folders. If you select a wave, collapse its data folder, Shift-select another wave, and drag it to another data folder, both waves will be moved there.

However, when a selected object is hidden by deselecting the relevant Display checkbox, actions, such as deletion and duplication, are not taken upon it.

You can rename data objects by clicking the name of the object and editing the name.

The Data Browser also supports icon dragging to move or copy data objects from one data folder to another. You can move data objects from one data folder to another by dragging them. You can copy data objects from one data folder to another by pressing Option (*Macintosh*) or Alt (*Windows*) while dragging.

You can duplicate data objects within a data folder by choosing Duplicate from the Edit menu or by pressing Command-D (*Macintosh*) or Ctrl+D (*Windows*).

You can copy the full paths of all selected data objects, quoted if necessary, to the command line by dragging the objects onto the command line.

You can select one or more waves in the list and drag them into an existing graph or table window to append them to the target window. See **Appending Traces by Drag and Drop** on page II-280 for details.

The Current Data Folder

The “current data folder” is the data folder that Igor uses by default for storing newly-created variables, strings, waves and other data folders. Commands that create or access data objects operate in the current data folder unless you use data folder paths to specify other data folders.

Above the main list there is a textbox that shows the full path to the current data folder. The main list displays a red arrow overlayed on the icon of the current data folder. When the current data folder is contained inside another data folder, a white arrow indicator is overlayed on the icons of the ancestors of the current data folder.

To set the current data folder, right-click any data folder and select Set Current Data Folder. You can also set the current data folder by dragging the red arrow or by Option-clicking (*Macintosh*) or Alt-clicking (*Windows*) a data folder icon.

The Display Checkboxes

The Display checkbox group lets you control which object types are shown in the main list. Data folders are always shown. They also allow you to show or hide the info pane and the plot pane.

The Info Pane

To view the info pane, check the Info checkbox.

The info pane appears below the main list. When you select an object in the main list, its properties or contents appear in the info pane. For example, when you select a variable, its name and value are displayed in the info pane.

If you select a wave in the main list, the info pane displays various properties of the wave, such as type, size, dimensions, units, start X, delta Xand note. Each of these fields is displayed as a blue label followed by a plain text value. You can control which properties are displayed using the Data Browser category of the Miscellaneous Settings Dialog.

If you select a data folder, the info pane may display notes for the data folder. See **Data Folder Notes** on page II-120 for details.

You can edit the name and value of a numeric or string variable, or the name and properties of a wave, by clicking the Edit Object Properties button at the top right corner of the info pane. The button appears only when a single object is selected. Clicking it displays the Edit Object Properties dialog.

The info pane can also display statistics for any selected wave. To show wave statistics, click the sigma icon at the top of the info pane. To change back to normal mode, click the i icon.

You can copy the text displayed in the info pane to the clipboard by clicking the clipboard button at the top of the info pane.

You can control various aspects of the info pane display by right-clicking and choosing an item from the Settings submenu. The options include how white space is displayed, parenthesis matching, syntax coloring, and word wrap.

The Plot Pane

To view the plot pane, check the Plot checkbox.

The plot pane provides a graphical display of a single selected wave. The plot pane is situated below the main list and the info pane. It displays a small graph or image of a wave selected in the main list above it. The plot pane does not display text waves or more than one wave at a time.

You can control various properties of the plot pane by right-clicking and using the resulting pop-up menu.

You can toggle the display of axes by choosing Show Axes from the pop-up menu. You can also toggle between showing a plot of the selected wave and a plot of a 1D histogram by choosing Show Plot or Show Histogram.

Simple 1D real waves are drawn in red on a white background. Complex 1D waves are drawn as two traces with the real part drawn in red and the imaginary in blue. The mode, line style, line size, marker, marker size, and color of the trace can all be configured using the pop-up menu.

2D waves are displayed as an image that by default is scaled to the size of the plot pane and uses the Rainbow color table by default. You can change the aspect ratio and color table via the pop-up menu.

By default, images are displayed with the left axis reversed - that is, the Y value increases from top to bottom, consistent with the behavior of the **NewImage** operation. You can disable this by choosing Reverse Left Axis from the pop-up menu.

When you select a 3D or 4D wave in the main list, the plot pane displays one layer of the wave at a time. Controls at the top of the plot pane allow you to choose which layer is displayed. You can start the plot pane cycling through the layers using the play button and stop it using the stop button.

If the dimensionality of the selected wave is such that it could be interpreted as containing RGB or RGBA data, Igor displays the Automatically Detect RGB(A) checkbox. If you check it, Igor displays the data as an RGB(A) composite image. Otherwise the image data is displayed using a color table, just as with 2D data.

The New Data Folder Button

The New Data Folder button creates a new data folder inside the current data folder. The Data Browser displays a simple dialog to ask for the name of the new data folder and tests that the name provided is valid. When entering liberal object names in the dialog, do not use single quotes around the name.

The Browse Expt Button

The Browse Expt button loads data objects from an Igor packed experiment file or from a folder on disk into the current experiment.

To browse a packed experiment file, click Browse Expt. The Data Browser displays the Open File dialog from which you choose the packed experiment file to browse.

To browse a folder on disk, which may contain packed experiment files, standalone Igor wave (.ibw) files, and other folders, press Option (*Macintosh*) or Alt (*Windows*) while clicking Browse Expt. The Data Browser displays the Choose Folder dialog from which you choose the folder to browse.

Once you select the packed experiment file or the folder to browse, the Data Browser displays an additional list to the right of the main list. The righthand list displays icons representing the data in the file or folder that you selected for browsing. To load data into the current experiment, select icons in the righthand list and drag them into a data folder in the lefthand list.

Click the Done Browsing button to close the righthand list and return to normal operating mode.

The Save Copy Button

The Save Copy button copies data objects from the current experiment to an Igor packed experiment file on disk or as individual files to a folder on disk. Most users will not need to do this because the data is saved when the current experiment is saved.

Before clicking Save Copy, select the items that you want to save. When you click Save Copy, the Data Browser presents a dialog in which you specify the name and location of the packed Igor experiment file which will contain a copy of the saved data.

If you press Option (*Macintosh*) or Alt (*Windows*) while clicking Save Copy, the Data Browser presents a dialog for choosing a folder on disk in which to save the data in unpacked format. The unpacked format is intended for advanced users with specialized applications.

When saving as unpacked, the Data Browser uses "mix-in" mode. This means that the items saved are written to the chosen disk folder but pre-existing items on disk are not affected unless they conflict with items being saved in which case they are overwritten.

When saving as unpacked, if you select a data folder in the Data Browser, it and all of its contents are written to disk. The name of the disk folder is the same as the name of the selected data folder except for the root data folder which is written with the name Data. If you don't select a data folder but just select some items, such as a few waves, the Data Browser writes files for those items but does not create a disk folder.

By default, objects are written to the output without regard to the state of the Waves, Variables and Strings checkboxes in the Display section of the Data Browser. However, there is a preference that changes this behavior in the Data Browser category of the Miscellaneous Settings Dialog. If you check the 'Save Copy saves ONLY displayed object types' checkbox, then Save Copy writes a particular type of object only if the corresponding Display checkbox is checked.

The Data Browser does not provide a method for adding or deleting data to or from a packed experiment file on disk. It can only overwrite an existing file. To add or delete, you need to open the experiment, make additions and deletions and then save the experiment.

The Delete Button

The Delete button is enabled whenever data objects are selected in the main list. If you click it, the Data Browser displays a warning listing the number of items that will be deleted.

To skip the warning, press Option (*Macintosh*) or Alt (*Windows*) when clicking in the Delete button.

Warning: Use this skip feature with great care. If you accidentally delete something, you cannot undo it except by reverting the entire experiment to its last saved state.

Clicking the Delete button when the root data folder is selected deletes all data objects in the current experiment.

Warning: If you accidentally delete something, you cannot undo it except by reverting the entire experiment to its last saved state.

If you try to delete an object that cannot be deleted, such as a wave used in a graph or table, Igor displays a warning message that some objects could not be deleted. Click the Show Details button in this dialog to get a list of objects that were not deleted.

The Execute Cmd Button

The Execute Cmd button provides a shortcut for executing a command on selected items in the Data Browser window. When you click the Execute Cmd button, Igor displays a dialog in which you can specify the command to be executed and the execution mode. Once you have set the command, you can skip the dialog and just execute the command by pressing Option (*Macintosh*) or Alt (*Windows*) while clicking the button.

The format of the command is the same as other Igor commands except that you use %s where the selection is to be inserted. For example:

```
Display %s
```

For the case where the command to be executed exceeds the maximum length for an Igor command, you can specify a secondary command. For example:

```
AppendToGraph %s
```

When 'Execute for each selected item' is enabled, Igor executes the primary command once for each selected item, substituting the full path to the item in place of %s. So, for example, if you select wave0, wave1 and wave2 in the root data folder, Igor executes:

```
Display root:wave0
Display root:wave1
Display root:wave2
```

When 'Execute once for all selected items' is enabled, Igor executes the primary command once, like this:

```
Display root:wave0, root:wave1, root:wave2
```

If the command would exceed the maximum length of the command line, Igor executes the primary and secondary commands, like this:

```
Display root:wave0, root:wave1
AppendToGraph root:wave2
```

Data Browser Command String Limitations

The command strings set in the Execute Cmd dialog, as well as those set via the **ModifyBrowser** operation, must be shorter than the maximum command length of 2500 bytes and may not contain "printf", "sprintf" or "sscanf".

When executing the command on all selected objects at once, the primary and secondary command strings may contain at most one "%s" token.

The commands must not close the Data Browser window.

The command string mechanism is sometimes convenient but somewhat kludgy. It is cleaner and more transparent to get the list of selected objects using the **GetBrowserSelection** function and then process the list using normal list-processing techniques.

Using the Find in Data Browser Dialog

If you choose the Edit→Find, the Data Browser displays the Find in Data Browser dialog. This dialog allows you to find waves, variables and data folders that might be buried in sub-data folders. It also provides a convenient way to select a number of items at one time, based on a search string. You can then use the Execute Cmd button to operate on the selection.

The Find in Data Browser dialog allows you to specify the text used for name matching. Any object whose name contains the specified text is considered a match.

You can also use the wildcard character "*" to match zero or more characters regardless of what they are. For example, "w*3" matches "wave3", and "w3", ""

The dialog also allows you to specify whether the search should be case sensitive, use whole word matching, and wrap around.

DataBrowser Pop-Up Menu

You can apply various Igor operations to objects by selecting the objects in the Data Browser, right-clicking, and choosing the operation from the resulting pop-up menu.

Using the Display and New Image pop-up items, you can create a new graph or image plot of the selected wave. You can select multiple waves, in the same or different data folders, to display together in the same graph.

The Copy Full Path item copies the complete data folder paths of the selected objects, quoted if necessary, to the clipboard.

The Show Where Object Is Used item displays a dialog that lists the dependencies in which the selected object is used and, for waves, the windows in which the wave is used. This item is available only when just one data object is selected.

The Adopt Wave item adopts a shared wave into the current experiment.

Data Browser Preferences

Various Data Browser settings are controlled from the Data Browser category of the Miscellaneous Settings Dialog. You access them by choosing Misc→Miscellaneous Settings and clicking Data Browser in the lefthand list.

Programming the Data Browser

The Data Browser can be controlled from Igor procedures using the following operations and functions:

CreateBrowser, **ModifyBrowser**, **GetBrowserSelection**, **GetBrowserLine**

Advanced Igor programmers can use the Data Browser as an input device via the **GetBrowserSelection** function. For an example, choose File→Example Experiments→Tutorials→Data Folder Tutorial.

Using the Data Browser as a Modal Dialog

You can use the Data Browser as a modal dialog to let the user interactively choose one or more objects to be processed by a procedure. The modal Data Browser is separate from the regular Data Browser that you invoke by choosing Data→Data Browser. The modal Data Browser normally exists only briefly, while you solicit user input from a procedure.

There are two methods for creating and displaying the modal Data Browser. Examples are provided in the help for the **CreateBrowser** operation.

The first and simplest method is to use the **CreateBrowser** operation with the prompt keyword. You can optionally use any of the other keywords supported by CreateBrowser.

The second method is to use the CreateBrowser operation with the /M flag. This creates the modal Data Browser but does not display it. You then call **ModifyBrowser** with the /M flag to configure the modal browser as you like. When you are ready to display the browser, call ModifyBrowser with the /M flag and the showModalBrowser keyword.

Using either method, when the user clicks OK or Cancel or closes the modal browser using the close box, the browser sets the V_Flag and S_BrowserList variables. If the user clicks OK, V_Flag is set to 1 and S_BrowserList is set to contain a semicolon-separated list of the full paths of the selected items. If the user clicks Cancel or the close box, V_Flag is set to 0 and S_BrowserList is set to "".

The Data Browser stores full paths, quoted if necessary, in S_BrowserList. Each full path is followed by a semicolon. You can extract the full paths one-by-one using the **StringFromList** function.

The output variables are local when CreateBrowser is executed from a procedure. There is no reason for you to create the modal Data Browser from the command line, but if you do, the output variables are global and are created in the current data folder at the time the CreateBrowser operation was invoked.

The user can change the current data folder in the modal browser. In most cases this is not desirable. The examples for the **CreateBrowser** operation show how to save and restore the current data folder.

When the user clicks the OK button, the Data Browser executes the commands that you specify using the command1 and command2 keywords to **CreateBrowser** or **ModifyBrowser**. If you omit these keywords, it does not execute any commands.

Managing Data Browser User-Defined Buttons

User-defined buttons allow you to customize your work environment and provide quick access to frequently-used operations. You add a button using ModifyBrowser appendUserButton and delete a button using ModifyBrowser deleteUserButton. The appendUserButton keyword allows you to specify the button name as well as the command to be executed when the button is clicked.

The user button command allows you to invoke operations or functions on the objects that are currently selected in the Data Browser or to invoke some action that is completely unrelated to the selection. For example, the command string "Display %s" will display each of the currently selected waves while the command string "Print IgorInfo(0)" prints some general information in the history window.

When you click a user button, the command is executed once for each selected item if the command string contains a %s. Otherwise the command is executed once regardless of the current selection. If you want to operate once on the whole selection you must not use %s but instead call GetBrowserSelection from your function.

User buttons are drawn in the order that they are appended to the window. If you want to change their position you must delete them and then append them in the desired order.

Buttons are not saved with the experiment or in preferences so they must be added to the Data Browser when Igor starts. To add a set of buttons so that they are available in any experiment, you must write an **IgorStartOrNewHook** hook function. See **User-Defined Hook Functions** on page IV-280 for more information.

Data Folder Notes

You can store notes describing a particular data folder in a string variable in that data folder and view those notes in the Data Browser.

When a single data folder is selected in the Data Browser and the info pane is visible, if the data folder contains a string whose name matches a preset list, the contents of the string are displayed in the info pane. You can edit the list of string names in the Info Pane tab of the Data Browser category of the Miscellaneous Settings dialog. The default list of names is "readme;notes;".

For example, execute the following commands on the command line:

```
NewDataFolder/O TestDF
String root:TestDF:readme = "This is a data folder note"
```

Now open the Data Browser, make sure the Info Pane checkbox is checked, and select TestDF in the main list. The contents of root:TestDF:readme are displayed in the info pane.

If a data folder contains more than one string matching the list of string names to use as data folder notes, only the first matching string in the list is used.

This feature was added in Igor Pro 9.00. To disable it, set the list of names in the Miscellaneous Settings dialog to an empty string.

Data Browser Shortcuts

Action	Shortcut
To set the current data folder	Control-click (*Macintosh*) or right-click (*Windows*) the desired data folder and choose Set Current Data Folder.
To reveal the current data folder in the main list	Click the arrow button at the right end of the Current Data Folder readout.
To display a graph, image or 3D plot of a wave	Control-click (*Macintosh*) or right-click (*Windows*) and select an option from the pop-up menu.
To select contiguous objects	Click the first object and Shift-click the last object.
To select discontiguous objects	Command-click (*Macintosh*) or Ctrl-click (*Windows*) the objects to be selected.
To move objects from one data folder to another	Select the objects and drag them onto the destination data folder.
To copy an object from one data folder to another	Drag the object while pressing Option (*Macintosh*) or Alt (*Windows*).
To duplicate an object	Select the object and press Command-D (*Macintosh*) or Ctrl+D (*Windows*).
To rename an object	Click the object's name and type a new name. To finish, press Return or Enter, or click outside the name.
To edit the properties of the selected object	Click the Edit Object Properties icon in the top/right corner of the info pane.
To view a wave in a table	Double-click the wave's icon.
To append a wave to an existing graph or table	Drag the wave's icon to the graph or table window.
To delete an object without the confirmation dialog	Press Option (*Macintosh*) or Alt (*Windows*) while clicking the Delete button.
To find objects in the browser list	Choose Find from the Edit menu, or press Command-F (*Macintosh*) or Ctrl+F (*Windows*).
To find the same thing again	Choose Find Same from the Edit menu, or press Command-G (*Macintosh*) or Ctrl+G (*Windows*).
To execute a command on a set of waves	Select the icon for each wave that the command is to act on and click the Execute Cmd button.
To execute a command on a set of waves without going through the Execute Cmd dialog	Select the icon for each wave that the command is to act on, press Option (*Macintosh*) or Alt (*Windows*), and click the Execute Cmd button. This reexecutes the command entered previously in the Execute Cmd dialog.
To see statistics for a selected wave	Click the sigma icon in the info pane.
To change styles used in the plot pane	Control-click (*Macintosh*) or right-click (*Windows*) in the plot pane and select an option from the pop-up menu.
To start or stop the animation in the plot pane	Click the Start or Stop button in the plot pane.

Action	Shortcut
To navigate between displayed layers or chunks	Stop the animation in the plot pane and use the up and down cursor keys to change the displayed layer and the right and left cursor keys to change the displayed chunk.
To browse a folder containing unpacked Igor data such as waves	Press Option (*Macintosh*) or Alt (*Windows*) while clicking the Browse Expt button.
To save a copy of selected data in unpacked format	Press Option (*Macintosh*) or Alt (*Windows*) while clicking the Save Copy button.

Chapter

II-9

Importing and Exporting Data

Importing Data

Most Igor users create waves by loading data from a file created by another program. The process of loading a file creates new waves and then stores data from the file in them. Optionally, you can overwrite existing waves instead of creating new ones. The waves can be numeric or text and of dimension 1 through 4.

Igor provides a number of different routines for loading data files. There is no single file format for numeric or text data that all programs can read and write.

There are two broad classes of files used for data interchange: text files and binary files. Text files are usually used to exchange data between programs. Although they are called text files, they may contain numeric data, text data or both. In any case, the data is encoded as plain text that you can read in a text editor. Binary files usually contain data that is efficiently encoded in a way that is unique to a single program and can not be viewed in a text editor.

The closest thing to a universally accepted format for data interchange is the “delimited text” format. This consists of rows and columns of numeric or text data with the rows separated by carriage return characters (CR - *Macintosh*), linefeed return characters (LF - *Unix*), or carriage return/linefeed (CRLF - *Windows*) and the columns separated by tabs or commas. The tab or comma is called the “delimiter character”. The CR, LF, or CRLF characters are called the “terminator”. Igor can read delimited text files written by most programs.

FORTRAN programs usually create fixed field text files in which a fixed number of bytes is used for each column of data with spaces as padding between columns. The Load Fixed Field Text routine is designed to read these files.

Text files are convenient because you can create, inspect or edit them with any text editor. In Igor, you can use a notebook window for this purpose. If you have data in a text file that has an unusual format, you may need to manually edit it before Igor can load it.

Text files generated by scientific instruments or custom programs often have “header” information, usually at the start of the file. The header is not part of the block of data but contains information associated with it. Igor’s text loading routines are designed to load the block of data, not the header. The Load General Text routine can usually automatically skip the header. The Load Delimited Text and Load Fixed Field Text routines needs to be told where the block of data starts if it is not at the start of the file.

An advanced user could write an Igor procedure to read and parse information in the header using the Open, FReadLine, StrSearch, sscanf and Close operations as well as Igor’s string manipulation capabilities. Igor includes an example experiment named Load File Demo which illustrates this.

If you will be working on a Macintosh, and loading data from files on a PC, or vice-versa, you should look at **File System Issues** on page III-450.

The following table lists the data loading routines available in Igor and their salient features.

File Type	Description
Delimited text	Created by spreadsheets, database programs, data acquisition programs, text editors, custom programs. This is the most commonly used format for exchanging data between programs. Row Format: `<data><delimiter><data><terminator>` Contains one block of data with any number of rows and columns. A row of column labels is optional. Can load numeric, text, date, time, and date/time columns. Can load columns into 1D waves or blocks into 2D waves. Columns may be equal or unequal in length. See **Loading Delimited Text Files** on page II-129.

File Type	Description
Fixed field text	Created by FORTRAN programs. Row Format: <data><padding><data><padding><`terminator`> Contains one block of data with any number of rows and columns. Each column consists of a fixed number of bytes including any space characters which are used for padding. Can load numeric, text, date, time and date/time columns. Can load columns into 1D waves or blocks into 2D waves. Columns are usually equal in length but do not have to be. See **Loading Fixed Field Text Files** on page II-137.
General text	Created by spreadsheets, database programs, data acquisition programs, text editors, custom programs. Row Format: `<number><white space><number><terminator>` Contains one or more blocks of numbers with any number of rows and columns. A row of column labels is optional. Can not handle columns containing non-numeric text, dates and times. Can load columns into 1D waves or blocks into 2D waves. Columns must be equal in length. Igor's Load General Text routine has the ability to automatically skip nonnumeric header text. See **Loading General Text Files** on page II-138.
Igor Text	Created by Igor, custom programs. Used mostly as a means to feed data and commands from custom programs into Igor. Format: See **Igor Text File Format** on page II-151. Can load numeric and text data. Can load data into waves of dimension 1 through 4. Contains one or more wave blocks with any number of waves and rows. Consists of special Igor keywords, numbers and Igor commands. See **Loading Igor Text Files** on page II-150.
Igor Binary	Created by Igor, custom programs. Used by Igor to store wave data. Each file contains data for one Igor wave of dimension 1 through 4. Format: See Igor Technical Note #003, "Igor Binary Format". See **Loading Igor Binary Data** on page II-154.
Image	Created by a wide variety of programs. Format: Always binary. Varies according to file type. Can load JPEG, PNG, TIFF, BMP, Sun Raster graphics files. Can load data into matrix waves, including TIFF image stacks. See **Loading Image Files** on page II-157.
General binary	General binary files are binary files created by other programs. If you understand the binary file format, it is possible to load the data into Igor. See **Loading General Binary Files** on page II-166.

File Type	Description
Excel	Supports the .xls and .xlsx file formats. See **Loading Excel Files** on page II-159.
HDF4	Requires activating an Igor extension. For help, execute this in Igor: `DisplayHelpTopic "HDF Loader XOP"`
HDF5	For help, execute this in Igor: `DisplayHelpTopic "HDF5 in Igor Pro"`
Matlab	See **Loading Matlab MAT Files** on page II-163.
JCAMP-DX	The JCAMP-DX format is used primarily in infrared spectroscopy. See **Loading JCAMP Files** on page II-168.
Sound	Supports a variety of sound file formats. See **Loading Sound Files** on page II-170.
TDMS	Loads data from National Instruments TDMS files. Requires activating an extension. Supported on Windows only. See the “TDM Help.ihf” help file for details.
Nicolet WFT	Loads data written by old Nicolet oscilloscopes. Requires activating an extension. See the “NILoadWave Help.ihf” help file for details.
SQL Databases	Loads data from SQL databases. Requires activating an extension and expertise in database programming. See **Accessing SQL Databases** on page II-181.

Load Waves Submenu

You access all of these routines via the Load Waves submenu of the Data menu.

The Load Waves item in this submenu leads to the Load Waves dialog. This dialog provides access to the built-in routines for loading Igor binary wave files, Igor text files, delimited text files, general text files, and fixed field text files, and provides access to all available options.

The Load Igor Binary, Load Igor Text, Load General Text, and Load Delimited Text items in the Load Waves submenu are shortcuts that access the respective file loading routines with default options. We recommend that you start with the Load Waves item so that you can see what options are available.

The precision of numeric waves created by Data→Load General Text and Data→Load Delimited Text is controlled by the Default Data Precision setting in the Data Loading section of the Miscellaneous Settings dialog.

There are no shortcut items for loading fixed field text or image data because these formats require that you specify certain parameters.

The Load Image item leads to the Load Image dialog which provides the means to load various kinds of image files.

Line Terminators

The character or sequence of characters that marks the end of a line of text is known as the “line terminator” or “terminator” for short. Different computer systems use different terminator.

Mac OS 9 used the carriage-return character (CR).

Unix uses linefeed (LF).

Windows uses a carriage-return and linefeed (CRLF) sequence.

When loading waves, Igor treats a single CR, a single LF, or a CRLF as the end of a line. This allows Igor to load text data from file servers on a variety of computers without translation.

LoadWave Text Encodings

This section applies to loading a text file using Load General Text, Load Delimited Text, Load Fixed Field Text, or Load Igor Text.

If your file uses a byte-oriented text encoding (i.e., a text encoding other than UTF-16 or UTF-32), and if the file contains just numbers or just ASCII text, then you don't need to be concerned with text encodings.

If your file uses UTF-16, UTF-32, or contains non-ASCII text, you may neeed to tell the LoadWave operation which text encoding the file uses. For details, see **LoadWave Text Encoding Issues** on page II-149.

Loading Delimited Text Files

A delimited text file consists of rows of values separated by tabs or commas with a carriage return, linefeed or carriage return/linefeed sequence at the end of the row. There may optionally be a row of column labels. Igor can load each column in the file into a separate 1D wave or it can load all of the columns into a single 2D wave. There is no limit to the number of rows or columns except that all of the data must fit in available memory.

In addition to numbers and text, the delimited text file may contain dates, times or date/times. The Load Delimited Text routine attempts to automatically determine which of these formats is appropriate for each column in the file. You can override this automatic determination if necessary.

A numeric column can contain, in addition to numbers, NaN and [±]INF. NaN means "Not a Number" and is the way Igor represents a blank or missing value in a numeric column. INF means "infinity". If Igor finds text in a numeric or date/time column that it can't interpret according to the format for that column, it treats it as a NaN.

If Igor encounters, in any column, a delimiter with no data characters preceding it (i.e., two tabs in a row) it takes this as a missing value and stores a blank in the wave. In a numeric wave, a blank is represented by a NaN. In a text wave, it is represented by an element with zero characters in it.

Determining Column Formats

The Load Delimited Text routine must determine the format of each column of data to be loaded. The format for a given column can be numeric, date, time, date/time, or text. Text columns are loaded into text waves while the other types are loaded into numeric waves with dates being represented as the number of seconds since 1904-01-01.

There are four methods for determining column formats:

- Auto-identify column type
- Treat all columns as numeric
- Treat all columns as text
- Use the LoadWave /B flag to explicitly specify the format of each column

You can choose from the first three of these methods using the Column Types pop-up menu in the Tweaks subdialog of the Load Waves dialog. To use the /B flag, you must manually add the flag to a LoadWave command. This is usually done in a procedure.

In the "auto-identify column type" method, Igor attempts to determine the format of each column by examining the file. This is the default method when you choose Data→Load Waves→Load Delimited Text. Igor looks for the first non-blank value in each column and makes a determination based on the column's content. In most cases, the auto-identify method works and there is no need for the other methods.

In the "treat all columns as numeric" method, Igor loads all columns into numeric waves. If some of the data is not numeric, you get NaNs in the output wave. For backward compatibility, this is the default method when you use the LoadWave/J operation from the command line or from an Igor procedure. To use the "auto-identify column type" method, you need to use LoadWave/J/K=0.

In the "treat all columns as text" method, Igor loads all columns into text waves. This method may have use in rare cases in which you want to do text-processing on a file by loading it into a text wave and then using Igor's string manipulation capabilities to massage it.

For details on the /B method, see the section **Specifying Characteristics of Individual Columns** on page II-145.

Date/Time Formats

The Load Delimited Text routine can handle dates in many formats. A few "standard" formats are supported and in addition, you can specify a "custom" format (see **Custom Date Formats** on page II-130).

The standard date formats are:

mm/dd/yy	(month/day/year)
mm/yy	(month/year)
dd/mm/yy	(day/month/year)

To use the dd/mm/yy format instead of mm/dd/yy, you must set a tweak. See **Delimited Text Tweaks** on page II-136.

You can also use a dash or a dot as a separator instead of a slash.

Igor can also handle times in the following forms:

[+][-]hh:mm:ss [AM PM]	(hours, minutes, seconds)
[+][-]hh:mm:ss.ff [AM PM]	(hours, minutes, seconds, fractions of seconds)
[+][-]hh:mm [AM PM]	(hours, minutes)
[+][-]hhhh:mm:ss.ff	(hours, minutes, seconds, fractions of seconds)

As of Igor Pro 6.23, Igor also accepts a colon instead of a dot before the fractional seconds.

The first three forms are time-of-day forms. The last one is the elapsed time. In an elapsed time, the hour is in the range 0 to 9999.

The year can be specified using two digits (99) or four digits (1999). If a two digit year is in the range 00 … 39, Igor treats this as 2000 … 2039. If a two digit year is in the range 40 … 99, Igor treats this as 1940 … 1999.

The Load Delimited Text routine can also handle date/times which consist of one of these date formats, a single space or the letter T, and then one of the time formats. To load <date><space><time> as a date/time value, space must not be specified as a delimiter character.

Custom Date Formats

If your data file contains dates in a format other than the "standard" format, you can use Load Delimited Text to specify exactly what date format to use. You do this using the Delimited Text Tweaks dialog which you access through the Tweaks button in the Load Waves dialog. Choose Other from the Date Format pop-up menu. This leads to the Date Format dialog.

By clicking the Use Common Format radio button, you can choose from a pop-up menu of common formats. After choosing a common format, you can still control minor properties of the format, such as whether to use 2 or 4 digits years and whether to use leading zeros or not.

In the rare case that your file's date format does not match one of the common formats, you can use a full custom format by clicking the Use Custom Format radio button. It is best to first choose the common format that is closest to your format and then click the Use Custom Format button. Then you can make minor changes to arrive at your final format.

When you use either a common format or a full custom format, the format that you specify must match the date in your file exactly.

When loading data as delimited text, if you use a date format containing a comma, such as "October 11, 1999", you must make sure that LoadWave operation does not treat the comma as a delimiter. You can do this using the Delimited Text Tweaks dialog.

When loading a date format that consists entirely of digits, such as 991011, you should use the LoadWave/B flag to specify that the data is a date. Otherwise, LoadWave will treat it as a regular number. The /B flag can not be generated from the dialog — you need to use the LoadWave operation from the command line. Another approach is to use the dialog to generate a LoadWave command without the /B flag and then specify that the column is a date column in the Loading Delimited Text dialog that appears when the LoadWave operation executes.

Column Labels

Each column may optionally have a column label. When loading 1D waves, if you read wave names and if the file has column labels, Igor will use the column labels for wave names. Otherwise, Igor will automatically generate wave names of the form wave0, wave1 and so on.

Igor considers text in the label line to be a column label if that text can not be interpreted as a data value (number, date, time, or datetime) or if the text is quoted using single or double quotes.

When loading a 2D wave, Igor optionally uses the column labels to set the wave's column dimension labels. The wave name does not come from column labels but is automatically assigned by Igor. You can rename the wave after loading if you wish.

Igor expects column labels to appear in a row of the form:

```
<label><delimiter><label><delimiter>…<label><terminator>
```

where <column label> may be in one of the following forms:

<label>	(label with no quotes)
"<label>"	(label with double quotes)
'<label>'	(label with single quotes)

The default delimiter characters are tab and comma. There is a tweak (see **Delimited Text Tweaks** on page II-136) for using other delimiters.

Igor expects that the row of column labels, if any, will appear at the beginning of the file. There is a tweak (see **Delimited Text Tweaks** on page II-136) that you can use to specify if this is not the case.

Igor cleans up column labels found in the file, if necessary, so that they are legal wave names using standard name rules. The cleanup consists of converting illegal characters into underscores and truncating long names to the maximum of 255 bytes.

Examples of Delimited Text

Here are some examples of text that you might find in a delimited text file. These examples are tab-delimited.

Simple delimited text

```
ch0          ch1          ch2          ch3          (optional row of labels)
2.97055      1.95692      1.00871      8.10685
3.09921      4.08008      1.00016      7.53136
3.18934      5.91134      1.04205      6.90194
```

Loading this text would create four waves with three points each or, if you specify loading it as a matrix, a single 3 row by 4 column wave.

Delimited text with missing values

```
ch0          ch1          ch2          ch3          (optional row of labels)
2.97055      1.95692                   8.10685
3.09921      4.08008      1.00016      7.53136
             5.91134      1.04205
```

Loading this text as 1D waves would create four waves. Normally each wave would contain three points but there is an option to ignore blanks at the end of a column. With this option, ch0 and ch3 would have two points. Loading as a matrix would give you a single 3 row by 4 column wave with blanks in columns 0, 2 and 3.

Delimited text with a date column

```
Date         ch0          ch1          ch2          (optional row of labels)
2/22/93      2.97055      1.95692      1.00871
2/24/93      3.09921      4.08008      1.00016
2/25/93      3.18934      5.91134      1.04205
```

Loading this text as 1D waves would create four waves with three points each. Igor would convert the dates in the first column into the appropriate number using the Igor system for storing dates (number of seconds since 1/1/1904). This data is not suitable for loading as a matrix.

Delimited text with a nonnumeric column

```
Sample       ch0          ch1          ch2          (optional row of labels)
Ge           2.97055      1.95692      1.00871
Si           3.09921      4.08008      1.00016
GaAs         3.18934      5.91134      1.04205
```

Loading this text as 1D waves would normally create four waves with three points each. The first wave would be a text wave and the remaining would be numeric. You could also load this as a single 3x3 matrix, treating the first row as column labels and the first column as row labels for the matrix. If you loaded it as a matrix but did not treat the first column as labels, it would create a 3 row by 4 column text wave, not a numeric wave.

Delimited text with quoted strings

Sample	ch0	ch1	ch2	Comment
Ge	2.97055	1.95692	1.00871	"Run 17, station 1"
Si	3.09921	4.08008	1.00016	"Run 17, station 2"
GaAs	3.18934	5.91134	1.04205	"Run 17, station 3"

Starting with Igor Pro 8.00, Load Delimited Text (LoadWave/J) recognizes ASCII double-quote characters as enclosing a string that may contain delimiter characters. In this case, the Comment column contains text which contains commas. Comma is normally a delimiter character but, because the column text is quoted, LoadWave does not treat it as a delimiter. See **Quoted Strings in Delimited Text Files** on page II-135 for details.

The Load Waves Dialog for Delimited Text — 1D

The basic process of loading 1D data from a delimited text file is as follows:

1. Choose Data→Load Waves→Load Waves to display the Load Waves dialog.
2. Choose Delimited Text from the File Type pop-up menu.

3. Click the File button to select the file containing the data.
4. Click Do It.

When you click Do It, the LoadWave operation runs. It executes the Load Delimited Text routine which goes through the following steps:

1. Optionally, determine if there is a row of column labels.
2. Determine the number of columns.
3. Determine the format of each column (number, text, date, time or date/time).
4. Optionally, present another dialog allowing you to confirm or change wave names.
5. Create waves.
6. Load the data into the waves.

Igor looks for a row of labels only if you enable the “Read wave names” option. If you enable this option and if Igor finds a row of labels then this determines the number of columns that Igor expects in the file. Otherwise, Igor counts the number of data items in the first row in the file and expects that the rest of the rows have the same number of columns.

In step 3 above, Igor determines the format of each column by examining the first data item in the column. Igor tries to interpret all of the remaining items in a given column using the format that it determines from the first item in the column.

If you choose Data→Load Waves→Load Delimited Text instead of choosing Data→Load Waves→Load Waves, Igor displays the Open File dialog in which you can select the delimited text file to load directly. This is a shortcut that skips the Load Waves dialog and uses default options for the load. This always loads 1D waves, not a matrix. The precision of numeric waves is controlled by the Default Data Precision setting in the Data Loading section of the Miscellaneous Settings dialog. Before you use this shortcut, take a look at the Load Waves dialog so you can see what options are available.

Editing Wave Names

The “Auto name & go” option is used mostly when you are loading 1D data under control of an Igor procedure and you want everything to be automatic. When loading 1D data manually, you normally leave the “Auto name & go” option deselected. Then Igor presents an additional Loading Delimited Text dialog in which you can confirm or change wave names.

The context area of the Loading Delimited Text dialog gives you feedback on what Igor is about to load. You can’t edit the file here. If you want to edit the file, abort the load and open the file as an Igor notebook or open it in a text editor.

Set Scaling After Loading Delimited Text Data

If your 1D numeric data is uniformly spaced in the X dimension then you will be able to use the many operations and functions in Igor designed for waveform data. You will need to set the X scaling for your waves after you load them, using the Change Wave Scaling dialog.

Note: If your 1D data is uniformly spaced it is *very important* that you set the X scaling of your waves. Many Igor operations depend on the X scaling information to give you correct results.

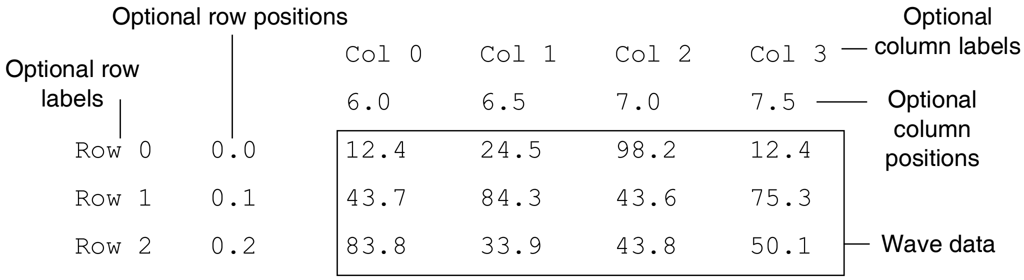
If your 1D data is not uniformly spaced then you will use XY pairs and you do not need to change X scaling. You may want to use Change Wave Scaling to set the data units.

The Load Waves Dialog for Delimited Text — 2D

To load a delimited text file as a 2D wave, choose the Load Waves menu item. Then, select the “Load columns into matrix” checkbox.

When you load a matrix (2D wave) from a text file, Igor creates a single wave. Therefore, there is no need for a second dialog to enter wave names. Instead, Igor automatically names the wave based on the base name that you specify. After loading, you can then rename the wave if you want.

To understand the row/column label/position controls, you need to understand Igor's view of a 2D delimited text file:



```
                Col 0    Col 1    Col 2    Col 3
                6.0      6.5      7.0      7.5
Row 0    0.0    12.4     24.5     98.2     12.4
Row 1    0.1    43.7     84.3     43.6     75.3
Row 2    0.2    83.8     33.9     43.8     50.1
```

In the simplest case, your file has just the wave data — no labels or positions. You would indicate this by deselecting all four label/position checkboxes.

2D Label and Position Details

If your file does have labels or positions, you would indicate this by selecting the appropriate checkbox. Igor expects that row labels appear in the first column of the file and that column labels appear in the first line of the file unless you instruct it differently using the Tweaks subdialog (see **Delimited Text Tweaks** on page II-136). Igor loads row/column labels into the wave's dimension labels (described in Chapter II-6, **Multidimensional Waves**).

Igor can treat column positions in one of two ways. It can use them to set the dimension scaling of the wave (appropriate if the positions are uniformly-spaced) or it can create separate 1D waves for the positions. Igor expects row positions to appear in the column immediately after the row labels or in the first column of the file if the file contains no row labels. It expects column positions to appear immediately after the column labels or in the first line of the file if the file contains no column labels unless you instruct it differently using the Tweaks subdialog.

A row position wave is a 1D wave that contains the numbers in the row position column of the file. Igor names a row position wave "RP_ " followed by the name of the matrix wave being loaded. A column position wave is a 1D wave that contains the numbers in the column position line of the file. Igor names a column position wave "CP_" followed by the name of the matrix wave being loaded. Once loaded (into separate 1D waves or into the matrix wave's dimension scaling), you can use row and column position information when displaying a matrix as an image or when displaying a contour of a matrix.

If your file contains header information before the data, column labels and column positions, you need to use the Tweaks subdialog to specify where to find the data of interest. The "Line containing column labels" tweak specifies the line on which to find column labels. The "First line containing data" tweak specifies the first line of data to be stored in the wave itself. The first line in the file is considered to be line zero.

If you instruct LoadWave to read column positions, it determines which line contains them in one of two ways, depending on whether or not you also instructed it to read column labels. If you do ask LoadWave to read column labels, then LoadWave assumes that the column positions line immediately follows the column labels line. If you do not ask LoadWave to read column labels, then LoadWave assumes that the column positions line immediately precedes the first data line.

Loading Text Waves from Delimited Text Files

There are a few issues relating to special characters that you may need to deal with when loading data into text waves.

By default, the Load Delimited Text operation considers comma and tab characters to be delimiters which separate one column from the next. If the text that you are loading may contain commas or tabs as values rather than as delimiters, you will need to change the delimiter characters. You can do this using the Tweaks subdialog of the Load Delimited Text dialog.

The Load Delimited Text operation always considers carriage return and linefeed characters to mark the end of a line of text. It would be quite unusual to find a data file that uses these characters as values. In the extremely rare case that you need to load a carriage return or linefeed as a value, you can use an escape sequence. Replace the carriage return value with "\r" (without the quotes) and the linefeed value with "\n". Igor will convert these to carriage return and linefeed and store the appropriate character in the text wave.

In addition to "\r" and "\n", Igor will also convert "\t" into a tab value and do other escape sequence conversions (see **Escape Sequences in Strings** on page IV-14). These conversions create a possible problem which should be quite rare. You may want to load text that contains "\r", "\n" or "\t" sequences which you do not want to be treated as escape sequences. To prevent Igor from converting them into carriage return and tab, you will need to replace them with "\\r", "\\n" and "\\t".

Igor does not remove quotation marks when loading data from delimited text files into text waves. If necessary, you can do this by opening the file as a notebook and doing a mass replace before loading or by displaying the loaded waves in a table and using Edit→Replace.

Quoted Strings in Delimited Text Files

Comma-separated values (CSV) text files can be loaded in Igor as delimited text files with comma as the delimiter. Here is some text that might appear in a CSV text file:

```
1,London
2,Paris
3,Rome
```

Sometimes double-quotes are used in CSV files to enclose an item. For example:

```
1,"London"
2,"Paris"
3,"Rome"
```

In Igor6 and Igor7, double-quotes in a delimited text file received no special treatment. Thus, when loading the second example, Igor would create a numeric wave containing 1, 2, and 3, and a text wave containing "London", "Paris", and "Rome". The text wave would include the double-quote characters.

In Igor8 and later, by default, the Load Delimited Text routine treats plain ASCII double-quote characters as enclosing characters that are not loaded into the wave. So, in the second example, the text wave contains London, Paris, and Rome, with no double-quote characters.

This feature is especially useful when the quoted strings contain commas, as in this example:

```
1,"123 First Street, London, England"
2,"59 Rue Poncelet, Paris, France"
3,"Viale Europa 22, 00144 Rome, Italy"
```

Prior to Igor8, Igor would treat this text as containing four columns, because double-quotes received no special treatment and comma is a delimiter character by default. Igor8 loads this as two columns creating a numeric wave with three numbers and a text wave with three addresses.

Because of previously-established rules regarding column names in delimited text files, if you specify that the file includes column names using the LoadWave /W flag, LoadWave interprets quoted text as column names even if the text is all numeric. For example, if you use LoadWave/W and the file contains:

```
"1","2","3"
"4","5","6"
```

LoadWave treats the first line as column names. However, if you use LoadWave/W and the file contains:

```
1,2,3
4,5,6
```

LoadWave treats the first line as data, not column names. So, if your file contains quoted strings, you must omit the /W flag if the file does not contain column names.

Delimited Text Compatibility With Igor7

As of Igor8, LoadWave/J provides better handling of quoted strings in delimited text files, such as comma-separated values files. See **Quoted Strings in Delimited Text Files** on page II-135 for details.

As a side-effect, in rare cases, Igor8 and later may produce different results for a given file compared to Igor7. For example, Igor7 would automatically identify this data as containing two numeric columns:

```
1;,2;
```

Igor8 automatically identifies this as containing two text columns, because of the unexpected semicolons.

If this change interferes with your file loading, you can cause Igor8 and later to work like Igor7 by setting bit 4 of the loadFlags parameter of the LoadWave /V flag. For example:

```
LoadWave/J/V={",", " ", 0, 16}       // 16 means bit 4 is set
```

This disables Igor8's support for quoted strings which causes LoadWave to behave more like it did in Igor7.

Another workaround is to force LoadWave to use a given data type for a given column. You can do this using the Column Format popup menu in the Loading Delimited Text dialog (see **Editing Wave Names** on page II-133) or by using the LoadWave /B flag.

Delimited Text Tweaks

There are many variations on the basic form of a delimited text file. We've tried to provide tweaks that allow you to guide Igor when you need to load a file that uses one of the more common variations. To do this, use the Tweaks button in the Load Waves dialog.

The Tweaks dialog can specify the space character as a delimiter. Use the LoadWave operation to specify other delimiters as well.

The main reason for allowing space as a delimiter is so that we can load files that use spaces to align columns. This is a common format for files generated by FORTRAN programs. Normally, you should use the fixed field text loader to load these files, not the delimited text loader. If you do use the delimited text loader and if space is allowed as a delimiter then Igor treats any number of consecutive spaces as a single delimiter. This means that two consecutive spaces do not indicate a missing value as two consecutive tabs would.

When loading a delimited file, by default Igor expects the first line in the file to contain either column labels or the first row of data. There are several tweaks that you can use for a file that doesn't fit this expectation.

Lines and columns in the tweaks dialog are numbered starting from zero.

Using the "Line containing column labels" tweak, you can specify on what line column labels are to be found if not on line zero. Using this and the "First line containing data" tweak, you can instruct Igor to skip garbage, if any, at the beginning of the file.

The "First line containing data", "Number of lines containing data", "First column containing data", and "Number of columns containing data" tweaks are designed to allow you to load any block of data from anywhere within a file. This might come in handy if you have a file with hundreds of columns but you are only interested in a few of them.

If "Number of lines containing data" is set to "auto" or 0, Igor will load all lines until it hits the end of the file. If "Number of columns containing data" is set to "auto" or 0, Igor will load all columns until it hits the last column in the file.

The proper setting for the "Ignore blanks at the end of a column" tweak depends on the kind of 1D data stored in the file. If a file contains some number of similar columns, for example four channels of data from a digital oscilloscope, you probably want all of the columns in the file to be loaded into waves of the same length. Thus, if a particular column has one or more missing values at the end, the corresponding points in the wave should contain NaNs to represent the missing value. On the other hand, if the file contains a number of dissimilar columns, then you might want to ignore any blank points at the end of a column so that the resulting waves will not necessarily be of equal length. If you enable the "Ignore blanks at the end of a column" tweak then LoadWave will not load blanks at the end of a column into the 1D wave. If this option is enabled and a particular column has nothing but blanks then the corresponding wave is not loaded at all.

Troubleshooting Delimited Text Files

You can examine the waves created by the Load Delimited Text routine using a table. If you don't get the results that you expected, you can to try other LoadWave options or inspect and edit the text file until it is in a form that Igor can handle. Remember the following points:

- Igor expects the file to consist of numeric values, text values, dates, times or date/times separated by tabs or commas unless you set tweaks to the contrary.
- Igor expects a row of column labels, if any, to appear in the first line of the file unless you set tweaks to the contrary. It expects that the column labels are also delimited by tabs or commas unless you set tweaks to the contrary. Igor will not look for a line of column labels unless you enable the Read Wave Names option for 1D waves or the Read Column Labels options for 2D waves.
- Igor determines the number of columns in the file by inspecting the column label row or the first row of data if there is no column label row.

If merely inspecting the file does not identify the problem then you should try the following troubleshooting technique.

- Copy just the first few lines of the file into a test file.
- Load the test file and inspect the resulting waves in a table.
- Open the test file as a notebook.
- Edit the file to eliminate any irregularities, save it and load it again. Note that you can load a file as delimited text even if it is open as a notebook. Make sure that you have saved changes to the notebook before loading it.
- Inspect the loaded waves again.

This process usually sheds some light on what aspect of the file is irregular. Working on a small subset of your file makes it easier to quickly do some trial and error investigation.

If you are unable to get to the bottom of the problem, email a zipped copy of the file or of a representative subset of it to support@wavemetrics.com along with a description of the problem. Do not send the segment as plain text because email programs may strip out or replace unusual control characters in the file.

Loading Fixed Field Text Files

A fixed field text file consists of rows of values, organized into columns, that are a fixed number of bytes wide with a carriage return, linefeed, or carriage return/linefeed sequence at the end of the row. Space characters are used as padding to ensure that each column has the appropriate number of bytes. In some cases, a value will fill the entire column and there will be no spaces after it.

FORTRAN programs typically generate fixed field text files. A normal Fortran data file contains consists of values followed by spaces to pad to the field width. For example, the contents of a file using a field width of 10 might look like this (using dashes to represent spaces for clarity):

```
0.000-----1.000-----2.000-----<CRLF>
1.000-----2.000-----3.000-----<CRLF>
```

Non-Fortran programs sometimes write fixed-field data right-justified instead of left-justified, like this (using dashes to represent spaces for clarity):

```
-----0.000-----1.000-----2.000<CRLF>
-----1.000-----2.000-----3.000<CRLF>
```

To accommodate such files, Igor's Load Fixed Field routine strips leading and trailing spaces from the field before reading the value.

Stripping leading and trailing spaces also allows Igor's Load Fixed Field routine to load values that are left-justified or right-justified, so long as each value for a given row is in a consistent width field.

Igor’s Load Fixed Field Text routine works just like the Load Delimited Text routine except that, instead of looking for a delimiter character to determine where a column ends, it counts the number of bytes in the column. All of the features described in the section **Loading Delimited Text Files** on page II-129 apply also to loading fixed field text.

The Load Waves Dialog for Fixed Field Text

To load a fixed field text file, invoke the Load Waves dialog by choosing Data→Load Waves→Load Waves. The dialog is the same as for loading delimited text except for three additional items.

In the Number of Columns item, you must enter the total number of columns in the file. In the Field Widths item, you must enter the number of bytes in each column of the file, separated by commas. The last value that you enter is used for any subsequent columns in the file. If all columns in the file have the same number of bytes, just enter one number.

If you select the All 9’s Means Blank checkbox then Igor will treat any column that consists entirely of the digit 9 as a blank. If the column is being loaded into a numeric wave, Igor sets the corresponding wave value to NaN. If the column is being loaded into a text wave, Igor sets the corresponding wave value to "" (empty string).

Specifying Fixed Field Widths Programmatically

If all of the columns in the file consist of the same number of bytes, you can specify this number using the LoadWave /F flag. If different columns consist of different numbers of bytes, you have to use the LoadWave /B flag to specify the width of each column.

Loading General Text Files

We use the term “general text” to describe a text file that consists of one or more blocks of numeric data. A block is a set of rows and columns of numbers. Numbers in a row are separated by one or more tabs or spaces. One or more consecutive commas are also treated as white space. A row is terminated by a carriage return character, a linefeed character, or a carriage return/linefeed sequence.

The Load General Text routine handles numeric data only, not date, time, date/time or text. Use Load Delimited Text or Load Fixed Field Text for these formats. Load General Text can handle 2D numeric data as well as 1D.

The first block of data may be preceded by header information which the Load General Text routine automatically skips.

If there is a second block, it is usually separated from the first with one or more blank lines. There may also be header information preceding the second block which Igor also skips.

When loading 1D data, the Load General Text routine loads each column of each block into a separate wave. It treats column labels as described above for the Load Delimited Text routine, except that spaces as well as tabs and commas are accepted as delimiters. When loading 2D data, it loads all columns into a single 2D wave.

The Load General Text routine determines where a block starts and ends by counting the number of numbers in a row. When it finds two rows with the same number of numbers, it considers this the start of a block. The block continues until a row which has a different number of numbers.

Examples of General Text

Here are some examples of text that you might find in a general text file.

Simple general text

```
ch0         ch1         ch2         ch3         (optional row of labels)
2.97055     1.95692     1.00871     8.10685
3.09921     4.08008     1.00016     7.53136
```

```
3.18934     5.91134     1.04205     6.90194
```

The Load General Text routine would create four waves with three points each or, if you specify loading as a matrix, a single 3 row by 4 column wave.

General text with header

```
Date: 3/2/93
Sample: P21-3A
ch0         ch1         ch2         ch3         (optional row of labels)
2.97055     1.95692     1.00871     8.10685
3.09921     4.08008     1.00016     7.53136
3.18934     5.91134     1.04205     6.90194
```

The Load General Text routine would automatically skip the header lines (Date: and Sample:) and would create four waves with three points each or, if you specify loading as a matrix, a single 3 row by 4 column wave.

General text with header and multiple blocks

```
Date: 3/2/93
Sample: P21-3A
ch0_1       ch1_1       ch2_1       ch3_1       (optional row of labels)
2.97055     1.95692     1.00871     8.10685
3.09921     4.08008     1.00016     7.53136
3.18934     5.91134     1.04205     6.90194

Date: 3/2/93
Sample: P98-2C
ch0_2       ch1_2       ch2_2       ch3_2       (optional row of labels)
2.97055     1.95692     1.00871     8.10685
3.09921     4.08008     1.00016     7.53136
3.18934     5.91134     1.04205     6.90194
```

The Load General Text routine would automatically skip the header lines and would create eight waves with three points each or, if you specify loading as a matrix, two 3 row by 4 column waves.

Comparison of General Text, Fixed Field and Delimited Text

You may wonder whether you should use the Load General Text routine, Load Fixed Field routine or the Load Delimited Text routine. Most commercial programs create simple tab-delimited files which these routines can handle. Files created by scientific instruments, mainframe programs, custom programs, or exported from spreadsheets are more diverse. You may need to try these routines to see which works better. To help you decide which to try first, here is a comparison.

Advantages of the Load General Text compared to Load Fixed Field and to Load Delimited Text:

- It can automatically skip header text.
- It can load multiple blocks from a single file.
- It can tolerate multiple tabs or spaces between columns.

Disadvantages of the Load General Text compared to Load Fixed Field and to Load Delimited Text:

- It can not handle blanks (missing values).
- It can not tolerate columns of non-numeric text or non-numeric values in a numeric column.
- It can not load text values, dates, times or date/times.
- It can not handle comma as the decimal point (European number style).

The Load General Text routine *can* load missing values if they are represented in the file explicitly as “NaN” (Not-a-Number). It can not handle files that represent missing values as blanks because this confounds the technique for determining where a block of numbers starts and ends.

The Load Waves Dialog for General Text — 1D

The basic process of loading data from a general text file is as follows:

1. Choose Data→Load Waves→Load Waves to display the Load Waves dialog.
2. Choose General Text from the File Type pop-up menu.
3. Click the File button to select the file containing the data.
4. Click Do It.

When you click Do It, Igor's LoadWave operation runs. It executes the Load General Text routine which goes through the following steps:

1. Locate the start of the block of data using the technique of counting numbers in successive lines. This step also skips the header, if any, and determines the number of columns in the block.
2. Optionally, determine if there is a row of column labels immediately before the block of numbers.
3. Optionally, present another dialog allowing you to confirm or change wave names.
4. Create waves.
5. Load data into the waves until the end of the file or a until a row that contains a different number of numbers.
6. If not at the end of the file, go back to step 1 to look for another block of data.

Igor looks for a row of column labels only if you enable the "Read wave names" option. It looks in the line immediately preceding the block of data. If it finds labels and if the number of labels matches the number of columns in the block, it uses these labels as wave names. Otherwise, Igor automatically generates wave names of the form wave0, wave1 and so on.

If you choose Data→Load Waves→Load General Text instead of choosing Data→Load Waves→Load Waves, Igor displays the Open File dialog in which you can select the general text file to load directly. This is a shortcut that skips the Load Waves dialog and uses default options for the load. This will always load 1D waves, not a matrix. The precision of numeric waves is controlled by the Default Data Precision setting in the Data Loading section of the Miscellaneous Settings dialog. Before you use this shortcut, take a look at the Load Waves dialog so you can see what options are available.

Editing Wave Names for a Block

In step 3 above, the Load General Text routine presents a dialog in which you can change wave names. This works exactly as described above for the Load Delimited Text routine except that it has one extra button: "Skip this block".

Use "Skip this block" to skip one or more blocks of a multiple block general text file.

Click the Skip Column button to skip loading of the column corresponding to the selected name box. Shift-click the button to skip all columns except the selected one.

The Load Waves Dialog for General Text — 2D

Igor can load a 2D wave using the Load General Text routine. However, Load General Text does not support the loading of row/column labels and positions. If the file has such rows and columns, you must load it as a delimited text file.

The main reason to use the Load General Text routine rather than the Load Delimited Text routine for loading a matrix is that the Load General Text routine can automatically skip nonnumeric header information. Also, Load General Text treats any number of spaces and tabs, as well as one comma, as a single delimiter and thus is tolerant of less rigid formatting.

Set Scaling After Loading General Text Data

If your 1D data is uniformly spaced in the X dimension then you will be able to use the many operations and functions in Igor designed for waveform data. You will need to set the X scaling for your waves after you load them, using the Change Wave Scaling dialog.

Note: If your data is uniformly spaced it is *very important* that you set the X scaling of your waves. Many Igor operations depend on the X scaling information to give you correct results.

If your 1D data is not uniformly spaced then you will use XY pairs and you do not need to change X scaling. You may want to use Change Wave Scaling to set the waves' data units.

General Text Tweaks

The Load General Text routines provides some tweaks that allow you to guide Igor as it loads the file. To do this, use the Tweaks button in the Load Waves dialog.

The items at the top of the dialog are hidden because they apply to the Load Delimited Text routine only. Load General Text always skips any tabs and spaces between numbers and will also skip a single comma. The "decimal point" character is always period and it can not handle dates.

The items relating to column labels, data lines and data columns have two potential uses. You can use them to load just a part of a file or to guide Igor if the automatic method of finding a block of data produces incorrect results.

Lines and columns in the tweaks dialog are numbered starting from zero.

Igor interprets the "Line containing column labels" and "First line containing data" tweaks differently for general text files than it does for delimited text files. For delimited text, zero means "the first line". For general text, zero for these parameters means "auto".

Here is what "auto" means for general text. If "First line containing data" is auto, Igor starts the search for data from the beginning of the file without skipping any lines. If it is not "auto", then Igor skips to the specified line and starts its search for data there. This way you can skip a block of data at the beginning of the file. If "Line containing column labels" is auto then Igor looks for column labels in the line immediately preceding the line found by the search for data. If it is not auto then Igor looks for column labels in the specified line.

If the "Number of lines containing data" is not "auto" then Igor stops loading after the specified number of lines or when it hits the end of the first block, whichever comes first. This behavior is necessary so that it is possible to pick out a single block or subset of a block from a file containing more than one block.

If a general text file contains more than one block of data and if "Number of lines containing data" is "auto" then, for blocks after the first one, Igor maintains the relationship between the line containing column labels and first line containing data. Thus, if the column labels in the first block were one line before the first line containing data then Igor expects the same to be true of subsequent blocks.

You can use the "First column containing data" and "Number of columns containing data" tweaks to load a subset of the columns in a block. If "Number of columns containing data" is set to "auto" or 0, Igor loads all columns until it hits the last column in the block.

Troubleshooting General Text Files

You can examine the waves created by the Load General Text routine using a table. If you don't get the results that you expected, you will need to inspect and edit the text file until it is in a form that Igor can handle. Remember the following points:

- Load General Text can not handle dates, times, date/times, commas used as decimal points, or blocks of data with non-numeric columns. Try Load Delimited Text instead.
- It skips any tabs or spaces between numbers and will also skip a single comma.
- It expects a line of column labels, if any, to appear in the first line before the numeric data unless you set tweaks to the contrary. It expects that the labels are also delimited by tabs, commas or spaces. It will not look for labels unless you enable the Read Wave Names option.
- It works by counting the number of numbers in consecutive lines. Some unusual formats (e.g., 1,234.56 instead of 1234.56) can throw this count off, causing it to start a new block prematurely.
- It can not handle blanks or non-numeric values in a column. Each of these cause it to start a new block of data.

- If it detects a change in the number of columns, it starts loading a new block into a new set of waves.

If merely inspecting the file does not identify the problem then you should try the technique of loading a subset of your data. This is described under **Troubleshooting Delimited Text Files** on page II-137 and often sheds light on the problem. In the same section, you will find instructions for sending the problem file to WaveMetrics for analysis, if necessary.

LoadWave Generation of Wave Names

When loading an Igor binary file or an Igor Text file, LoadWave uses the wave name or names stored in the file being loaded.

When loading files as delimited text (/J), as fixed field text (/F), and as general text (/G), wave names are determined by the /A, /N, /W, /B, and /NAME flag. This section provides describes how these naming flags work.

If you omit all of the naming flags, LoadWave generates wave names like wave0, wave1, and wave2 but if such wave already exist, it generates unique names like wave3, wave4, and wave5. LoadWave then displays a dialog in which you can edit the names.

The /A flag behaves the same except that it turns on "auto name and go" which skips the dialog in which you can edit the names. /A=baseName is the same as /A except that allows you to specify a base name other than 'wave'.

The /N flag is the same as /A except that it always uses suffix numbers starting from zero and increments by one for each wave loaded from the file. If the resulting name conflicts with an existing wave, the existing wave is overwritten. For example, /N=wave gives wave names like wave0, wave1, and wave2.

The /W flag loads wave names from the file itself. By default, LoadWave expects the wave names to be in the first line of the file but the /L flag allows you to specify another line. If the names in the file conflict with existing waves and you specify overwrite (/O), the existing waves are overwritten; if you do not specify overwrite, LoadWave displays a dialog in which you can enter unique names.

The /B flag, used when calling LoadWave from a user-defined function, allows you to specify explicit names for each column. See **Specifying Characteristics of Individual Columns** on page II-145 for details.

The /NAME flag provides an easy way to incorporate the file name in the wave names. See the next section for details.

/NAME overrides /B which overrides /W which overrides /N which overrides /A.

Using the File Name in Wave Names

The LoadWave /NAME flag was added in Igor Pro 9.00 primarily to provide an easy way to incorporate the file name in the wave names.

The Load Waves dialog (Data→Load Waves→Load Waves) supports the /NAME flag through the Use File Name in Wave Names and Include Normal Name checkboxes. The dialog does not provide access to all features of /NAME but is sufficient for most common uses.

This section provides a general description of the /NAME flag. Subsequent sections with examples which should clarify how to use it.

The format of the flag is:

```
/NAME={namePrefix, nameSuffix, nameOptions}
```

The generated wave names consist of the following components:

```
<namePrefix><normal name><nameSuffix><suffix number>
```

namePrefix and *nameSuffix* can be empty (""), literal text like "Run1_", the special pattern ":filename:" or a combination of literal text and the special pattern like ":filename:_". LoadWave replaces the special pattern ":filename:" with the name of the file being loaded minus the file name extension. If both *namePrefix* and *nameSuffix* are empty, LoadWave acts as if the /NAME flag were omitted.

<normal name> refers to the wave name that would be used if /NAME were omitted.

The rest of this section discusses the nameOptions bitwise parameter (see **Setting Bit Parameters** on page IV-12) which provides flexibility in naming across various scenarios. In the abstract, *nameOptions* may be confusing; examples shown in subsequent sections should clarify its meaning and use.

If bit 0 of *nameOptions* is set, LoadWave includes <normal name>. If cleared, it omits <normal name>.

Bits 1, 2, and 3 control the use of suffix numbers. A suffix number is a number like 0, 1, 2, and so on, used to make the wave names unique. When loading a single wave, LoadWave includes <suffix number> if bit 1 of *nameOptions* is set unless it is suppressed by bit 3 as explained below. When loading multiple waves, LoadWave includes <suffix number> if bit 2 of *nameOptions* is set unless it is suppressed by bit 3 as explained below. Often you want to include suffix numbers when loading multiple waves, because the numbers are necessary to distinguish the names of the waves you are loading, but you want to exclude the suffix number when loading a single wave. For that case you would leave bit 1 cleared and set bits 2 and 3.

Bit 3 of *nameOptions* overrides bits 1 and 2 to prevent prevent appending suffix numbers if they are not needed to prevent name conflicts. When loading a single wave, bit 3 overrides bit 1 to prevent appending a suffix number if there is no name conflict. When loading multiple waves, bit 3 overrides bit 2 to prevent appending a suffix numbers if there are no name conflicts.

If bit 4 of *nameOptions* is set, LoadWave chooses the suffix number, if enabled, to avoid conflicts with existing waves and other objects. If it is cleared, the suffix number, if enabled, starts from 0 and increments for each wave being loaded.

If bit 5 of *nameOptions* is cleared, LoadWave cleans up the wave name to make it a standard name. Otherwise it allows liberal names. We recommend standard names because programming with liberal names is tricky. See **Object Names** on page III-501 for details.

Loading a Single Wave Using the File Name

In this section, we assume that we are loading a file named "Data.txt" and that we are loading a single wave from the file.

```
// nameOptions=0 means omit the normal name
/NAME={":filename:","",0}
```

LoadWave creates a wave named Data if it does not already exist. If it exists and you include the /O (overwrite) flag, Data is overwritten. If it exists and you omit /O, LoadWave displays a dialog in which you can enter a unique name.

```
// nameOptions=26 means include a unique suffix number
// but only if there is a name conflict
/NAME={":filename:","",26}          // 26 = 2 | 8 | 16 (bits 1, 3, and 4 set)
```

LoadWave creates a wave named Data if it does not already exist. If it exists LoadWave creates a wave named Data0, or Data1, or ... where the suffix number is chosen so that the resulting wave name is unique.

Loading Multiple Waves Using the File Name

In this section, we assume that we are loading a file named "Data.txt" and that we are loading three waves from the file.

```
// nameOptions=0 means omit the normal name
/NAME={":filename:","",0}
```

In this case, since we requested no suffix number, all of the generated wave names are Data and LoadWave displays a dialog in which you can enter unique names.

```
// nameOptions=4 means always include a sequential suffix number
/NAME={":filename:","",4}           // Bit 2 set
```

LoadWave generates wave names Data0, Data1, and Data2. If any of these waves exist and you include the /O (overwrite) flag, they are overwritten. If they exist and you omit /O, LoadWave displays a dialog in which you can enter unique names.

```
// nameOptions=20 means always include a unique suffix number
/NAME={":filename:","",20}          // 20 = 4 | 16 (bits 2 and 4 set)
```

LoadWave generates wave names like Data0, Data1, and Data2 where the suffix numbers are chosen to make the names unique. If you execute the same command on the same file a second time, LoadWave generates wave names Data3, Data4, and Data5.

Loading Multiple Waves Using the File Name and the Normal Name

In this section, we assume that we are loading a file named "Data.txt" and that we are loading three waves from the file. We further assume that the file contains the column names ColumnA, ColumnB, and ColumnC and that we include the /W flag to load the column names from the file.

```
// nameOptions=1 means include the normal name
/W /NAME={":filename:_","",1}
```

This generates wave names Data_ColumnA, Data_ColumnB, and Data_ColumnC. If any of these waves exist and you include the /O (overwrite) flag, they are overwritten. If they exist and you omit /O, LoadWave displays a dialog in which you can enter unique names.

```
// nameOptions=21 means always include the normal name and a unique suffix number
/W /NAME={":filename:_","",21}      // 21 = 1 | 4 | 16 (bits 0, 2 and 4 set)
```

LoadWave generates wave names Data_ColumnA0, Data_ColumnB0, and Data_ColumnC0 where the suffix numbers are chosen to make the wave names unique. If you execute the same command on the same file a second time, LoadWave generates wave names Data_ColumnA1, Data_ColumnB1, and Data_ColumnC1.

This technique could also be used with the /B flag instead of the /W flag to create wave names combining the file name and additional names explicitly specified by /B. See **Setting Wave Names When Loading Data Files** on page II-176 for a functional example.

Other LoadWave Features

This section discusses other issues that apply to loading text data files.

Loading Custom Date Formats

This section applies to loading delimited text (/J), fixed field text (/F) and general text (/G) files.

Here are some examples showing custom date formats and how you would specify them using the **LoadWave** /R flag:

October 11, 1999	/R={English, 2, 4, 1, 1, "Month DayOfMonth, Year", 40}
Oct 11, 1999	/R={English, 2, 3, 1, 1, "Month DayOfMonth, Year", 40}
11 October 1999	/R={English, 2, 4, 1, 1, "DayOfMonth Month Year", 40}
11 Oct 1999	/R={English, 2, 3, 1, 1, "DayOfMonth Month Year", 40}
10/11/99	/R={English, 1, 2, 1, 1, "Month/DayOfMonth/Year", 40}
11-10-99	/R={English, 1, 2, 2, 1, "DayOfMonth-Month-Year", 40}
11-Jun-99	/R={English, 1, 3, 2, 1, "DayOfMonth-Month-Year", 40}
991011	/R={English,1,2,2,1,"YearMonthDayOfMonth", 40}

When loading data as delimited text, if you use a date format containing a comma, such as "October 11, 1999", you must use the /V flag to make sure that LoadWave will not treat the comma as a delimiter.

When loading a date format that consists entirely of digits, such as 991011, you must use the LoadWave/B flag to tell LoadWave that the data is a date. Otherwise, LoadWave will treat it as a regular number.

Specifying Characteristics of Individual Columns

The LoadWave /B=*columnInfoStr* flag provides information to LoadWave for each column in a delimited text (/J), fixed field text (/F) or general text (/G) file. The flag overrides LoadWave's normal behavior. In most cases, you will not need to use it. /B is useful in user-defined functions when you need additional control.

columnInfoStr is constructed as follows:

```
"<column info>;<column info>; . . .;<column info>;"
```

where <column info> consists of one or more of the following:

C=<number> The number of columns controlled by this column info specification. <number> is an integer greater than or equal to one.

F=<format> A code that specifies the data type of the column or columns. <format> is an integer from -2 to 10. The meaning of the <format> is:

-2: Text. The column will be loaded into a text wave.

-1: Format unknown. Igor will deduce the format.

0 to 5: Numeric.

6: Date

7: Time

8: Date/Time

9: Octal number

10: Hexadecimal number

The F= flag is used for delimited text and fixed field text files only. It is ignored for general text files.

N=<name> A name to use for the column. <name> can be a standard name (e.g., wave0) or a quoted liberal name (e.g., 'Heart Rate'). If <name> is '_skip_' then LoadWave will skip the column.

The N= flag works for delimited text, fixed field text and general text files.

See **LoadWave Generation of Wave Names** on page II-142 for further discussion.

T=<numtype> A number that specifies what the numeric type for the column should be. This flag overrides the LoadWave/D flag. It has no effect on columns whose format is text. <numtype> must be one of the following:

2: 32-bit float

4: 64-bit float

8: 8-bit signed integer

16: 16-bit signed integer

32: 32-bit signed integer

72: 8-bit unsigned integer

80: 16-bit unsigned integer

96: 32-bit unsigned integer

W=<width> The column field width for fixed field files. <width> is an integer greater than or equal to one. Fixed width files are FORTRAN-style files in which a fixed number of bytes is allocated for each column and spaces are used as padding.

The W= flag is used for fixed field text only.

Here is an example of the /B=*columnInfoStr* flag:

```
/B="C=1,F=-2,T=2,W=20,N=Factory;  C=1,F=6,W=16,T=4,N=MfgDate;
C=1,F=0,W=16,T=2,N=TotalUnits;  C=1,F=0,W=16,T=2,N=DefectiveUnits;"
```

This example is shown on two lines but in a real command it would be on a single line. In a procedure, it could be written as:

```
String columnInfoStr = ""
columnInfoStr += "C=1,F=-2,T=2,W=20,N=Factory;"
columnInfoStr += "C=1,F=6,T=4,W=16,N=MfgDate;"
columnInfoStr += "C=1,F=0,T=2,W=16,N=TotalUnits;"
columnInfoStr += "C=1,F=0,T=2,W=16,N=DefectiveUnits;"
columnInfoStr += "C=1,F=0,T=2,W=16,N=DefectiveUnits;"
```

Note that each flag inside the quoted string ends with either a comma or a semicolon. The comma separates one flag from the next within a particular column info specification. The semicolon marks the end of a column info specification. The trailing semicolon is required. Spaces and tabs are permitted within the string.

This example provides information about a file containing four columns.

The first column info specification is "C=1;F=-2,T=2,W=20,N=Factory;". This indicates that the specification applies to one column, that the column format is text, that the numeric format is single-precision floating point (but this has no effect on text columns), that the column data is in a fixed field width of 20 bytes, and that the wave created for this column is to be named Factory.

The second column info specification is "C=1;F=6,T=4,W=16,N=MfgDate;". This indicates that the specification applies to one column, that the column format is date, that the numeric format is double-precision floating point (double precision should always be used for dates), that the column data is in a fixed field width of 16 bytes, and that the wave created for this column is to be named MfgDate.

The third column info specification is "C=1;F=0,T=2,W=16,N=TotalUnits;". This indicates that the specification applies to one column, that the column format is numeric, that the numeric format is single-precision floating point, that the column data is in a fixed field width of 16 bytes, and that the wave created for this column is to be named TotalUnits.

The fourth column info specification is the same as the third except that the wave name is DefectiveUnits.

All of the items in a column specification are optional. The default value for each item in the column info specification is as follows:

C=<number> C=1. Specifies that the column info describes one column.

F=<format> F=-1. Determines the format as dictated by the /K flag. If /K=0 is used, LoadWave will automatically determine the column format.

N=<name> N=_auto_. Generates the wave name as it would if the /B flag were omitted.

T=<numtype> Defaults to T=4 (double precision) if the LoadWave/D flag is used or to T=2 (single precision) if the /D flag is omitted.

W=<width> W=0. For a fixed width file, LoadWave will use the default field width specified by the /F flag unless you provide an explicit field width greater than 0 using W=<width>.

Taking advantage of the default values, we could abbreviate the example as follows:

```
/B="F=-2,W=20,N=Factory;  F=6,T=4,W=16,N=MfgDate;
W=16,N=TotalUnits; W=16,N=DefectiveUnits;"
```

If the file were not a fixed field text file, we would omit the W= flag and the example would become:

```
/B="F=-2,N=Factory;  F=6,T=4,N=MfgDate;  N=TotalUnits;  N=DefectiveUnits;"
```

Here are some more examples and discussion that illustrate the use of the /B=*columnInfoStr* flag.

In this example, the /B flag is used solely to specify the name to use for the waves created from the columns in the file:

```
/B="N=WaveLength;  N=Absorbance;"
```

The wave names in the previous example are standard names. If you want to use liberal names, such as names containing spaces or dots, you must use single quotes. For example:

```
/B="N='Wave Number';  N='Reflection Angle';"
```

The name that you specify via N= can not be used if overwrite is off and there is already a wave with this name or if the name conflicts with a macro, function or operation or variable. In these cases, LoadWave generates a unique name by adding one or more digits to the name specified by the N= flag for the column in question. You can avoid the problem of a conflict with another wave name by using the overwrite (/O) flag or by loading your data into a newly-created data folder. You can minimize the likelihood of a name conflict with a function, operation or variable by avoiding vague names.

If you specify the same name in two N= flags, LoadWave will generate an error, so make sure that the names are unique.

Except if the specified name is '_skip_', the N= flag generates a name for one column only, even if the C= flag is used to specify multiple columns. Consider this example:

```
/B="C=10,N=Test;"
```

This ostensibly uses the name Test for 10 columns. However, wave names must be unique, so LoadWave will not do this. It will use the name Test for just the first column and the other columns will receive default names.

You can load a subset of the columns in the file using the /L flag. Even if you do this, the column info specifications that you provide via the /B flag start from the first column in the file, not from the first column to be loaded. For example, if you are using /L to skip columns 0 and 1, you must skip columns 0 and 1 in the column info specification, like this:

```
// Skip column 0 and 1 and name the successive columns
/L={0,0,0,2,0} /B="C=2;N=Column2;N=Column3;"
```

The "C=2;" part accepts default specifications for columns 0 and 1 and the subsequent specifications apply to subsequent columns.

You can achieve the same thing using /B without /L, like this:

```
/B="C=2,N='_skip_';N=Column2;N=Column3;"
```

Also, when loading data into a matrix wave, LoadWave uses only one name. If you specify more than one name, only the first is used. If you are loading data into a matrix and also skipping columns, the explanation above about skipping applies.

In this example, the /B flag solely specifies the format of each column in the file. The file in question starts with a text column, followed by a date column, followed by 3 numeric columns.

```
/B="F=-2;  F=6;  C=3,F=0"
```

In most cases, it is not necessary to use the F= flag because LoadWave can automatically deduce the formats. The flag is useful for those cases where it deduces the column formats incorrectly. It is also useful to force LoadWave to interpret a column as octal or hexadecimal because LoadWave can not automatically deduce these formats.

The numeric codes (0...10) used by the F= flag are the same as the codes used by the ModifyTable operation. If you create a table using the /E flag, the F= flag controls the numeric format of table columns.

The code -1 is not a real column format code. If you use F=-1 for a particular column, LoadWave will deduce the format for that column from the column text.

In this example, the /B flag is used solely to specify the width of each column in a fixed field file. This file contains a 20 byte column followed by ten 16 byte columns followed by three 24 byte columns.

```
/B="C=1,W=20;  C=10,W=16;  C=3,W=24"
```

The field widths specified via W= override the default field width specified by the /F flag. If all of the columns in the file have the same field width then you can use just the /F flag.

You can load a subset of the columns in the file using the /L flag. Even if you do this, the column info specifications that you provide via the /B flag start from the first column in the file, not from the first column to be loaded.

Other LoadWave Issues

This section discusses other issues that apply to the LoadWave operation.

LoadWave Text Encoding Issues

This section discusses LoadWave text encoding issues of interest to advanced users. It assumes that you are familiar with the general topic of text encodings as explained under **Text Encodings** on page III-459.

Since Igor stores all text internally as UTF-8, it must convert text read from a file from the source text encoding to UTF-8. In order to do this it needs to know the source text encoding.

When loading an Igor binary wave file LoadWave ignores the /ENCG=*textEncoding* flag. The loaded wave's text encoding is determined as described under **LoadWave Text Encodings for Igor Binary Wave Files** on page III-475. The rest of this section applied to loading data from plain text files, not from Igor binary wave files.

When loading a text data file you can use the /ENCG=*textEncoding* flag to tell Igor what that text encoding is. See **Text Encoding Names and Codes** on page III-490 for a list of accepted values for textEncoding.

LoadWave uses the text encoding specified by /ENCG and the rules described under **Determining the Text Encoding for a Plain Text File** on page III-467 to determine the source text encoding for conversion of the text file's data to UTF-8. If you omit /ENCG or specify /ENCG=0, the specified text encoding is unknown and does not factor into the determination of the source text encoding. If following the rules does not identify a text encoding that works for converting the file's text to UTF-8, Igor displays the Choose Text Encoding dialog.

If the file contains nothing but ASCII characters, as is often the case, then any byte-oriented text encoding will work and there is no need to use the /ENCG flag.

When you are loading a huge file (e.g., hundreds of megabytes), finding a valid source text encoding may add a noticeable amount to the time it takes to load the file. If you know that the file is either all ASCII or is valid UTF-8, you can tell LoadWave to skip text encoding conversion altogether using an optional parameter, like this:

```
/ENCG={1,4}
```

"1" tells LoadWave that the text is valid as UTF-8, meaning that it is all ASCII or, if it contains non-ASCII characters, they are properly encoded as UTF-8.

"4" tells LoadWave to assume that the text is valid as UTF-8 and skip all validation and conversion.

In testing with a 200 MB delimited text file containing 1 million rows and 20 columns, we found that using /ENCG={1,4} saved about 10% of the time.

NOTE: If you use this flag but the file is not valid UTF-8 and you are loading data into text wave, the text waves will wind up with invalid data which will result in errors when you use the waves later.

As noted above, if following the rules does not identify a text encoding that works for converting the file's text to UTF-8, Igor displays the Choose Text encoding dialog. If you are loading many files using an unattended, automated procedure, displaying this dialog will cause your procedure to grind to a halt. You can prevent this by using another optional flag, like this:

```
/ENCG={1,8}
```

If you use this flag and LoadWave can not determine the source text encoding for a file, it will return an error. If you want your procedure to continue with other files you must check for and handle the error using GetRTError.

Loading Very Large Files

The number of waves (columns) or points (rows) that LoadWave can handle when loading a text file is limited only by available memory.

You can improve the speed and efficiency of loading very large files (i.e., more than 50,000 lines of data) using the *numLines* parameter of the /L flag. Normally this parameter is used to load a section of the file instead of the whole file. However, in delimited, general text and fixed field text loads, the *numLines* parameter also specifies how many rows the waves should initially have. Thus all of the required memory is allocated at the start of the load, rather than increasing the number of wave rows over and over as more lines of data are loaded. When loading very large files, if you know the exact number of lines of data in the file, use the *numLines* parameter of the /L flag. If you don't know the exact number of lines, you can provide a number that is guaranteed to be larger.

If you omit the /L flag or if the *numLines* parameter is zero, and if you are loading a file greater than 500,000 bytes, LoadWave automatically counts the lines of data in the file so that the entire wave can be allocated before data loading starts. This acts as if you used /L and set *numLines* to the exact correct value which normally speeds the loading process considerably. You can disable this feature by using the /V flag and setting bit 2 of the *loadFlags* parameter to 1.

Escape Sequences

An escape sequence is a two-character sequence used to represent special characters in plain text. Escape sequences are introduced by a backslash character.

By default, in a text column, LoadWave interprets the following escape sequences: \t (tab), \n (linefeed), \r (carriage-return), \\ (backslash), \" (double-quote) and \' (single-quote). This works well with Igor's Save operation which uses escape sequences to encode the first four of these characters.

If you are loading a file that does not use escape sequences but which does contain backslashes, you can disable interpretation of these escape sequences by setting bit 3 of the loadFlags parameter of the /V flag. This is mainly of use for loading a text file that contains unescaped Windows file system paths.

Loading Igor Text Files

An Igor Text file consists of keywords, data and Igor commands. The data can be numeric, text or both and can be of dimension 1 to 4. Many Igor users have found this to be an easy and powerful format for exporting data from their own custom programs into Igor.

The file name extension for an Igor Text file is ".itx". Old versions of Igor used ".awav" and this is still accepted.

Examples of Igor Text

Here are some examples of text that you might find in an Igor Text file.

Simple Igor Text

```
IGOR
WAVES/D unit1, unit2
BEGIN
      19.7  23.9
      19.8  23.7
      20.1  22.9
END
X SetScale x 0,1, "V", unit1; SetScale d 0,0, "A", unit1
X SetScale x 0,1, "V", unit2; SetScale d 0,0, "A", unit2
```

Loading this would create two double-precision waves named unit1 and unit2 and set their X scaling, X units and data units.

Igor Text with extra commands

```
IGOR
WAVES/D/O xdata, ydata
BEGIN
      98.822      486.528
      109.968     541.144
      119.573     588.21
      133.178     654.874
      142.906     702.539
END
X SetScale d 0,0, "V", xdata
X SetScale d 0,0, "A", ydata
X Display/N=TempGraph ydata vs xdata
X ModifyGraph mode=2,lsize=5
X CurveFit line ydata /X=xdata /D
X Textbox/A=LT/X=0/Y=0 "ydata= \\{W_coef[0]}+\\{W_coef[1]}*xdata"
X PrintGraphs TempGraph
X KillWindow TempGraph              // Kill the graph
X KillWaves xdata, ydata, fit_ydata // Kill the waves
```

Loading this would create two double-precision waves and set their data units. It would then make a graph, do a curve fit, annotate the graph and print the graph. The last two lines do housekeeping.

Igor Text File Format

An Igor Text file starts with the keyword **IGOR**. The rest of the file may contain blocks of data to be loaded into waves or Igor commands to be executed and it must end with a blank line.

A block of data in an Igor Text file must be preceded by a declaration of the waves to be loaded. This declaration consists of the keyword **WAVES** followed by optional flags and the names of the waves to be loaded. Next the keyword **BEGIN** indicates the start of the block of data. The keyword **END** marks the end of the block of data.

A file can contain any number of blocks of data, each preceded by a declaration. If the waves are 1D, the block can contain any number of waves but waves in a given block must all be of the same data type. Multidimensional waves must appear one wave per block.

A line of data in a block consists of one or more numeric or text items with tabs separating the numbers and a terminator at the end of the line. The terminator can be CR, LF, or CRLF. Each line should have the same number of items.

You can’t use blanks, dates, times or date/times in an Igor Text file. To represent a missing value in a numeric column, use “NaN” (not-a-number). To represent dates or times, use the standard Igor date format (number of seconds since 1904-01-01).

There is no limit to the number of waves or number of points except that all of the data must fit in available memory.

The WAVES keyword accepts the following optional flags:

Flag	Effect
/N=(…)	Specifies size of each dimension for multidimensional waves.
/O	Overwrites existing waves.
/R	Makes waves real (default).
/C	Makes waves complex.
/S	Makes waves single precision floating point (default).

Flag	Effect
/D	Makes waves double precision floating point.
/I	Makes waves 32 bit integer.
/W	Makes waves 16 bit integer.
/B	Makes waves 8 bit integer.
/U	Makes integer waves unsigned.
/T	Specifies text data type.

Normally you should make single or double precision floating point waves. Integer waves are normally used only to contain raw data acquired via external operations. They are also appropriate for storing image data.

The /N flag is needed only if the data is multidimensional but the flag is allowed for one-dimensional data, too. Regardless of the dimensionality, the dimension size list must always be inside parentheses. Examples:

```
WAVES/N=(5) wave1D
WAVES/N=(3,3) wave2D
WAVES/N=(3,3,3) wave3D
```

Integer waves are signed unless you use the /U flag to make them unsigned.

If you use the /C flag then a pair of numbers in a line supplies the real and imaginary value for a single point in the resulting wave.

If you specify a wave name that is already in use and you don’t use the overwrite option, Igor displays a dialog so that you can resolve the conflict.

The /T flag makes text rather than numeric waves. See **Loading Text Waves from Igor Text Files** on page II-154.

A command in an Igor Text file is introduced by the keyword **X** followed by a space. The command follows the X on the same line. When Igor encounters this while loading an Igor Text file it executes the command.

Anything that you can execute from Igor’s command line is acceptable after the X. Introduce comments with “X //”. There is no way to do conditional branching or looping. However, you can call an Igor procedure defined in a built-in or auxiliary procedure window.

Commands, introduced by X, are executed as if they were entered on the command line or executed via the Execute operation. Such command execution is not thread-safe. Therefore, you can not load an Igor text file containing a command from an Igor thread.

Setting Scaling in an Igor Text File

When Igor writes an Igor Text file, it always includes commands to set each wave’s scaling, units and dimension labels. It also sets each wave’s note.

If you write a program that generates Igor Text files, you should set at least the scaling and units. If your 1D data is uniformly spaced in the X dimension, you should use the SetScale operation to set your waves X scaling, X units and data units. If your data is not uniformly spaced, you should set the data units only. For multidimensional waves, use SetScale to set Y, Z and T units if needed.

The Load Waves Dialog for Igor Text

The basic process of loading data from an Igor Text file is as follows:

1. Choose Data→Load Waves→Load Waves to display the Load Waves dialog.
2. Choose Igor Text from the File Type pop-up menu.
3. Click the File button to select the file containing the data.

4. Click Do It.

When you click Do It, Igor’s LoadWave operation runs. It executes the Load Igor Text routine which loads the file.

If you choose Data→Load Waves→Load Igor Text instead of choosing Data→Load Waves→Load Waves, Igor displays the Open File dialog in which you can select the Igor Text file to load directly. This is a shortcut that skips the Load Waves dialog.

Loading MultiDimensional Waves from Igor Text Files

In an Igor Text file, a block of wave data is preceded by a WAVES declaration. For multidimensional data, you must use a separate block for each wave. Here is an example of an Igor Text file that defines a 2D wave:

```
IGOR
WAVES/D/N=(3,2) wave0
BEGIN
      1     2
      3     4
      5     6
END
```

The “/N=(3,2)” flag specifies that the wave has three rows and two columns. The first line of data (1 and 2) contains data for the first row of the wave. This layout of data is recommended for clarity but is not required. You could create the same wave with:

```
IGOR
WAVES/D/N=(3,2) wave0
BEGIN
      1     2     3     4     5     6
END
```

Igor merely reads successive values and stores them in the wave, storing a value in each column of the first row before moving to the second row. All white space (spaces, tabs, return and linefeed characters) are treated the same.

When loading a 3D wave, Igor expects the data to be in column/row/layer order. You can leave a blank line between layers for readability but this is not required.

Here is an example of a 3 rows by 2 columns by 2 layers wave:

```
IGOR
WAVES/D/N=(3,2,2) wave0
BEGIN
      1     2
      3     4
      5     6

      11    12
      13    14
      15    16
END
```

The first 6 numbers define the values of the first layer of the 3D wave. The second 6 numbers define the values of the second layer. The blank line improves readability but is not required.

When loading a 4D wave, Igor expects the data to be in column/row/layer/chunk order. You can leave a blank line between layers and two blank lines between chunks for readability but this is not required.

If loading a multidimensional wave, Igor expects that the dimension sizes specified by the /N flag are accurate. If there is more data in the file than expected, Igor ignores the extra data. If there is less data than expected, some of the values in the resulting waves will be undefined. In either of these cases, Igor prints a message in the history area to alert you to the discrepancy.

Loading Text Waves from Igor Text Files

Loading text waves from Igor Text files is similar to loading them from delimited text files except that in an Igor Text file you declare a wave's name and type. Also, text strings are quoted in Igor Text files as they are in Igor's command line. Here is an example of Igor Text that defines a text wave:

```
IGOR
WAVES/T textWave0, textWave1
BEGIN
      "This"      "Hello"
      "is"        "out"
      "a test"    "there"
END
```

All of the waves in a block of an Igor Text file must have the same number of points and data type. Thus, you can not mix numeric and text waves in the same block. You can have any number of blocks in one Igor Text file.

As this example illustrates, you must use double quotes around each string in a block of text data. If you want to embed a quote, tab, carriage return or linefeed within a single text value, use the escape sequences \", \t, \r or \n. Use \\ to embed a backslash. For less common escape sequences, see **Escape Sequences in Strings** on page IV-14.

Loading Igor Binary Data

This section discusses loading Igor Binary data into memory.

Igor stores Igor Binary data in two ways: one wave per Igor binary wave file in unpacked experiments and multiple waves within a packed experiment file.

When you open an experiment, Igor *automatically* loads the Igor Binary data to recreate the experiment's waves. The main reason to *explicitly* load an Igor binary wave file is if you want to access the same data from multiple experiments. The easiest way to load data from another experiment is to use the Data Browser (see **The Data Browser** on page II-114).

Warning: You can get into trouble if two Igor experiments load data from the same Igor binary wave file. See **Sharing Versus Copying Igor Binary Wave Files** on page II-156 for details.

There are a number of ways to load Igor Binary data into the current experiment in memory. Here is a summary. For most users, the first and second methods — which are simple and easy to use — are sufficient.

Method	Loads	Action	Purpose
Open Experiment	Packed and unpacked files	Restores the experiment to the state in which it was last saved.	To restore experiment.
Data Browser	Packed and unpacked files	Copies data from one experiment to another. See **The Browse Expt Button** on page II-117 for details.	To collect data from different sources for comparison.
Desktop Drag and Drop	Unpacked files only	Copies data from one experiment to another or shares between experiments.	To collect data from different sources for comparison.
Load Waves Dialog	Unpacked files only	Copies data from one experiment to another or shares between experiments.	To create a LoadWave command that can be used in an Igor procedure.
LoadWave Operation	Unpacked files only	Copies data from one experiment to another or shares between experiments. See **LoadWave** on page V-508 for details.	To automatically load data using an Igor Procedure.

Method	Loads	Action	Purpose
LoadData Operation	Packed and unpacked files	Copies data from one experiment to another. See **LoadData** on page V-500 for details.	To automatically load data using an Igor Procedure.

The Igor Binary Wave File

The Igor binary wave file format is Igor's native format for storing waves. This format stores one wave per file very efficiently. The file includes the numeric contents of the wave (or text contents if it is a text wave) as well as all of the auxiliary information such as the dimension scaling, dimension and data units and the wave note. In an Igor packed experiment file, any number of Igor Binary wave files can be packed into a single file.

The file name extension for an Igor binary wave file is ".ibw". Old versions of Igor used ".bwav" and this is still accepted. The Macintosh file type code is IGBW and the creator code is IGR0 (last character is zero).

The name of the wave is stored *inside* the Igor binary wave file. It does not come from the name of the file. For example, wave0 might be stored in a file called "wave0.ibw". You could change the name of the file to anything you want. This does not change the name of the wave stored in the file.

The Igor binary wave file format was designed to save waves that are part of an Igor experiment. In the case of an unpacked experiment, the Igor binary wave files for the waves are stored in the experiment folder and can be loaded using the LoadWave operation. In the case of a packed experiment, data in Igor Binary format is packed into the experiment file and can be loaded using the **LoadData** operation.

.ibw files do not support waves with more than 2 billion elements. you can use the **SaveData** operation or the Data Browser Save Copy button to save very large waves in a packed experiment file (.pxp) instead.

Some Igor users have written custom programs that write Igor binary wave files which they load into an experiment. Igor Technical Note #003, "Igor Binary Format", provides the details that a programmer needs to do this. See also Igor Pro Technical Note PTN003.

The Load Waves Dialog for Igor Binary

The basic process of loading data from an Igor binary wave file is as follows:

1. Choose Data→Load Waves→Load Waves to display the Load Waves dialog.
2. Choose Igor Binary from the File Type pop-up menu.
3. Click the File button to select the file containing the data.
4. Check the "Copy to home" checkbox.
5. Click Do It.

When you click Do It, Igor's LoadWave operation runs. It executes the Load Igor Binary routine which loads the file. If the wave that you are loading has the same name as an existing wave or other Igor object, Igor presents a dialog in which you can resolve the conflict.

Notice the "Copy to home" checkbox in the Load Waves dialog. It is very important.

If it is checked, Igor will disassociate the wave from its source file after loading it into the current experiment. When you next save the experiment, Igor will store a new copy of the wave with the current experiment. The experiment will not reference the original source file. We call this "copying" the wave to the current experiment.

If "Copy to home" is unchecked, Igor will keep the connection between the wave and the file from which it was loaded. When you save the experiment, it will contain a *reference* to the source file. We call this "sharing" the wave between experiments.

We strongly recommend that you copy waves rather than share them. See **Sharing Versus Copying Igor Binary Wave Files** on page II-156 for details.

If you choose Data→Load Waves→Load Igor Binary instead of choosing Data→Load Waves→Load Waves, Igor displays the Open File dialog in which you can select the Igor binary wave file to load directly. This is a shortcut that skips the Load Waves dialog. When you take this shortcut, you lose the opportunity to set the “Copy to home” checkbox. Thus, during the load operation, Igor presents a dialog from which you can choose to copy or share the wave.

The LoadData Operation

The LoadData operation provides a way for Igor programmers to automatically load data from packed Igor experiment files or from a file-system folder containing unpacked Igor binary wave files. It can load not only waves but also numeric and string variables and a hierarchy of data folders that contains waves and variables.

The Data Browser’s Browse Expt button provides interactive access to the LoadData operation and permits you to drag a hierarchy of data from one Igor experiment into the current experiment in memory. To achieve the same functionality in an Igor procedure, you need to use the LoadData operation directly. See the **LoadData** operation (see page V-500).

LoadData, accessed from the command line or via the Data Browser, has the ability to overwrite existing waves, variables and data folders. Igor automatically updates any graphs and tables displaying the overwritten waves. This provides a very powerful and easy way to view sets of identically structured data, as would be produced by successive runs of an experiment. You start by loading the first set and create graphs and tables to display it. Then, you load successive sets of identically named waves. They overwrite the preceding set and all graphs and tables are automatically updated.

Sharing Versus Copying Igor Binary Wave Files

There are two reasons for loading a binary file that was created as part of another Igor experiment: you may want your current experiment to *share* data with the other experiment or, you may want to *copy* data to the current experiment from the other experiment.

There is a potentially serious problem that occurs if two experiments share a file. The file can not be in two places at one time. Thus, it will be stored *with* the experiment that created it but *separate from* the other. The problem is that, if you move or rename files or folders, the second experiment will be unable to find the binary file.

Here is an example of how this problem can bite you.

Imagine that you create an experiment at work and save it as an unpacked experiment file on your hard disk. Let’s call this “experiment A”. The waves for experiment A are stored in individual Igor binary wave files in the experiment folder.

Now you create a new experiment. Let’s call this “experiment B”. You use the Load Igor Binary routine to load a wave from experiment A into experiment B. You elect to share the wave. You save experiment B on your hard disk. Experiment B now contains a *reference* to a file in experiment A’s home folder.

Now you decide to use experiment B on another computer so you copy it to the other computer. When you try to open experiment B, Igor can’t find the file it needs to load the shared wave. This file is back on the hard disk of the original computer.

A similar problem occurs if, instead of moving experiment B to another computer, you change the name or location of experiment A’s folder. Experiment B will still be looking for the shared file under its old name or in its old location and Igor will not be able to load the file when you open experiment B.

Because of this problem, we recommend that you *avoid file sharing* as much as possible. If it is necessary to share a binary file, you will need to be very careful to avoid the situation described above.

The Data Browser always copies when transferring data from disk into memory.

For more information on the problem of sharing files, see **References to Files and Folders** on page II-24.

Copy or Share Wave Dialog

When you load an Igor binary wave file interactively (i.e., not via a command), by default Igor displays the Copy or Share Wave dialog which allows you to choose to copy the wave into the current experiment or to share it with other experiments. You can change the default behavior to always copy or always share using the Data Loading section of the Miscellaneous Settings dialog.

If you interactively load multiple Igor binary wave files at one time, by default, you will see the Copy or Share Wave dialog once for each file being loaded. In Igor Pro 9 and later, you can apply your choice to all of the files by checking the Apply to All Igor Binary Wave Files in the Batch Currently Being Loaded checkbox. This feature is available only when you:

- Choose Data→Load Waves→Load Igor Binary
- Drag multiple Igor binary wave files into the Igor command window
- Drag multiple Igor binary wave files into the Igor frame window (Windows only)
- Drag multiple Igor binary wave files into the Data Browser (Windows only)

The Copy or Share Wave dialog is not displayed when you load Igor binary wave files using the Data Browse Browse Expt button. In that case, the waves are always copied to the current experiment.

When loading multiple Igor binary wave files, the output variables V_Flag, S_waveNames, S_path, and S_fileName reflect only the last file loaded.

Loading Image Files

You can load JPEG, PNG, TIFF, BMP, and Sun Raster image files into Igor Pro using the Load Image dialog.

You can load numeric plain text files containing image data using the Load Waves dialog via the Data menu. Check the "Load columns into matrix" checkbox.

You can load images from HDF5 files. For help, execute this in Igor:

```
DisplayHelpTopic "HDF5 in Igor Pro"
```

You can load images from HDF4 files. For help, execute this in Igor:

```
DisplayHelpTopic "HDF Loader XOP"
```

You can also load images by grabbing frames. See the **NewCamera** operation.

The Load Image Dialog

To load an image file into an Igor wave, choose Data→Load Waves→Load Image to display the Load Image dialog.

When you choose a particular type of image file from the File Type pop-up menu, you are setting a file filter that is used when displaying the image file selection dialog. If you are not sure that your image file has the correct file name extension, choose “Any” from the File Type pop-up menu so that the filter does not restrict your selection.

The name of the loaded wave can be the name of the file or a name that you specify. If you enter a wave name in the dialog that conflicts with an existing wave name and you do not check the Overwrite Existing Waves checkbox, Igor appends a numeric suffix to the new wave name.

Loading PNG Files

There are two menu choices for the PNG format: Raw PNG and PNG. When Raw PNG is selected, the data is read directly from the file into the wave. When PNG is selected, the file is loaded into memory, an offscreen image is created, and the wave data is set by reading the offscreen image. In nearly all cases, you should choose Raw PNG.

When loading a PNG file, the image data is loaded into a 3D Igor RGB wave containing unsigned byte RGB elements in layers 0, 1, and 2. If the image file includes an alpha channel, the resulting 3D RGBA wave includes an alpha layer.

You can convert a 3D waves containing an RGB image into a grayscale image using the **ImageTransform** operation with the rgb2gray keyword.

Loading JPEG File

When loading a JPEG file, the image data is loaded into a 3D Igor RGB wave containing unsigned byte RGB elements in layers 0, 1, and 2. JPEG does not support alpha.

You can convert a 3D waves containing an RGB image into a grayscale image using the **ImageTransform** operation with the rgb2gray keyword.

Loading BMP Files

When loading a BMP file, the image data is loaded into a 3D Igor RGB wave containing unsigned byte RGB elements in layers 0, 1, and 2. BMP does not support alpha.

You can convert a 3D waves containing an RGB image into a grayscale image using the **ImageTransform** operation with the rgb2gray keyword.

Loading TIFF Files

A TIFF file can store one or more images in many formats. The most common formats are:

- Bilevel
- Grayscale
- Palette color
- Full color (RGB, RGBA, CMYK)

A bilevel image consists of one plane of data in which each pixel can represent black or white. Igor loads a bilevel image into a 2D wave.

A grayscale image consists of one plane of data in which each pixel can represent a range of intensities. Igor loads a grayscale image into a 2D wave.

A palette color image is like a grayscale but includes a color palette. Igor loads the grayscale image into a 2D wave and also creates a colormap wave named with the suffix "_CMap".

RGB, RGBA, and CMYK images are loaded into 3D waves with 3 or 4 layers. Each layer stores the pixels for one color component.

TIFF files that contain multiple images are called TIFF stacks. There are two options for loading them:

- Load the images into a single 3D wave.

  This works with grayscale images only. Each grayscale image is loaded into a layer of the 3D output wave.
- Load each image into its own wave.

  This works with any kind of image. Each grayscale image is loaded into its own 2D wave. Each RGB, RGBA, or CMYK image is loaded into its own 3D wave.

You can specify a particular image, or range of images, to be loaded from a multi-image TIFF file. In the Load Image dialog, enter the zero-based index of the first image to load and the number of images to load from the TIFF stack.

You can display TIFF images using the NewImage operation and convert image waves into other forms using the ImageTransform operation.

You can convert a 3D waves containing an RGB image into a grayscale image using the **ImageTransform** operation with the rgb2gray keyword.

You can convert a number of 2D image waves into a 3D stack using the **ImageTransform** operation with the stackImages keyword.

Loading Sun Raster Files

Sun Raster files are loaded as 2D waves.

If the Sun Raster file includes a color map, Igor creates, in addition to the image wave, a colormap wave, named with the suffix "_CMap".

Loading Row-Oriented Text Data

All of the built-in text file loaders are column-oriented — they load the columns of data in the file into 1D waves. There is a row-oriented format that is fairly common. In this format, the file represents data for one wave but is written in multiple columns. Here is an example:

```
350         2.97         1.95         1.00         8.10         2.42
351         3.09         4.08         1.90         7.53         4.87
352         3.18         5.91         1.04         6.90         1.77
```

In this example, the first column contains X values and the remaining columns contain data values, written in row/column order.

Igor Pro does not have a file-loader extension to handle this format, but there is a WaveMetrics procedure file for it. To use it, use the Load Row Data procedure file in the “WaveMetrics Procedures:File Input Output” folder. It adds a Load Row Data item to the Macros menu. When you choose this item, Igor presents a dialog that offers several options. One of the options treats the first column as X values or as data. If you specify treating the column as X values, Igor will use it to determine the X scaling of the output wave, assuming that the values in the first column are evenly spaced. This is usually the case.

Loading Excel Files

You can load data from Excel files into Igor using the **XLLoadWave** operation directly or by choosing Data→Load Waves→Load Excel File which displays the Load Excel File dialog.

XLLoadWave loads numeric, text, date, time and date/time data from Excel files into Igor waves. It can load data from .xls and .xlsx files. It does not support .xlsb (binary format for large files) files. It also can not load password-protected Excel files.

On Macintosh, it is possible to have a worksheet open in Excel and to use XLLoadWave to load the worksheet into Igor at the same time. When you do this, Igor loads the most recently saved version of the worksheet. On Windows, you must close the worksheet in Excel before loading it in Igor.

Some programs unfortunately save tab-delimited or other non-Excel type files using the ".xls" extension. If you try to load one of these files, XLLoadWave will tell you that it is not an Excel binary file.

What XLLoadWave Loads

A worksheet can be very simple, consisting of just a rectangular block of numbers, or it can be very complex, with blocks of numbers, strings, and formulas mixed up in arbitrary ways. XLLoadWave is designed to pick a rectangular block of cells out of a worksheet, converting the columns into Igor waves.

XLLoadWave can load both numeric and text (string) data. An Excel column can contain a mix of numeric and text cells. An Igor wave must be all numeric or all text. When you load an Excel column into an Igor wave, you need to decide whether to load the data into a numeric wave or into a text wave. XLLoadWave can also load date, time, and date/time data into numeric waves.

Column and Wave Types

XLLoadWave provides the following methods of determining the type of wave that it will create for a given column. These methods are presented in the Load Excel File dialog and are controlled by the /C and /COLT flags of the XLLoadWave command line operation.

Treat all columns as numeric

This is the default method. If you have a simple block of numbers that you want to load into waves, this is the method to use, and you can forget about the others.

XLLoadWave creates a numeric wave for each Excel column that you are loading. If the column contains numeric cells, their values are stored in the corresponding point of the wave. If the column contains text cells, XLLoadWave stores NaNs (blanks) in the corresponding point of the wave.

Treat all columns as date

This is the same as the preceding method except that XLLoadWave converts the numeric data from Excel date/time format into Igor date/time format. See Excel Date/Time Versus Igor Date/Time for details.

When XLLoadWave creates a numeric wave that is to store dates or times, it always creates a double-precision wave, because double precision is required to accurately store dates. Also, XLLoadWave sets the data units of the wave to "dat". Igor recognizes "dat" as signifying that the wave contains dates and/or times when you use the wave in a graph as the X part of an XY pair.

In this method, when XLLoadWave displays the wave in a table, it uses date/time formatting for the table column. You can change the column format to just date or just time using the ModifyTable operation.

Treat all columns as text

XLLoadWave loads all columns into text waves.

If you load a column containing numeric cells into a text wave, Igor converts the numeric cell value into text and stores the resulting text in the wave.

Deduce from row

This is a good method to use for loading a mix of columns of different types (numeric and/or date and/or text) into Igor.

You tell XLLoadWave what row to look at. XLLoadWave examines the cells in that row. For a given column, if the cell is numeric then XLLoadWave creates a numeric wave and if the cell is text then XLLoadWave creates a text wave.

If a numeric cell uses an Excel built-in date, time, or date/time format, XLLoadWave converts the numeric data from Excel date/time format into Igor date/time format. XLLoadWave can not deduce date and time formatting for cells that are governed by custom cell formats. In this case, see Excel Date/Time Versus Igor Date/Time for details on manually conversion.

When XLLoadWave deduces the column type using this method, it sets the Igor table column format for date/time waves to either date, time or date/time, depending on the built-in cell format for the corresponding column in the Excel file.

Use column type string

Use this method if you have a mix of columns of different types (numeric and/or date and/or text) and the "deduce from row" method does not make the correct deduction. For example, in some files there may be no single row that is suitable for deducing the column type.

In this method, you provide a string that identifies the type of each column to be loaded. For example, the string "1T1D3N" tells XLLoadWave that the first column loaded is to be loaded into a text wave, the next column is to be loaded into a numeric date/time wave, and the next three columns are to be loaded into numeric waves. If you load more columns than are covered by the string, extra columns are loaded as numeric. Also, the string "N" means all columns are numeric, the string "D" means all columns are numeric date/time, and the string "T" means all columns are text. The string must not contain any blanks or other extraneous characters.

Here are examples of suitable strings:

"N"	All columns are numeric.
"T"	All columns are text.
"1T1D3N"	One text column followed by one numeric date/time column followed by three or more numeric columns.
"1T1N3T25N"	One text column followed by one numeric column followed by three text columns followed by 25 or more numeric columns.

When loading numeric columns, the "use column type string" method differs from the "treat all columns as numeric" method in one way. In the "Treat all columns as numeric" method, any text cells in the numeric column are treated as blanks. This behavior is compatible with previous versions of XLLoadWave. In the "use column type string" method, if XLLoadWave encounters a text cell in a numeric column, it converts the text cell into a number. If the text represents a valid number (e.g., "1.234"), this will produce a valid number in the Igor wave. If the text does not represent a valid number (e.g., "January"), this will produce a blank in the Igor wave. This is useful if you have a file that inadvertently contains a text cell in a numeric column.

XLLoadWave and Wave Names

As you can see in the Load Excel File dialog, XLLoadWave uses one of three ways to generate names for the Igor waves that it creates. First, it can take wave names from a row that you specify in the worksheet. In this case XLLoadWave expects that the row contains string values. Second, it can generate default wave names of the form ColumnA, ColumnB and so on, where the letter at the end of the name indicates the column in the worksheet from which the wave was created. Third, XLLoadWave can generate wave names of the form wave0, wave1 and so on using a base name, "wave" in this case, that you specify.

XLLoadWave supports a fourth wave naming method that is not available from the dialog: the /NAME flag. This flag allows you to specify the desired name for each column using a semicolon-separated string list.

There are several situations, described below, in which XLLoadWave changes the name of the wave that it creates from what you might expect. When this happens, XLLoadWave prints the original and new names in Igor's history area. After the load, you can use Igor's Rename operation to pick another name of your choice, if you wish.

If a name in the worksheet is too long, XLLoadWave truncates it to a legal length. If a name contains characters that are not allowed in standard Igor wave names, XLLoadWave replaces them with the underscore character.

If two names in the worksheet conflict with each other, XLLoadWave makes the second name unique by adding a prefix such as “D_” where the letter indicates the Excel column from which the wave is being loaded.

If a name in the worksheet conflicts with the name of an existing wave, XLLoadWave makes the name of the incoming wave unique by adding one or more digits unless you use the overwrite option. With the overwrite option on, the incoming data overwrites the existing wave.

If XLLoadWave needs to add one or more digits to a name to make it unique and if the length of the name is already at the limit for Igor wave names, XLLoadWave removes one or more characters from the middle of the name.

It is possible that a name taken from a cell in the worksheet might conflict with the name of an Igor operation, function or macro. For example, Date and Time are built-in Igor functions so a wave can not have these names. If such a conflict occurs, XLLoadWave changes the name and prints a message in Igor's history area showing the original and the new names.

XLLoadWave Output Variables

XLLoadWave sets the standard Igor file-loader output variables, V_flag, S_path, S_fileName, and S_waveNames. In addition it sets S_worksheetName to the name of the loaded worksheet within the workbook file.

Excel Date/Time Versus Igor Date/Time

Excel stores date/time information in units of days since January 1, 1900 or January 1, 1904. 1900 is the default on Windows and 1904 is the default on Macintosh. Igor stores dates in units of seconds since January 1, 1904.

If you use the Treat all columns as date, Deduce from row, or Use column type string methods for determining the column type, XLLoadWave automatically converts from the Excel format into the Igor format. If you use the Treat all columns as numeric method, you need to manually convert from Excel to Igor format.

If the Excel file uses 1904 as the base year, the conversion is:

```
wave *= 24*3600              // Convert days to seconds
```

If the Excel file uses 1900 as the base year, the conversion is:

```
wave *= 24*3600              // Convert days to seconds
wave -= 24*3600*365.5*4      // Account for four year difference
```

The use of 365.5 here instead of 365 accounts for a leap year plus the fact that the Microsoft 1900 date system represents 1/1/1900 as day 1, not as day 0.

When displaying time data, you may see a one second discrepancy between what Excel displays and what Igor displays in a table. For example, Excel may show "9:00:30" while Igor shows "9:00:29". The reason for this is that the Excel data is just short of the nominal time. In this example, the Excel cell contains a value that corresponds to, "9:00:30" minus a millisecond. When Excel displays times, it rounds. When Igor displays times, it truncates. If this bothers you, you can round the data in the Igor wave:

```
wave = round(wave)
```

In doing this rounding, you eliminate any fractional seconds in the data. That is why XLLoadWave does not automatically do the rounding.

Loading Excel Data Into a 2D Wave

XLLoadWave creates 1D waves. Here is an Igor function that converts the 1D waves into a 2D wave.

```
Function LoadExcelNumericDataAsMatrix(pathName, fileName, worksheetName,
                        startCell, endCell)
   String pathName         // Name of Igor symbolic path or "" to get dialog
   String fileName         // Name of file to load or "" to get dialog
   String worksheetName
   String startCell        // e.g., "B1"
   String endCell          // e.g., "J100"

   if ((strlen(pathName)==0) || (strlen(fileName)==0))
      // Display dialog looking for file.
      Variable refNum
      String filters = "Excel Files (*.xls,*.xlsx,*.xlsm):.xls,.xlsx,.xlsm;"
      filters += "All Files:.*;"
      Open/D/R/P=$pathName /F=filters refNum as fileName
      fileName = S_fileName         // S_fileName is set by Open/D
      if (strlen(fileName) == 0)    // User cancelled?
         return -2
      endif
   endif
   // Load row 1 into numeric waves
   XLLoadWave/S=worksheetName/R=($startCell,$endCell)/COLT="N"/O/V=0/K=0/Q fileName
   if (V_flag == 0)
      return -1                    // User cancelled
   endif

   String names = S_waveNames       // S_waveNames is created by XLLoadWave
   String nameOut = UniqueName("Matrix", 1, 0)
   Concatenate /KILL /O names, $nameOut   // Create matrix and kill 1D waves

   String format = "Created numeric matrix wave %s containing cells %s to %s in
                    worksheet \"%s\"\r"
   Printf format, nameOut, startCell, endCell, worksheetName
End
```

Loading Matlab MAT Files

The MLLoadWave operation loads Matlab MAT-files into Igor Pro. You can access it directly via the MLLoadWave operation or by choosing Data→Load Waves→Load Matlab MAT File which displays the Load Matlab MAT File dialog.

MLLoadWave relies on dynamic libraries provided by the Matlab application. You must have a compatible version of Matlab installed on your machine to use MLLoadWave. If you don't have Matlab or if your Matlab version is not compatible with Igor, or if you simply prefer to work with HDF5 files, see **Loading Version 7.3 MAT Files as HDF5 Files** on page II-165 for a workaround.

The MLLoadWave operation was incorporated into Igor for Igor Pro 7.00. In earlier versions it was implemented as an XOP. The XOP was originally created by Yves Peysson and Bernard Saoutic.

Finding Matlab Dynamic Libraries

MLLoadWave dynamically links with libraries supplied by The Mathworks when you install Matlab. You will need to tell Igor where to look as follows:

1. Choose Data→Load Waves→Load Matlab MAT File.

   This displays the Load Matlab MAT File dialog.
2. Click the Find 32-bit Matlab Libraries button or the Find 64-bit Matlab Libraries button.

   The button title depends on whether you are running IGOR32 (Windows only) or IGOR64. Clicking it displays the Find Matlab dialog.
3. Click the Folder button to display a Choose Folder dialog.
4. Navigate to your Matlab folder and select it.

   This will be something like:

   ```
   C:\Program Files\MATLAB\<version>                  // 64-bit Windows
   C:\Program Files (x86)\MATLAB\<version>            // 32-bit Windows
   /Applications/MATLAB_<version>.app/bin/maci64      // 64-bit Macintosh
   ```

   where <version> is your Matlab version, for example, R2015a.
5. Click the Choose button.

   Igor searches your Matlab folder to find the required dynamic libraries. If found, Igor attempts to load them. If the search and loading succeeds, the Accept button is enabled. If the search and loading fails, the Accept button is disabled. The search will fail if Igor can not find the required Matlab dynamic libraries or if the system can not find other dynamic libraries required by the Matlab dynamic libraries.

   If you have selected a valid Matlab folder but the Accept button remains disabled, see **Matlab Dynamic Library Issues**.

6. Click the Accept button.

   Igor records the location of the Matlab dynamic libraries in preferences for use in future sessions.

If you call MLLoadWave before you specify the Matlab dynamic library locations, MLLoadWave displays the Find Matlab dialog. Follow the steps above to locate your Matlab installation.

Matlab Dynamic Library Issues

NOTE: MLLoadWave requires compatible Matlab dynamic libraries built for the same architecture as your version of Igor Pro. IGOR32 (32-bit Igor Pro, Windows only) requires the 32-bit Matlab libraries, while IGOR64 (64-bit Igor Pro) requires the 64-bit Matlab libraries.

If you don't have Matlab or if your Matlab version is not compatible with Igor, or if you simply prefer to work with HDF5 files, see **Loading Version 7.3 MAT Files as HDF5 Files** on page II-165 for a workaround.

Matlab Dynamic Library Issues on Macintosh

On Macintosh, MLLoadWave has been verified to work with Matlab version 2010b and 2015b. It should work with later versions. It may or may not work with earlier versions.

For Matlab 2010b, you need to create an alias to the libraries as described next. Matlab 2015b does not require the alias. We do not know whether the alias is necessary for versions between 2010b and 2015b.

On Macintosh it sometimes happens that you point Igor to valid Matlab dynamic libraries but Igor still can't link with them. This occurs when the dynamic libraries to which Igor directly links cannot find other dynamic libraries which they require. To address this problem, create an alias pointing to the Matlab libraries directory as follows:

1. In the Finder, open the Applications folder and locate the 64-bit Matlab application.
2. Right click on the Matlab application and open it by selecting "Show Package Contents".
3. Inside the Matlab package, navigate to the folder containing your Matlab dynamic libraries. This will be one of the following:

   ```
   /Applications/MATLAB_<version>.app/bin/maci64    // 64-bit Macintosh
   ```

   where <version> is your Matlab version, for example, R2010b.
4. Right click the maci64 folder and select Make Alias.
5. Rename the alias as MLLoadWave64Support.
6. Move the alias to your Applications folder.
7. Restart Igor and try Finding Matlab Dynamic Libraries again.

Matlab Dynamic Library Issues on Windows

Prior to Igor Pro 7.02, it was required that the path to the Matlab dynamic libraries directory be in the Windows PATH environment variable. As of 7.02, this should no longer be necessary.

However, we can not test with all versions of Matlab and future versions may behave differently. If you follow the steps listed under **Finding Matlab Dynamic Libraries** on page II-163 but Igor is still unable to link with the Matlab dynamic libraries, try adding the Matlab libraries path to your Windows PATH environment variable. Remember that IGOR32 requires 32-bit Matlab libraries and IGOR64 requires 64-bit Matlab libraries. Restart Igor before re-testing.

We have received reports that Matlab 2021 is not compatible with Igor Pro 9. This appears to be caused by a dynamic library conflict. See **Loading Version 7.3 MAT Files as HDF5 Files** on page II-165 for a workaround.

Supported Matlab Data Types

MLLoadWave can load 1D, 2D, 3D and 4D numeric and string data. MLLoadWave can not load data of dimension greater than 4.

When loading Matlab string data into an Igor wave, the Igor wave will be of dimension one less than the Matlab data set. This is because each element in a Matlab string data set is a single byte whereas each element in an Igor string wave is a string (any number of bytes).

MLLoadWave does not support loading of the following types of Matlab data: cell arrays, structures, sparse data sets, objects, 64 bit integers.

Numeric Data Loading Modes

The Load Matlab MAT File dialog presents a popup menu that controls how numeric data is loaded into Igor. The items in the menu are:

Load columns into 1D wave	Each column of the Matlab matrix is loaded into a separate 1D Igor wave.
Load rows into 1D wave	Each row of the Matlab matrix is loaded into a separate 1D Igor wave.
Load matrix into one 1D wave	The entire Matlab matrix is loaded into a single 1D Igor wave.
Load matrix into matrix	The Matlab matrix is loaded into an Igor multi-dimensional wave*.
Load matrix into transposed matrix	The Matlab matrix is loaded into an Igor multi-dimensional wave* but the rows and columns are transposed.

* Starting with Igor Pro 8.00, after loading a matrix that results in an Mx1 2D wave, MLLoadWave automatically redimensions the wave as an M-row 1D wave.

When loading data of dimension 3 or 4, the first three modes treat each layer (“page” in Matlab terminology) as a separate matrix. For 3D Matlab data, this gives the following behavior:

Load columns into 1D wave	Each column of each layer of the Matlab data set is loaded into a separate 1D Igor wave.
Load rows into 1D wave	Each row of each layer of the Matlab data set is loaded into a separate 1D Igor wave.
Load matrix into one 1D wave	The layer of the Matlab data set is loaded into a 1D Igor wave.
Load matrix into matrix	The Matlab 3D data set is loaded into an Igor 3D wave.
Load matrix into transposed matrix	The Matlab 3D data set is loaded into an Igor 3D wave but the rows and columns are transposed.

When loading 3D or 4D data sets, the term "matrix" in the last two modes is not really appropriate. MLLoadWave loads the entire 3D or 4D data set into a 3D or 4D Igor wave.

Loading Version 7.3 MAT Files as HDF5 Files

In 2006 Matlab added version 7.3 of their MAT file format. A version 7.3 MAT file is an HDF5 file with 512 bytes of Matlab-specific information at the start of the file. The HDF5 library allows applications to prepend application-specific data, so version 7.3 MAT files can be loaded as HDF5 files.

You may find it useful to load such files as HDF5 files because Igor has better HDF5 support than MAT-file support, because you don't have Matlab on your machine, or because Igor's Matlab support does not work with your Matlab installation. See **Igor HDF5 Guide** on page II-183 for information on Igor's HDF5 support.

A version 7.3 MAT file contains an HDF5 signature at byte offset 512. The HDF5 signature is an 8-byte pattern described at https://support.hdfgroup.org/HDF5/doc/H5.format.html.

In order to open a 7.3 MAT file in the HDF5 Browser you will need to select All Files from the popup menu in the HDF5 Browser Open File dialog. If you try to open a MAT file that is not version 7.3 as an HDF5 file, the HDF5 library will return an error.

For an example of saving Matlab data in the version 7.3 format, see https://www.mathworks.com/help/matlab/ref/save.html. You can also set a preference in Matlab to make version 7.3 your default format.

Loading General Binary Files

General binary files are binary files created by other programs. If you understand the binary file format, it is possible to load the data into Igor. However, you must understand the binary file format *precisely*. This is usually possible only for relatively simple formats.

There are two ways to load data from general binary files into Igor:

- Using the **FBinRead** operation
- Using the **GBLoadWave** operation

Using FBinRead is somwhat more difficult and more flexible than using GBLoadWave. It is especially useful for loading data stored as structures into Igor. For details, see **FBinRead**. This section focuses on using the GBLoadWave operation.

GBLoadWave loads data from general binary files into Igor waves. "GB" stands for "general binary".

You can invoke the GBLoadWave operation directly or by choosing Data→Load Waves→Load General Binary File which displays the Load General Binary dialog.

You need to know the format of the binary file precisely in order to successfully use GBLoadWave. Therefore, it is of use mostly to load binary files that you have created from your own program. You can also use GBLoadWave to load third party files if you know the file format precisely.

Files GBLoadWave Can Handle

GBLoadWave handles the following types of binary data:

- 8 bit, 16 bit, 32 bit, and 64 bit signed and unsigned integers
- 32 and 64 bit IEEE floating point numbers
- 32 and 64 bit VAX floating point numbers

In addition, GBLoadWave handles high-byte-first (Motorola) and low-byte-first (Intel) type binary numbers.

GBLoadWave currently can not handle IEEE or VAX extended precision values. See **VAX Floating Point** for more information.

GBLoadWave can create waves using any of numeric data types that Igor supports (64-bit and 32-bit IEEE floating point, 64-bit, 32-bit, 16-bit and 8-bit signed and unsigned integers). The data type of the wave does not need to be the same as the data type of the file. For example, if you have a file containing integer A/D readings, you can load that data into a single-precision or double-precision floating point wave.

In general, it is best to load waves as floating point since nearly all Igor operations work faster on floating point. One exception is when you are dealing with images, especially stacks of images. For example, if you have a 512x512x1024 byte image stack in a file, you should load it into a byte wave. This takes one quarter of the memory and disk space of a single-precision floating point wave.

GBLoadWave knows nothing about Igor multi-dimensional waves. It knows about 1D only. The term "array", used in the GBLoadWave dialog, means "1D array". However, after loading data as a 1D wave, you can redimension it as required.

GBLoadWave can load one or more 1D arrays from a file. When multiple arrays are loaded, they can be stored sequentially in the file or they can be interleaved. Sequential means that all of the points of one array appear in the file followed by all of the points of the next array. Interleaved means that point zero of each array appears in the file followed by point one of each array.

GBLoadWave And Very Big Files

Most data files are not so large as to present major issues for GBLoadWave or Igor. However, if your data file approaches hundreds of millions or billions of bytes, size and memory issues may arise.

If you want GBLoadWave to convert the type of the data, for example from 16-bit signed to 32-bit floating point, this requires an extra buffer during the load process which takes more memory.

When dealing with extremely large files, you may need to load part of your data file into Igor at a time using the GBLoadWave /S and /U flags.

The Load General Binary Dialog

When you choose Data→Load Waves→Load General Binary File, Igor displays the Load General Binary dialog. This dialog allows you to choose the file to load and to specify the data type of the file and the data type of the wave or waves to be created.

A few of the items in the dialog require some explanation.

The Number of Arrays in File textbox and the Number of Points in Array textbox are both initially set to 'auto'. Auto means that GBLoadWave automatically determines these based on the number of bytes in the file.

If you leave both on auto, GBLoadWave assumes that there is one array in the file with the number of points determined by the number of bytes in the file and the data length of each point.

If you set Number of Arrays in File to a number greater than zero and leave Number of Points in Array on auto, GBLoadWave determines the number of points in each array based on the total number of bytes in the file and the specified number of arrays in the file.

If you set Number of Points in Array to a number greater than one and leave Number of Arrays in File on auto, GBLoadWave determines the number of arrays in the file based on the total number of bytes in the file and the specified number of points in each array.

You can also specify the number of arrays in the file and the number of points in each array explicitly by entering a number in place of 'auto' for each of these settings.

GBLoadWave creates one or more 1D waves and gives the waves names which it generates by appending a number to the specified base name. For example, if the base name is "wave", it creates waves with names like wave0, wave1, etc.

If the Overwrite Existing Waves checkbox is checked, GBLoadWave uses names of existing waves, overwriting them. If it is unchecked, GBLoadWave skips names already in use.

Checking the Apply Scaling checkbox allows you to specify an offset and multiplier so that GBLoadWave can scale the data into meaningful units. If this checkbox is unchecked, GBLoadWave does no scaling.

VAX Floating Point

GBLoadWave can load VAX "F" format (32 bit, single precision) and "G" format (64 bit, double precision) numbers.

Do not use the GBLoadWave byte-swapping feature (/B flag) for VAX data. This does Intel-to-Motorola byte swapping, also called little-endian to big-endian. VAX data is byte-swapped relative to the way Igor stores data, but not in the same sense. Specifically, each 16-bit word is big-endian but each 8-bit byte is little-endian. When you specify that the input data is VAX data, using /J=2, GBLoadWave does the swapping required for VAX data.

GBLoadWave can not currently read VAX "D" (another 64 bit format). However, VAX D format is the same as F with an additional 4 bytes of fraction. This makes it possible to load VAX D format as F format, throwing away the extra fractional bits. Here is an example:

```
GBLoadWave/W=2/V/P=VAXData/T={2,2}/J=2/N=temp "VAX D File"
KillWaves temp1
Rename temp0, VAXDData_WithoutExtraFractBits
```

The /W=2 flag tells GBLoadWave that there are two arrays in the file. The /V flag tells it that they are interleaved. The first four bytes of each data point in the file wind up in the temp0 wave. The seconds four bytes, which contain the extra fractional bits in the D format, wind up in temp1 which we discard.

Loading JCAMP Files

Igor can load JCAMP-DX files using the **JCAMPLoadWave** operation. The JCAMP-DX format is used primarily in infrared spectroscopy. It is a plain text format that uses only ASCII characters.

You can invoke the JCAMPLoadWave operation directly or by choosing Data→Load Waves→Load JCAMP-DX File which displays the Load JCAMP-DX File dialog.

JCAMPLoadWave understands JCAMP-DX file headers well enough to read the data and set the wave scaling appropriately. Because JCAMP-DX is intended primarily for evenly-spaced data, a single wave is produced for each data set. The wave's X scaling is set based on information in the JCAMP-DX file header. The header information is optionally stored in the wave note, and optionally in a series of Igor variables. If you choose to create these variables, there will be one variable for each JCAMP-DX label in the header.

Files JCAMPLoadWave Can Handle

JCAMPLoadWave can load one or more waves from a single file. The JCAMP-DX standard calls for each new data set to start with a new header. Each header should start with the ##TITLE= label. As far as we can tell, most spectrometer systems write only one data set per file.

In addition, the JCAMP-DX standard includes simple optional compression techniques which JCAMPLoadWave supports. Files that do not use compression are human-readable.

We believe that JCAMPLoadWave should load most files stored in standard JCAMP-DX format. If you have a JCAMP-DX file that does not load correctly, please send it to support@wavemetrics.com.

Some systems produce a hybrid format in which the data itself is stored in a binary file, accompanied by an ASCII file that contains just a JCAMP-DX style header. We know that certain Bruker NMR spectrometers do this. To accomodate these systems, it is possible to select an option to load the header information only. You would then have to load the data separately, most likely using **GBLoadWave**.

Loading JCAMP Header Information

JCAMPLoadWave provides two mechanisms to load the header information into Igor:

- Storing all header text in the wave note
- Creating one Igor variable for each JCAMP label encountered in the header

In the Load JCAMP-DX File dialog, checking the Make Wave Note checkbox invokes the /W flag which stores the entire header in the wave note.

Checking Set JCAMP Variables invokes the /V flag which creates one Igor variable for each JCAMP label encountered in the header. This is explained in the next section.

Variables Set By JCAMPLoadWave

JCAMPLoadWave sets the standard Igor file-loader output variables: S_fileName, S_path, V_flag and S_waveNames. These are described in the JCAMPLoadWave reference documentation.

If you use the /V flag, which corresponds to the the Set JCAMP Variables checkbox in the dialog, it also sets "header variables". Header variables are variables that contain data which JCAMPLoadWave gleans from the JCAMP header.

When JCAMPLoadWave is called from a macro, it creates the header variables as local variables. When it is called from the command line or from a user-defined function, it creates the header variables as global variables. The section **Using Header Variables From a Function** on page II-169 explains this in more detail.

The header variable names are set based on the JCAMP label with a prefix of "SJC_" for string variables or "VJC_" for numeric variables. Thus, when it encounters the ##TITLE label, JCAMPLoadWave creates a string variable named SJC_TITLE which contains the label.

Certain JCAMP labels are parsed for numeric information and a numeric variable is created. Numeric variables that might be created include:

VJC_NPOINTS Set to the number of points in the data set. This is set from the header information. If the actual number of data points in the file is different, this variable will not reflect this fact.

VJC_FIRSTX Set to the X value of the first data point in the data set.

VJC_LASTX Set to the X value of the last data point in the data set.

VJC_DELTAX Set to the interval between successive abscissa values. This is calculated from (VJC_LASTX -VJC_FIRSTX)/ (VJC_NPOINTS - 1), and so might be slightly different from the value given by the ##DELTAX=label.

VJC_XFACTOR Set to the multiplier that must be applied to the X data values in the file to give real-world values.

VJC_YFACTOR Set to the multiplier that must be applied to the Y data values in the file to give real-world values.

VJC_MINY Set to the minimum Y value found in the data set.

VJC_MAXY Set to the maximum Y value found in the data set.

If you are loading Fourier domain data, these variables may be created to reflect the fact that the data represent optical retardation and amplitude: VJC_FIRSTR, VJC_LASTR, VJC_DELTR, VJC_RFACTOR, VJC_AFACTOR.

Any other labels found in the header result in a string variable with name SJC_<label> where <label> is replaced with the name of the JCAMP label. For instance, the ##YUNITS label results in a string variable named SJC_YUNITS.

Since successive data sets in a single file have the same standard labels, the contents of the variables are set by the last instance of a given label in the file.

Using Header Variables From a Function

If you execute JCAMPLoadWave from a user-defined function and tell it to create header variables via the /V flag, the variables are created as global variables in the current data folder. To access these variables, you must use **NVAR** and **SVAR** references. These references must appear *after* the call to JCAMPLoadWave. For example:

```
Function LoadJCAMP()
   JCAMPLoadWave/P=JCAMPFiles "JCAMP1.dx"
   if (V_Flag == 0)
      Print "No waves were loaded"
      return -1
   endif

   NVAR VJC_NPOINTS
   Printf "Number of points: %d\r", VJC_NPOINTS
   SVAR SJC_YUNITS
   Printf "Y Units: %s\r", SJC_YUNITS

   return 0
End
```

The code above assumes that the header contains the ##NPOINTS label from which the variables VJC_NPOINTS and SJC_YUNITS are created. If you can't guarantee that the file contains such a label, then you must use **NVAR**/Z and **NVAR_Exists** to test for the existence of the variable before using it.

If you need to determine which variables were created at runtime, use the **GetIndexedObjName** function and test each name for the SJC_ or VJC_ prefix.

Another problem with header variables in functions is that they leave a lot of clutter around. You can clean up like this:

```
KillVariables/Z VJC_NPOINTS
KillStrings/Z SJC_YUNITS
```

Loading Sound Files

The **SoundLoadWave** operation, which was added in Igor Pro 7, loads data from various sound file formats.

See **Sound** on page IV-245 for general information on Igor's sound-related features.

Loading Waves Using Igor Procedures

One of Igor's strong points is that it you can write procedures to automatically load, process and graph data. This is useful if you have accumulated a large number of data files with identical or similar structures or if your work generates such files on a regular basis.

The input to the procedures is one or more data files. The output might be a printout of a graph or page layout or a text file of computed results.

Each person will need procedures customized to his or her situation. In this section, we present some examples that might serve as a starting point.

Variables Set by File Loaders

The LoadWave operation creates the numeric variable V_flag and the string variables S_fileName, S_path, and S_waveNames to provide information that is useful for procedures that automatically load waves. When used in a function, the LoadWave operation creates these as local variables.

Most other file loaders create the same or similar output variables.

LoadWave sets the string variable S_fileName to the name of the file being loaded. This is useful for annotating graphs or page layouts.

LoadWave sets the string variable S_path to the full path to the folder containing the file that was loaded. This is useful if you need to load a second file from the same folder as the first.

LoadWave sets the variable V_flag to the number of waves loaded. This allows a procedure to process the waves without knowing in advance how many waves are in a file.

LoadWave also sets the string variable S_waveNames to a semicolon-separated list of the names of the loaded waves. From a procedure, you can use the names in this list for subsequent processing.

Loading and Graphing Waveform Data

Here is a very simple example designed to show the basic form of an Igor function for automatically loading and graphing the contents of a data file. It loads a delimited text file containing waveform data and then makes a graph of the waves.

This example uses an Igor symbolic path. If you are not familiar with the concept, see **Symbolic Paths** on page II-22.

In this function, we make the assumption that the files that we are loading contain three columns of waveform data. Tailoring the function for a specific type of data file allows us to keep it very simple.

```
Function LoadAndGraph(fileName, pathName)
   String fileName      // Name of file to load or "" to get dialog
   String pathName      // Name of path or "" to get dialog

   // Load the waves and set the local variables.
   LoadWave/J/D/O/P=$pathName fileName
   if (V_flag==0)       // No waves loaded. Perhaps user canceled.
      return -1
   endif

   // Put the names of the three waves into string variables
   String s0, s1, s2
   s0 = StringFromList(0, S_waveNames)
   s1 = StringFromList(1, S_waveNames)
   s2 = StringFromList(2, S_waveNames)

   Wave w0 = $s0                 // Create wave references.
   Wave w1 = $s1
   Wave w2 = $s2

   // Set waves' X scaling, X units and data units
   SetScale/P x, 0, 1, "s", w0, w1, w2
   SetScale d 0, 0, "V", w0, w1, w2

   Display w0, w1, w2            // Create a new graph

   // Annotate graph
   Textbox/N=TBFileName/A=LT "Waves loaded from " + S_fileName

   return 0                      // Signifies success.
End
```

s0, s1 and s2 are local string variables into which we place the names of the loaded waves. We then use the $ operator to create a reference to each wave, which we can use in subsequent commands.

Once the function is entered in the procedure window, you can execute it from the command line or call it from another function. If you execute

```
   LoadAndGraph("", "")
```

the LoadWave operation displays an Open File dialog allowing you to choose a file. If you call LoadAndGraph with the appropriate parameters, LoadWave loads the file without presenting a dialog.

You can add a “Load And Graph” menu item by putting the following menu declaration in the procedure window:

```
Menu "Macros"
   "Load And Graph...", LoadAndGraph("", "")
End
```

Because we have not used the “Auto name & go” option for the LoadWave operation, LoadWave displays another dialog in which you can enter names for the new waves. If you want the procedure to be more automatic, use /A or /N to turn “Auto name & go” on. If you want the procedure to specify the names of the loaded waves, use the /B flag. See the description of the **LoadWave** operation (see page V-508) for details.

To keep the function simple, we have hard-coded the X scaling, X units and data units for the new waves. You would need to change the parameters to the SetScale operation to suit your data. For more flexibility, you would add additional parameters to the function.

It is possible to write LoadAndGraph so that it can handle files with any number of columns. This makes the function more complex but more general.

For more advanced programmers, here is the more general version of LoadAndGraph.

```
Function LoadAndGraph(fileName, pathName)
   String fileName      // Name of file to load or "" to get dialog
   String pathName      // Name of path or "" to get dialog

   // Load the waves and set the variables.
   LoadWave/J/D/O/P=$pathName fileName
   if (V_flag==0)       // No waves loaded. Perhaps user canceled.
      return -1
   endif

   Display              // Create a new graph

   String theWave
   Variable index=0
   do                   // Now append waves to graph
      theWave = StringFromList(index, S_waveNames) // Next wave
      if (strlen(theWave) == 0)             // No more waves?
            break                           // Break out of loop
      endif
      Wave w = $theWave
      SetScale/P x, 0, 1, "s", w            // Set X scaling
      SetScale d 0, 0, "V", w               // Set data units
      AppendToGraph w
      index += 1
   while (1)         // Unconditionally loop back up to “do”

   // Annotate graph
   Textbox/A=LT "Waves loaded from " + S_fileName

   return 0          // Signifies success.
End
```

The do-loop picks each successive name out of the list of names in S_waveNames and adds the corresponding wave to the graph. S_waveNames will contain one name for each column loaded from the file.

There is one serious shortcoming to the LoadAndGraph function. It creates a very plain, default graph. There are four approaches to overcoming this problem:

- Use preferences
- Use a style macro
- Set the graph formatting directly in the procedure
- Overwrite data in an existing graph

Normally, Igor does not use preferences when a procedure is executing. To get preferences to take effect during the LoadAndGraph function, you would need to put the statement “Preferences 1” near the beginning of the function. This turns preferences on just for the duration of the function. This will cause the Display and AppendToGraph operations to use your graph preferences.

Using preferences in a function means that the output of the function will change if you change your preferences. It also means that if you give your function to a colleague, it will produce different results. This dependence on preferences can be seen as a feature or as a problem, depending on what you are trying to achieve. We normally prefer to keep procedures independent of preferences.

Using a style macro is a more robust technique. To do this, you would first create a prototype graph and create a style macro for the graph (see **Graph Style Macros** on page II-350). Then, you would put a call to the style macro at the end of the LoadAndGraph macro. The style macro would apply its styles to the new graph.

To make your code self-contained, you can set the graph formatting directly in the code. You should do this in a subroutine to avoid cluttering the LoadAndGraph function.

The last approach is to overwrite data in an existing graph rather than creating a new one. The simplest way to do this is to always use the same names for your waves. For example, imagine that you load a file with three waves and you name them wave0, wave1, wave2. Now you make a graph of the waves and set everything in the graph to your taste. You now load another file, use the same names and use LoadWave's overwrite option. The data from the new file will replace the data in your existing waves and Igor will automatically update the existing graph. Using this approach, the function simplifies to this:

```
Function LoadAndGraph(fileName, pathName)
   String fileName      // Name of file to load or "" to get dialog
   String pathName      // Name of path or "" to get dialog

   // load the waves, overwriting existing waves
   LoadWave/J/D/O/N/P=$pathName fileName
   if (V_flag==0)       // No waves loaded. Perhaps user canceled.
      return -1
   endif

   Textbox/C/N=TBFileName/A=LT "Waves loaded from " + S_fileName

   return 0             // Signifies success.
End
```

There is one subtle change here. We have used the /N option with the LoadWave operation, which autonames the incoming waves using the names wave0, wave1, and wave2.

You can see that this approach is about as simple as it can get. The downside is that you wind up with uninformative names like wave0. You can use the LoadWave /B flag to provide better names.

If you are loading data from Igor binary wave files or from packed Igor experiments, you can use the LoadData operation instead of LoadWave. This is a powerful operation, especially if you have multiple sets of identically structured data, as would be produced by multiple runs of an experiment. See **The LoadData Operation** on page II-156 above.

Loading and Graphing XY Data

In the preceding example, we treated all of the columns in the file the same: as waveforms. If you have XY data then things change a bit. We need to make some more assumptions about the columns in the file. For example, we might have a collection of files with four columns which represent two XY pairs. The first two columns are the first XY pair and the second two columns are the second XY pair.

Here is a modified version of our function to handle this case.

```
Function LoadAndGraphXY(fileName, pathName)
   String fileName   // Name of file to load or "" to get dialog
   String pathName   // Name of path or "" to get dialog

   // load the waves and set the globals
   LoadWave/J/D/O/P=$pathName fileName
   if (V_flag==0)    // No waves loaded. Perhaps user canceled.
      return -1
   endif
   // Put the names of the waves into string variables.
   String sx0, sy0, sx1, sy1
   sx0 = StringFromList(0, S_waveNames)
   sy0 = StringFromList(1, S_waveNames)
   sx1 = StringFromList(2, S_waveNames)
   sy1 = StringFromList(3, S_waveNames)

   Wave x0 = $sx0                    // Create wave references.
   Wave y0 = $sy0
   Wave x1 = $sx1
   Wave y1 = $sy1

   SetScale d 0, 0, "s", x0, x1      // Set wave data units
   SetScale d 0, 0, "V", y0, y1

   Display y0 vs x0                  // Create a new graph
   AppendToGraph y1 vs x1

   Textbox/A=LT "Waves loaded from " + S_fileName  // Annotate graph

   return 0          // Signifies success.
End
```

The main difference between this and the waveform-based LoadAndGraph function is that here we append waves to the graph as XY pairs. Also, we don’t set the X scaling of the waves because we are treating them as XY pairs, not as waveforms.

It is possible to write a more general function that can handle any number of XY pairs. Once again, adding generality adds complexity. Here is the more general version of the function.

```
Function LoadAndGraphXY(fileName, pathName)
   String fileName      // Name of file to load or "" to get dialog
   String pathName      // Name of path or "" to get dialog

   // Load the waves and set the globals
   LoadWave/J/D/O/P=$pathName fileName
   if (V_flag==0)       // No waves loaded. Perhaps user cancelled.
      return -1
   endif

   Display              // Create a new graph

   String sxw, syw
   Variable index=0
   do       // Now append waves to graph
      sxw=StringFromList(index, S_waveNames) // Next name
      if (strlen(sxw) == 0)                  // No more?
            break       // break out of loop
      endif
      syw=StringFromList(index+1, S_waveNames)// Next name

      Wave xw = $sxw                    // Create wave references.
      Wave yw = $syw

      SetScale d 0, 0, "s", xw          // Set X wave's units
      SetScale d 0, 0, "V", yw          // Set Y wave's units
      AppendToGraph yw vs xw

      index += 2
   while (1)         // Unconditionally loop back up to “do”

   // Annotate graph
   Textbox/A=LT "Waves loaded from " + S_fileName
   return 0          // Signifies success.
End
```

Loading All of the Files in a Folder

In the next example, we assume that we have a folder containing a number of files. Each file contains three columns of waveform data. We want to load each file in the folder, make a graph and print it. This example uses the LoadAndGraph function as a subroutine.

```
Function LoadAndGraphAll(pathName)
   String pathName      // Name of symbolic path or "" to get dialog

   String fileName
   String graphName
   Variable index=0

   if (strlen(pathName)==0)      // If no path specified, create one
      NewPath/O temporaryPath    // This will put up a dialog
      if (V_flag != 0)
         return -1               // User cancelled
      endif
      pathName = "temporaryPath"
   endif

   Variable result
   do       // Loop through each file in folder
      fileName = IndexedFile($pathName, index, ".dat")
      if (strlen(fileName) == 0)    // No more files?
         break                      // Break out of loop
      endif
      result = LoadAndGraph(fileName, pathName)
      if (result == 0)              // Did LoadAndGraph succeed?
                                    // Print the graph
         graphName = WinName(0, 1)  // Get the name of the top graph
         String cmd
         sprintf cmd, "PrintGraphs %s", graphName
         Execute cmd                // Explained below

         KillWindow $graphName      // Kill the graph
         KillWaves/A/Z              // Kill all unused waves
      endif
      index += 1
   while (1)

   if (Exists("temporaryPath"))     // Kill temp path if it exists
      KillPath temporaryPath
   endif
   return 0          // Signifies success.
End
```

This function relies on the IndexedFile function to find the name of successive files of a particular type in a particular folder. The last parameter to IndexedFile says that we are looking for files with a ".dat" extension.

Once we get the file name, we pass it to the LoadAndGraph function. After printing the graph, we kill it and then kill all the waves in the current data folder so that we can start fresh with the next file. A more sophisticated version would kill only those waves in the graph.

To print the graphs, we use the PrintGraphs operation. PrintGraphs is one of a few built-in operations that can not be directly used in a function. Therefore, we put the PrintGraphs command in a string variable and call Execute to execute it.

If you are loading data from Igor binary wave files or from packed Igor experiments, you can use the LoadData operation. See **The LoadData Operation** on page II-156 above.

Setting Wave Names When Loading Data Files

In this section we show how to programmatically set the names of waves loaded from a delimited text file. For background information, see **LoadWave Generation of Wave Names** on page II-142.

We assume that the file contains three columns of numbers with no column labels and we want to create waves named Stimulus, CellA, and CellB. We use the LoadWave /B flag to set the wave names.

```
Function/S GetColumnInfoStr1()
   String columnInfoStr = ""
   columnInfoStr += "N='Stimulus';"
   columnInfoStr += "N='CellA';"
   columnInfoStr += "N='CellB';"
   return columnInfoStr
End

Function LoadAndSetNames1(pathName, fileName)
   String pathName      // Name of symbolic path or "" to get dialog
   String fileName      // Name of file or "" to get dialog

   String columnInfoStr = GetColumnInfoStr1()
   LoadWave/J/O/P=$pathName/B=columnInfoStr fileName
   if (V_Flag == 0)
      return -1         // Failure
   endif

   return 0             // Success
End
```

Next we include the name of the file being loaded, minus the file name extension, in the wave names. Given a file named "Data.txt", this creates waves named Data_Stimulus, Data_CellA, and Data_CellB.

```
Function/S GetColumnInfoStr2(String baseName)
   String columnInfoStr = ""
   columnInfoStr += "N='" + baseName + "_" + "Stimulus';"
   columnInfoStr += "N='" + baseName + "_" + "CellA';"
   columnInfoStr += "N='" + baseName + "_" + "CellB';"
   return columnInfoStr
End

Function LoadAndSetNames2(pathName, fileName)
   String pathName      // Name of symbolic path
   String fileName      // Name of file

   // This version does requires that you provide the actual symbolic path
   // and file name.
   if (strlen(pathName)==0 || strlen(fileName)==0)
      return -1         // Failure
   endif

   String fileNameMinusExtension = ParseFilePath(3, fileName, ":", 0, 0)
   String baseName = CleanupName(fileNameMinusExtension, 0)

   String columnInfoStr = GetColumnInfoStr2(baseName)
   LoadWave/J/A/O/P=$pathName/B=columnInfoStr fileName
   if (V_Flag == 0)
      return -1         // Failure
   endif

   return 0             // Success
End
```

Finally we include the name of the file being loaded, minus the file name extension, in the wave names using the LoadWave /NAME flag which requires Igor Pro 9.00 or later. Given a file named "Data.txt", this creates waves named Data_Stimulus, Data_CellA, and Data_CellB, the same as the preceding example.

```
Function/S GetColumnInfoStr3()
   String columnInfoStr = ""
   columnInfoStr += "N='Stimulus';"
   columnInfoStr += "N='CellA';"
   columnInfoStr += "N='CellB';"
   return columnInfoStr
End

Function LoadAndSetNames3(pathName, fileName)
   String pathName      // Name of symbolic path or "" to get dialog
   String fileName      // Name of file or "" to get dialog

   String columnInfoStr = GetColumnInfoStr3()
   LoadWave/J/A/O/P=$pathName/B=columnInfoStr/NAME={":filename:_","",1} 
fileName
   if (V_Flag == 0)
      return -1      // Failure
   endif

   return 0          // Success
End
```

See **Using the File Name in Wave Names** on page II-142 for details on the /NAME flag.

Exporting Data

Igor automatically saves the waves in the current experiment on disk when you save the experiment. Many Igor users load data from files into Igor and then make and print graphs or layouts. This is the end of the process. They have no need to explicitly save waves.

You can save waves in an Igor packed experiment file for archiving using the SaveData operation or using the Save Copy button in the Data Browser. The data in the packed experiment can then be reloaded into Igor using the LoadData operation or the Load Expt button in Data Browser. Or you can load the file as an experiment using File→Open Experiment. See the **SaveData** operation on page V-815 for details.

The main reason for saving a wave separate from its experiment is to export data from Igor to another program. To explicitly save waves to disk, you would use Igor's Save operation.

You can access all of the built-in routines via the Save Waves submenu of the Data menu.

The following table lists the available data saving routines in Igor and their salient features.

File type	Description
Delimited text	Used for archiving results or for exporting to another program. Row Format: `<data><delimiter><data><terminator>`* Contains one block of data with any number of rows and columns. A row of column labels is optional. Columns may be equal or unequal in length. Can export 1D or 2D waves. See **Saving Waves in a Delimited Text File** on page II-178.

File type	Description
General text	Used for archiving results or for exporting to another program. Row Format: `<number><tab><number><terminator>`* Contains one or more blocks of numbers with any number of rows and columns. A row of column labels is optional. Columns in a block must be equal in length. Can export 1D or 2D waves. See **Saving Waves in a General Text File** on page II-179.
Igor Text	Used for archiving waves or for exporting waves from one Igor experiment to another. Format: See **Igor Text File Format** on page II-151 above. Contains one or more wave blocks with any number of waves and rows. A given block can contain either numeric or text data. Consists of special Igor keywords, numbers and Igor commands. Can export waves of dimension 1 through 4. See **Saving Waves in an Igor Text File** on page II-179.
Igor Binary	Used for exporting waves from one Igor experiment to another. Contains data for one Igor wave. Format: See Igor Technical Note #003, “Igor Binary Format”. See **Sharing Versus Copying Igor Binary Wave Files** on page II-156.
Image	Used for exporting waves to another program. Format: TIFF, PNG, raw PNG, JPEG. See **Saving Waves in Image Files** on page II-180.
HDF4	Igor does not support exporting data in HDF4 format.
HDF5	For help, execute this in Igor: `DisplayHelpTopic "HDF5 in Igor Pro"`
Sound	Used for exporting waves to another program. Format: AIFC, WAVE. See **Saving Sound Files** on page II-180.
TDMS	Saves data to National Instruments TDMS files. Requires activating an extension. Supported on Windows only. See the “TDM Help.ihf” help file for details.
SQL Databases	Writes data to SQL databases. Requires activating an extension and expertise in database programming. See **Accessing SQL Databases** on page II-181.

* `<terminator>` can be carriage return, linefeed or carriage return/linefeed. You would use carriage return for exporting to a Macintosh program, carriage return/linefeed for Windows systems, and linefeed for Unix systems.

Saving Waves in a Delimited Text File

To save a delimited text file, choose Data→Save Waves→Save Delimited Text to display the Save Delimited Text dialog.

The Save Delimited Text routine writes a file consisting of numbers separated by tabs, or another delimiter of your choice, with a selectable line terminator at the end of each line of text. When writing 1D waves, it can optionally include a row of column labels. When writing a matrix, it can optionally write row labels as well as column labels plus row and column position information.

Save Delimited Text can save waves of any dimensionality. Multidimensional waves are saved one wave per block. Data is written in row/column/layer/chunk order. Multidimensional waves saved as delimited text can not be loaded back into Igor as delimited text because the Load Delimited Text routine does not support multiple blocks. They can be loaded back in as general text. However, for data that is intended to be loaded back into Igor later, the Igor Text, Igor Binary or Igor Packed Experiment formats are preferable.

The order of the columns in the file depends on the order in which the wave names appear in the Save command. This dialog generates the wave names based on the order in which you select waves in the Source Waves list.

By default, the Save operation writes numeric data using the "%.15g" format for double-precision data and "%.7g" format for data with less precision. These formats give you up to 15 or 7 digits of precision in the file.

To use different numeric formatting, create a table of the data that you want to export. Set the numeric formatting of the table columns as desired. Be sure to display enough digits in the table because the data will be written to the file as it appears in the table. In the Save Delimited Text dialog, select the "Use table formatting" checkbox. When saving a multi-column wave (1D complex wave or multi-dimensional wave), all columns of the wave are saved using the table format for the first table column from the wave.

The **SaveTableCopy** and **wfprintf** operations can also be used to save waves to text files using a specific numeric format.

The Save operation is capable of appending to an existing file, rather than overwriting the file. This is useful for accumulating results of a analysis that you perform regularly in a single file. You can also use this to append a block of numbers to a file containing header information that you generated with the fPrintf operation. The append option is not available through the dialog. If you want to do this, see the discussion of the **Save** operation (see page V-812).

Saving Waves in a General Text File

Saving waves in a general text file is very similar to saving a delimited text file. The Save General Text dialog is identical to the Save Delimited Text dialog.

All of the columns in a single block of a general text file must have the same length. The Save General Text routine writes as many blocks as necessary to save all of the specified waves. For example, if you ask it to save two 1D waves with 100 points and two 1D waves with 50 points, it will write two blocks of data. Multidimensional waves are written one wave per block.

Saving Waves in an Igor Text File

The Igor Text format is capable of saving not only the data of a wave but its other properties as well. It saves each wave's dimension scaling, units and labels, data full scale and units and the wave's note, if any. All of this data is saved more efficiently as binary data when you save as an Igor packed experiment using the SaveData operation.

As in the general text format, all of the columns in a single block of an Igor Text file must have the same length. The Save Igor Text routine handles this requirement by writing as many blocks as necessary.

Save Igor Text can save waves of any dimensionality. Multidimensional waves are saved one wave per block. The /N flag at the start of the block identifies the dimensionality of the wave. Data is written in row/column/layer/chunk order.

Saving Waves in Igor Binary Wave Files

Igor's Save Igor Binary routine saves waves in Igor binary wave files, one wave per file. Most users will not need to do this since Igor automatically saves waves when you save an Igor experiment. You might want to save a wave in an Igor binary wave file to send it to a colleague.

The Save Igor Binary dialog is similar to the Save Delimited Text dialog. There is a difference in file naming since, in the case of Igor Binary, each wave is saved in a separate file. If you select a single wave from the dialog's list, you can enter a name for the file. However, if you select multiple waves, you can not enter a file name. Igor will use default file names of the form "wave0.ibw".

When you save an experiment in a packed experiment file, all of the waves are saved in Igor Binary format. The waves can then be loaded into another Igor experiment using **The Data Browser** (see page II-114) or **The LoadData Operation** (see page II-156).

.ibw files do not support waves with more than 2 billion elements. you can use the **SaveData** operation or the Data Browser Save Copy button to save very large waves in a packed experiment file (.pxp) instead.

Saving Waves in Image Files

To save a wave in TIFF, PNG, raw PNG, or JPEG format, choose Data→Save Waves→Save Image to display the Save Image dialog.

JPEG uses lossy compression. TIFF, PNG and raw PNG use lossless compression. To avoid compression loss, don't use JPEG.

JPEG supports only 8 bits per sample.

PNG supports 24 and 32 bits per sample. Raw PNG supports 8 and 16 bits per sample.

The extended TIFF file format supports 8, 16, and 32 bits per sample and you can use image stacks to export 3D and 4D waves.

See the **ImageSave** operation on page V-405 for details.

Saving Sound Files

You can save waves as sound files using the **SoundSaveWave** operation.

Exporting Text Waves

Igor does not quote text when exporting text waves as a delimited or general text file. It does quote text when exporting it as an Igor Text file.

Certain special characters, such as tabs, carriage returns and linefeeds, cause problems during exchange of data between programs because most programs consider them to separate one value from the next or one line of text from the next. Igor Text waves can contain any character, including special characters. In most cases, this will not be a problem because you will have no need to store special characters in text waves or, if you do, you will have no need to export them to other programs.

When Igor writes a text file containing text waves, it replaces the following characters, when they occur within a wave, with their associated escape codes:

Character	**Name**	**ASCII Code**	**Escape Sequence**
CR	carriage return	13	\r
LF	linefeed	10	\n
tab	tab	9	\t
\	backslash	92	\\

Igor does this because these would be misinterpreted if not changed to escape sequences. When Igor loads a text file into text waves, it reverses the process, converting escape sequences into the associated ASCII code.

This use of escape codes can be suppressed using the /E flag of the **Save** operation (see page V-812). This is necessary to export text containing backslashes to a program that does not interpret escape codes.

At present, the Save operation always uses the UTF-8 text encoding when writing text files. If your waves contain non-ASCII text, and if you need to import into a program that does not support UTF-8, you will need to convert the file's text encoding after saving it. You can do this by opening the file as a notebook, changing the text encoding, and saving it again, or using an external text editor.

Exporting MultiDimensional Waves

When exporting a multidimensional wave as a delimited or general text file, you have the option of writing row labels, row positions, column labels and column positions to the file. Each of these options is controlled by a checkbox in the Save Waves dialog. There is a discussion of row/column labels and positions under **2D Label and Position Details** on page II-134.

Igor writes multidimensional waves in column/row/layer/chunk order.

Accessing SQL Databases

Igor Pro includes an XOP, called SQL XOP, which provides access to relational databases from IGOR procedures. It uses ODBC (Open Database Connectivity) libraries and drivers on Mac OS X and Windows to provide this access.

For details on configuring and using SQL XOP, open the SQL Help file in "Igor Pro 7 Folder:More Extensions:Utilities".

Chapter

II-10

Igor HDF5 Guide

HDF5 in Igor Pro

HDF5 is a widely-used, very flexible data file format. The HDF5 file format is a creation of the National Center for Supercomputing Applications (NCSA). HDF5 is now supported by The HDF Group (http://www.hdfgroup.org).

Igor Pro includes deep support for HDF5, including:

- Browsing HDF5 files
- Loading data from HDF5 files
- Saving data to HDF5 files
- Saving Igor experiments as HDF5 files
- Loading Igor experiments from HDF5 files

Prior to Igor Pro 9.00, HDF5 support was provided by an XOP that you had to activate. HDF5 support is now built into Igor and activation is no longer required.

Support for saving and loading Igor experiments as HDF5 files was added in Igor Pro 9.00. For details see **HDF5 Packed Experiment Files** on page II-223.

Index of HDF5 Topics

Here are the main sections in the following material on HDF5:

HDF5 Guided Tour on page II-183

The HDF5 Browser on page II-193

HDF5 Operations and Functions on page II-197

HDF5 Procedure Files on page II-198

HDF5 Attributes on page II-198

HDF5 Dataset Subsets on page II-202

Igor Versus HDF5 on page II-203

Advanced HDF5 Data Types on page II-207

Other HDF5 Issues on page II-219

HDF5 Packed Experiment Files on page II-223

HDF5 Guided Tour

This section will get you off on the right foot using HDF5 in Igor Pro.

In the following material you will see numbered steps that you should perform. Please perform them exactly as written so you stay in sync with the guided tour.

This tour in intended to help experienced Igor users learn how to access HDF5 files from Igor but also to entice non-Igor users to buy Igor. Therefore the tour is written assuming no knowledge of Igor.

HDF5 Overview

HDF5 is a very powerful but complex file format that is designed to be capable of storing almost any imaginable set of data and to encapsulate relationships between data sets.

An HDF5 file can contain within it a hierarchy similar to the hierarchy of directories and files on your hard disk. In HDF5, the hierarchy consists of "groups" and "datasets". There is a root group named "/". Each group can contain datasets and other groups.

An HDF5 dataset is a one-dimensional or multi-dimensional set of elements. Each element can be an "atomic" datatype (e.g., 16-bit signed integer or a 32-bit IEEE float) or a "composite" datatype such as a structure or an array. A "compound" datatype is a composite datatype similar to a C structure. Its members can be atomic datatypes or composite datatypes. For now, forget about composite datatypes - we will deal with atomic datatypes only.

Each dataset can have associated with it any number of "attributes". Attributes are like datasets but are attached to datasets, or to groups, rather than being part of the hierarchy.

Igor Pro HDF5 Support

The Igor Pro HDF5 package consists of built-in HDF5 support and a set of Igor procedures.

The Igor procedures, which are automatically loaded when Igor is launched, implement an HDF5 browser in Igor. The browser supports:

- Previewing HDF5 file data
- Loading HDF5 datasets and groups into Igor
- Saving Igor waves and data folders in HDF5 files

In Igor Pro 9 and later, Igor can save Igor experiments as HDF5 packed experiment files and reload experiments from them.

Using the HDF5 Browser

1. **Choose Data→Load Waves→New HDF5 Browser.**

   This displays an HDF5 browser control panel.

   As you can see, the HDF5 Browser takes up a bit of screen space. You will need to arrange it and this help window so you can see both.

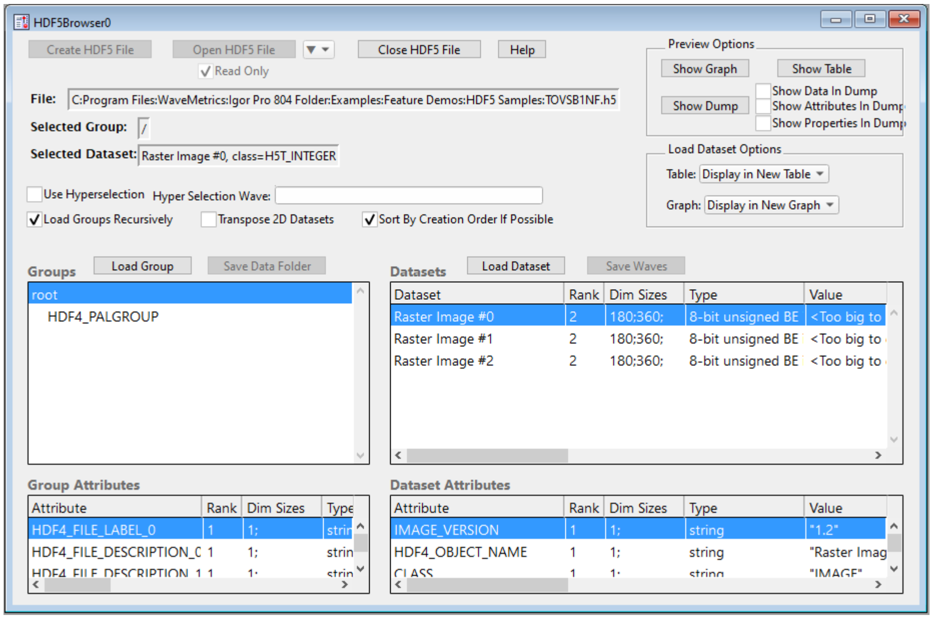
2. **Click the Open HDF5 File button and open the following file:**

   ```
   Igor Pro Folder\Examples\Feature Demos\HDF5 Samples\TOVSB1NF.h5
   ```

   (If you're not sure where your Igor Pro Folder is, choose Misc→Path Status, click on the Igor symbolic path, and note the path to the Igor Pro Folder.)

   We got this sample from the NCSA web site.

You should now see something like this:



☑ Read Only

File: C:Program Files:WaveMetrics:Igor Pro 804 Folder:Examples:Feature Demos:HDF5 Samples:TOVSB1NF.h5

Selected Group: /

Selected Dataset: Raster Image #0, class=H5T_INTEGER

□ Use Hyperselection   Hyper Selection Wave:

☑ Load Groups Recursively   □ Transpose 2D Datasets   ☑ Sort By Creation Order If Possible

□ Show Data In Dump
□ Show Attributes In Dump
□ Show Properties In Dump

Table: Display in New Table

Graph: Display in New Graph

Groups:

root
HDF4_PALGROUP

Dataset	Rank	Dim Sizes	Type	Value
Raster Image #0	2	180;360;	8-bit unsigned BE	<Too big to
Raster Image #1	2	180;360;	8-bit unsigned BE	<Too big to
Raster Image #2	2	180;360;	8-bit unsigned BE	<Too big to

Attribute	Rank	Dim Sizes	Type
HDF4_FILE_LABEL_0	1	1;	strin
HDF4_FILE_DESCRIPTION_C	1	1;	strin
HDF4_FILE_DESCRIPTION_1	1	1;	strin

Attribute	Rank	Dim Sizes	Type	Value
IMAGE_VERSION	1	1;	string	"1.2"
HDF4_OBJECT_NAME	1	1;	string	"Raster Imag
CLASS	1	1;	string	"IMAGE"

The browser contains four lists.

The top/left list is the Groups list and shows the groups in the HDF5 file. Groups in an HDF5 file are analogous to directories in a hard disk hierarchy. In this case there are two groups, root (which is called "/" in HDF5 terminology) and HDF4_PALGROUP. HDF4_PALGROUP is a subgroup of root.

This file contains a number of objects with names that begin with HDF4 because it was created by converting an HDF4 file to HDF5 format using a utility supplied by The HDF Group.

Below the Groups list is the Group Attributes list. In the picture above, the root group is selected so the Group Attributes list shows the attributes of the root group. An attribute is like a dataset but is attached to a group or dataset instead of being part of the HDF5 file hierarchy. Attributes are usually used to save small snippets of information related to a group or dataset.

The top/right list is the Datasets list. This lists the datasets in the selected group, root in this case. In the root group of this file we have three datasets all of which are images.

Below the Datasets list is the Dataset Attributes list. It shows the attributes of the selected dataset, Raster Image #0 in this case.

Three of the lists have columns that show information about the items in the list.

3. **Familiarize yourself with the information listed in the columns of the lists.**

   To see all the information you will need to either scroll the list and/or resize the entire HDF5 browser window to make it larger.

4. **In the Groups list, click the subgroup and notice that the information displayed in the other lists changes.**

5. **In the Groups list, click the root group again.**

   Now we will see how to browse a dataset.

6. **Click the Show Graph, Show Table and Show Dump buttons and arrange the three resulting windows so that they can all be seen at least partially.**

   These browser preview windows should typically be kept fairly small as they are intended just to provide a preview. It is usually convenient to position them to the right of the HDF5 browser.

   The three windows are blank now. They display something only when you click on a dataset or attribute.

7. **In the Datasets list, click the top dataset (Raster Image #0).**

The dataset is displayed in each of the three preview windows.

The dump window shows the contents of the HDF5 file in "Data Description Language" (DDL). This is useful for experts who want to see the format details of a particular group, dataset or attribute. The dump window will be of no interest in most everyday use.

If you check the Show Data in Dump checkbox and then click a very large dataset, it will take a very long time to dump the data into the dump window. Therefore you should avoid checking the Show Data in Dump checkbox.

The preview graph and table, not surprisingly, allow you to preview the dataset in graphical and tabular form.

This dataset is a special case. It is an image formatted according to the HDF5 Image and Palette Specification which requires that the image have certain attributes that describe it. You can see these attributes in the Dataset Attributes list. They are named CLASS, IMAGE_VERSION, IMAGE_SUBCLASS and PALETTE. The HDF5 Browser uses the information in these attributes to make a nice preview graph.

An HDF5 file can contain a 2D dataset without the dataset being formatted according to the HDF5 Image and Palette Specification. In fact, most HDF5 files do not follow that specification. We use the term "formal image" to make clear that a particular dataset is formatted according to the HDF5 Image and Palette Specification and to distinguish it from other 2D datasets which may be considered to be images.

8. **In the Dataset Attributes list, click the CLASS attribute.**

   The value of the selected attribute is displayed in the preview windows.

   Try clicking the other image attributes, IMAGE_VERSION, IMAGE_SUBCLASS and PALETTE.

   So far we have just previewed data, we have not loaded it into Igor. (Actually, it was loaded into Igor and stored in the root:Packages:HDF5Browser data folder, but that is an HDF5 Browser implementation detail.)

   Now we will load the data into Igor for real.

9. **Make sure that the Raster Image #0 dataset is selected, that the Table popup menu is set to Display In New Table and that the Graph popup menu is set to Display In New Graph. Then click the Load Dataset button.**

   The HDF5 Browser loads the dataset (and its associated palette, because this is a formal image with an associated palette dataset) into the current data folder in Igor and creates a new graph and a new table.

10. **Choose Data→Data Browser and note the two "waves" in the root data folder.**

    "Wave" is short for "waveform" and is our term for a dataset. This terminology stems from our roots in time series signal processing.

    The two waves, 'Raster Image #0' and 'Raster Image #0Pal' were loaded when you clicked the Load Dataset button. The graph was set up to display 'Raster Image #0' using 'Raster Image #0Pal' as a palette wave.

11. **Back in the HDF5 Browser, with the root group still selected, click the Load Group button.**

    The HDF5 Browser created a new data folder named TOVSB1NF and loaded the contents of the HDF5 root group into the new data folder which can be seen in the Data Browser. The name TOVSB1NF comes from the name of the HDF5 file whose root group we just loaded.

    When you load a group, the HDF5 Browser does not display the loaded data in a graph or table. That is done only when you click Load Dataset and also depends on the Load Dataset Options controls.

12. **Click the Close HDF5 File button and then click the close box on the HDF5 Browser.**

    If you had closed the HDF5 browser without clicking the Close HDF5 File button, the browser would have closed the file anyway. It will also automatically close the file if you choose File→New Experiment or File→Open Experiment or if you quit Igor.

Next we will learn how to write an Igor procedure to access an HDF5 file programmatically. Before you start that, feel free to play with the HDF5 Browser on your own HDF5 files. Start by choosing Data→Load Waves→New HDF5 Browser.

Igor can handle most HDF5 files that you will encounter, but it can not handle all possible HDF5 files. If you receive an error while examining your own files, it may be because of a bug or because Igor does not support a feature used in your file. In this case you can send the file along with a brief explanation of the problem to support@wavemetrics.com and we will determine whether the problem is a bug or just a limitation.

Loading HDF5 Data Programmatically

Igor includes a number of operations and functions for accessing HDF5 files. To get an idea of the range and scope of the operations, click the link below. Then return here to continue the guided tour.

HDF5 Operations and Functions on page II-197

In this section, we will load an HDF5 dataset using a user-defined function.

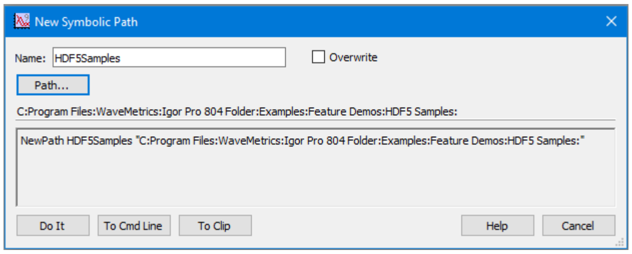
1. **Choose File→New Experiment.**

   You will be asked if you want to save the current experiment. Click No (or Yes if you want to save your current work environment).

   This closes the current experiment, killing any waves and windows.

2. **Choose Misc→New Path and create an Igor symbolic path named HDF5Samples and pointing to the HDF5 Samples directory which contains the TOVSB1NF.h5 file that we used in the preceding section.**

   The New Path dialog will look something like this:



Name: HDF5Samples

C:Program Files:WaveMetrics:Igor Pro 804 Folder:Examples:Feature Demos:HDF5 Samples:

NewPath HDF5Samples "C:Program Files:WaveMetrics:Igor Pro 804 Folder:Examples:Feature Demos:HDF5 Samples:"

   Click Do It to create the symbolic path.

   An Igor symbolic path is a short name that references a specific directory on disk. We will use this symbolic path in a command that opens an HDF5 file.

   In HDF5, you must first open a file or create a new file. The HDF5 library returns a file ID that you use in subsequent calls to the library. When you are finished, you must close the file. The following example illustrates this three step process.

3. **Choose Windows→Procedure Window and paste the following into the Procedure window:**

```
Function TestLoadDataset(datasetName)
    String datasetName              // Name of dataset to be loaded

    Variable fileID                 // HDF5 file ID will be stored here

    Variable result = 0             // 0 means no error

    // Open the HDF5 file
    HDF5OpenFile /P=HDF5Samples /R /Z fileID as "TOVSB1NF.h5"
    if (V_flag != 0)
        Print "HDF5OpenFile failed"
        return -1
    endif

    // Load the HDF5 dataset
```

```
	HDF5LoadData /O /Z fileID, datasetName
	if (V_flag != 0)
		Print "HDF5LoadData failed"
		result = -1
	endif

	// Close the HDF5 file
	HDF5CloseFile fileID

	return result
End
```

Read through the procedure. Notice that we use the symbolic path HDF5Samples created above as a parameter to the HDF5OpenFile operation.

The /R flag was used to open the file for read-only so that we don't get an error if the file can not be opened for writing.

The /Z flag tells HDF5OpenFile that, if there is an error, it should set the variable V_flag to a non-zero value but return to Igor as if no error occurred. This allows us to handle the error gracefully rather than having Igor abort execution and display an error dialog.

HDF5OpenFile sets the local variable fileID to a value returned from the HDF5 library. We pass that value to the HDF5LoadData operation. We also provide the name of a dataset within that file to be loaded. Again we check the V_flag variable to see if an error occurred.

Finally we call HDF5CloseFile to close the file that we opened. It is important to always close files that we open. If, during development of a procedure, you forget to close a file or fail to close a file because of an error, you can close all open HDF5 files by executing:

```
HDF5CloseFile /A 0
```

Also, Igor automatically closes all open files if you choose File→New Experiment, File→Revert Experiment or File→Open Experiment or if you quit Igor.

4. **Click the Compile button at the bottom of the Procedure window.**

   If you don't see a Compile button at the bottom of the Procedure window, that is because the procedures were already automatically compiled when you deactivated the Procedure window.

   If you get a compile error then you have not pasted the right text into the Procedure window or you have entered other incorrect text. Try going back to step 1 of this section.

   Now we are ready to run our procedure.

5. **Execute the following command in the Igor command line either by typing it and pressing Enter, by copy/paste followed by Enter, or by selecting it and pressing Control-Enter (Macintosh) or Ctrl-Enter (Windows):**

   ```
   TestLoadDataset("Raster Image #0")
   ```

   At this point the procedure should have correctly executed and you should see no error messages printed in the history area (just above the command line). This history area of the command window should contain:

   ```
   Waves loaded by HDF5LoadData: Raster Image #0
   ```

   Now we will verify that the data was loaded.

6. **Choose Data→Data Browser. Then double-click the wave named 'Raster Image #0'.**

   This creates a table showing the data just loaded.

   At this point Igor's root data folder contains just one wave, 'Raster Image #0', and does not contain the corresponding palette wave. That's because we loaded the dataset as a plain dataset, not as a formal image. The HDF5LoadData operation does not know about formal images. For that we need to use HDF5LoadImage.

7. **In the Data Browser, Control-click (Macintosh) or right-click (Windows) the 'Raster Image #0' wave and choose New Image.**

   Igor creates an image plot of the wave.

Because we loaded 'Raster Image #0' as a plain dataset, not as a formal image, the image plot is gray instead of colored. Also, it is rotated 90 degrees relative to the image plot we saw earlier in the tour. That's because Igor plots 2D data different from most programs and the HDF5LoadData operation does not compensate by default.

Our procedure is of rather limited value because it is hard-coded to use a specific symbolic path and a specific file name. We will now make it more general.

Before we do that, we will save the current work environment so that, if you make a mistake, you can revert to a version of it that worked.

8. **Choose File→Save Experiment As and save the current work environment in a new Igor experiment file named "HDF5 Tour.pxp" in a directory where you store your own files.**

   This saves all of your data and procedures in a single file. If need be, later you can revert to the saved state by choosing File→Revert Experiment.

9. **Open the Procedure window (Windows→Procedure Window) and replace the first two lines of the TestLoadDataset function with this:**

```
Function TestLoadDataset(pathName, fileName, datasetName)
    String pathName         // Name of symbolic path
    String fileName         // Name of HDF5 file
    String datasetName      // Name of dataset to be loaded
```

10. **Change the HDF5OpenFile command to this:**

```
HDF5OpenFile /P=$pathName /R /Z fileID as fileName
```

   Here we replaced HDF5Samples with $pathName. $pathName tells Igor to use the contents of the string parameter pathName as the parameter for the /P flag. We also replaced "TOVSB1NF.h5" with fileName so that we can specify the file to be loaded when we call the function instead of when we code it.

11. **Click the Compile button and then execute this in the command line:**

```
TestLoadDataset("HDF5Samples", "TOVSB1NF.h5", "Raster Image #0")
```

   This does the same thing as the earlier version of the function but now we have a more general function that can be used on any file in any directory.

   The command above reloaded the 'Raster Image #0' dataset into Igor, overwriting the previous contents. Since the new contents is identical to the previous contents, the graph and table did not change.

12. **Choose File→Save Experiment to save your current work environment in the HDF5 Tour.pxp experiment file that you created earlier.**

   You can now take a break and quit Igor if you want.

Saving HDF5 Data Programmatically

Now that we have seen how to programmatically load HDF5 data we will turn our attention to saving Igor data in an HDF5 file.

In this section, we will create some Igor data and save it in an HDF5 dataset from a user-defined function.

1. **If the HDF5 Tour.pxp experiment that you previously saved is not already open, open it by choosing File→Recent Experiments→HDF5 Tour.pxp.**

   Next we will create some data that we can save in an HDF5 file. We will create the data in a new Igor data folder to keep it separate from our other data.

2. **Choose Data→Data Browser. Click the New Data Folder button. Enter the name Test Data, click the Set As Current Data Folder and click OK.**

   The Data Browser shows the new data folder and a red arrow pointing to it. The red arrow indicates the current data folder. Operations that do not explicitly address a specific data folder work in the current data folder.

3. **On Igor's command line, execute these commands:**

```
Make/N=100 data0, data1, data2
```

```
SetScale x 0, 2*PI, data0, data1, data2
data0=sin(x); data1=2*sin(x+PI/6); data2=3*sin(x+PI/4)
Display data0, data1, data2
```

The SetScale operation set the X scaling for each wave to run from 0 to 2π. The symbol x in the wave assignment statements takes on the X value for each point in the destination wave as the assignment is executed.

(If you have not already done it, later you should do the Igor guided tour by choosing Help→Getting Started. It explains X scaling and wave assignment statements as well as many other Igor features.)

4. **Choose Misc→New Path and create an Igor symbolic path named HDF5Data pointing to a directory on your hard disk where you will save data.**

   Choose Misc→Path Status and verify that the HDF5Data symbolic path exists and points to the intended directory.

   Now that we have done some more work worth saving we will save the current experiment so we can revert to a known good state if necessary.

5. **Choose File→Save Experiment to save your current work environment in the HDF5 Tour.pxp experiment file that you created earlier.**

   If need be, later you can revert to the saved state by choosing File→Revert Experiment.

6. **Choose Windows→Procedure Window and paste the following into the Procedure window below the TestLoadDataset function:**

```
Function TestSaveDataset(pathName, fileName, w)
    String pathName         // Name of symbolic path
    String fileName         // Name of HDF5 file
    Wave w                  // The wave to be saved

    Variable result = 0     // 0 means no error

    Variable fileID

    // Create a new HDF5 file, overwriting if same-named file exists
    HDF5CreateFile/P=$pathName /O /Z fileID as fileName
    if (V_flag != 0)
        Print "HDF5CreateFile failed"
        return -1
    endif

    // Save wave as dataset
    HDF5SaveData /O /Z w, fileID
    if (V_flag != 0)
        Print "HDF5SaveData failed"
        result = -1
    endif

    // Close the HDF5 file
    HDF5CloseFile fileID

    return result
End
```

Read through the procedure. It should look familiar as it is similar to the TestLoadDataset function we wrote before. Again it has parameters that specify a symbolic path and file name. It does not have a parameter to specify the dataset name because HDF5SaveData uses the wave name as the dataset name unless instructed otherwise. This function has a wave parameter through which we will specify the wave whose data is to be written to the HDF5 file.

Here we use HDF5CreateFile to create a new file rather than HDF5OpenFile. HDF5CreateFile creates a new file and opens it, returning a file ID. If you wanted to add a dataset to an existing file you would use HDF5OpenFile instead of HDF5CreateFile.

HDF5CreateFile sets the local variable fileID to a value returned from the HDF5 library. We pass that value to the HDF5SaveData operation. The /O flag means that, if there is already a dataset with the same name, it will be overwritten.

Finally we call HDF5CloseFile to close the file that we opened via HDF5CreateFile.

7. **Click the Compile button at the bottom of the Procedure window.**

   If you get an error then you have not pasted the right text into the Procedure window or you have entered other incorrect text. If you can not find the error, choose File→Revert Experiment and go back to step 6 of this section.

   Now we are ready to run our procedure.

8. **Execute the following command in the Igor command line either by typing it and pressing Enter, by copy/paste followed by Enter, or by selecting it and pressing Control-Enter (Macintosh) or Ctrl-Enter (Windows):**

```
TestSaveDataset("HDF5Data", "SaveTest.h5", data0)
```

   At this point the procedure should have correctly executed and you should see no error messages printed in the history area of the command window (just above the command line).

   Now we will verify that the data was saved.

9. **Choose Data→Load Waves→New HDF5 Browser. Click the Open HDF5 File button and open the SaveTest.h5 file that we just created.**

   Verify that the file contains a dataset named data0.

   The data0 dataset has some attributes. These attributes allow HDF5LoadData to fully recreate the wave and all of its properties if you ever load it back into Igor. If you don't want to save these attributes you can use the /IGOR=0 flag when calling HDF5SaveData.

10. **Click the Close HDF5 File button.**

    We can't write more data to the file while it is open in the HDF5 Browser.

    The TestSaveDataset function is rather limited because it saves just one wave. We will now make it more general so that it can save any number of waves. But first we will save our work since it is in a known good state.

11. **Choose File→Save Experiment to save your current work environment in the HDF5 Tour.pxp experiment file that you created earlier.**

    If need be, later you can revert to the saved state by choosing File→Revert Experiment.

    Next we will create a new, more general user function with a different name (TestSaveDatasets instead of TestSaveDataset).

12. **Choose Windows→Procedure Window and paste the following into the Procedure window below the TestSaveDataset function:**

```
Function TestSaveDatasets(pathName, fileName, listOfWaves)
    String pathName            // Name of symbolic path
    String fileName            // Name of HDF5 file
    String listOfWaves         // Semicolon-separated list of waves

    Variable result = 0        // 0 means no error

    Variable fileID

    // Create a new HDF5 file, overwriting if same-named file exists
    HDF5CreateFile/P=$pathName /O /Z fileID as fileName
    if (V_flag != 0)
        Print "HDF5CreateFile failed"
        return -1
    endif

    String listItem
    Variable index
    index = 0
    do
        listItem = StringFromList(index, listOfWaves)
        if (strlen(listItem) == 0)
            break  // No more waves
        endif

        // Create a local reference to the wave
        Wave w = $listItem

        // Save wave as dataset
        HDF5SaveData /O /Z w, fileID
        if (V_flag != 0)
            Print "HDF5SaveData failed"
            result = -1
            break
        endif

        index += 1
    while(1)

    // Close the HDF5 file
    HDF5CloseFile fileID

    return result
End
```

Read through the procedure. It is similar to the TestSaveDataset function.

The first difference is that, instead of passing a wave, we pass a list of wave names in a string parameter. This parameter is a semicolon-separated list which is a commonly-used programming technique in Igor.

The next difference is that we have a do-while loop which extracts a name from the list and saves the corresponding wave as a dataset in the HDF5 file.

The statement

```
Wave w = $listItem
```

creates a local "wave reference" and allows us to use w to refer to a wave whose identity is determined by the contents of the listItem string variable. This also is a common Igor programming technique and is explained in detail in the Programming help file.

13. **Click the Compile button at the bottom of the Procedure window.**

    If you get an error then you have not pasted the right text into the Procedure window or you have entered other incorrect text. If you can not find the error, choose File→Revert Experiment and go back to step 12 of this section.

    Now we are ready to run our procedure.

14. **Enter the following command in the Igor command line either by typing it and pressing enter, by copy/paste followed by Enter, or by selecting it and pressing Control-Enter (Macintosh) or Ctrl-Enter (Windows):**

    ```
    TestSaveDatasets("HDF5Data", "SaveTest.h5", "data0;data1;data2;")
    ```

    The third parameter is a string in this function. In the previous example, it was a name, specifically the name of a wave. Strings are entered in double-quotes but names are not.

    At this point the procedure should have correctly executed and you should see no error messages printed in the history area of the command window (just above the command line).

    Now we will verify that the data was saved.

15. **Activate the HDF5 Browser window. Click the Open HDF5 File button and open the SaveTest.h5 file that we just created.**

    Verify that the file contains datasets named data0, data1 and data2.

16. **Click the Close HDF5 File button.**

17. **Choose File→Save Experiment and save the current work environment in the HDF5 Tour.pxp experiment file.**

    This concludes the HDF5 Guided Tour.

Where To Go From Here

If you are new to Igor or have never done it you should definitely do the **Guided Tour of Igor Pro** on page I-11. If you are in a hurry, do just the first half of it.

You should also read the following chapters which explain the basics of Igor in more detail:

Getting Help on page II-1

Experiments, Files and Folders on page II-15

Windows on page II-43

Waves on page II-61

To get started with Igor programming you need to read these chapters:

Working with Commands on page IV-1

Programming Overview on page IV-23

User-Defined Functions on page IV-29

For HDF5-specific programming, you need to have at least a basic understanding of HDF5. See the links in the next section. Then you need to familiarize yourself with the HDF5-related operations and functions listed in **HDF5 Operations and Functions** on page II-197.

If you run into problems, send a sample HDF5 file along with a description of what you are trying to do to support@wavemetrics.com and we will try to get you started in the right direction.

Learning More About HDF5

In order to use HDF5 operations, you must have at least a basic understanding of HDF5. The HDF5 web site provides an abundance of material. To get started, visit this web page:

```
https://portal.hdfgroup.org/display/HDF5/Learning+HDF5
```

The HDF Group provides a Java-based program called HDFView. You may want to download and install HDFView so that you can easily browse HDF5 files as you read the introductory material. Or you may prefer to use the HDF5 browser provided by Igor.

The HDF5 Browser

Igor Pro includes an automatically-loaded procedure file, "HDF5 Browser.ipf", which implements an HDF5 browser. The browser lets you interactively examine HDF5 files to get an idea of what is in them. It also lets you load HDF5 datasets and groups into Igor and save Igor waves and data folders in HDF5 files.

The browser currently does not support creating attributes. For that you must use the **HDF5SaveData** operation.

The HDF5 browser includes lists which display:

- All groups in the file
- All attributes of the selected group
- All datasets in the selected group
- All attributes of the selected dataset

In addition, the HDF5 browser optionally displays:

- A graph displaying the selected dataset or attribute
- A table displaying the selected dataset or attribute
- A notebook window containing a dump of the selected group, dataset or attribute

Using The HDF5 Browser

To browse HDF5 files, choose Data→Load Waves→New HDF5 Browser. This creates an HDF5 browser control panel. You can create additional browsers by choosing the same menu item again.

Each browser control panel lets you browse one HDF5 file at a time. For most users, one browser will be sufficient.

After creating a new browser, click the Open HDF5 File button to choose the file to browse.

The HDF5 browser contains four lists which display the groups, group attributes, datasets and dataset attributes in the file being browsed.

HDF5 Browser Basic Controls

Here is a description of the basic controls in the HDF5 browser that most users will use.

Create HDF5 File

Creates a new HDF5 file and opens it for read/write.

Open HDF5 File

Opens an existing HDF5 file for read-only or read/write, depending on the state of the Read Only checkbox.

Close HDF5 File

Closes the HDF5 file.

Show Graph

If you click the Show Graph button, the browser displays a preview graph of subsequent datasets or attributes that you select.

Show Table

If you click the Show Table button, the browser displays a preview table of subsequent datasets or attributes that you select.

Load Dataset

The Load Dataset button loads the currently selected dataset into the current data folder.

Load Dataset Options

This section of the browser contains two popup menus that determine if data that you load by clicking the Load Dataset button is displayed in a table or graph.

The Table popup menu contains three items: No Table, Display in New Table, and Append To Top Table. If you choose Append To Top Table and there are no tables, it acts as if you chose Display in New Table.

The Graph popup menu contains three items: No Graph, Display in New Graph, and Append To Top Graph. If you choose Append To Top Graph and there are no graphs, it acts as if you chose Display in New Graph. Appending is useful when you are loading 1D data but of little use when appending multi-dimensional data. Multi-dimensional data is appended as an image plot which obscures anything that was already in the graph.

Text waves are not displayed in graphs even if Display in New Graph or Append To Top Graph is selected.

Save Waves

Displays a panel that allows you to select and save waves in the HDF5 file, provided the file was open for read/write.

Transpose 2D

If the Transpose 2D checkbox is checked, 2D datasets are transposed to compensate for the difference in how Igor and other programs treat rows and columns in an image plot. See **HDF5 Images Versus Igor Images** on page II-204 for details. This does not affect the loading of "formal" images (images formatted according to the HDF5 Image and Palette Specification).

Sort By Creation Order If Possible

If checked, and if the file supports listing by creation order, the HDF5 Browser displays and loads groups and datasets in creation order.

Most HDF5 files do not include creation-order information and so are listed and loaded in alphabetical order even if this checkbox is checked. However, HDF5 files written by Igor Pro 9 or later include creation-order information and so can be listed and loaded in creation order.

Load Group

The Load Group button loads all of the datasets in the currently selected group into a new data folder inside current data folder.

The Load Group button calls the **HDF5LoadGroup** operation using the /IMAG flag. This means that, if the group contains a formal image (see **HDF5 Images Versus Igor Images** on page II-204), it is be loaded as a formal image.

The Hyperselection controls do not apply to the operation of the Load Group button.

Load Groups Recursively

If the Load Groups Recursively checkbox is checked, when the Load Group button is pressed any sub-groups in the currently selected group are loaded as sub-datafolders.

Load Group Only Once

Because an HDF5 file is a "directed graph" rather than a strict hierarchy, a given group in an HDF5 file can appear in more than one location in the file's hierarchy.

If the Load Group Only Once checkbox is checked, the Load Group button loads a given group only the first time it is encountered. If it is unchecked, the Load Group button loads a given group each time it appears in the file's hierarchy resulting in duplicated data. If in doubt, leave Load Group Only Once checked.

Save Data Folder

Displays a panel that allows you to select a single data folder and save it in the HDF5 file, provided the file was open for read/write.

HDF5 Browser Contextual Menus

If you right-click a row in the Datasets, Dataset Attributes or Group Attributes lists, the browser displays a contextual menu that allows you to perform the following actions:

- Copy the selected dataset or attribute's value to the clipboard as text

- Load the selected dataset or attribute as a wave

- Load all datasets or attributes as a waves

These popup menus also appear if you click the triangle icons above the lists. They work the same as the HDF5 Browser contextual menus described above.

When the Datasets list is active, choosing Load Selected Dataset as Wave does the same thing as clicking the Load Dataset button.

When copying data to the clipboard, the data is copied as text and consequently may not represent the full precision of the underlying dataset or attribute. Not all datatypes are supported. If the browser supports the datatype that you right clicked, the contextual menu shows "Copy to Clipboard as Text". If the datatype is not supported, it shows "Can't Copy This Data Type".

When loading as waves, any pre-existing waves with the same names are overridden.

When loading waves from datasets, the table and graph options in the Load Dataset Options section of the browser apply. If you check Apply to Attributes also, the options apply when loading waves from attributes.

HDF5 Browser Advanced Controls

Here is a description of the advanced controls in the HDF5 browser. These are for use by people familiar with the HDF5 file format.

Show Dump

If you click the Show Dump button, the browser displays a notebook in which you can see additional details about subsequent groups, datasets or attributes that you select. The dump window is updated each time you select a group, dataset or attribute from any of the lists.

Show Data In Dump

When unchecked, the dump shows header information but not the actual data of a dataset. When checked it shows data as well as header information for a dataset.

WARNING: If you check the Show Data In Dump checkbox and choose to dump a very large dataset, the dump could take a very long time. If the dump seems to be taking forever, clicking the Abort button in the Igor status bar.

Even if the Show Data In Dump checkbox is checked, the dump for a group consist of the header information only and omits the actual data for datasets and attributes.

Show Attributes In Dump

The Show Attributes In Dump checkbox lets you determine whether attributes are dumped when you select a group or dataset. When checked, information about any attributes associated with the dataset is included in the dump. This checkbox does not affect what is dumped when you select an item in the group or dataset attribute lists.

Show Properties In Dump

The Show Properties In Dump checkbox lets you see properties such as storage layout and filters (compression). This information is usually of little interest but is useful when investigating the effects of compression.

Use Hyperselection

If you check the Use Hyperselection checkbox and enter a path to a "hyperslab wave", the HDF5 Browser uses the hyperselection in the wave to load a subset of subsequent datasets or attributes that you click. This is a feature for advanced users who understand HDF5 hyperselections and have read the **HDF5 Dataset Subsets** discussion below.

The hyperselection is used when you click the Load Dataset button but not when you click the Load Group button.

HDF5 Browser Dump Technical Details

The dump notebook displays a dump of the selected group, dataset or attribute in "Data Description Language" (DDL) format. For most purposes you will not need the dump window. It is useful for experts who are trying to debug a problem or for people who are trying to understand the nuts and bolts of HDF5.

Sometimes strings in HDF5 files contain a large number of trailing nulls. These are not displayed in the dump window.

Sometimes strings in HDF5 files contain the literal strings "\r", "\n" and "\t" to represent carriage return, linefeed and tab. To improve readability, in the dump window these literal strings are displayed as actual carriage returns, linefeeds and tabs.

HDF5 Operations and Functions

This section lists Igor's HDF5-related operations and functions:

HDF5CreateFile	Creates a new HDF5 file or overwrites an existing file.
HDF5OpenFile	Opens an HDF5 file, returning a file ID that is passed to other operations and functions.
HDF5CloseFile	Closes an HDF5 file or all open HDF5 files.
HDF5FlushFile	Flushes an HDF5 file or all open HDF5 files.
HDF5CreateGroup	Creates a group in an HDF5 file, returning a group ID that can be passed to other operations and functions.
HDF5OpenGroup	Opens an existing HDF5 group, returning a group ID that can be passed to other operations and functions.
HDF5ListGroup	Lists all objects in a group.
HDF5CloseGroup	Closes an HDF5 group.
HDF5LinkInfo	Returns information about an HDF5 link.
HDF5ListAttributes	Lists all attributes associated with a group, dataset or datatype.
HDF5AttributeInfo	Returns information about an HDF5 attribute.
HDF5DatasetInfo	Returns information about an HDF5 dataset.
HDF5LoadData	Loads data from an HDF5 dataset or attribute into Igor.
HDF5LoadImage	Loads an image written according to the HDF5 Image and Palette Specification.
HDF5LoadGroup	Loads an HDF5 group and its datasets into an Igor Pro data folder.
HDF5SaveData	Saves Igor waves in an HDF5 file.
HDF5SaveImage	Saves an image in the HDF5 Image and Palette Specification format.
HDF5SaveGroup	Saves an Igor data folder in an HDF5 group.
HDF5TypeInfo	Returns information about an HDF5 datatype.
HDF5CreateLink	Creates a new hard, soft or external link.
HDF5UnlinkObject	Unlinks an object (a group, dataset, datatype or link) from an HDF5 file. This deletes the object from the file's hierarchy but does not free up the space in the file used by the object.
HDF5DimensionScale	Supports the creation and querying of HDF5 dimension scales.

HDF5LibraryInfo	Returns information about the HDF5 library used by Igor. This is of interest to advanced programmers only.
HDF5Control	Provides control of aspects of Igor's use of the HDF5 file format.
HDF5Dump	Returns a DDL-format dump of a group, dataset or attribute.
HDF5DumpErrors	Returns information about HDF5-related errors encountered by Igor. This is a diagnostic tool for experts that is needed only in rare cases.

HDF5 Procedure Files

Igor ships with two procedure files to support HDF5 use and programming. Both files are automatically loaded by Igor on launch and consequently are always available.

"HDF5 Browser.ipf" implements the HDF5 Browser. This procedure file is an independent module and consequently is normally hidden. If you are an Igor programmer who wants to inspect the procedure file, see Independent Modules for background information. However, there is no reason for you to call routines in "HDF5 Browser.ipf" from your own code.

"HDF5 Utilities.ipf" is a public procedure file (i.e., not an independent module) that defines HDF5-related constants and provides HDF5-related utility routines that may be of use if you write procedures that use HDF5 features.

If you write your own procedure file, you can use the constants and utility routines in "HDF5 Utilities.ipf" without #including anything. However, if you are creating your own independent module for HDF5 programming, you will need to #include "HDF5 Utilities.ipf" into your independent module - see "HDF5 Browser.ipf" for an example.

HDF5 Attributes

An attribute is a piece of data attached to an HDF5 group, dataset or named datatype.

To load an attribute, you need to use the HDF5LoadData operation with the /A=attributeNameStr flag and the /TYPE=objectType flag.

Loading attributes of type H5T_COMPOUND (compound - i.e., structure) is not supported.

Loading an HDF5 Numeric Attribute

This function illustrates how to load a numeric attribute of a group or dataset. The function result is an error code. The value of the attribute is returned via the pass-by-reference attributeValue numeric parameter.

```
Function LoadHDF5NumericAttribute(pathName, filePath, groupPath, objectName,
                        objectType, attributeName, attributeValue)
   String pathName            // Symbolic path name - or ""
   String filePath            // File name, relative path or full path
   String groupPath           // Path to group, such as "/", "/my_group"
   String objectName          // Name of group or dataset
   Variable objectType        // 1=group, 2=dataset
   String attributeName       // Name of attribute
   Variable& attributeValue   // Output - pass-by-reference parameter

   attributeValue = NaN

   Variable result = 0

   // Open the HDF5 file
   Variable fileID            // HDF5 file ID will be stored here
   HDF5OpenFile /P=$pathName /R /Z fileID as filePath
   if (V_flag != 0)
      Print "HDF5OpenFile failed"
      return -1
   endif

   Variable groupID           // HDF5 group ID will be stored here
   HDF5OpenGroup /Z fileID, groupPath, groupID
   if (V_flag != 0)
      Print "HDF5OpenGroup failed"
      HDF5CloseFile fileID
      return -1
   endif

   HDF5LoadData /O /A=attributeName /TYPE=(objectType) /N=tempAttributeWave /Q
                      /Z groupID, objectName
   result = V_flag                  // 0 if OK or non-zero error code

   if (result == 0)
      Wave tempAttributeWave
      if (WaveType(tempAttributeWave) == 0)
         attributeValue = NaN       // Attribute is string, not numeric
         result = -1
      else
         attributeValue = tempAttributeWave[0]
      endif
      KillWaves/Z tempAttributeWave
   endif

   // Close the HDF5 group
   HDF5CloseGroup groupID

   // Close the HDF5 file
   HDF5CloseFile fileID

   return result
End
```

Loading an HDF5 String Attribute

This function illustrates how to load a string attribute of a group or dataset. The function result is an error code. The value of the attribute is returned via the pass-by-reference attributeValue string parameter.

```
Function LoadHDF5StringAttribute(pathName, filePath, groupPath, objectName,
                        objectType, attributeName, attributeValue)
   String pathName            // Symbolic path name - or ""
   String filePath            // File name, relative path or full path
   String groupPath           // Path to group, such as "/", "/metadata_group"
   String objectName          // Name of group or dataset
   Variable objectType        // 1=group, 2=dataset
   String attributeName       // Name of attribute
   String& attributeValue     // Output - pass-by-reference parameter

   attributeValue = ""

   Variable result = 0

   // Open the HDF5 file
   Variable fileID               // HDF5 file ID will be stored here
   HDF5OpenFile /P=$pathName /R /Z fileID as filePath
   if (V_flag != 0)
      Print "HDF5OpenFile failed"
      return -1
   endif

   Variable groupID              // HDF5 group ID will be stored here
   HDF5OpenGroup /Z fileID, groupPath, groupID
   if (V_flag != 0)
      Print "HDF5OpenGroup failed"
      HDF5CloseFile fileID
      return -1
   endif

   HDF5LoadData /O /A=attributeName /TYPE=(objectType) /N=tempAttributeWave /Q
                        /Z groupID, objectName
   result = V_flag               // 0 if OK or non-zero error code

   if (result == 0)
      Wave/T tempAttributeWave
      if (WaveType(tempAttributeWave) != 0)
         attributeValue = ""     // Attribute is numeric, not string
         result = -1
      else
         attributeValue = tempAttributeWave[0]
      endif
      KillWaves/Z tempAttributeWave
   endif

   // Close the HDF5 group
   HDF5CloseGroup groupID

   // Close the HDF5 file
   HDF5CloseFile fileID

   return result
End
```

Loading All Attributes of an HDF5 Group or Dataset

This function illustrates loading all of the attributes of a given group or dataset. The attributes are loaded into waves in the current data folder.

```
Function LoadHDF5Attributes(pathName, filePath, groupPath, objectName,
                        objectType, verbose)
   String pathName            // Symbolic path name - or ""
   String filePath            // File name, relative path or full path
   String groupPath           // Path to group, such as "/", "/metadata_group"
   String objectName          // Name of object whose attributes you want or "."
                        for the group specified by groupPath
   Variable objectType        // The type of object referenced by objectPath:
                              // 1=group, 2=dataset
   Variable verbose           // Bit 0: Print errors; Bit 1: Print warnings;
                              // Bit 2: Print routine info

   Variable printErrors = verbose & 1
   Variable printWarnings = verbose & 2
   Variable printRoutine = verbose & 4

   Variable result = 0        // 0 means no error

   // Open the HDF5 file
   Variable fileID            // HDF5 file ID will be stored here
   HDF5OpenFile /P=$pathName /R /Z fileID as filePath
   if (V_flag != 0)
      if (printErrors)
         Print "HDF5OpenFile failed"
      endif
      return -1
   endif

   Variable groupID           // HDF5 group ID will be stored here
   HDF5OpenGroup /Z fileID, groupPath, groupID
   if (V_flag != 0)
      if (printErrors)
         Print "HDF5OpenGroup failed"
      endif
      HDF5CloseFile fileID
      return -1
   endif

   HDF5ListAttributes /TYPE=(objectType) groupID, objectName
   if (V_Flag != 0)
      if (printErrors)
         Print "HDF5ListAttributes failed"
      endif
      HDF5CloseGroup groupID
      HDF5CloseFile fileID
      return -1
   endif

   Variable numAttributes = ItemsInList(S_HDF5ListAttributes)
   Variable i
   for(i=0; i<numAttributes; i+=1)
      String attributeNameStr = StringFromList(i, S_HDF5ListAttributes)

      STRUCT HDF5DataInfo di
      InitHDF5DataInfo(di)    // Initialize structure
      HDF5AttributeInfo(groupID,objectName,objectType,attributeNameStr,0,di)
      Variable doLoad = 0
      switch(di.datatype_class)
         case H5T_INTEGER:
         case H5T_FLOAT:
         case H5T_TIME:       // Not yet tested
         case H5T_STRING:
         case H5T_BITFIELD:   // Not yet tested
         case H5T_OPAQUE:     // Not yet tested
         case H5T_REFERENCE:
         case H5T_ENUM:       // Not yet tested
         case H5T_ARRAY:      // Not yet tested
            doLoad = 1
            break
         case H5T_COMPOUND:   // HDF5LoadData can not load a compound attribute
            doLoad = 0
            break
      endswitch
      if (!doLoad)
         if (printWarnings)
            Printf "Not loading attribute %s - class %s not supported\r",
                        attributeNameStr, di.datatype_class_str
         endif
         continue
      endif

      HDF5LoadData /O /A=attributeNameStr /TYPE=(objectType) /Q /Z groupID,
                        objectName
      if (V_flag != 0)
         if (printErrors)
            Print "HDF5LoadData failed"
         endif
         result = -1
         break
      endif
      if (printRoutine)
         Printf "Loaded attribute %d, name=%s\r", i, attributeNameStr
      endif
   endfor

   // Close the HDF5 group
   HDF5CloseGroup groupID

   // Close the HDF5 file
   HDF5CloseFile fileID

   return result
End
```

HDF5 Dataset Subsets

It is possible, although usually not necessary, to load a subset of an HDF5 dataset using a "hyperslab". To use this feature, you must have a good understanding of hyperslabs which are explained in the HDF5 documentation. The examples below assume that you have read and understood that documentation.

HDF5 does not support loading a subset of an attribute.

To load a subset of a dataset, use the /SLAB flag of the **HDF5LoadData** operation. The /SLAB flag takes as its parameter a "slab wave". This is a two-dimensional wave containing exactly four columns and at least as many rows as there are dimensions in the dataset you are loading.

The four columns of the slab wave correspond to the start, stride, count and block parameters to the HDF5 H5Sselect_hyperslab routine from the HDF5 library.

The following examples illustrate how to use a hyperslab. The examples use a sample file provided by NCSA and stored in Igor's HDF5 Samples directory.

Create an Igor symbolic path (Misc→New Symbolic Path) named HDF5Samples which points to the folder containing the i32le.h5 file (Igor Pro X Folder:Examples:Feature Demos:HDF5 Samples).

In the first example, we load a 2D dataset named "TestArray" which has 6 rows and 5 columns. We start by loading the entire dataset without using a hyperslab.

```
Variable fileID
HDF5OpenFile/P=HDF5Samples /R fileID as "i32le.h5"
HDF5LoadData /N=TestWave fileID, "TestArray"
Edit TestWave
```

Now we create a slab wave and set its dimension labels to make it easier to remember which column holds which type of information. We will use a utility routine in the automatically-loaded "HDF5 Utiliies.ipf" procedure file which is automatically loaded when Igor is launched:

```
HDF5MakeHyperslabWave("root:slab", 2   )  // In HDF5 Utilities.ipf
Edit root:slab.ld
```

Now we set the values of the slab wave to give the same result as before, that is, to load the entire dataset, and then we load the data again using the slab.

```
slab[][%Start] = 0          // Start at zero for all dimensions
slab[][%Stride] = 1         // Use a stride of 1 for all dimensions
slab[][%Count] = 1          // Use a count of 1 for all dimensions
slab[0][%Block] = 6         // Set block size for the dimension 0 to 6
slab[1][%Block] = 5         // Set block size for the dimension 1 to 5
TestWave = -1               // So we can see the next command change it
HDF5LoadData /N=TestWave /O /SLAB=slab fileID, "TestArray"
```

Here we set the stride, count and block parameters to load every other element from both dimensions.

```
slab[][%Start] = 0          // Start at zero for all dimensions
slab[][%Stride] = 2         // Use a stride of 2 for all dimensions
slab[0][%Count] = 3         // Load three blocks of dimension 0
slab[1][%Count] = 2         // Load two blocks of dimension 1
slab[0][%Block] = 1         // Set block size for dimension 0 to 1
slab[1][%Block] = 1         // Set block size for dimension 1 to 1
TestWave = -1               // So we can see the next command change it
HDF5LoadData /N=TestWave /O /SLAB=slab fileID, "TestArray"
```

Finally we close the file.

```
HDF5CloseFile fileID
```

Each row of the slab wave holds a set of parameters (start, stride, count and block) for the corresponding dimension of the dataset in the file. Row 0 of the slab wave holds the parameters for dimension 0 of the dataset, and so on.

The start values must be greater than or equal to zero. All of the other values must be greater than or equal to 1. All values must be less than 2 billion.

HDF5LoadData clips the values supplied for the block sizes to the corresponding sizes of dataset being loaded.

HDF5LoadData requires that the slab wave have exactly four columns. The HDF5MakeHyperslabWave function, from the automatically-loaded "HDF5 Utiliies.ipf" procedure file, creates a four-column wave with the column dimension labels Start, Stride, Count, and Block. HDF5LoadData does not require the column dimension labels. As of Igor Pro 9.00, HDF5MakeHyperslabWave returns a free wave if you pass "" for the path parameter. It also returns a wave reference as the function result whether the wave is free or not.

The slab wave must have at least as many rows as the dataset has dimensions. Extra rows are ignored.

HDF5LoadData creates a wave just big enough to hold all of the loaded data. So in the first example, it created a 6 x 5 wave whereas in the last example it created a 3 x 2 wave.

Igor Versus HDF5

This section documents differences in how Igor and HDF5 organize data and how Igor reconciles them.

Case Sensitivity

Igor is not case-sensitive but HDF5 is. So, for example, when you specify the name of an HDF5 data set, case matters. "/Dataset1" is not the same as "/dataset1".

If you load /Dataset1 and /dataset1 into Igor using the default wave name, the second load overwrites the wave created by the first.

HDF5 Object Names Versus Igor Object Names

The forward slash character is not allowed in HDF5 object names. If you create an Igor wave or data folder with a name containing a forward slash and attempt to save the object to an HDF5 file, you will get an error. An object name that starts with a dot may also create an error.

As of this writing, the section 4.2.3, "HDF5 Path Names" of the HDF5 User's Guide says:

Component link names may be any string of ASCII characters not containing a slash or a dot ("/"and ".", which are reserved as noted above). However, users are advised to avoid the use of punctuation and non-printing characters, as they may create problems for other software.

HDF5 Data Types Versus Igor Data Types

HDF5 supports many data types that Igor does not support. When loading such data types, Igor attempts to convert to a data type that it supports, if possible. In the process, precision may be lost.

By default and for backward compatibility, Igor saves and loads HDF5 64-bit integer data as double-precision floating point. Precision may be lost in this conversion. To save and load 64-bit integer data as 64-bit integer, use /OPTS=1 with **HDF5SaveData**, **HDF5SaveGroup**, **HDF5LoadData** and **HDF5LoadGroup**. Because most operations are carried out in Igor in double-precision floating point, we recommend loading 64-bit integer as double if it fits in 53 bits and as 64-bit integer if it may exceed 53 bits.

Since Igor does not currently support 128-bit floating point data (long double), Igor loads HDF5 128-bit floating point data as double-precision floating point. Precision is lost in this conversion.

HDF5 Max Dimensions Versus Igor Max Dimensions

The HDF5 library supports data with up to 32 dimensions. Igor supports only four dimensions. If you load HDF5 data whose dimensionality is greater than four, the **HDF5LoadData** operation creates a 4D wave with enough chunks to hold all of the data. ("Chunk" is the name of the fourth dimension in Igor, after "row", "column" and "layer".)

For example, if you load a dataset whose dimensions are 7x6x5x4x3x2, HDF5LoadData creates a wave with dimensions 7x6x5x24. The wave has 24 chunks. The number 24 comes from multiplying the number of chunks in the HDF5 dataset (4 in this case) by the size of each higher dimension (3 and 2 in this case).

Row-major Versus Column-major Data Order

HDF5 and Igor store multi-dimensional data differently in memory. For almost all purposes, this difference is immaterial. For those rare cases in which it matters, here is an explanation.

HDF5 stores multi-dimensional data in "row-major" order. This means that the index associated with the highest dimension changes fastest as you move in memory from one element to the next. For example, if you have a two-dimensional dataset with 5 rows and 4 columns, as you move sequentially through memory, the column index changes fastest and the row index slowest.

Igor stores data in "column-major" order. This means that the index associated with the lowest dimension changes fastest as you move in memory from one element to the next. For example, if you have a two-dimensional dataset with 5 rows and 4 columns, as you move sequentially through memory, the row index changes fastest and the column index slowest.

To work around this difference, after the **HDF5LoadData** or **HDF5LoadGroup** operation initially loads data with two or more dimensions, it shuffles the data around in memory. The result is that the data when viewed in an Igor table looks just as it would if viewed in the HDF Group's HDFView program although its layout in memory is different. A similar accomodation occurs when you save Igor data using **HDF5SaveData** or **HDF5SaveGroup**.

HDF5 Images Versus Igor Images

The HDF5 Image and Palette Specification provides detailed guidelines for storing an image and its associated information (e.g., palette, color model) in an HDF5 file. However many HDF5 users do not follow the specification and just write image data to a regular 2D dataset. To distinguish between these two ways of storing an image, we use the term "formal image" to refer to an image written to the specification and "regular image" to refer to regular 2D datasets which the user thinks of as an image.

In an Igor image plot the wave's column data is plotted horizontally while in HDFView and most other programs the row data is plotted horizontally. Therefore, without special handling, a regular image would appear rotated in Igor relative to most programs.

The **HDF5LoadImage** and **HDF5SaveImage** operations handle loading and saving formal images. These operations automatically compensate for the difference in image orientation.

If you are dealing with a regular image, you will use the **HDF5LoadData** and **HDF5SaveData** operations, or **HDF5LoadGroup** and **HDF5SaveGroup**. These operations have a /TRAN flag which causes 2D data to be transposed. When you use /TRAN with HDF5LoadData, images viewed in Igor and in programs like HDFView will have the same orientation but will appear transposed when viewed in a table.

The /TRAN flag works with 2D and higher-dimensioned data. When used with higher-dimensioned data (3D or 4D), each layer of the data is treated as a separate image and is transposed. In other words, /TRAN treats higher-dimensioned data as a stack of images.

Saving and Reloading Igor Data

The **HDF5SaveData** and **HDF5SaveGroup** operations can save Igor waves, numeric variables and string variables in HDF5 files. All of these Igor objects are written as HDF5 datasets.

The datasets saved from Igor waves are, by default, marked with attributes that store wave properties such as the wave data type, the wave scaling and the wave note. The attributes have names like IGORWaveType and IGORWaveScaling. This allows **HDF5LoadData** and **HDF5LoadGroup** to fully recreate the Igor wave if it is later read from the HDF5 file back into Igor. You can suppress the creation of these attributes by using the /IGOR=0 flag when calling HDF5SaveData or HDF5SaveGroup.

Wave text is always written using UTF-8 text encoding. See **HDF5 Wave Text Encoding** on page II-221 for details.

Wave reference waves and data folder reference waves are read as such when you load an HDF5 packed experiment but HDF5LoadData and HDF5LoadGroup load these waves as double-precision numeric. The reason for this is that restoring such waves so that they point to the correct wave or data folder is is possible only when an entire experiment is loaded.

The datasets saved by HDF5SaveGroup from Igor variables are marked with an "IGORVariable" attribute. This allows **HDF5LoadData** and **HDF5LoadGroup** to recognize these datasets as representing Igor variables if you reload the file. In the absence of this attribute, these operations load all datasets as waves.

The value of the IGORVariable attribute is the data type code for the Igor variable. It is one of the following values:

0:	Igor string variable
4:	Igor real numeric variable
5:	Igor complex numeric variable

See also **HDF5 String Variable Text Encoding** on page II-221.

Handling of Complex Waves

Igor Pro supports complex waves but HDF5 does not support complex datasets. Therefore, when saving a complex wave, **HDF5SaveData** writes the wave as if its number of rows were doubled. For example, HDF5SaveData writes the same data to the HDF5 file for these waves:

```
Make wave0 = {1,-1,2,-2,3,-3}                              // 6 scalar points
Make/C cwave0 = {cmplx(1,-1),cmplx(2,-2),cmplx(3,-3)}     // 3 complex points
```

When reading an HDF5 file written by HDF5SaveData, you can determine if the original wave was complex by checking for the presence of the IGORComplexWave attributes that HDF5SaveData attaches to the dataset for a complex wave. **HDF5LoadData** and **HDF5LoadGroup** do this automatically if you use the appropriate /IGOR flag.

Handling of Igor Reference Waves

Igor Pro supports wave reference waves and data folder reference waves. Each element of a wave reference wave is a reference to another wave. Each element of a data folder reference wave is a reference to a data folder.

Igor correctly writes Igor reference waves when you save an experiment as HDF5 packed format, but the **HDF5SaveData** and **HDF5SaveGroup** do not support saving Igor reference waves.

The HDF5SaveData operation returns an error if you try to save a reference wave.

The behavior of the HDF5SaveGroup operation when it is asked to save a reference wave depends on the /CONT flag. By default (/CONT=1), it prints a note in the history saying it can not save the wave and then continues saving the rest of the objects in the data folder. If /CONT=0 is used, HDF5SaveGroup returns an error if asked to save a reference wave.

HDF5 Multitasking

You can call HDF5 operations and functions from an Igor preemptive thread.

The HDF5 library is limited to accessing one HDF5 file at a time. This precludes loading multiple HDF5 files concurrently in a given Igor instance but it does allow you to load an HDF5 file in a preemptive thread while you do something else in Igor's main thread. If you create multiple threads that attempt to access HDF5 files, one of your threads will gain access to the HDF5 library. Your other HDF5-accessing threads will wait until the first thread finishes at which time another thread will gain access to the HDF5 library.

For further information on multitasking and for examples, see the Demo Thread Safe HDF5 example experiment.

Igor HDF5 Capabilities

Igor supports only a subset of all of the HDF5 capability.

Here is a partial list of HDF5 features that Igor does support:

- Loading of all atomic datatypes.
- Loading of strings.
- Loading of array datatypes.
- Loading of variable-length datasets where the base type is integer or float.
- Loading of compound datasets (datasets consisting of C-like structures), including compound datasets containing members that are arrays.
- Use of hyperslabs to load subsets of datasets.
- Loading and saving object references.
- Loading dataset region references.

Igor HDF5 Limitations

Here is a partial list of HDF5 features that Igor does not support:

- Creating or appending to VLEN datasets (ragged arrays).
- Loading of deeply-nested datatypes. See **HDF5 Nested Datatypes** below.
- Saving dataset region references.

If Igor does not work with your HDF5 file, it could be due to a limitation in Igor. Send a sample file along with a description of what you are trying to do to support@wavemetrics.com and we will try to determine what the problem is.

Advanced HDF5 Data Types

This section is mostly of interest to advanced HDF5 users.

HDF5 Variable-Length Data

Most HDF5 files do not use variable-length datatypes so most users do not need to know this information.

Variable-length data consists of an array where each element is a 1D set of elements of another datatype, called the "base" datatype. The number of elements of the base datatype in each set can be different. For example, a 5 element variable-length dataset whose base type is H5T_STD_I32LE contains 5 1D sets of 32-bit, little-endian integers and the length of each set is independent.

HDF5LoadData loads variable-length datasets where the base type is integer or float only. The data for each element is loaded into a separate wave.

When loading most types of data, HDF5LoadData creates just one wave. When loading a variable-length dataset or attribute, one wave is created for each loaded element. If more than one element is to be loaded, the proposed name for the wave (the name of the dataset or attribute being loaded or a name specified by /N=name) is treated as a base name. For example, if the dataset or attribute has three elements and name is test and the /O flag is used, waves named test0, test1 and test2 are created. If the /O flag is not used, names of the form test<n> are created where <n> is a number chosen to make the wave names unique.

HDF5LoadData operation supports loading a subset of a variable-length dataset. You do this by supplying a slab wave using the HDF5LoadData /SLAB flag. In the example from the previous paragraph, if you loaded just one element, its name would be test, not test0. If you loaded two elements, they would be named test0 and test1, regardless of which two elements you loaded.

This function demonstrates loading one element of a variable-length dataset. We assume that a symbolic path named Data and a file named "Vlen.h5" exist and that the file contains a 1D variable-length dataset named TestVlen that contains at least two elements. The function loads the second variable-length element into a wave named TestWave.

```
Function DemoVlenLoad()
   Variable fileID
   HDF5OpenFile /P=Data /R fileID as "Vlen.h5"

   HDF5MakeHyperslabWave("root:slab", 1)  // In HDF5 Utilities.ipf
   Wave slab = root:slab
   slab[0][%Start] = 1                    // Start at second vlen element
   slab[0][%Stride] = 1                   // Use a stride of 1
   slab[0][%Count] = 1                    // Load 1 block
   slab[0][%Block] = 1                    // Set block size to 1
   HDF5LoadData /N=TestWave /O /SLAB=slab fileID, "TestVlen"

   HDF5CloseFile fileID
End
```

HDF5 Array Data

Most HDF5 files do not use array datatypes so most users do not need to know this information.

An HDF5 dataset (or attribute) consists of elements organized in one or more dimensions. Each element can be an atomic datatype, such as an unsigned short or a double-precision float, or it can be a composite datatype, such as a structure or an array. Thus, an HDF5 dataset can be an array of unsigned shorts, an array of doubles, an array of structures or an array of arrays. This section discusses loading this last type - an array of arrays.

In this case, the class of the dataset is H5T_ARRAY. The type of the dataset is something like "5 x 4 array of signed long" and "signed long" is said to be the "base type" of the array datatype. If the dataset itself is 1D with 10 rows then you would have a 10-row dataset, each element of which consists of one 5 x 4 array of signed longs.

In Igor this could be treated as a 3D wave consisting of 5 rows, 4 columns and 10 layers. However, since Igor supports only 4 dimensions while HDF5 supports 32, you could easily run out of dimensions in Igor. Therefore when you load an H5T_ARRAY-type dataset, HDF5LoadData creates a wave whose dimensionality is that of the array type, not that of the underlying dataset, except that the highest dimension is increased to make room for all of the data. This reduces the likelihood of running out of dimensions.

HDF5LoadData can not load array data whose base type is compound or array (see **HDF5 Nested Datatypes** on page II-211 for details).

The following example illustrate how to load an array datatype. The example uses a sample file provided by NCSA and stored in Igor's HDF5 Samples directory.

Create an Igor symbolic path (Misc→New Symbolic Path) named HDF5Samples which points to the folder containing the SDS_array_type.h5 file (Igor Pro X Folder:Examples:Feature Demos:HDF5 Samples).

The sample file contains a 10 row 1D data set, each element of which is a 5 row by 4 column matrix of 32-bit signed big-endian integers (a.k.a., big-endian signed longs or I32BE).

When we load the dataset, we get 5 rows and 40 columns.

```
Variable fileID
HDF5OpenFile/P=HDF5Samples /R fileID as "SDS_array_type.h5"
HDF5LoadData fileID, "IntArray"
HDF5CloseFile fileID
Edit IntArray
```

The first four columns contain the 5x4 matrix from row zero of the dataset. The next four columns contain the 5x4 matrix from row one of the dataset. And so on.

In this case, Igor has enough dimensions so that you could, if you want, reorganize it as a 3D wave consisting of 10 layers of 5x4 matrices. You would do that using this command:

```
Redimension /N=(5,4,10) /E=1 IntArray
```

The /E=1 flag tells Redimension to change the dimensionality of the wave without actually changing the stored data. In Igor, the layout in memory of a 5x4x10 wave is the same as the layout of a 5x40 wave. The redimension merely changes the way we look at the wave from 40 columns of 5 rows to 10 layers of 4 columns of 5 rows.

Although you can load a dataset with an array datatype, Igor currently provides no way to write a datatype with an array datatype.

Loading HDF5 Reference Data

Most HDF5 files do not use reference datatypes so most users do not need to know this information.

An HDF5 dataset or attribute can contain references to other datasets, groups and named datatypes. There are two types of references: "object references" and "dataset region references". **HDF5LoadData** loads both types of references into text waves.

An element of an object reference dataset can refer to a dataset in the same or in another file. An element of a dataset region reference dataset can refer only to a dataset in the same file.

Loading HDF5 Object Reference Data

For each object reference to a dataset, HDF5LoadData returns "D:" plus the full path of the dataset within the HDF5 file, for example, "D:/GroupA/DatasetB".

For references to groups and named datatypes, HDFLoadData returns "G:" and "T:" respectively, followed by the path to the object.

Loading HDF5 Dataset Region Reference Data

This section is for advanced HDF5 users. A demo experiment, "HDF5 Dataset Region Demo.pxp", provides examples and utilities for dealing with dataset region references.

Prior to Igor Pro 9.00, HDF5LoadData returned the same thing when loading an object reference or a dataset region reference. It had this form:

```
<object type character>:<full path>
```

For a dataset, this might be something like

```
D:/Group1/Dataset3
```

In Igor Pro 9.00 and later, HDF5LoadData returns additional information for a dataset region reference. It has this form with the additional information shown in red:

```
<object type character>:<full path>.<region info>
```

For a dataset, this might be something like this with the additional information shown in red:

```
D:/Group1/Dataset3.REGIONTYPE=BLOCK;NUMDIMS=2;NUMELEMENTS=2;COORDS=0,0-
0,2/0,11-0,13;
```

If you have code that depends on the pre-Igor9 behavior, you can make it work with Igor9 by using StringFromList with "." as list separator string to obtain the text preceding the dot character.

The region info string is a semicolon-separated keyword-value string constructed for ease of programmatic parsing. It consists of these parts:

```
REGIONTYPE=<type>
NUMDIMS=<number of dimensions>
NUMELEMENTS=<number of elements>
COORDS=<list of points> or <list of blocks>
```

<type> is POINT for a region defined as a set of points, BLOCK for a region defined as a set of blocks.

<number of dimensions> is the number of dimensions in the dataset.

<number of elements> is the number of points in a region defined as a set of points or the number of blocks in a region defined as a set of blocks.

<list of points> has the following format:

```
<point coordinates>/<point coordinates>/...
```

where <point coordinates> is a comma-separated list of coordinates.

For a 2D dataset with three selected points, this might look like this:

```
3,4/11,13/21,30
```

which specifies these three points:

```
row 3, column 4
row 11, column 13
row 21, column 30
```

<list of blocks> has the following format:

```
<coordinates>-<coordinates>/<coordinates>-<coordinates>/...
```

where <coordinates> is a comma-separated list of coordinates.

A dash appears between pairs of <coordinates>. The first set of coordinates of a pair specifies the starting coordinates of a block while the second set of coordinates of a pair specifies the ending coordinates of the block.

For a 2D dataset with three selected blocks, this might look like this:

```
3,4-6,7/11,13-15,17/21,30-37,38
```

which specifies these three blocks:

```
row 3, column 4 to row 6, column 7
row 11, column 13 to row 15, column 17
row 21, column 30 to row 37, column 38
```

Here is an example of a complete point dataset region info string with the additional information shown in red:

```
D:/Group1/Dataset3.REGIONTYPE=POINT;NUMDIMS=2;NUMELEMENTS=3;COORDS=3,4/11,13/2
1,30;
```

Here is an example of a complete block dataset region info string with the additional information shown in red:

```
D:/Group1/Dataset4.REGIONTYPE=BLOCK;NUMDIMS=2;NUMELEMENTS=3;COORDS=3,4-
6,7/11,13-15/17/21,30-37/38;
```

The wave returned after calling HDF5LoadData on a two-row dataset region reference dataset would contain two rows of text like the examples just shown. Each row in the dataset region reference dataset refers to one dataset and to a set of points or blocks within that dataset.

The "HDF5 Dataset Region Demo.pxp" experiment provides further information including examples and utilities for dealing with dataset region references.

Saving HDF5 Object Reference Data

Most HDF5 files do not use reference datatypes so most users do not need to know this information.

An HDF5 dataset or attribute can contain references to other datasets, groups and named datatypes. These are called "object references". You can instruct HDF5SaveData to save a text wave as an object reference using the /REF flag. The /REF flag requires Igor Pro 8.03 or later.

The text to save as a reference must be formatted with a prefix character identifying the type of the referenced object followed by a full or partial path to the object: "G:", "D", or "T:" for groups, datatypes, and datasets respectively. For example:

```
Function DemoSaveReferences(pathName, fileName)
   String pathName         // Name of symbolic path
   String fileName         // Name of HDF5 file

   Variable fileID
   HDF5CreateFile/P=$pathName /O fileID as fileName

   // Create a group to target using a reference
   Variable groupID
   HDF5CreateGroup fileID, "GroupA", groupID

   // Create a dataset to target using a reference
   Make/O/T textWave0 = {"One", "Two", "Three"}
   HDF5SaveData /O /REF=(0) /IGOR=0 textWave0, groupID

   // Write reference dataset to root using full paths
   Make/O/T refWaveFull = {"G:/GroupA", "D:/GroupA/textWave0"}
   HDF5SaveData /O /REF=(1) /IGOR=0 refWaveFull, fileID
   HDF5CloseGroup groupID
   HDF5CloseFile fileID
End
```

Partial paths are relative to the file ID or group ID passed to HDF5SaveData.

Saving HDF5 Dataset Region Reference Data

Igor Pro does not currently support saving dataset region references.

HDF5 Enum Data

Most HDF5 files do not use enum datatypes so most users do not need to know this information.

Enum data values are stored in an HDF5 file as integers. The datatype associated with an enum dataset or attribute defines a mapping from an integer value to a name. HDF5LoadData can load either the integer data or the name for each element of enum data. You control this using the /ENUM=enumMode flag.

If /ENUM is omitted or enumMode is zero, HDF5LoadData creates a numeric wave with data type signed long and loads the integer enum values into it.

If enumMode is 1, HDF5LoadData creates a text wave and loads the name associated with each enum value into it. This is slower than loading the integer enum values but the speed penalty is significant only if you are loading a very large enum dataset or very many enum datasets.

HDF5 Opaque Data

Most HDF5 files do not use opaque datatypes so most users do not need to know this information.

Opaque data consists of elements that are treated as a string of bytes of a specified length. HDF5LoadData loads opaque data into an unsigned byte wave. If an element of opaque data is n bytes then each element occupies n contiguous rows in the wave.

HDF5 Bitfield Data

Most HDF5 files do not use bitfield datatypes so most users do not need to know this information.

Bitfield data consists of elements that are a sequence of bits treated as individual values. HDF5LoadData loads bitfield data of any length into an unsigned byte wave with as many bytes per element as are needed. For example, if the bitfield is two bytes then two rows of the unsigned byte wave are used for each element.

The data is loaded using the byte order as stored in the file. This is appropriate if you think of the data as a stream of bytes. If you think of the data as a short (2 bytes) or long (4 bytes), you can use the Redimension command. For example, if you just loaded a 2D data set containing 5 two-byte bitfields per row and 3 columns, you wind up with a 10x3 unsigned byte wave. You can change it to a 5x3 wave of shorts like this:

```
Redimension /N=(5,3) /W /E=1 bitfieldWave
```

If you need to change the byte order, use /E=2 instead of /E=1.

HDF5 Nested Datatypes

Most HDF5 files do not use nested datatypes so most users do not need to know the following information.

HDF5 supports "atomic" data classes, such as integers (called class H5T_INTEGER in the HDF5 library) and floats (class H5T_FLOAT), and "composite" data classes, such as structures (class H5T_COMPOUND), arrays (class H5T_ARRAY) and variable-length (class H5T_VLEN) types.

In a dataset whose datatype is atomic, each element of the dataset is a single value. In a dataset whose datatype is composite, each element is a collection of elements of one or more other datatypes.

For example, consider a 5x3 array of data of type H5T_STD_I16BE. H5T_STD_I16BE is a built-in atomic datatype of class H5T_INTEGER. Each element of such an array is a 16-bit, big-endian integer value. This dataset has an atomic datatype.

Now consider a 4 row dataset of class H5T_ARRAY where each element of the dataset is a 5x3 array of data of type H5T_STD_I16BE. This dataset has a composite datatype. Each of the 4 elements of the dataset is an array, in this case, an array of 16-bit, big-endian integer values. H5T_STD_I16BE is the "base datatype" of each of the 4 arrays.

Now consider a 4 row dataset of class H5T_COMPOUND with one integer member, one float member and one string member. This dataset has a composite datatype. Each of the 4 elements of this dataset is a structure with members of three different datatypes.

In HDF5 it is possible to nest datatypes to any depth. For example, you can create an H5T_ARRAY dataset where each element is an H5T_COMPOUND dataset where the members are H5T_ARRAY datasets of H5T_STD_I16BE, H5T_STD_I32BE and other datatypes.

Igor does not support indefinitely nested datasets. It supports only the following:

- Atomic datasets of almost any type.
- Array datasets where the base type is integer, float, string, bitfield, opaque or enum.
- Compound datasets where the member's base type is integer, float, string, bitfield, opaque, enum or reference.
- Compound datasets with array members where the base type of the array is integer, float, string, bitfield, opaque, enum or reference.
- Variable-length datasets where the base type is integer or float.

HDF5 Compound Data

This is an advanced feature that most users will not need.

In an HDF5 compound dataset, each element of the dataset consists of a set of named fields, like an instance of a C structure. Loading compound datasets is problematic because their structure can be arbitrarily complex.

A compound data set may contain a collection of disparate datatypes, arrays of disparate datatypes, and sub-compound structures.

HDF5LoadData can load either a single member from a compound dataset into a single wave, or it can load all members of the compound dataset into a set of waves. However, if a member is too complex, HDF5LoadData can not load it and returns an error. For the most part, "too complex" means that the member is itself compound (a sub-structure).

You instruct the HDF5LoadData operation to load a single member from each element of the compound dataset by using the /COMP flag with a mode parameter set to one and with the name of the member to load. The member must be an atomic datatype or an array datatype but can not be another compound datatype (see HDF5 Nested Datatypes for details). HDF5LoadData creates an Igor wave with a data type that is compatible with the datatype of the HDF5 dataset. The name of the wave is based on the name of the dataset or attribute being loaded or on the name specified by the /N flag.

You instruct HDF5LoadData to load all members into separate waves by omitting the /COMP flag or specifying /COMP={0,""}. The names of the waves created consist of a base name concatenated with a possibly cleaned up version of the member name. The base name is based on the name of the dataset or attribute being loaded or on the name specified by the /N flag.

Although you can load an HDF5 dataset with a compound datatype, Igor currently provides no way to write a datatype with a compound datatype.

HDF5 Compression

Igor can use compression when writing datasets to HDF5 files. Compressed datasets usually take less space on disk but more time to write and more time to read.

The amount of disk space saved by compression depends on the size and nature of your data. If your data is very noisy, compression may save little disk space or even increase the amount of disk space required. If your data is nearly all zero or some other constant value, compression may save 99 percent of disk space. With typical real-world data, compression may save from 10 to 50 percent of disk space.

The time required to save a compressed dataset is typically in the range of 2 to 10 times the time required to save an uncompressed dataset. The time required to load a compressed dataset is typically 2 times the time required to load an uncompressed dataset. These times can vary widely depending on your data.

Compression may be performed when you

- Save an HDF5 packed experiment file
- Save a dataset or group using the HDF5 Browser
- Call the HDF5SaveData operation
- Call the HDF5SaveGroup operation

To enable compression Igor must provide certain parameters to the HDF5 library:

- The level of compression for GZIP from 0 to 9
- Whether shuffling should be performed or not
- The chunk size to use for each dimension of the wave being saved

Compression parameters for a given dataset are set when the dataset is created and can not be changed when appending to an existing dataset.

There are several ways through which Igor obtains these parameters:

- From the HDF5SaveData operation /GZIP and /LAYO flags
- From the HDF5SaveGroup, SaveExperiment, and SaveData operation /COMP flags
- From the SaveGraphCopy, SaveTableCopy, and SaveGizmoCopy operation /COMP flags
- From the HDF5SaveDataHook (see **Using HDF5SaveDataHook** on page II-215 for details)
- Via HDF5 default compression for manual saves (see **HDF5 Default Compression** on page II-214 for details)

HDF5 supports compression of datasets through three categories of filters:

- Internal filters: Shuffle, Fletcher32, ScaleOffset, and NBit

  Internal filters are implemented in the HDF5 library source code.
- External filters: GZIP, SZIP

  External filters are linked to the HDF5 library when the library is compiled.
- Third-party filters

  Third-party filters are detected and loaded by the HDF5 library at runtime.

Igor Pro currently supports the following HDF5 filters:

- GZIP
- SZIP for reading only - not supported for writing
- Shuffle

  Shuffle reorders the bytes of multi-byte data elements and can result in higher compression ratios.

HDF5 Layout Chunk Size

Layout refers to how data for a given dataset is arranged on disk. The HDF5 library supports three types of layout: contiguous, compact, chunked. The HDF5SaveData and HDF5SaveGroup operations default to contiguous layout which is appropriate for uncompressed datasets.

The HDF5 library requires that compressed datasets use the chunked layout. This means that the data for the dataset is written in some number of discrete chunks rather than in one contiguous block. In the context of compression, chunking is mainly useful for speeding up accessing subsets of a dataset. Without chunking, the entire dataset has to be read and decompressed to read any subset. If the dataset is always read in its entirety, such as when Igor loads an HDF5 packed experiment file, chunking does not enhance speed.

Chunked storage requires telling the HDF5 library the size of a chunk for each dimension of the dataset. There is no general way to choose appropriate chunk sizes because it depends on the dimensions and nature of the data as well as tradeoffs that require the judgement of the user. In Igor chunk sizes can be specified

- Using the HDF5SaveData operation /LAYO flag (see HDF5SaveData for details)
- Using HDF5SaveDataHook (see **Using HDF5SaveDataHook** on page II-215 for details)

The HDF5SaveGroup, SaveExperiment, SaveData, SaveGraphCopy, SaveTableCopy, and SaveGizmoCopy operations and HDF5 default compression always save a compressed dataset in one chunk, as described in the following sections.

Compression parameters for a given dataset are set when the dataset is created and can not be changed when appending to an existing dataset.

HDF5SaveGroup Compression

In Igor Pro 9.00 and later, you can tell the HDF5SaveGroup operation to compress numeric datasets using the /COMP flag.

HDF5SaveGroup applies compression only to numeric waves, not to text waves or other non-numeric waves nor to numeric waves with fewer than the number of elements specified by the /COMP flag.

HDF5SaveGroup compression uses chunk sizes equal to the size of each dimension of the wave. Such chunk sizes mean that the entire wave is written using chunked layout as one chunk. This is fine for datasets that are to be read all at once. For finer control you can use the HDF5SaveDataHook function to override compression specified by /COMP but this is usually not necessary.

SaveExperiment Compression

In Igor Pro 9.00 and later, you can tell the SaveExperiment operation to compress numeric datasets using the /COMP flag. This works the same as HDF5SaveGroup compression discussed in the preceding section.

The same is true for the SaveData, SaveGraphCopy, SaveTableCopy, and SaveGizmoCopy operations.

See **HDF5 Compression for Saving Experiment Files** on page II-16 for step-by-step instructions.

HDF5 Default Compression

Igor can perform default dataset compression when you save HDF5 files via the user interface. Default compression does not apply when saving HDF5 files via the HDF5SaveData, HDF5SaveGroup, SaveExperiment, SaveData, SaveGraphCopy, SaveTableCopy, or SaveGizmoCopy operations. HDF5 default compression was added in Igor Pro 9.00.

Default compression is disabled by default because compression can significantly increase the time required to save. Whether compression is worthwhile depends on the size and nature of your data and on how you trade off time versus disk space.

You can enable default compression by choosing Misc→Miscellaneous Settings and clicking the Experiment icon. This allows you to set the following settings:

- Whether default compression is enabled or disabled
- The minimum size a wave must be before compression is used
- A compression level from 0 (no compression) to 9 (max)
- Whether you want to enable shuffle (an optional process before compression)

You can programmatically determine the default compression currently in effect like this:

```
Print IgorInfo(14)      // Print default compression settings
```

Once you have turned default compression on, it applies to saving HDF5 packed experiment files via the user interface, namely via the Data Browser Save Copy button, via the HDF5 Browser Save Waves and Save Data Folder buttons, and via the following File menu items: Save Experiment, Save Experiment As, Save Experiment Copy, Save Graph Copy, Save Table Copy, and Save Gizmo Copy.

The HDF5SaveDataHook function, if it exists, can override default compression settings.

Igor applies default compression only to numeric waves, not to text waves or other non-numeric waves nor to numeric waves with fewer than the number of elements specified by the settings.

Default compression uses chunk sizes equal to the size of each dimension of the wave. Such chunk sizes mean that the entire wave is written using chunked layout as one chunk. This is fine for datasets that are to be read all at once.

Igor's HDF5 default compression is sufficient for most purposes. If necessary, for example if you intend to read subsets rather than the entire dataset at once, you can override it using the HDF5SaveData operation or an HDF5SaveDataHook function.

Using HDF5SaveDataHook

HDF5SaveDataHook is a user-defined function that Igor calls to determine what kind of compression, if any, should be applied when saving a given wave as a dataset.

Support for HDF5SaveDataHook was added in Igor Pro 9.00.

NOTE: HDF5SaveDataHook is an experimental feature for advanced users only. The feature may be changed or removed. If you find it useful, please let us know, and send your function and an explanation of what purpose it serves.

The hook function takes a single HDF5SaveDataHookStruct parameter containing input and output fields. The version, operation, and waveToSave fields are inputs. The gzipLevel, shuffle, and chunkSizes fields are outputs.

The main use for HDF5SaveDataHook is to provide a way by which you can specify chunk sizes on a wave-by-wave basis taking into account the wave dimension sizes and your tradeoff of disk space versus save time. The HDF5SaveData operation allows you to specify chunk sizes via the /LAYO flag. However, the other operations that support HDF5 compression through the /COMP flag (HDF5SaveGroup, SaveExperiment, SaveData, SaveGraphCopy, SaveTableCopy, and SaveGizmoCopy) always save datasets as one chunk as does HDF5 default compression. HDF5SaveDataHook allows you to change that, though in most cases there is no need to.

If the operation field is non-zero, then the hook function is being called from the HDF5SaveData or HDF5SaveGroup operations. You should respect the compression settings specified to those operations unless there is good reason to override them.

To set the compression parameters for the specified wave, set the output fields and return 1. To allow the save to proceed without altering compression, return 0.

Here is an example:

```
ThreadSafe Function HDF5SaveDataHook(s)
   STRUCT HDF5SaveDataHookStruct& s
   // Print "Global ThreadSafe HDF5SaveDataHook called"// For debugging only
   // return 0             // Uncomment this to make this hook a NOP

   if (s.version >= 2000)  // Structure version is 1000 as of this writing
      return 0             // The structure changed in an incompatible way
   endif

   if (s.operation != 0)   // Called from HDF5SaveData or HDF5SaveGroup?
      return 0             // Respect their settings unless we have
   endif                   // good reason to change them

   // Set compression parameters only for numeric waves
   int isNumeric = WaveType(s.waveToSave) == 1
   if (!isNumeric)
      return 0
   endif

   // Set compression parameters only if the total number of elements
   // in the wave exceeds a particular threshold
   int numElements = numpnts(s.waveToSave)
   if (numElements < 10000)
      return 0
   endif

   // Set compression parameters
   s.gzipLevel = 5         // 0 (no compression) to 9 (max compression)
   s.shuffle = 0           // 0=shuffle off, 1=shuffle on
   s.chunkSizes[0] = 200   // These values are arbitrary. Igor clips them.
   s.chunkSizes[1] = 200   // In a real function you would have some more
   s.chunkSizes[2] = 50    // systematic way to choose them.
   s.chunkSizes[3] = 50

   // Indicate that we want to set compression parameters for this wave
   return 1
End
```

The HDF5SaveDataHook function must be threadsafe. If it is not threadsafe, Igor will not call it.

The HDF5SaveDataHook function is called when a dataset is created and not when appending to an existing dataset.

The HDF5SaveDataHook function must not alter the state of Igor. That is, it must have no side effects such as creating waves or variables, modifying waves or variables, or killing waves or variables. It is called when HDF5SaveData is called, when HDF5SaveGroup is called, and also when Igor is saving an HDF5 packed experiment file. If you change the state of Igor in your HDF5SaveDataHook function, this may cause file corruption or a crash.

It is possible but not recommended to define more than one HDF5SaveDataHook function. For example, you can define one in the ProcGlobal context, another in a regular module and a third in an independent module. If you do this, each function is called until one returns 1. The order in which the functions are called is not defined.

We recommend that you define your HDF5SaveDataHook function in an independent module so that it can be called when normal procedures are in an uncompiled state. Otherwise, if you save an HDF5 packed experiment while procedures are not compiled, for example because of an error, your HDF5SaveDataHook function will not be called and all datasets will be saved with default compression or uncompressed.

For testing purposes, you can prevent Igor from calling any HDF5SaveDataHook function by executing:

```
SetIgorOption HDF5SaveDataHook=0
```

This prevents Igor from calling the hook function via HDF5SaveData and HDF5SaveGroup operations and when saving an HDF5 packed experiment file.

You can re-enable calling the hook function by executing:

```
SetIgorOption HDF5SaveDataHook=1
```

HDF5 Compression References

This section lists documents that discuss HDF5 filtering and compression that may be of interest to advanced HDF5 users.

Using Compression in HDF5

Chunking in HDF5

HDF5 Advanced Topics:Chunking in HDF5

Dataset Chunking Issues

HDF5 Compression Demystified #1

HDF5 Compression Demystified #2

Improving I/O Performance When Working with HDF5 Compressed Datasets

HDF5 Compression Troubleshooting

HDF5 Dynamically Loaded Filters

The HDF5 library can use third-party dynamically loaded filter plugins which are used for forms of compression that are not built into the library itself. "Dynamically loaded" means that the filter plugins are not compiled into the HDF5 library but reside in separate library files. The HDF5 library looks for these plugins and, if it finds them, loads them, and their features become available. (For HDF5 experts, dynamically loaded filter plugins are described at https://portal.hdfgroup.org/display/HDF5/HDF5+Dynamically+Loaded+Filters.)

The default locations where the HDF5 libraries look for filter plugins are:

Macintosh: `/usr/local/hdf5/lib/plugin`

Windows: `%ALLUSERSPROFILE%/hdf5/lib/plugin`
(%ALLUSERSPROFILE% is `C:\ProgramData` on most systems)

The user can override the default locations by setting the HDF5_PLUGIN_PATH environment variable to the path to the user's plugins prior to launching Igor.

Starting with Igor Pro 9.01, Igor Pro ships with the plugins provided by The HDF Group at https://www.hdfgroup.org/downloads/hdf5. These plugins include:

BLOSC, BSHUF, BZ2, JPEG, LZ4, LZF, and ZFP

Igor supports decoding datasets written with these filters. It does not yet support encoding with these filters. Also, this is supported with the 64 bit version of Igor, not with the 32 bit version because we have not found 32 bit filter plugin libraries.

On Macintosh, the filter libraries as provided by The HDF Group's download page (https://www.hdfgroup.org/downloads/hdf5) do not work with Igor Pro. The filter libraries that ship with Igor are tweaked to allow them to work with Igor Pro. (For Macintosh programming experts, this is explained at https://forum.hdfgroup.org/t/dynamically-loaded-filters-on-mac-os/9159/2.)

On Macintosh, the filters are shipped in "Igor64.app/MacOS/hdf5plugins". On Windows they are shipped in "IgorBinaries_x64/hdf5plugins". When Igor starts, it tells the HDF5 library to look in these locations first for plugins, before looking in the default filter location or in the locations specified by the HDF5_PLUGIN_PATH environment variable if it exists. This means that the HDF5 library will find the filter plugins that ship with Igor, not those in other locations.

There are other "registered filtered plugins" besides the ones listed above. A comprehensive list can be found at https://portal.hdfgroup.org/display/support/Registered+Filter+Plugins. It is unlikely that you will need other filter plugins. If you do, you can put them in the default location or the location specified by the HDF5_PLUGIN_PATH environment variable. The HDF5 library will look for them there after not finding them in Igor's hdf5plugins folder.

As noted above, filter plugin libraries on Macintosh need to be tweaked to work with Igor. Consequently, on Macintosh, other filter plugins will probably not work no matter where you put them.

HDF5 Dimension Scales

The HDF5 library supports dimension scales through the H5DS API. A dimension scale is a dataset that provides coordinates for another dataset. A dimension scale for dimension 0 of a dataset is analogous to an X wave in an Igor XY pair. The analogy applies to higher dimensions also.

Dimension scales are primarily of use in connection with the netCDF-4 format which is based on HDF5. Most Igor users do not need to know about dimension scales.

In the netCDF file format, the term "variable" is used like the term "dataset" in HDF5. Each variable of dimension n is associated with n named dimensions. Another variable, called a "coordinate variable", can supply values for the indices of a dimension, like an X wave supplies values for an XY pair in Igor.

The association between a variable and its coordinate variables is established when the variable is created. A given coordinate variable typically provides values for a given dimension of multiple variables. For example, a netCDF file may define coordinate variables named "latitude" and "longitude" and multiple 2D variables (images) whose X and Y dimensions are associated with "latitude" and "longitude".

The netCDF-4 file format is implemented using HDF5. The netCDF library uses the HDF5 dimension scale feature to implement coordinate variables and their associations with variables.

HDF5 Dimension Scale Support in Igor

The HDF5DimensionScale operation supports the creation and querying of HDF5 dimension scales. It was added in Igor Pro 9.00.

The operation is implemented using keyword=value syntax for setting parameters and simple keywords without values for invoking an action. For example, these commands convert a particular dataset into a dimension scale and then attach that dimension scale to another dataset:

```
// Convert a dataset named XScale into a dimension scale with name "X"
HDF5DimensionScale dataset={fileID,"XScale"}, dimName="X", setScale

// Attach dimension scale XScale to dimension 0 of dataset Data0
HDF5DimensionScale scale={fileID,"XScale"}, dataset={fileID,"Data0"},
                                                    dimIndex=0, attachScale
```

For details, see HDF5DimensionScale in Igor’s online help.

HDF5 Dimension Scale Implementation Details

If you view netCDF-4 files using Igor's HDF5 Browser, this section will help you to understand what you see in the browser.

The H5DS (dimension scale) API, which is part of the HDF5 "high-level" library, uses attributes to designate a dataset as a dimension scale, to associate a dimension scale with other datasets, and to keep track of the associations.

The most important attributes of a dimension scale are:

CLASS	Set to "DIMENSION_SCALE" to indicate that the dataset is a dimension scale.
NAME	Name of dimension such as "X".
REFERENCE_LIST	An array of structures used to keep track of the datasets and dimensions to which the dimension scale is attached. Each element of the structure includes a reference to a dataset and a dimension index.

The most important attributes of a dataset to which a dimension scale is attached are are:

DIMENSION_LIST	A variable-length array of references used to keep track of the dimensions used by each dimension of the dataset. The array has one column for each dataset dimension. Each column has one row for each dimension scale attached to the corresponding dataset dimension. A given dataset dimension can have multiple attached dimension scales.
DIMENSION_LABELS	An 1D array containing labels for the dimensions of the dataset.

HDF5 Dimension Scale Reference

For experts, some sources of additional information on HDF5 dimension scales and related netCDF-4 features are available in Igor’s online help topic "HDF5 Dimension Scale Reference".

Other HDF5 Issues

This section is mostly of interest to advanced HDF5 users.

HDF5 String Formats

Strings can be written to HDF5 files as datasets or as attributes using several formats:

- Variable length
- Fixed length with null termination
- Fixed length with null padding
- Fixed length with space padding

In addition, strings can be marked as either ASCII or UTF-8.

Usually you do not need to know or care about the format used to write strings. However, some programs do not support all of the formats. If you attempt to load an HDF5 file written by Igor into one of those programs, you may get an error. In that event, you may be able to fix the problem by controlling the string format used to write the file. The rest of this section provides information that may help in that event.

The variable-length string format is most useful when writing a dataset or attribute containing multiple strings of different lengths. For example, when a dataset containing the two strings "ABC" and "DEFGHIJKLMNOP" is written as variable length, each string and its length is stored in the HDF5 file. That requires 3 bytes for "ABC" and 13 bytes for "DEFGHIJKLMNOP" plus the space required to store the length for each string. If these strings were written as fixed length, the fixed length for the dataset or attribute would have to be at least 13 bytes and at least 10 bytes would be wasted by padding when writing "ABC".

The fixed-length string format is most useful when writing a dataset or attribute consisting of a single string. For example, "ABC" can be written using a 3-byte fixed-length datatype with 0 padding bytes.

If more than one string is written as fixed length, the padding mode determines how extra bytes are filled. In null-terminated mode, the extra bytes are filled with null (0) and the first null marks the end of a string. In null-padded mode, the extra bytes are filled with null (0) all consecutive nulls at the end of the string mark the end of the string. In space-padded mode, the extra bytes are filled with space all consecutive spaces at the end of the string mark the end of the string.

In addition to the variable-length versus fixed-length issue, there is a text encoding issue with HDF5. A string dataset or attribute is marked as either ASCII or UTF-8 depending on the software that wrote it. The marking does not guarantee that the text is valid ASCII or valid UTF-8 as the HDF5 library does not check to make sure that written text is valid as marked nor does it do any text encoding conversions. The marking is merely a statement of the intended text encoding. Some software may fail when reading datasets or attributes marked as ASCII or UTF-8 because the HDF5 library does require that the reading program use a compatible datatype.

With some exceptions explained below, you can control the string format used to save datasets and attributes using the /STRF={*fixedLength*,*paddingMode*,*charset*} flag with the **HDF5SaveData** and **HDF5SaveGroup** operations. The /STRF flag was added in Igor Pro 9.00.

If *fixedLength* is 0, HDF5SaveData writes strings using a variable-length HDF5 string datatype. If *fixedLength* is greater than 0, HDF5SaveData writes strings using a fixed-length HDF5 string datatype of the specified length with padding specified by padding mode. If *fixedLength* is -1, HDF5SaveData determines the length of the longest string to be written for a given dataset or attribute and writes strings using a fixed-length HDF5 string datatype of that length with padding specified by *paddingMode*.

If *paddingMode* is 0, HDF5SaveData writes fixed-length strings as null terminated strings. If *paddingMode* is 1, HDF5SaveData writes fixed-length strings as null-padded strings. If *paddingMode* is 2, HDF5SaveData writes fixed-length strings as space-padded strings. When writing strings as variable length (*fixedLength*=0), *paddingMode* is ignored.

If *charset* is 0, HDF5SaveData writes strings marked as ASCII. If *charset* is 1, HDF5SaveData writes strings marked as UTF-8.

An exception is zero-length datasets or attributes which are always written as variable-length UTF-8. Another exception is string variables written by HDF5SaveGroup which are always written as fixed-length null padded UTF-8.

This table shows the default string format used for various situations if you omit /STRF and whether or not HDF5SaveData and HDF5SaveGroup honor the /STRF flag for the corresponding situation:

	HDF5SaveData Default	**HDF5SaveData Behavior**
Text Wave Zero Element as Dataset	Variable,NULLPAD,UTF-8	Ignores /STRF
Text Wave Single Element as Dataset	Variable,NULLPAD,UTF-8	Honors /STRF
Text Wave Multiple Elements as Dataset	Variable,NULLPAD,UTF-8	Honors /STRF
String Variable	N/A	HDF5SaveData can not save string variables
Text Wave Zero Element as Attribute	Variable,NULLPAD,UTF-8	Ignores /STRF
Text Wave Single Element as Attribute	Fixed,NULLPAD,UTF-8	Honors /STRF
Text Wave Multiple Elements as Attribute	Variable,NULLPAD,UTF-8	Honors /STRF

	HDF5SaveGroup Default	**HDF5SaveGroup Behavior**
Text Wave Zero Element as Dataset	Variable,NULLPAD,UTF-8	Ignores /STRF
Text Wave Single Element as Dataset	Variable,NULLPAD,UTF-8	Honors /STRF
Text Wave Multiple Elements as Dataset	Variable,NULLPAD,UTF-8	Honors /STRF
Zero-Length String Variable	Variable,NULLPAD,UTF-8	Ignores /STRF
String Variable > Zero-Length	Fixed,NULLPAD,UTF-8	Ignores /STRF

Text Wave Zero Element as Attribute	N/A	HDF5SaveGroup can not save attributes
Text Wave Single Element as Attribute	N/A	HDF5SaveGroup can not save attributes
Text Wave Multiple Elements as Attribute	N/A	HDF5SaveGroup can not save attributes

HDF5 String Variable Text Encoding

Igor writes string variables as UTF-8 fixed-length string datasets.

String variables may contain null bytes and text that is invalid as UTF-8. This would occur, for example, if a variable were used to contain binary data. Such string variables are still written as UTF-8 fixed-length string datasets.

HDF5 Wave Text Encoding

For background information on wave text encoding, see **Wave Text Encodings** on page III-472.

Igor text wave contents are written as variable-length string datasets using UTF-8 text encoding. Other wave elements (units, note, dimension labels) are written as UTF-8 fixed-length string attributes.

Text wave contents may contain null bytes and text that is invalid as UTF-8. This would occur, for example, if a text wave were used to contain binary data. Such contents are still written as UTF-8 variable-length string datasets.

Wave text elements can be marked in Igor as using any supported text encoding. No matter how a wave's text elements are marked, they are written to HDF5 as UTF-8 strings. Consequently, if you save a wave that uses non-UTF-8 text encodings to an HDF5 file and the load it back into Igor, its text encodings change but the characters represented by the text do not.

An exception applies to wave elements marked as binary (see **Text Waves Containing Binary Data** on page III-475 for background information). When you save a wave containing one or more binary elements as an HDF5 dataset, Igor adds the IGORWaveBinaryFlags attribute to the dataset. This attribute identifies the wave elements marked as binary using bits as defined for the **WaveTextEncoding** function. When you load the dataset from an HDF5 file, Igor restores the binary marking for the wave elements corresponding to the bits set in the attribute. The IGORWaveBinaryFlags attribute was added in Igor Pro 9.00.

In Igor Pro 9.00 and later, the HDF5LoadData and HDF5LoadGroup operations can check for binary data loaded into text waves from string datasets and mark such text waves as containing binary. If you load binary data from string datasets, see bit 1 of the /OPTS flag of those operations for details.

HDF5 File Paths on Windows

The HDF5 library does not support Unicode file paths on Windows. This is explained under "Filenames" at

https://support.hdfgroup.org/HDF5/doc/Advanced/UsingUnicode/index.html

Instead, on Windows the HDF5 library uses system text encoding (also known as "system locale" and "language for non-Unicode programs").

This worked fine with Igor6 because Igor6 also used system text encoding internally. However, Igor7 and later store all text internally as UTF-8. Consequently, on Windows Igor converts file paths from UTF-8 to system text encoding before passing those paths to HDF5 library routines. This allows you to use paths containing non-ASCII characters. However, you are limited to those characters that are supported by the current system text encoding on your system. For example, if your system is set to use Windows-1252 ("western" code page) as the system text encoding, then you will get an error if the path to your file contains, for example, Japanese, because Windows-1252 does not support Japanese characters.

This issue does not affect Macintosh because both Igor7 or later and the HDF5 library use UTF-8 on Macintosh.

Igor Compatibility With HDF5

This section discusses issues relating to various HDF5 library versions.

You don't need to know this information unless you are using very old software based on HDF5 library versions earlier than 1.8.0.

The following table lists various Igor and HDF5 versions:

Igor Version	**HDF5 Library Version**
Igor Pro 6.03	1.6.3 (released on 2004-09-22)
Igor Pro 6.10 to 6.37	1.8.2 (released on 2008-11-10)
Igor Pro 7.00 to 7.08	1.8.15 (released on 2015-05-04)
Igor Pro 8.00 to 8.04	1.10.1 (released on 2017-04-27)
Igor Pro 9.00 to 9.0x	1.10.7 (released on 2020-09-15)

HDF5 Compatibility Terminology

For the purposes of this discussion, the term "old HDF5 version" means a version of the HDF5 library earlier than 1.8.0. The term "old HDF5 program" means a program that uses an old HDF5 version. The term "old HDF5 file" means an HDF5 file written by an old HDF5 program.

Reading Igor HDF5 Files With Old HDF5 Programs

By default, Igor Pro 9 and later create HDF5 files that work with programs compiled with HDF5 library version 1.8.0 or later. HDF5 1.8.0 was released in February of 2008. For most uses, this default compatibility will work fine.

The rest of this section is of interest only if you need to read Igor HDF5 files using very old HDF5 programs which use HDF5 library versions earlier than 1.8.0.

As explained at https://support.hdfgroup.org/HDF5/faq/bkfwd-compat.html#162unable, software compiled with HDF5 1.6.2 (released on 2004-02-12) or before is incompatible with HDF5 files produced by HDF5 1.8.0 or later.

If you need to read files written by Igor Pro 9 and later using software that uses HDF5 1.6.3 (released on 2004-09-22) through 1.6.10 (released on 2009-11-10), you need to tell Igor to use compatible HDF5 formats. You do this by executing this command:

```
SetIgorOption HDF5LibVerLowBound=0
```

When you do this, you lose these features:

- The ability to save attributes larger than 65,535 bytes ("large attributes")

  You will get an error in the unlikely event that you attempt to write a large attribute to a group or dataset.

  This includes saving a wave with a very large wave note or with a very large number of dimension labels via HDF5SaveData or HDF5SaveGroup or by saving an HDF5 packed experiment file. It also includes adding a large attribute using HDF5SaveData/A.
- The ability to sort groups and datasets by creation order

  This feature is provided by the HDF5 browser but is supported only for files in which it is enabled when the file is created.
- The ability to write attributes so that they can be read in creation order

To restore Igor to normal operation, execute:

```
SetIgorOption HDF5LibVerLowBound=1
```

The effect of SetIgorOption lasts only until you restart Igor.

Typically, if you need to set HDF5LibVerLowBound at all, you would do this once at startup. Do not call SetIgorOption to set HDF5LibVerLowBound while an Igor preemptive thread is making HDF5 calls.

Writing to Old HDF5 Files

The HDF5 documentation does not spell out all of the myriad compatibility issues between various HDF5 library versions. The information in this section is based on our empirical testing.

See **HDF5 Compatibility Terminology** on page II-222 for a definition of terms used in this section.

If you use Igor to open an old HDF5 file and write a dataset to it, the new dataset is not readable by old HDF5 programs. Also, the group containing the new dataset can not be listed by old HDF5 programs. In other words, writing to an old file in new format makes the old file at least partially unreadable by old software.

If you have old HDF5 files and you rely on old HDF5 programs to read them, open such files for read only, not for read/write. Also make sure the files are thoroughly backed up.

HDF5 Packed Experiment Files

In Igor Pro 9 and later, you can save an Igor experiment in an HDF5 file. The main advantage is that the data is immediately accessible to a wide array of programs that support HDF5. The main disadvantage is that you will need Igor Pro 9 or later to open the file in Igor. Also HDF5 is considerably slower than PXP for experiments with very large numbers of waves.

To save an experiment in HDF5 packed experiment format, choose File→Save Experiment As. In the resulting Save File dialog, choose "HDF5 Packed Experiment Files (*.h5xp)" from the pop-up menu under the list of files. Then click Save.

If you want to make HDF5 packed experiment format your default format for saving new experiments, choose Misc→Miscellaneous Settings. In the resulting dialog, click the Experiment category on the left. Then choose "HDF5 Packed (.h5xp)" from the Default Experiment Format pop-up menu.

For normal use you don't need to know the details of how Igor stores data in HDF5 packed experiment files, but, if you are curious, you can get a sense by opening such a file using the HDF5 Browser (Data→Load Waves→New HDF5 Browser).

The rest of this topic is for the benefit of programmers who want to read or write HDF5 packed experiment files from other programs. It is assumed that you are an experienced Igor and HDF5 user.

HDF5 Packed Experiment File Organization

An HDF5 packed experiment file has the following general organization:

```
/
   History
      History                 // Contents of the history area window
   Packed Data
      <Waves, variables and data folders here>
   Shared Data
      <Paths to shared wave files here>
   Free Waves
      <Free waves here>
   Free Data Folders
      <Free data folders here>
   Packed Procedure Files
      Procedure              // Contents of the built-in procedure window
      <Zero or more packed procedure files>
   Shared Procedure Files
      <Paths to shared procedure files>
   Packed Notebooks
      <Packed notebook files>
   Shared Notebooks
      <Paths to shared notebook files>
   Symbolic Paths
      <Paths to folders associated with symbolic paths>
   Recreation
      Recreation Procedures   // Experiment recreation procedures
   Miscellaneous
         Dash Settings (dataset)
         Recent Windows Settings (dataset)
         Running XOPs (dataset)
         Pictures
            <Picture datasets here>
         Page Setups
            Page Setup for All Graphs (dataset)
            Page Setup for All Tables (dataset)
            Page Setup for Built-in Procedure Window (dataset)
            <Datasets for page layout page setups here>
         XOP Settings
            <XOP settings datasets here>
```

Any of the top-level groups can be omitted if it is not needed. For example, if the experiment has no free waves, Igor writes no Free Waves group.

The Shared Data group contains datasets containing full paths to Igor binary wave (.ibw) files referenced by the experiment. The paths are expressed as Posix paths on Macintosh, Windows paths on Windows.

The Shared Procedure Files group contains datasets containing full paths to procedure (.ipf) files referenced by the experiment. The paths are expressed as Posix paths on Macintosh, Windows paths on Windows. Global procedure files and #included procedure files are not part of the expeirment and are not included.

The Shared Notebooks group contains datasets containing full paths to notebook (.ifn or any plain text file type) files referenced by the experiment. The paths are expressed as Posix paths on Macintosh, Windows paths on Windows.

The Symbolic Paths group includes only user-created symbolic paths, not the built-in symbolic paths Igor, IgorUserFiles, or home which by definition points to the folder containing the packed experiment file. The paths are expressed as Posix paths on Macintosh, Windows paths on Windows. If there are no user-created symbolic paths, Igor writes no Symbolic Paths group.

The Recreation Procedures dataset contains the experiment recreation procedures written by Igor to recreate the experiment.

The Miscellaneous group contains datasets and subgroups. Any of these objects can be omitted if not needed. The format of the data in the Miscellaneous group is subject to change and not documented. If your program reads or writes HDF5 packed experiment files, we recommend that you not attempt to read or write these objects.

HDF5 Packed Experiment File Hierarchy

An HDF5 file is a "directed graph" rather than a strict hierarchy. This means that it is possible to create strange relationships between objects, such as a group being a child of itself, or a dataset being a child of more than one group. Such strange relationships are not allowed for HDF5 packed experiment files - they must constitute a strict hierarchy.

Storage of Plain Text in HDF5 Packed Experiment Files

Several kinds of plain text items are stored in experiment files, including:

- The contents of the built-in procedure window
- The contents of the built-in history window
- The contents of packed procedure files
- The contents of packed plain text notebook files
- Text data contents of text waves
- Text properties of numeric waves (e.g., units, wave note, dimension labels)
- String variables

Igor stores such text as strings (HDF5 class H5T_STRING) using UTF-8 text encoding (H5T_CSET_UTF8) when writing an HDF5 packed experiment file.

Occasionally an item that is expected to be valid UTF-8 will contain byte sequences that are invalid in UTF-8 or contain null bytes. Examples include text waves and string variables that are being used to carry binary rather than text data. Such data is still written as UTF-8 string datasets.

Plain Text Files

"Plain text files" in this context refers to history text, recreation procedures, procedure files, and plain text notebooks. Igor writes plain text files as UTF-8 string datasets.

Waves

Igor writes text waves as UTF-8 string datasets. For details, see **HDF5 Wave Text Encoding** on page II-221.

String Variables

Igor writes string variables as UTF-8 fixed-length string datasets. For details, see **HDF5 String Variable Text Encoding** on page II-221.

Writing HDF5 Packed Experiments

This section is for programmers who want to write HDF5 packed experiment files from programs other than Igor. This section assumes that you understand the information presented in the preceding sections of **HDF5 Packed Experiment Files** on page II-223.

If you open an HDF5 packed experiment using the HDF5 Browser you will see that the file (top-level group displayed as "root" in the HDF5 file) has a number of attributes. The only required attribute is the IGORRequiredVersion attribute which specifies the minimum version of Igor required to open the experiment. Write 9.00 for this attribute, or a larger value if your HDF5 packed experiment requires a later version of Igor.

Waves in HDF5 Files

Igor writes waves to HDF5 files as datasets with attributes representing wave properties. For example, the IGORWaveType attribute represents the data type of the wave - see **WaveType** for a list of data types. See the discussion of the /IGOR flag of the **HDF5SaveData** operation for a list of HDF5 wave attributes.

Wave Reference Waves and Data Folder Reference Waves

A wave reference wave (see **Wave Reference Waves** on page IV-77), or "wave wave" for short, is a wave whose elements are references to other waves. Each wave referenced by a wave wave has an IGORWaveID attribute that specifies the wave ID of the referenced wave. The contents of each element of a wave wave is the wave ID for the referenced wave. A wave ID of zero indicates a null wave reference. On loading the experiment, Igor restore each element of each wave wave so that it points to the appropriate wave.

A data folder reference wave (see **Data Folder Reference Waves** on page IV-82), or "DFREF wave" for short, is a wave whose elements are references to data folders. Each data folder referenced by a DFREF wave has an IGORDataFolderID attribute that specifies the data folder ID of the referenced data folder. The contents of each element of a DFREF wave is the data folder ID for the referenced data folder. A data folder ID of zero indicates a null data folder reference. On loading the experiment, Igor restore each element of each DFREF wave so that it points to the appropriate data folder.

Writing and restoring wave waves and DFREF waves is complicated and tricky. If your program writes HDF5 packed experiment files, we recommend that you not attempt to write these objects. If your program loads HDF5 packed experiment files, we recommend that you ignore ignore wave waves (IGORWaveType=16384) and DFREF waves (IGORWaveType=256).

Free Waves and Free Data Folders

Igor writes representations of **Free Waves** and **Free Data Folders** to HDF5 packed experiment files and restores free waves and free data folders when the experiment is loaded.

Each free wave is written as a dataset in the Free Waves group. For a free wave to exist, it must be referenced by a wave wave in the experiment.

Each free root data folder and its descendents are written as a group and subgroups in the Free Data Folders group. For a free root data folder to exist, it must be referenced by a DFREF wave in the experiment.

Writing and restoring free waves and free data folders is a complicated and tricky. If your program writes HDF5 packed experiment files, we recommend that you not attempt to write these objects. If your program loads HDF5 packed experiment files, we recommend that you ignore the Free Waves and Free Data Folders groups.

HDF5 Packed Experiment Issues

In rare cases, Igor experiments can not be written in HDF5 packed format because of name conflicts. For details, see **Object Name Conflicts and HDF5 Files** on page III-505.

Chapter

II-11

Dialog Features

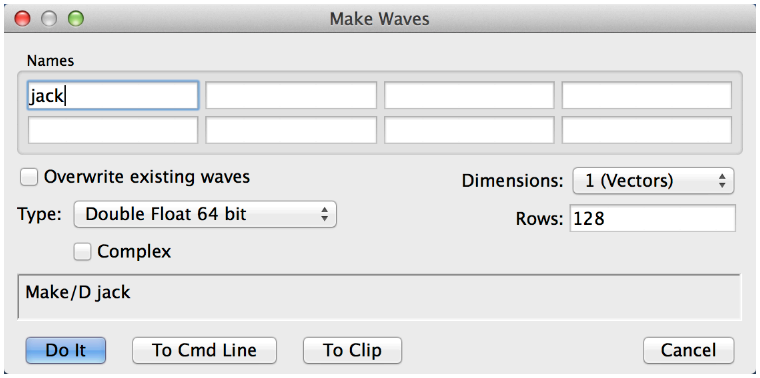
Overview

Most Igor Pro dialogs are designed with common features. This chapter describes some of those common features.

Operation Dialogs

Menus and dialogs provide easy access to many of Igor Pro's operations.

When you choose a menu item, like Data-Make waves, Igor presents a dialog:



Names: jack

□ Overwrite existing waves

Dimensions: 1 (Vectors)

Type: Double Float 64 bit

Rows: 128

□ Complex

```
Make/D jack
```

As you click and type in the items in the dialog, Igor generates an appropriate command. The command being generated is displayed in the command box near the bottom of the dialog.

As you become more proficient, you will find that some commands are easier to invoke from a dialog and others are easier to enter directly in the command line. There are some menus and dialogs that bypass the command line, usually because they perform functions that have no command line equivalents.

Dialog Wave Browser

In dialogs in which a wave must be selected, Igor presents a list of suitable waves in a dialog wave browser. The browser also lets you navigate through the data folder hierarchy.

Double-click a data folder icon to make that data folder the top level for the hierarchical display. To return to levels closer to the root, use the menu at the top of the browser.

In dialogs that support selection of multiple items, you can drag the mouse over items to select more than one. Press Shift while clicking to add additional contiguous items to a selection. Press Command (*Macintosh*) or Ctrl (*Windows*) while clicking to add additional discontiguous items to a selection.

You can change the columns that are displayed in the browser by right-clicking the header and enabling or disabling the different columns. If you click on a column in the header area, the waves in the browser are sorted by that column. Click the column in the header area again to reverse the sort direction.

After toggling the display of one or more columns, you may wish to right-click the header again and choose Size All Columns to Fit. You can also resize the columns by dragging the vertical bar that separates columns.

Click the gear icon at the bottom of the browser to display the options menu. It allows you to control whether data folders are displayed and how the waves are sorted and grouped.

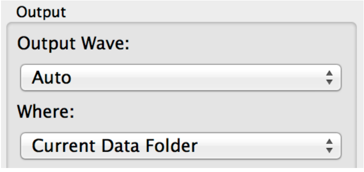
To filter waves by name, type a name in the Filter edit box. Only waves whose name match the filter string are displayed. The filtering algorithm supports the following features:

?	Matches any single character.
*	Matches zero or more of any characters. For example, "w*3" matches wave3, wave30 and wave300.
[...]	Matches a single character if it is included in the set of characters specified in square brackets. For example, [A-Z] matches any character from A to Z, case-insensitive. [0-9] matches any single digit.

Click the question mark icon at the bottom right corner of the browser to display a tooltip containing information about the current filter. This tip shows you the filtering criteria currently in place and may help you to figure out why waves you expect to be able to select are not displayed in the browser.

Operation Result Chooser

In most Igor dialogs that perform numeric operations (Analysis menu: Integrate, Smooth, FFT, etc.) there is a group of controls allowing you to choose what to do with the result. Here is what the Result Chooser looks like in the Integrate dialog:



Output

Output Wave: Auto

Where: Current Data Folder

The Output Wave menu offers choices of a wave to receive the result of the operation:

Auto	Igor will create a new wave to receive the results. The source wave is not changed. The new wave will have a name derived from the source wave by adding a suffix that depends on the operation. choosing Auto makes the Where menu available.
Overwrite Source	The source wave (the wave that contains the input data) will be overwritten with the results of the operation. This will destroy the original data. The Where menu will not be available.
Make New Wave	This is like the Auto choice, but an edit box is presented that you use to type a name of your own choosing. Igor will make a new wave with this name to receive the results of the operation. This selection makes the Where menu available.
Select Existing Wave	A Wave Browser will be presented allowing you to choose any existing wave to be overwritten with the results. This choice preserves the contents of the source wave, but destroys the contents of the wave chosen to receive the results.

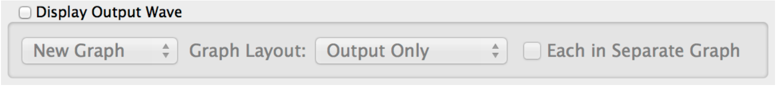
The Where menu offers choices for the location of a new wave created when you choose Auto or Make New Wave. Usually you will want to choose Current Data Folder.

Current Data Folder	The new wave is created in the current data folder. If you don't know about data folders, this is probably the best choice.
Source Wave Data Folder	The new wave is created in the same data folder as the source wave. It is quite likely that the source wave will be in the current data folder, in which case this choice is the same as choosing Current Data Folder.
Select Data Folder	This choice presents a Wave Browser in which you can choose a data folder where the new wave will be created.

Operation Result Displayer

In some Igor dialogs that perform numeric operations (Analysis menu: Integrate, Smooth, FFT, etc.) there is a group of controls allowing you to choose how to display the result. Choices are offered to put the result into the top graph, a new graph, the top table, or a new table. For two-dimensional results, New Image and New Contour are also offered. If the result is complex, as is the case for an FFT, New Contour is not available.

Here is what the Result Displayer looks like in the Smooth dialog:

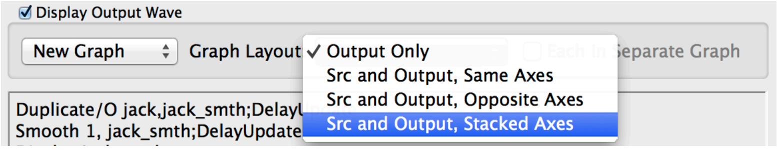


☐ Display Output Wave

New Graph   Graph Layout: Output Only   ☐ Each in Separate Graph

The contents of the displayer are not available here because the Display Output Wave checkbox is not selected. This is the default state.

When you choose New Graph, there are four choices in the Graph menu for the contents and layout of the new graph. In this menu, *Src* stands for Source. It is the wave containing the input data; Output is the wave containing the result of the operation.



☑ Display Output Wave

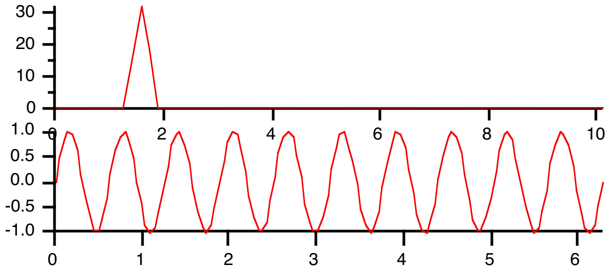
New Graph   Graph Layout

✓ Output Only
Src and Output, Same Axes
Src and Output, Opposite Axes
Src and Output, Stacked Axes
Each in Separate Graph

```
Duplicate/O jack,jack_smth;DelayU
Smooth 1, jack_smth;DelayUpdate
```

In many cases, the second choice, Src and Output, Same Axes, will not be appropriate because the operation changes the magnitude of data values or the range of the X values.

This picture shows the result of an FFT operation when Src and Output, Stacked Axes is chosen:

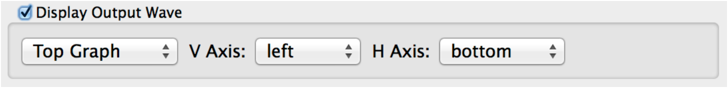


When you choose New Image or New Contour to display matrix results, the Graph Layout menu allows only Output Only or Src and Output, Stacked Axes. The axes aren't really stacked - it makes side-by-side graphs. It makes little sense to put two images or two contours on one set of axes.

The Result Displayer doesn't give you many options for formatting the graph, and doesn't allow any control over trace style, placement of axes, etc. It is intended to be a convenient way to get started with a graph. You can then modify the graph in any way you choose.

If you want a more complex graph, you may need to use the New Graph dialog (choose New Graph from the Windows menu) after you have clicked Do It in an operation dialog.

If you choose Top Graph instead of New Graph, the output wave will be appended to the top graph. It is assumed that this graph will already contain the source wave, so there is no option to append the source wave to the top graph. The Graph layout menu disappears, and two menus are presented to let you choose axes for the new wave:



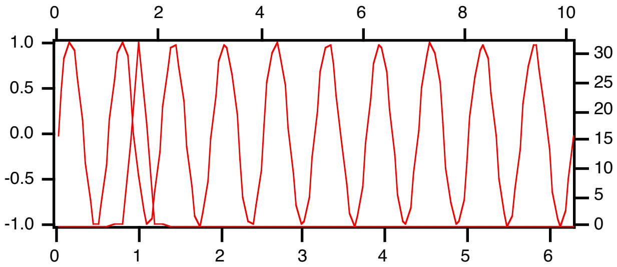
☑ Display Output Wave

Top Graph V Axis: left H Axis: bottom

The menus allow you to choose the standard axes: left and right in the V Axis menu; top and bottom in the H Axis menu. If the top graph includes any free axes (axes you defined yourself) they will be listed in the appropriate menu as well.

In most cases the source wave will be plotted on the left and bottom axes. You will usually want to select the right axis because of the differing magnitude of data values that result from most operations. You may also want to select the top axis if the operation (like the FFT) changes the X range as well.

Here is the result of choosing right and top when doing an FFT (this is the same input data as in the graph above):



Note that the format of the graph is poor. We leave it to you to format it as you wish. If you want a stacked graph, it may be better to choose the New Graph option.

Chapter

II-12

Tables

Overview

Tables are useful for entering, modifying, and inspecting waves. You can also use a table for presentation purposes by exporting it to another program as a picture or by including it in a page layout. However, it is not optimized for this purpose.

If your data has a small number of points you will probably find it most convenient to manually enter it in a table. In this case, creating a new empty table will be your first step.

If your data has a large number of points you will most likely load it into Igor from a file. In this case it is not necessary to make a table. However, you may want to display the waves in a table to inspect them. Igor Pro tables can handle virtually any number of rows and columns provided you have sufficient memory.

A table in Igor is similar to but not identical to a spreadsheet in other graphing programs. The main difference is that in Igor data exists independent of the table. You can create new waves in Igor's memory by entering data in a table. Once you have entered your data, you may, if you wish, kill the table. The waves exist independently in memory so killing the table does not kill the waves. You can still display them in a graph or in a new table.

In a spreadsheet, you can create a formula that makes one cell dependent on another. You can not create cell-based dependencies in Igor. You can create dependencies that control entire waves using Analysis→Compose Expression, Misc→Object Status, or the **SetFormula** operation (see page V-847), but this is not recommended for routine work.

To make a table, choose Windows→New Table.

When the active window is a table, the Table menu appears in Igor's menu bar. Items in this menu allow you to append and remove columns, change the formatting of columns, and sets table preferences.

Waves in tables are updated dynamically. Whenever the values in a wave change, Igor automatically updates any tables containing that wave. Because of this, tables are often useful for troubleshooting number-crunching procedures.

Creating Tables

Table Creation with New Experiment

By default, when you create a new experiment, Igor automatically creates a new, empty table. This is convenient if you generally start working by entering data manually. However, in Igor data can exist in memory without being displayed in a table. If you wish, you can turn automatic table creation off using the Experiment Settings category of the Miscellaneous Settings dialog (Misc menu).

Creating an Empty Table for Entering New Waves

Choose Windows→New Table and click the Do It button.

If you enter a numeric value in the table, Igor creates a numeric wave. If you enter a non-numeric value, Igor creates a text wave.

To create multidimensional waves you must use the Make Waves dialog (Data menu).

After creating the wave, you may want to rename it. Choose Rename from the Table pop-up menu or from the Data menu in the main menu bar.

Creating a Table to Edit Existing Waves

Choose Windows→New Table.

Select the waves to appear in the table from the Columns to Edit list. Press Shift while clicking to select a range of waves. Press Cmd (*Macintosh*) or Ctrl (*Windows*) to select an individual list item.

Click the Do It button to create the table.

Showing Index Values

As described in Chapter II-5, **Waves**, waves have built-in scaled index values. The New Table and Append to Table dialogs allow you to display just the data in the wave or the index values and the data. If you click the Edit Index And Data Columns radio button in either of these dialogs, Igor displays both index and data columns in the table.

A 1D wave's X index values are determined by its X scaling which is a property that you set using the Change Wave Scaling dialog or SetScale operation. A 2D wave has X and Y scaling, controlling X and Y scaled index values. Higher dimension waves have additional scaling properties and scaled index values. Displaying index values in a table is of use mostly if you are not sure what a wave's scaling is or if you want to see the effect of a SetScale operation.

Showing Dimension Labels

As described in **Dimension Labels** on page II-93, waves have dimension labels. The New Table and Append Columns to Table dialogs allow you to display just the data in the wave or the dimension labels and the data. If you click the Edit Dimension Label And Data Columns radio button in either of these dialogs, Igor displays both label and data columns in the table.

Dimension labels are of use only when individual rows or columns of data have distinct meanings. In an image, for example, this is not the case because the significance of one row or column is the same as any other row or column. It is the case when a multidimensional wave is really a collection of related but disparate data.

A 2D wave has row and column dimension labels. A 1D wave has row dimension labels only.

You can display dimension labels or dimension indices in a table, but you can not display both at the same time for the same wave.

The Horizontal Index Row

When a multidimensional wave is displayed in a table, Igor adds the horizontal index row, which appears below the column names and above the data cells. This row can display numeric dimension indices or textual dimension labels.

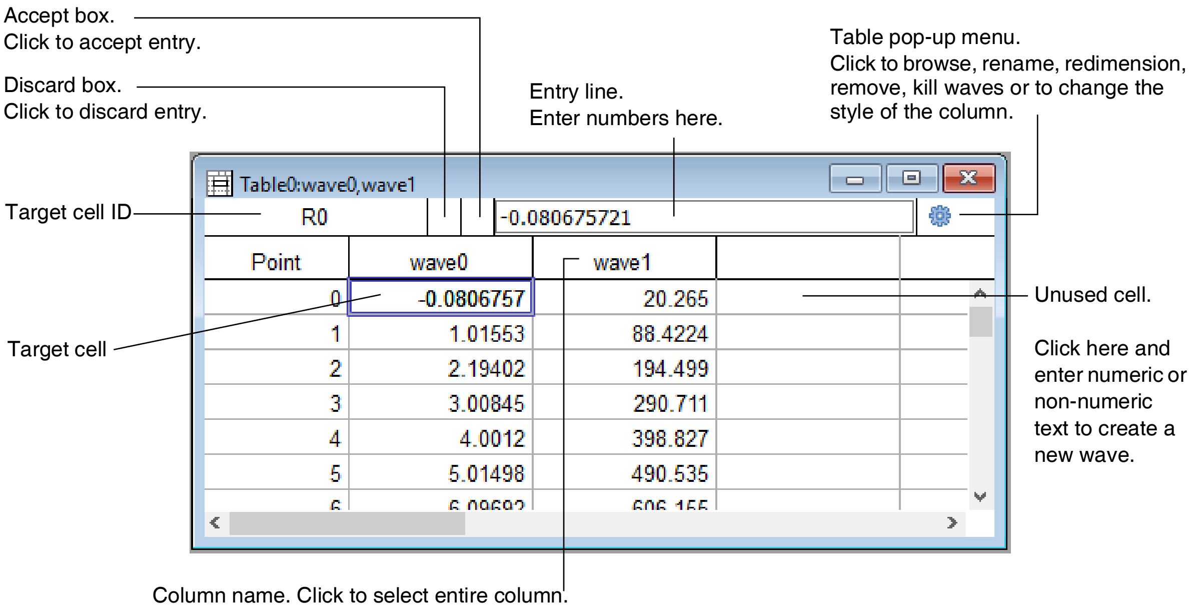
By default, the horizontal index row displays dimension labels if the wave's dimension label column is displayed in the table. Otherwise it displays numeric dimension indices. You can override this default using the Table→Horizontal Index submenu.

Creating a Table While Loading Waves From a File

The Load Waves dialog (Data menu) has an option to create a table to show the newly loaded waves.

Parts of a Table

This diagram shows the parts of a table displaying 1D waves. If you display multidimensional waves, Igor adds some additional items to the table, described under **Editing Multidimensional Waves** on page II-261.



Accept box.
Click to accept entry.

Discard box.
Click to discard entry.

Entry line.
Enter numbers here.

Table pop-up menu.
Click to browse, rename, redimension, remove, kill waves or to change the style of the column.

Target cell ID

Target cell

Unused cell.

Click here and enter numeric or non-numeric text to create a new wave.

Column name. Click to select entire column.

Table0:wave0,wave1

R0 -0.080675721

Point	wave0	wave1
0	-0.0806757	20.265
1	1.01553	88.4224
2	2.19402	194.499
3	3.00845	290.711
4	4.0012	398.827
5	5.01498	490.535
6	6.09692	606.155

The bulk of a table is the **cell area**. The cell area contains columns of numeric or text data values as well as the column of point numbers on the left. If you wish, it can also display index columns or dimension label columns. To the right are unused columns into which you can type or paste new data.

If the table displays multidimensional waves then it will include a row of column indices or dimension labels below the row of names. Use the Append Columns to Table dialog to switch between the indices and labels.

In the top left corner is the **target cell ID** area. This identifies a wave element corresponding to the target cell. For example, if a table displays a 2D wave, the ID area might show “R13 C22”, meaning that the target cell is on row 13, column 22 of the 2D wave. For 3D waves the target cell ID includes the layer (“L”) and for a 4D wave it includes the chunk (“Ch”).

If you scroll the target cell out of view you can quickly bring it back into view by clicking in the target cell ID

There is a special cell, called the **insertion cell**, at the bottom of each column of data values. You can add points to a wave by entering a value or pasting in the insertion cell.

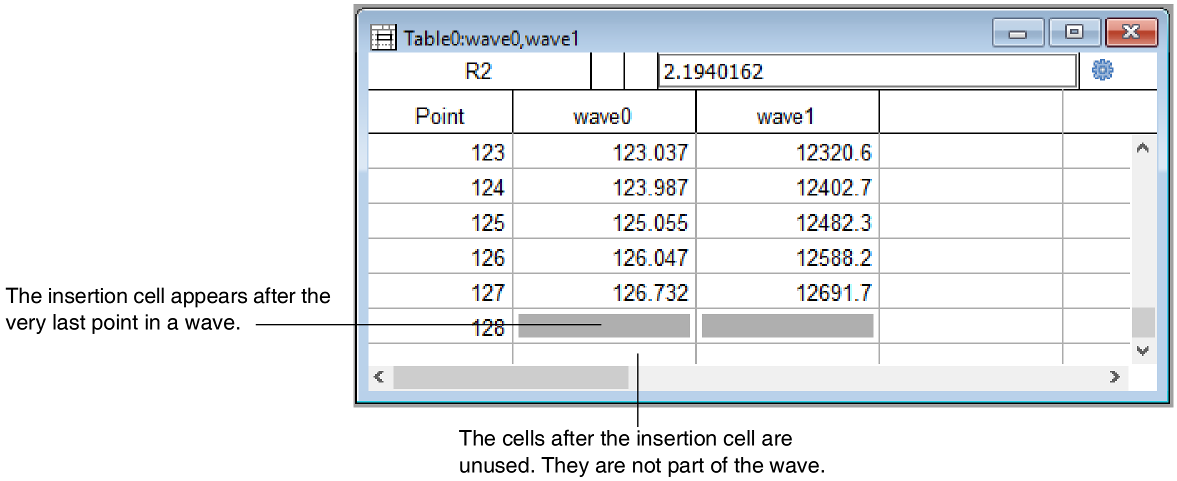


Table0:wave0,wave1

R2 2.1940162

Point	wave0	wave1
123	123.037	12320.6
124	123.987	12402.7
125	125.055	12482.3
126	126.047	12588.2
127	126.732	12691.7
128		

The insertion cell appears after the very last point in a wave.

The cells after the insertion cell are unused. They are not part of the wave.

The **Table pop-up menu** provides a quick way to inspect or change a wave, remove or kill a wave and change the formatting of one or more columns. You can invoke the Table pop-up menu by clicking the gear icon or right-clicking (*Windows*) or Control-clicking (*Macintosh*) a column.

For waves displayed in multiple columns (complex waves and multidimensional waves), if you change the display format of any data column from the wave, Igor changes the format for all data columns from that wave.

One of the items in the pop-up menu is Delay Update. Normally, when you change the value of a cell in the table, Igor immediately updates any other tables or graphs to reflect the new value. Enabling Delay Update forces this updating of other tables and graphs to be postponed until you click in another window or disable Delay Update. When Delay Update has been enabled, there is a checkmark next to the menu item. You can do this if you have a list of values to enter into a table and you don't want other tables or graphs to be updated until you are finished.

Delay Update does not delay updates when you remove or add cells to a wave. It only delays updates when you change the value of a cell.

Showing and Hiding Parts of a Table

The Table menu has a Show submenu that shows or hides various parts of a table. This is of use only in specialized situations such as when you are using a table subwindow in a control panel to display data but don't want the user to enter data. For normal use you should leave all of the items in the Show submenu checked so that all parts of the table will be visible.

When the entry line is hidden, the user can not change values in the table.

Arrow Keys in Tables

By default, if you are in the process of entering data, the arrow keys accept the entry as if you pressed Enter and then move the selected cell.

Some users prefer to use the arrow keys to move the selection in the entry line when an entry is in progress. You can specify your preference via the Table Settings category in the Miscellaneous Settings dialog (Misc menu).

Table Keyboard Navigation

The term "keyboard navigation" refers to selection and scrolling actions in response to the arrow keys and to the Home, End, Page Up, and Page Down keys. Macintosh and Windows have different conventions for these actions in windows containing text. You can use either Macintosh or Windows conventions on either platform.

By default, Macintosh conventions apply on Macintosh and Windows conventions apply on Windows. You can change this using the Keyboard Navigation menu in the Misc Settings section of the Miscellaneous Settings Dialog. If you use Macintosh conventions on Windows, use Ctrl in place of Command. If you use Windows conventions on Macintosh, use Command in place of Ctrl.

Macintosh Table Navigation

Key	No Modifier	Option	Command
Left Arrow	Move selection left one cell	Not used	Move selection to first column
Right Arrow	Move selection right one cell	Not used	Move selection to last column
Up Arrow	Move selection up one cell	Show previous layer	Scroll and move selection to first row
Down Arrow	Move selection down one cell	Show next layer	Scroll and move selection to last row
Home	Scroll to start of document	Scroll to start of document	Scroll to start of document
End	Scroll to end of document	Scroll to end of document	Scroll to end of document

Macintosh Table Navigation

Key	No Modifier	Option	Command
Page Up	Scroll up one screen	Scroll left one screen	Scroll up one screen
Page Down	Scroll down one screen	Scroll right one screen	Scroll down one screen

When viewing a 3D or 4D wave, Option-Up Arrow and Option-Down Arrow change the currently viewed layer.

When viewing a 4D wave, Command-Option-Up Arrow and Command-Option-Down Arrow change the currently viewed chunk.

Pressing shift-arrow-key extends the selection in the direction of the arrow key. Pressing cmd-shift-arrow-key extends selection as far as possible in the direction of the arrow key.

Windows Table Navigation

Key	No Modifier	Alt	Ctrl
Left Arrow	Move selection left one cell	Not used	Move selection to first column
Right Arrow	Move selection right one cell	Not used	Move selection to last column
Up Arrow	Move selection up one cell	Show previous layer	Scroll and move selection to first row
Down Arrow	Move selection down one cell	Show next layer	Scroll and move selection to last row
Home	Move selection to first visible cell	Not used	Scroll and move selection to first cell
End	Move selection to last visible cell	Not used	Scroll and move selection to last cell
Page Up	Scroll up one screen	Scroll left one screen	Scroll up one screen
Page Down	Scroll down one screen	Scroll right one screen	Scroll down one screen

When viewing a 3D or 4D wave, Alt+Up Arrow and Alt+Down Arrow change the currently viewed layer.

When viewing a 4D wave, Ctrl+Alt+Up Arrow and Ctrl+Alt+Down Arrow change the currently viewed chunk.

Pressing shift-arrow-key extends the selection in the direction of the arrow key. Pressing cmd-shift-arrow-key extends selection as far as possible in the direction of the arrow key.

Decimal Symbol and Thousands Separator in Tables

By default, the decimal symbol for entering a number in a table is period. You can change this to comma or Per System Setting using the Table→Table Misc Settings menu. The selected decimal symbol is used for entering, copying, and pasting data in tables.

The Decimal Symbol setting controls the character that is produced when you press the decimal key on the numeric keypad while entering a value in the entry area of a table.

If comma is selected as the decimal symbol then it is not supported as a column separator when creating new waves by pasting text into a table.

When you choose Per System Setting, the decimal symbol is determined by your system settings as of when Igor was launched. Only period and comma are supported as the decimal symbol. If you choose any decimal symbol other than comma, period will be used as the decimal symbol for tables.

When creating a new wave by entering data into an unused table cell, there are some rare situations when what you are trying to enter cannot be properly interpreted unless you first choose the appropriate column numeric format from the Table menu. For example, if the decimal symbol is comma and you want to enter a time or date/time value with fractional seconds, you must choose Time or Date/Time from the Table→Formats menu before entering the data.

If the decimal symbol is period then the thousands separator is comma. If the decimal symbol is comma then the thousands separator is period. The thousands separator is permitted when entering data in a table. You can also choose a column numeric format that displays thousands separators. However thousands separators are not permitted when creating new waves by pasting text into a table.

Using a Table to Create New Waves

If you click in any unused column, Igor selects the first cell in the first unused column. You can then create new waves by entering a value or pasting data that you have copied to the clipboard.

Creating a New Wave by Entering a Value

When you enter a data value in the first unused cell, Igor creates a single new 1D wave and displays it in the table. This is handy for entering a small list of numbers or text items. If you enter a numeric value, including date/time values, Igor creates a numeric wave. If you enter a nonnumeric value, Igor creates a text wave.

Igor gives the wave a default name, such as wave0 or wave1. You can rename the wave using the Rename item in the Data menu or the Rename item in the Table pop-up menu. You can also rename the wave from the command line by simply executing:

```
Rename oldName, newName
```

When you create a new wave, the wave has one data point — point 0. The cell in point number 1 appears gray. This is the **insertion cell**. It indicates that the preceding cell is the last point of the wave. You can click in the insertion cell and enter a value or do a paste. This adds one or more points to the wave.

If the new wave is numeric, it will be single or double precision, depending on the Default Data Precision setting in the Miscellaneous Settings dialog. The number of digits displayed, however, depends on the numeric format. See **Numeric Formats** on page II-255.

When entering a date, you must use the format selected in the Table Date Format dialog, accessible via the Table menu. The setting in this dialog affects all tables.

If you enter a date (e.g., 1993-01-26), time (e.g., 10:23:30) or date/time (e.g., 1993-01-26 10:23:30) value, Igor notices this. It sets the column's numeric format to display the value properly. It also forces the new wave to be double precision, regardless the Default Data Precision setting in the Miscellaneous Settings dialog. This is necessary because single precision does not have enough range to store date and time values.

Creating New Waves by Pasting Data from Another Program

If you have data in a spreadsheet program or other graphing program, you may be able to import that data into Igor using copy and paste.

This will work if the other program can copy its data to the clipboard as tab-delimited text or comma-delimited text. Most programs that handle data in columns can do this. Tab-delimited data consists of a number of lines of text with following format:

```
value <tab> value <tab> value <terminator>
```

It may start with a line containing column names. The end of a line is marked by a terminator which may be a carriage return, a linefeed, or a carriage return/linefeed combination. If pasted into a word processor, tab delimited text would look something like this:

```
column1     column2     column3     (this line is optional)
27.95       -13.738     12.74e3
31.37       -12.89      13.97e3
```

.	.	.
.	.	.
.	.	.

In the other program, select the cells containing the data of interest and copy them to the clipboard. In Igor, select the first cell in the first unused column in a table and then select Paste from Igor's Edit menu.

Igor scans the contents of the clipboard to determine the number of rows and columns of numeric text data. It also checks the first line of text in the clipboard to see if it contains column names. It creates waves and displays them in the table using the names found in the clipboard or default names. If the text contains names which conflict with existing names, Igor presents a dialog in which you can correct the problem.

If you paste text-only data, which does not contain numbers,dates, times or date/time values, Igor treats all of the pasted text as data instead of treating the first line as column names. This will usually produce the desired results. If you want to treat the first line as column names, use the Load Delimited Text routine to load the text from the clipboard and specify that you want to load wave names. See **Loading Delimited Text Files** on page II-129 for details.

Troubleshooting Table Copy and Paste

If the waves that are created when you paste don't contain the values you expect, chances are that the clipboard does not contain tab-delimited text. In this case you will need to undo the paste. To examine the contents of the clipboard, paste it into an Igor plain text notebook or into the word processor of your choice. After editing the text, copy it to the clipboard again and repaste it into the table.

Creating New Waves by Pasting Data from Igor

You can also create new waves by copying data from existing waves. When you copy wave data in a table, Igor stores not only the raw data but also the following properties of the wave or waves:

- Data units and dimension units
- Data full scale and dimension scaling
- Dimension labels
- The wave note

Thus you can duplicate a wave by copying it in a table and pasting into the unused area of the same table or a different table. You can also copy from a table in one experiment and paste in a table in another experiment.

You can copy and paste the wave note only if you copy the entire wave. If you copy part of the wave, it does not copy the wave note.

Table Names and Titles

Every table that you create has a name. The name is a short Igor object name that you or Igor can use to reference the table from a command or procedure. When you create a new table, Igor assigns it a name of the form Table0, Table1 and so on. You will most often use a table's name when you kill and recreate the table, as described in the next section.

A table also has a title. The title is the text that appears at the top of the table window. Its purpose is to identify the table visually. It is not used to identify the table from a command or procedure. The title can consist of any text, up to 255 bytes.

You can change the name and title of a table using the Window Control dialog. This dialog is a collection of assorted window-related things. Choose Windows→Control→Window Control to display the dialog.

Hiding and Showing a Table

You can hide a table by Shift-clicking the close button.

You can show a table by choosing its name from the Windows→Tables submenu.

Killing and Recreating a Table

Igor provides a way for you to kill a table and then later to recreate it. Use this to temporarily get rid of a table that you expect to be of use later.

You kill a table by clicking the table window's close button or by using the Close item in the Windows menu. When you kill a table, Igor offers to create a **window recreation macro**. Igor stores the window recreation macro in the procedure window of the current experiment. The name of the window recreation macro is the same as the name of the table. You can invoke the window recreation macro later to recreate the table by choosing its name from Windows→Table Macros.

A table does not contain waves but is just a way of viewing them. Killing a table does not kill the waves displayed in a table. If you want to kill the waves in a table, select all of them (Select All in Edit menu) and then choose Kill All Selected Waves from the Table pop-up menu.

For further details, see **Closing a Window** on page II-46 and **Saving a Window as a Recreation Macro** on page II-47.

Index Columns

There are two kinds of numeric values associated with a numeric wave: the stored data values and the computed index values. For example, each point in a real 1D wave has two values: a data value and an X index value. The data value is stored in memory. The X value is computed based on the point number and the wave's X scaling property. The correspondence between point numbers and X values is discussed in detail under **Waveform Model of Data** on page II-62.

Because the index values for a wave are computed, a value in an index column in a table can not be altered by editing the wave. Only values in data columns of a table can be edited. To alter the index values of a wave, use the Change Wave Scaling dialog.

Column Names

Column names are related to but not identical to wave names. You need to use column names to append, remove or modify table columns from the command line or from an Igor procedure.

A column name consists of a wave name and a suffix that identifies which part of the wave the column displays. For each real 1D wave there can be two columns: one for the X index values or dimension labels of the wave and one for the data values of the wave. For complex waves there can be three columns: one for the X index values or dimension labels of the wave, one for the real data values of the wave and one for the imaginary data values of the wave.

If we have a real 1D wave named "test" then there are three column names associated with that wave: test.i ("i" for "index"), test.l ("l" for "label") and test.d ("d" for "data"). If we have a complex 1D wave named "ctest" then there are four column names associated with that wave:ctest.i, ctest.l, ctest.d.real and ctest.d.imag.

Wave Name	Column Name	Column Contents
test	test.i	Index values of test
test	test.l	Dimension labels of test

Wave Name	Column Name	Column Contents
test	test.d	Data values of test
ctest	ctest.i	Index values of ctest
ctest	ctest.d.real	Real part of data values of ctest
ctest	ctest.d.imag	Imaginary part of data values of ctest

For multidimensional waves, the ".i" and ".l" suffixes still specify a single column of index values or dimension labels while the ".d" suffix specifies all of the data columns.

In the table-related commands, you can abbreviate column names as follows:

Full Column Specification	Abbreviated Column Specification
test.d	test
test.i, test.d	test.id
test.l, test.d	test.ld
ctest.d.real, ctest.d.imag	ctest.d or ctest
ctest.i, ctest.d.real	ctest.id.real
ctest.l, ctest.d.real	ctest.ld.real
ctest.i, ctest.d.imag	ctest.id.imag
ctest.l, ctest.d.imag	ctest.ld.imag
ctest.i,ctest.d.real,ctest.d.imag	ctest.id
ctest.l,ctest.d.real,ctest.d.imag	ctest.ld

A 2D wave has X and Y index values. A 3D wave has X, Y and Z index values. A 4D wave has X, Y, Z and T index values. Regardless of the dimensionality of the wave, however, it has only one index column in a table. The index column for a 2D wave, for example, may show the X values or the Y values, depending on how you are viewing the data. The index column will be labeled "wave.x" or "wave.y", depending on the view. However, when referring to the column from an Igor command, you can always use the generic column name "wave.i" as well as the specific column name "wave.x" or "wave.y". A dimension label column is always called "wave.l", regardless of which dimension is showing in the table.

See **Edit** on page V-192 for some examples of commands using column names.

Appending Columns

To append columns to a table, choose Table→Append Columns to Table. This displays the Append Columns dialog.

Igor appends columns to the right end of the table. You can drag a column to a new position by pressing Option (*Macintosh*) or Alt (*Windows*) and dragging the column name.

Removing Columns

To remove columns from a table, choose Table→Remove Columns from Table. This displays the Remove Columns dialog.

You can also select the columns in the table, and use the Table pop-up menu to remove the selected columns.

Removing a column from a table does not kill the underlying wave. The column is not the wave but just a *view* of the wave. Use the Kill Waves item in the Table pop-up menu to remove waves from the table and kill them. Use the Kill Waves item in the Data menu to kill waves you have already removed from a table.

Selecting Cells

If you click in a cell, it becomes the target cell. The old target cell is deselected and the cell you clicked on is highlighted. The target cell ID changes to reflect the row and column of the new target cell and the value of the target cell is shown in the entry line. You click in a cell and make it the target when you want to enter a new value for that cell or because you want to select a range of cells starting with that cell.

Here are the selections that you can make:

Click	Action
Click	Selects a single cell and makes it the target cell
Shift-click	Extends or reduces the selection range
Click in the point column	Selects the entire row
Click in a column name	Selects the entire column
Click in an unused column	Selects the first unused cell
Choose Select All (Edit menu)	Selects all cells (if possible)

The selection in a table must be rectangular. Igor will not let you select a range that is not rectangular. If you choose Select All, Igor will attempt to select all of the cells in the table. However, if you have columns of different length, Igor will be limited to selecting a rectangular array of cells.

If, after clicking in a cell to make it the target cell, you drag the mouse, the cells over which you drag are selected and highlighted to indicate that they are selected. You select a range of cells in preparation for copying, cutting, pasting or clearing those cells. While you drag, the cell ID area shows the number of rows and columns that you have currently selected. If you drag beyond the edges of the table, the cell area scrolls so that you can select as many cells as you want.

Moving the target cell accepts any data entry in progress.

You can change which cell is the target cell using Return, Enter, Tab, or arrow keys. If you are entering a value, these keys also accept the entry.

Key	Action (When a Single Cell is Selected)
Return, Enter, Down Arrow	Moves target cell down
Shift-Return, Shift-Enter, Up Arrow	Moves target cell up
Tab, Right Arrow	Moves target cell right
Shift-Tab, Left Arrow	Move target cell left

If you have a range of cells selected, these keys keep the target cell within that selected range. If it is at one extreme of the selected range it will wrap around to the other extreme.

By default, the arrow keys move the target cell. You can change it so they move the insertion point in the entry line using Table→Table Misc Settings.

The used columns in a table are always contiguous. If you click in any unused column, Igor selects the first unused cell. There are just two things you can do when the first unused cell is selected: create a new wave by entering a value or create new waves by pasting data from the clipboard. Igor will not allow you to select any unused cell other than the first cell in the first unused column.

The Insertion Cell

At the bottom of every column of data values is a special cell called the **insertion cell**. It appears as a gray box below the very last point in a wave.

Sometimes you know the number of points that you want a wave to contain and don't need to insert additional points into the wave. However, if you want to enter a short list of values into a table or to add new data to an existing wave, you can do this by entering data in the insertion cell.

When you enter a value in an insertion cell, Igor extends the wave by one point. Then the insertion cell moves down one position and you can insert another point.

The insertion cell can also be used to a extend a wave or waves by more than one point at a time. This is described under **Pasting Values** on page II-248.

You can also insert points in waves using the Insert Points item which appears in both the Table pop-up menu and the Data menu or using the InsertPoints operation from the command line.

Entering Values

You can alter the data value of a point in a wave by making the cell corresponding to that value the target cell, typing the new value in the entry line, and then confirming the entry.

You can also accept the entry by clicking in any cell or by pressing any of the keys that move the target cell: Return, Enter, Tab, or arrow keys. You can discard the entry by pressing Escape or by clicking the X icon.

If a range of cells is selected when you confirm an entry, the target cell will move within the range of selected cells unless you click in a cell outside this range.

While you are in the process of entering a value, the Clear, Copy, Cut and Paste items in the Edit menu as well as their corresponding command key shortcuts affect the entry line. If you are not in the process of entering, these operations affect the cells.

Entering a value in an insertion cell is identical to entering a value in any other cell except that when the entry is confirmed the wave is extended by one point.

Igor will not let you enter a value in an index column since index values are computed based on a waves dimension scaling.

Dimension labels are limited to 255 bytes. If you paste into a dimension label cell, Igor clips the pasted data to 255 bytes.

Prior to Igor Pro 8.00, dimension labels were limited to 31 bytes. If you use long dimension labels, your wave files and experiments will require Igor Pro 8.00 or later.

When entering a value in a numeric column, if what you have entered in the entry line is not a valid numeric entry, Igor will not let you confirm it. The check icon will be dimmed to indicate that the value can not be entered. To enter a date in a date column, you must use the date format specified in the Table Date Format dialog.

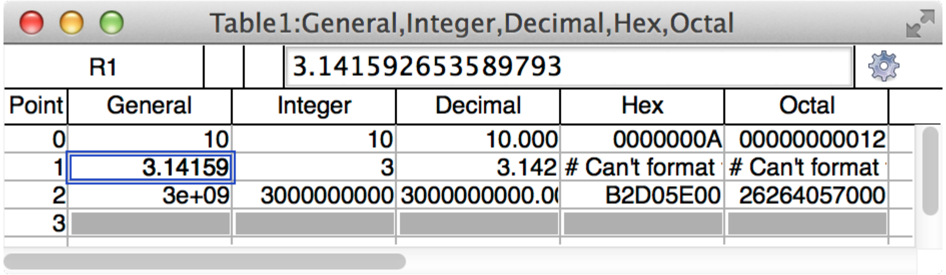
If you edit a text wave or dimension label which contains bytes that are not valid in the wave's text encoding, Igor displays a warning and substitutes escape codes for the invalid bytes. See **Editing Invalid Text** on page II-259 for details.

If you attempt to enter a character that can not be represented in the text encoding controlling a text wave or dimension label, Igor displays an error message. See **Entering Special Characters** on page II-261 for details.

Table Display Precision

The display of data in a column of a table depends on the data type of the displayed wave, the values stored in the wave, and the numeric format settings applied to the column. In some cases, the cell area displays less than full precision but the entry line always displays full precision.

For example, this table displays five double-precision floating point waves using different display formats. Each of the waves contains the values 10, 3.141592653589793, and 3 billion.



Point	General	Integer	Decimal	Hex	Octal
0	10	10	10.000	0000000A	00000000012
1	3.14159	3	3.142	# Can't format	# Can't format
2	3e+09	3000000000	3000000000.0	B2D05E00	26264057000
3					

The first column is set to general format with six digits of precision which is the factory default format for new table columns. The general format chooses either integer, floating point or scientific notation depending on the displayed value.

The value of General[1] is displayed in the body of the table as 3.14159 since the format specifies six digits of precision. Since this cell is selected as the target cell, its value appears in the entry line. However, the entry line displays the value using full 16-digit precision rather than using the column's format. This guarantees that, if you click in the entry line and accept the text that is already there, no precision is lost.

If the General wave were single-precision floating point instead of double-precision, the entry line would show 8 digits for non-integer values since 8 digits are sufficient for single-precision. Integer values from single-precision waves are displayed using up to 16 digits.

Because the Integer column is formatted to display as integer, 3.141592653589793 is displayed as 3. If you clicked on Integer[1], the entry line would show the full precision.

The story is the same for Decimal[1]. The cell area of the table follows the column's formatting but if you clicked Decimal[1], the full precision would appear in the entry line.

The Hex and Octal columns are set to hexadecimal and octal display respectively. The hex and octal formats do not support the display of fractional data so Hex[1] and Decimal[1] display error messages instead of values. This is further discussed under **Hexadecimal Numeric Formats** on page II-257 and **Octal Numeric Formats** on page II-257.

Date Values

Dates and times are represented in Igor date format — as a number of seconds since midnight, January 1, 1904. Dates before that are represented by negative values.

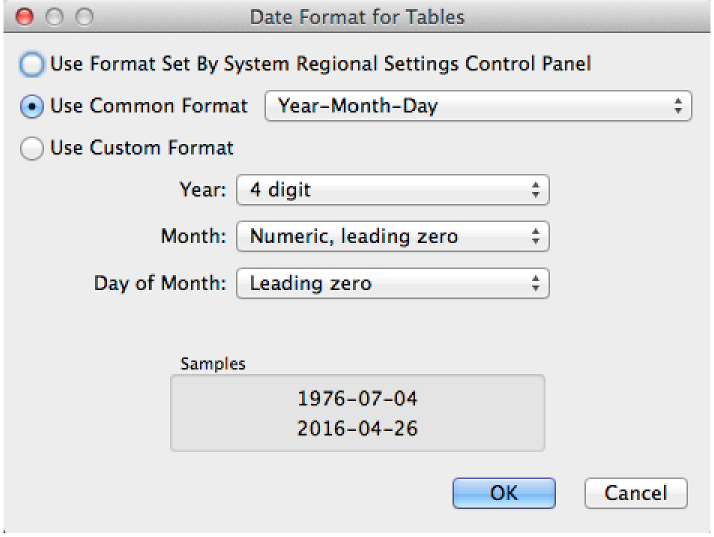
A date can not be accurately stored in the data values of a single precision wave. Make sure to use double precision to store dates and times.

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-85.

The way you enter dates in tables and the way that Igor displays them is controlled by the Table Date Format dialog which you invoke through the Table menu. This dialog sets a global preference that determines the date format for all tables.

If in the Table Date Format dialog you choose to use the system date format, which is the factory default setting, Igor displays dates in a table using the short date format as set by the Language & Region control panel (*Macintosh*) or by the Region control panel (*Windows*).

Alternatively, you can choose to use a common date format or a custom date format. Here is what the dialog looks like if the Use Common Format radio button is selected:



Date Format for Tables

○ Use Format Set By System Regional Settings Control Panel

● Use Common Format: Year-Month-Day

○ Use Custom Format

Year: 4 digit

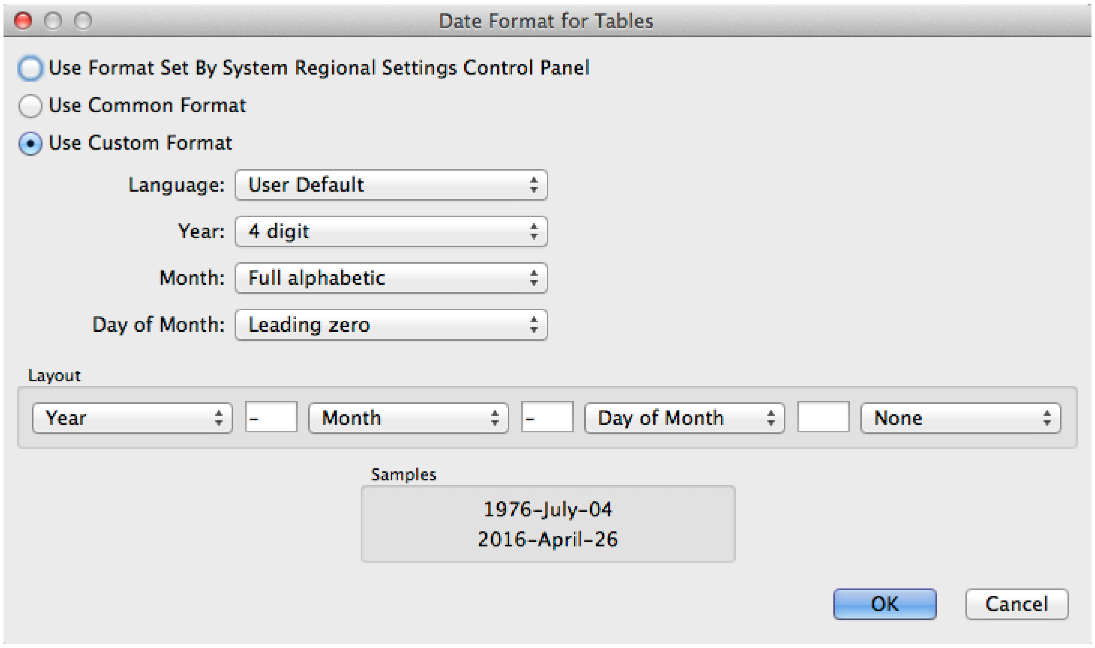
Month: Numeric, leading zero

Day of Month: Leading zero

Samples

1976-07-04
2016-04-26

You can access even more flexibility in those rare cases where it's needed by clicking the Use Custom Format radio button:



Date Format for Tables

○ Use Format Set By System Regional Settings Control Panel

○ Use Common Format

● Use Custom Format

Language: User Default

Year: 4 digit

Month: Full alphabetic

Day of Month: Leading zero

Layout

Year | - | Month | - | Day of Month | | None

Samples

1976-July-04
2016-April-26

When using a common or custom date format that includes separators (e.g., 10/05/99 or 10.05.99), Igor is lenient about the number of digits in the year and whether or not leading zeros are used. Igor will accept two or four digit years and leading zeros or no leading zeros for the year, month, and day of month. However, when using a format with no separators (e.g., 991005 or 19991005), Igor requires that you enter the date exactly as the format specifies.

When you enter a value in the first unused column in a table, Igor must deduce what kind of value you are entering (number, date, time, date/time, or text). It then sets the column format appropriately and interprets what you have entered accordingly. An ambiguity occurs if you use date formats with no separators. For example, if you enter 991005, are you trying to enter a date or a number? Igor has no way to know. Therefore, if you want to create a new column consisting of dates with no separators, you must choose Date from the Table Format submenu before you enter the value. This is not necessary for dates that include separators because Igor can distinguish them from numbers.

If you choose a date format that includes alphabetic characters, such as “October 11, 1999”, you must enter dates exactly as the format indicates, including spaces.

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-85.

Special Values

There are two special values that can be entered in any numeric data column. They are NaN and INF.

Missing Values (NaNs)

NaN stand for “Not a Number” and is the value Igor uses for missing or blank data. Igor displays NaNs in a table as a blank cell. NaN is a legal number in a text data file, in text pasted from the clipboard, and in a numeric expression in Igor’s command line or in a procedure.

A point will have the value NaN when a computation has produced a meaningless result, for example if you take the log of a negative number. You can enter a missing value in a cell of a table by entering NaN or by deleting all of the text in the entry line and confirming the entry.

You can also get NaNs in a wave if you load a delimited text data file or paste delimited text which contains two delimiters with no number in between.

Infinities (INFs)

INF stands for “infinity”. Igor displays infinities in a table as “INF”. INF is a legal number in a text data file, in text pasted from the clipboard and in a numeric expression in Igor’s command line or in a procedure.

A point will have the value INF or -INF when a computation has produced an infinity, for example if you divide by zero. You can enter an infinity in a cell of a table by entering INF or -INF.

Clearing Values

You invoke the clear operation by choosing Edit→Clear from. Clear sets all selected cells in numeric columns to zero. It sets all selected cells in text and dimension label columns to "" (empty string). It has no effect on selected cells in index columns.

To set a block of numeric values to NaN (or any other numeric value), select the block and then choose Analysis→Compose Expression. In the resulting dialog, choose “_table selection_” from the Wave Destination pop-up menu. Enter “NaN” as the expression and click Do It.

Copying Values

You invoke the copy operation by choosing Edit→Copy. This copies all selected cells to the clipboard as text and as Igor binary. It is useful for copying ranges of points from one wave to another, from one part of a wave to another part of that wave, and for exporting data to another application or to another Igor experiment (see **Exporting Data from Tables** on page II-252).

Copying and pasting in Igor tables uses the binary version of the data which represents the data with full precision and also includes wave properties such as scaling and units. If you paste anywhere other than an Igor table, for example to an Igor notebook or to another program, the text version of the data is used.

For technical reasons relating to 64-bit support, the binary clipboard format is different from the Igor Pro 6 format. Consequently you can not copy/paste binary table data between these versions.

The text version of the copied data uses as many digits as needed to represent the data with full precision. If you want to export text exactly as shown in the table you must use the Save operation with the /F flag, or File→Save Table Copy, or the **SaveTableCopy** operation.

You can also create new waves by copying data from existing waves. This is described earlier in this chapter under **Creating New Waves by Pasting Data from Igor** on page II-240.

See also **Multidimensional Copy/Cut/Paste/Clear** on page II-265.

Cutting Values

You invoke the cut operation by choosing Edit→Cut. Cut starts by copying all selected cells to the clipboard as text and as Igor binary. Then it deletes the selected points from their respective waves, thereby shortening the waves.

You cannot cut sections of an index column since index values are computed based on point numbers, not stored. However, if you cut a section of a data or dimension label column, the index column corresponding to the data column will also be shortened.

Pasting Values

You invoke the paste operation by choosing Edit→Paste. There are three kinds of paste operations: a replace-paste, an insert-paste and a create-paste.

Paste Type	What You Do	What Igor Does
Replace-paste	Choose Paste.	Replaces the selected cells with data from the clipboard.
Insert-paste	Press Shift and choose Paste.	Inserts data from clipboard as new cells.
Create-paste	Select the first cell in the first unused column and then choose Paste.	Creates new waves containing clipboard data.

When dealing with multidimensional waves, there are other options. See **Multidimensional Copy/Cut/Paste/Clear** on page II-265 for details.

When you do a paste, Igor starts by figuring out how many rows and columns of values are in the clipboard. The clipboard may contain binary data that you just copied from an Igor table or it may contain plain text data from another application such as a spreadsheet or a text editor.

If the data in the clipboard is plain text, Igor expects that rows of values be separated by carriage return characters, linefeed characters, or carriage return/linefeed pairs and that individual values in a row be separated by tabs or commas. This is normally no problem since most applications export data as tab-delimited text. If you have trouble with a paste and are not sure about the format of the data in the clipboard, you can paste it into an Igor notebook to inspect or edit it.

Once Igor has figured out how many rows and columns are in the clipboard, it proceeds to paste those values into the table and therefore into the waves that the table displays.

If you select the first cell in the first unused column, the paste will be a create-paste. In this case, Igor makes new waves, appends them to the table and then stores the data in the clipboard in the new waves. It makes one new wave for each column of text in the clipboard. If the text starts with a row of column names, Igor uses this row as the basis for the names of the new waves. Otherwise Igor uses default wave names.

Mismatched Number of Columns

If the number of columns in the clipboard is not the same as the number of columns selected in the table then Igor asks you how many columns to paste. This applies to the replace-paste but not to the insert-paste or create-paste.

For example, if you have M columns of text in the clipboard but you select N columns and then do a paste, Igor presents presents the Columns to Paste dialog which gives you two options:

- Paste M columns using the current selection
- Change the selection and paste N columns

Mismatched Number of Rows

If the number of rows in the clipboard is not the same as the number of rows selected in the table then Igor asks you how many rows to paste. This applies to the replace-paste but not to the insert-paste or create-paste.

For example, if you have M rows of text in the clipboard but you select N rows and then do a paste, Igor presents the Rows to Paste dialog which gives you three options:

- Paste M rows using the current selection
- Change the selection and paste N rows
- Replace the selected N rows with M rows from the clipboard

Pasting and Index Columns

Since the values of an index column are computed based on point numbers, they can not be altered by pasting. However, if index columns and data columns are adjacent in a range of selected cells, a paste can still be done. The data values will be altered by the paste but the index values will not be altered.

Pasting and Column Formats

When you paste plain text data into existing numeric columns, Igor tries to interpret the text in the clipboard based on the numeric format of the columns. For example, if a column is formatted as dates then Igor tries to interpret the data according to the table date format. If the column is formatted as time then Igor tries to interpret the text as a time values (e.g., 10:00:00). If the column has a regular number format, Igor tries to intrepret the text as regular numbers.

When you paste plain text data into unused columns, Igor does a create-paste. In this case, Igor inspects the text in the clipboard to determine if the data is in date format, time format, date and time format or regular number format. When it appends new columns to the table, it applies the appropriate numeric format.

When pasting octal or hexadecimal text in a table, you must first set the column format to octal or hexadecimal so that Igor will correctly interpret the text.

If the column does not appear to be in any of these formats, Igor creates a text wave rather than a numeric wave.

See **Date Values** on page II-245 for details on entering dates.

Copy-Paste Waves

You can copy and paste entire waves within Igor. This is described under **Creating New Waves by Pasting Data from Igor** on page II-240.

Inserting and Deleting Points

In addition to pasting and cutting, you can also insert and delete points from waves using the Insert Points and Delete Points dialogs via the Data menu or via the Table pop-up menu. You can use these dialogs to modify waves without using a table but they do work intelligently when a table is the top window.

Finding Table Values

You can search tables for specific contents by choosing Edit→Find. This displays the Find In Table dialog.

Find In Table can search the current selection in the active table, the entire active table or all table windows. You control this using the right-hand pop-up menu at the top of the dialog.

The All Table Windows mode searches standalone table windows only. It does not search table subwindows. It is possible to search a table subwindow in a control panel using the Top Table mode. Searching in tables embedded in graphs and page layouts is not supported.

Find In Table can search for the following types of values which you control using the left-hand pop-up menu.

Find Type	Description
Row	Displays the specified row but does not select it.
Text String	Finds the specified text string in any type of column: text, numeric, date, time, date/time, and dimension labels. For example, searching for “-1” would find numeric cells containing -1.234 and -1e6. A given cell is found only once, even if the search string occurs more than once in that cell. The target string is limited to 254 bytes.
Blank Cell	Finds blank cells in any type of column: text, numeric, date, time, date/time, and dimension labels. Finds blank cells in numeric columns (NaNs) and text columns (text elements containing zero characters).
Numeric Value	Finds numeric values within the specified range in numeric columns only. Does not search the following types of columns: text, date, time, date/time, and dimension labels.
Date	Finds date values within the specified range in date and date/time columns only. Does not search the following types of columns: text, numeric, time and dimension labels. Accepts input of dates in the format specified by Table Date Format dialog (Table menu).
Time of Day	Finds a time of day within the specified range in time and date/time columns only. Does not search the following types of columns: text, numeric, date and dimension labels. A time of day is a time between 00:00:00 and 24:00:00. Times are entered as hh:mm:ss.ff with the seconds part and fractional part optional.
Elapsed Time	Finds an elapsed time within the specified range in time columns only. Does not search the following types of columns: text, numeric, date, date/time and dimension labels. Unlike a time of day, an elapsed time can be negative and can be greater than 24:00:00. Times are entered as hh:mm:ss.ff with the seconds part and fractional part optional.
Date/Time	Finds a date/time within the specified range in date/time columns only. Does not search the following types of columns: text, numeric, date, time and dimension labels. Date/time values consist of a date, a space and a time.

Find In Table does not search the point column.

The search starts from the “anchor” cell. If you are searching the top table or the current selection, the anchor cell is the target cell. If you are searching all tables, the anchor cell is the first cell in the first-opened table, or the last cell in the last-opened table if you are doing a backward search.

When you do an initial search via the Find dialog, the search includes the anchor cell. When you do a subsequent search using Find Again, the search starts from the cell after the anchor cell, or before it if you are doing a backward search.

A find in the top table starts from the target cell and proceeds forward or backward, depending on the state of the Search Backwards checkbox. The search stops when it hits the end or beginning of the table, unless Wrap Around Search is enabled, in which case the whole table is searched.

A find in the current selection also starts from the target cell and proceeds forward or backward, depending on the state of the Search Backwards checkbox. The search stops when it hits the end or beginning of the selection, unless Wrap Around Search is enabled, in which case the whole selection is searched.

If Search Rows First is selected, all rows of a given column are searched, then all rows of the next column. If Search Columns First is selected, all columns of a given row are searched, then all columns of the next row.

To do a search of a 3D or 4D wave, you must create a table containing just that wave. Then the Table Find will search the entirety of the wave. If the table contains more than one wave, the Table Find will not search the parts (e.g. other layers of a 3D wave) of a 3D or 4D wave that are not shown in the table.

Choosing the Edit→Find Selection menu sets the Find mode to Find Text String, Find Blank Cells, Find Numeric Value, Find Date, Find Time Of Day, Find Elapsed Time, or Find Date/Time based on the format of the target cell except that, if the target cell is blank, the mode is set to Find Blank Cells regardless of the cell's format.

You may find it convenient to use Find Again (Command-G on Macintosh, Ctrl+G on Windows) after doing an initial find to find subsequent cells with the specified contents. Pressing Shift (Comand-Shift-G on Macintosh, Ctrl+Shift+G on Windows) does a Find Again in the opposite direction.

Replacing Table Values

You can perform a mass replace in a table by choosing Edit→Replace. This displays the Replace In Table dialog.

Unlike Find In Table, which can search all tables, Replace In Table is limited to the top table or the current selection, as set by the right-hand pop-up menu.

When you click Replace All, Replace In Table first finds the specified cell contents using the same rules as Find In Table. It then replaces the contents with the specified replace value. It then continues searching and replacing until it hits the end of the table or selection, unless Wrap Around Search is enabled, in which case the whole table or selection is searched.

You can undo all of the replacements by choosing Edit→Undo Replace.

Replace In Table does not affect X columns. You must use the Change Wave Scaling dialog (Data menu) for that.

Replace In Table goes through each candidate cell looking for the specified search value. If it finds the value, it extracts the text from the cell and does the replacement on the extracted text. If the resulting text is legal given the format of the cell, the replacement is done. If it is not legal, Replace In Table stops and displays an error dialog showing where the error occurred.

Here are some additional considerations regarding Replace In Table.

Find Type	Description
Text String	In each cell, all occurrences of the find string are replaced with the replace string. For example, if you replacing "22" with "33" and a cell contains the value 122.223, the resulting value will be 133.333. The replace string is limited to 254 bytes. Using text string replace, it is possible to come up with a value that is not legal given a cell's formatting. For example, if you replace "22" with "9.9" in the example above, you get "19.9.9.93". This is not a legal numeric value so Replace In Table displays an error dialog.
Date	You can replace a date in a date column or a date/time column. When replacing a date in a date/time column, the time component is not changed. Dates must use the format specified by the Table Date Format dialog (Table menu).
Time of Day	You can replace a time of day in a time column or a date/time column. When replacing a time of day in a date/time column, the date component is not changed.

Selectively Replacing Table Values

The Replace In Table dialog is designed to do a mass replace. You can do a selective replace using the Find In Table dialog followed by a series of Find Again and Paste operations. Here is the process:

1. Choose Edit→Find and find the first cell containing the value you want to replace.
2. Edit that cell so it contains the desired value.
3. Copy that cell’s contents to the clipboard.
4. Do Find Again (Command-G on Macintosh, Ctrl+G on Windows) to find the next cell you might want to replace.
5. If you want to replace the found cell, do Paste (Command-V on Macintosh, Ctrl+V on Windows).
6. If not done, go back to step 4.

Exporting Data from Tables

You can use the clipboard to export data from an Igor table to another application. If you do this you must be careful to preserve the precision of the exported data.

When you copy data from an Igor table, Igor puts the data into the clipboard in two formats: tab-delimited text and Igor binary.

If you later paste that data into an Igor table, Igor uses the Igor binary data so that you retain all precision through the copy-paste operation.

If you paste the data into another application, the other application uses the text data that Igor stored in the clipboard. To prevent losing precision, Igor uses enough digits to represent the data with full precision.

You can also export data via files by choosing Data→Save Waves→Save Delimited Text or File→Save Table Copy.

Changing Column Positions

You can rearrange the order of columns in the table. To do this, position the cursor over the name of the column that you want to move. Press Option (*Macintosh*) or Alt (*Windows*) and the cursor changes to a hand. If you now click the mouse you can drag an outline of the column to its new position.

When you release the mouse the column will be redrawn in its new position. Igor always keeps all of the columns for a particular wave together so if you drag a column, you will move all of the columns for that wave.

The point column can not be moved and is always at the extreme left of the cell area.

Changing Column Widths

You can change the width of a column by dragging the vertical boundary to the right of the column name.

You can influence the manner in which column widths are changed by pressing certain modifier keys.

If Shift is pressed, all table columns except the Point column are changed to the same width.

The Command (*Macintosh*) or Ctrl (*Windows*) key determines what happens when you drag the boundary of a data column of a multidimensional wave. If that key is not pressed, all data columns of the wave are set to the same width. If that key is pressed then just the dragged column is changed.

Autosizing Columns By Double-Clicking

You can autosize a column by double-clicking the vertical boundary to the right of the column name.

You can influence the manner in which column widths are changed by pressing certain modifier keys.

If the no modifier keys are pressed and you double-click the boundary of a data column of a multidimensional wave then the width of each data column is set individually.

When pressing Option (*Macintosh*) or Alt (*Windows*) and you double-click the boundary of a data column of a multidimensional wave then the width of all data columns are set the same.

If Shift is pressed, all table columns except the Point column are autosized. Shift pressed with Option (*Macintosh*) or Alt (*Windows*) will autosize all data columns of a given wave to the same width.

When pressing Command (*Macintosh*) or Ctrl (*Windows*), only the double-clicked column is autosized, not all data columns of a multidimensional wave.

Autosizing Columns Using Menus

You can autosize columns by selecting them and choosing Autosize Columns from the Table menu or from the table popup menu in the top-right corner of the table. You can also choose Autosize Columns from the contextual menu that you get when you Control-click (*Macintosh*) or right-click (*Windows*) on a column.

You can influence the manner in which column widths are changed by pressing certain modifier keys.

If the no modifier keys are pressed and a data column of a multidimensional wave is selected, all data columns of that wave are set individually.

If Option (*Macintosh*) or Alt (*Windows*) is pressed and a data column of a multidimensional wave is selected, all data columns of that wave are set the same.

If Shift is pressed, all table columns except the Point column are autosized. Shift pressed with Option (*Macintosh*) or Alt (*Windows*) will autosize all data columns of a given wave to the same width.

When pressing Command (*Macintosh*) or Ctrl (*Windows*), only the selected columns are autosized, not all data columns of a multidimensional wave.

Autosizing Limitations

When you autosize a column, the width of every cell in that column must determined. For very long columns (100,000 points or more), this may take a very long time. When this happens, cell checking stops and the autosize is based only on the checked cells.

Similarly, if you autosize the data columns of a multidimensional wave with a very large number of columns (10,000 or more columns), this could take a very long time. When this happens, columns checking stops and the autosize is based only on the checked columns.

If the default time limits are not suitable, use the ModifyTable autosize keyword to set the time limits.

Changing Column Styles

You can change the presentation style of columns in a table using the Modify Columns dialog, the Table menu in the main menu bar, the Table pop-up menu (gear icon), or the contextual menu that you get by right-clicking.

You can invoke the Modify Columns dialog from the Table menu, from the Table pop-up menu, from the contextual menu, or by double-clicking a column name.

You can select one or more columns in the Columns to Modify list. If you select more than one column, the items in the dialog reflect the settings of the first selected column.

Once you have made your selection, you can change settings for the selected columns. After doing this, you can then select a different column or set of columns and make more changes. Igor remembers all of the changes you make, allowing you to do everything in one trip to the dialog.

There is a shortcut for changing a setting for all columns at once without using the dialog: Press Shift while choosing an item from the Table menu, from the Table pop-up menu, or from the contextual menu.

Tables are primarily intended for on-screen data editing. You can use a table for presentation purposes by exporting it to another program as a picture or by including it in a page layout. However, it is not ideal for this purpose. For example, there no way to change the background color or the appearance of gridlines.

You can capture your favorite styles as preferences. See **Table Preferences** on page II-272.

Modifying Column Properties

You can independently set properties, such as color, column width, font, etc., for the index or dimension label column and the data column of 1D waves. Except in rare cases all of the data columns of a multidimensional wave should have the same properties. When you set the properties of one data column of a multidimensional wave using Igor's menus or using the Modify Columns dialog, Igor sets the properties of all data columns the same.

For example, if you are editing a 3 x 3 2D wave and you set the first data column to red, Igor will make the second and third data columns will red too.

Despite Igor's inclination to set all of the data columns of a multidimensional wave the same, it is possible to set them differently. Select the columns to be modified. Press Command (*Macintosh*) or Ctrl (*Windows*) before clicking the Table menu or Table pop-up menu and make a selection. Your selection will be applied to the selected columns only instead of to all data columns from the selected waves.

The ModifyTable operation supports a column number syntax that designates a column by its position in the table. Using this operation, you can set any column to any setting. For example:

```
Make/O/N=(3,3) mat
Edit mat
ModifyTable rgb[1]=(50000,0,0), rgb[2]=(0,50000,0)
```

This sets mat's first data column to red and its second to blue.

You can specify a range of columns using column number syntax:

```
ModifyTable rgb[1,3]=(50000,0,0)
```

The Modify Columns dialog sets the properties for both the real and imaginary columns of complex waves at the same time. If you really want to set the properties of the real and imaginary columns differently, you must use the column number syntax shown above.

Column Titles

The column title appears at the top of each column. By default, Igor automatically derives the column title from the name of the wave displayed in the column. For example, if we display the X index and data values of wave1 and data values of wave2, the table will have columns titled wave0.x, wave0.d, and wave1.

Igor uses the suffixes ".x" and ".d" only if this is necessary to distinguish columns. If there is no index column displayed, Igor omits the suffix.

Using the Title setting in the Modify Columns dialog, you can replace the automatically derived title with a title of your own. Remove all text from this item to turn the automatic title back on.

The title setting changes *only* the title of the column in this table. It does not change wave name. We provide the column title setting primarily to make tables look better in presentations (in layouts and exported pictures). If you are not using a table for a presentation, it is better to let Igor automatically generate column titles since this has less potential for confusion. If you want to rename a wave, use the Rename items in the Data and Table pop-up menus.

If you really need to use column titles for multidimensional waves, use the techniques described in **Modifying Column Properties** on page II-254 to set the title for individual columns.

Numeric Formats

Columns in tables display either text or numeric waves. For numeric waves, the column format determines how the data values in the wave are entered and displayed. The column format has no effect on data columns of text waves.

In addition to regular number formats, tables support date, time and date&time formats. The format is merely a way of displaying a number. Even dates and times are stored internally in Igor as numbers. You can enter a value in a numeric column of a table as a number, date, time or date&time if you set the format for the column appropriately.

The following table lists all of the numeric formats.

Numeric Format	Description
General	Displays numbers in a format appropriate to the number itself. Very large or small numbers are displayed in scientific notation. Other numbers are displayed in decimal form (e.g. 1234.567). The Digits setting controls the number of significant digits. Integers are displayed with no fractional digits.
Integer	Numbers are displayed as the nearest integer number. For example, 1234.567 is displayed as 1235.
Integer with comma	Numbers are displayed as the nearest integer number. In addition, commas are used to separate groups of three digits. For example, 1234.567 is displayed as 1,235.
Decimal	As many digits to the left of the decimal point as are required are used to display the number. The Digits setting controls the number of digits to the right of the decimal point. For example, if the number of digits is specified as two, 1234.567 is displayed as 1234.57.
Decimal with comma	Identical to the decimal format except that commas are used to separate groups of three digits to the left of the decimal point.
Scientific	Numbers are displayed in scientific notation. The Digits setting controls the number of digits to the right of the decimal point.
Date	Dates are displayed using the format set in the Table Date Format dialog. See **Date/Time Formats** on page II-256.
Time	[+][-]hhhh:mm:ss[.ff] [AM/PM]. See **Date/Time Formats** on page II-256.
Date & Time	Date format plus space plus time format. See **Date/Time Formats** on page II-256.
Octal	Numbers are displayed in octal (base 8) notation. Only integers are supported. The number of digits displayed depends on the wave data type and the Digits setting is ignored. See **Octal Numeric Formats** on page II-257 for details.
Hexadecimal	Numbers are displayed in hexadecimal (base 16) notation. Only integers are supported. The number of digits displayed depends on the wave data type and the Digits setting is ignored. See **Hexadecimal Numeric Formats** on page II-257 for details.

When you enter a number in a table, Igor expects either dot or comma as the decimal symbol, as determined by the Decimal Symbol setting in the Table Misc Settings dialog. The factory default is dot. This setting applies only to entering numbers in tables. To change it, choose Table→Table Misc Settings. If it is set to Per System Setting and you change the system decimal symbol, you must restart Igor for the change to take effect.

For most numeric formats you can control the number of digits displayed. You can set this using the Modify Columns dialog or using the Digits submenu of the Table menu, table pop-up menu (gear icon), or table contextual menu. The meaning of the number that you choose from the Digits submenu depends on the numeric format.

Numeric Format	You Specify
General	Number of displayed digits
Decimal (0.0...0)	Number of digits after the decimal point
Decimal with comma (0.0...0)	Number of digits after the decimal point
Time and Date&Time	Number of digits after the decimal point when displaying fractional seconds
Scientific (0.0...0E+00)	Number of digits after the decimal point
Octal	Total number of octal digits to display.
Hexadecimal	Total number of hexadecimal digits to display.

The Digits setting has no effect on columns displayed using the integer, octal and hexadecimal formats and also has no effect on columns displaying text waves. It affects time and date/time formats only if the display of fractional seconds is enabled.

With the General format, you can choose to display trailing zeros or not.

With the time format, Igor accepts and displays times from -9999:59:59 to +9999:59:59. This is the supported range of elapsed times. If you are entering a time-of-day rather than an elapsed time, you should restrict yourself to the range 00:00:00 to 23:59:59.

With the Time and Date&Time formats, you can choose to display fractional seconds. Most people dealing with time data use whole numbers of seconds. Therefore, by default, a table does not show fractional seconds. If you want to see fractional seconds in a table, you must choose Show Fractional Seconds from the Table→Format menu. Once you do this, the Table→Digits menu controls the number of digits that appear in the fractional part of the time.

If you always want to see fractional seconds, use the Capture Table Prefs dialog to capture columns whose Show Fractional Seconds setting is on. This applies to tables created after you capture the preference.

When displaying fractional seconds, Igor always displays trailing zeros and the Show Trailing Zeros menu item in the Table→Format menu has no effect.

When choosing a format, remember that single precision floating point data stores about 7 decimal digits and double-precision floating point data stores about 16 decimal digits. If you want to inspect your data down to the last decimal place, you need to select a format with enough digits.

The format does not affect the precision of data that you export via the clipboard from a table to another application. See **Exporting Data from Tables** on page II-252.

Date/Time Formats

As described under **Date Values** on page II-245, the way you enter dates in tables and the way Igor displays them is controlled by the Table Date Format dialog which you invoke through the Table menu. This dialog sets a global preference that determines the date format for all tables. By factory default, the table date format is controlled by the system Regional Settings control panel.

If you set the column format to time, then Igor displays time in elapsed time format. You can enter elapsed times from -9999:59:59 to +9999:59:59. You can precede an elapsed time with a minus sign to enter a negative elapsed time. You can also enter a fractional seconds value, for example 31:35:20.19. To view fractional seconds, choose Show Fractional Seconds from the Format submenu of the Table menu.

You can also enter times in time-of-day format, for example 1:45 PM or 1:45:00 PM or 13:45 or 13:45:00. A space before the AM or PM is allowed but not required. AM and am have no effect on the resulting time. PM adds 12 hours to the time. PM has no effect if the hour is already 12 or greater.

Igor stores times the same way whether they are time-of-day times or elapsed times. The difference is in how you think of the value and how you choose to display it in a graph.

Here are some valid time-of-day times:

```
00:00:00 (midnight)           12:00:00 (noon)
06:30:00 (breakfast time)     18:30:00 (dinner time)
06:30 (breakfast time)        18:30 (dinner time)
06:30 PM (dinner time)        18:30 PM (dinner time)
```

Here are some valid elapsed times:

```
-00:00:10    (T minus 10 and counting)
72:00:00     (an elapsed time covering three days)
4:17:33.25   (an elapsed time with fractional seconds)
```

The format for a date/time wave in a table defaults to Date, Time, or Date/Time depending on the content of the wave.

For displaying date/time data in a graph, see **Date/Time Axes** on page II-315.

Octal Numeric Formats

You can format any numeric column in a table as octal. Igor ignores the digits setting for columns displayed as octal and always displays all digits required for the supported range of values.

For floating point waves (single-precision and double-precision), negative values are displayed using a minus sign as this is necessary to accurately present the wave value. For example, -1 is displayed as -00000001.

It is possible to store in floating point waves a value that can't be displayed in octal because the value is fractional or exceeds the supported range of values. In these cases, the table will display an error message such as "# Can't format fractional value as octal" or "# Value too large to display as octal".

If you are using a floating point wave for data you think of as integer data of a given size, consider redimensioning the wave to the appropriate integer type using the **Redimension** operation. For example, if you are storing 16-bit signed integer data in a floating point wave, consider redimensioning the wave as 16-bit signed integer to get a more appropriate octal display in a table.

Hexadecimal Numeric Formats

You can format any numeric column in a table as hexadecimal. Igor ignores the digits setting for columns displayed as hexadecimal and always displays all digits required for the supported range of values.

For floating point waves (single-precision and double-precision), negative values are displayed using a minus sign as this is necessary to accurately present the wave value. For example, -1 is displayed as -00000001.

It is possible to store in floating point waves a value that can't be displayed in hexadecimal because the value is fractional or exceeds the supported range of values. In these cases, the table will display an error message such as "# Can't display fractional value as hex" or "# Value too large to display as hex".

Although SP waves are displayed as 32 bits using 8 hex digits, they have only 24 bits of precision so not all values greater than 2^24 (16,777,216) can be represented by an SP wave.

If you are using a floating point wave for data you think of as integer data of a given size, consider redimensioning the wave to the appropriate integer type using the **Redimension** operation. For example, if you are storing 16-bit signed integer data in a floating point wave, consider redimensioning the wave as 16-bit signed integer to get a more appropriate hexadecimal display in a table.

Object Reference Wave Formatting

This section is for advanced users only.

In addition to text and numeric waves, tables can also display object reference waves. An object reference wave is a WAVE wave or a DFREF wave.

The elements of object reference waves are displayed with full precision either as decimal integer, hexadecimal, or octal. If the column format is integer or octal, the column is displayed as integer or octal. If the column format is any other format, including general, decimal, scientific, date, time, and date&time, the column is displayed as hexadecimal.

Editing Text Waves

You can create a text wave in a table by clicking in the first unused cell and entering non-numeric text. For the most part, editing a text wave is self-evident. However, there are some issues, mostly relating to special characters or large amounts of text, that you may run into.

Large Amounts of Text in a Single Cell

A text wave is handy for storing short descriptions of corresponding data in numeric waves. In some cases, you may find it useful to store larger amounts of text. There is no limit on the number of characters in a point of a text wave. However, the entry area of a table can display no more than 100,000 bytes. You cannot edit in a table a cell containing more than 100,000 bytes.

The Edit Text Cell Dialog

You edit numeric wave data using the table entry line. To edit text wave data, you can use the the table entry line or the Edit Text Cell dialog. The dialog is convenient for editing multi-line data. It is also convenient for editing text to be used for annotations.

When a text wave cell is selected, an icon appears at the right end of the table entry line. Click the icon to invoke the dialog.

The next section discusses using the Edit Text Cell dialog to edit tabs and terminators.

Tabs, Terminators and Other Control Characters

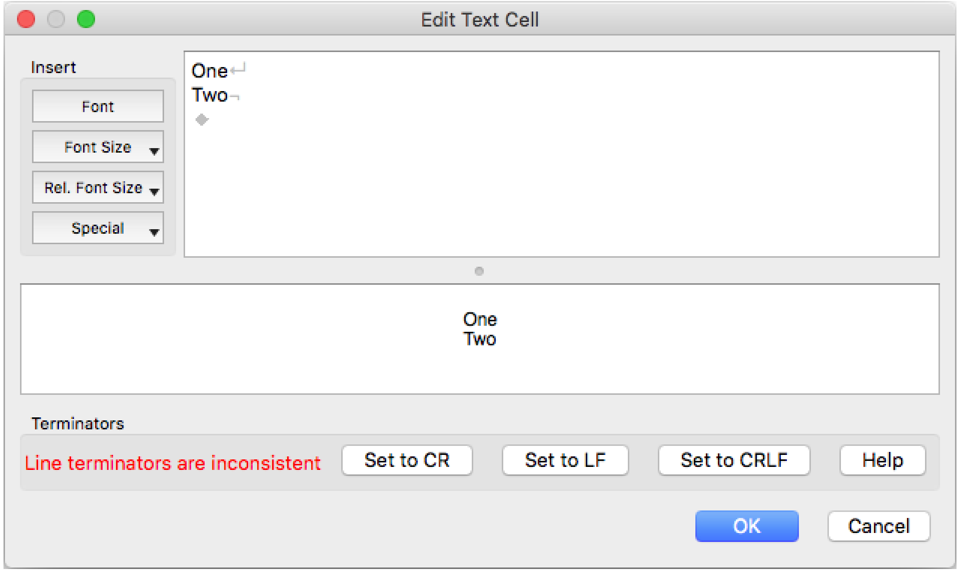
In rare cases, you may want to store text containing control characters such as tabs and terminators (carriage-returns and linefeeds) in text waves. You can’t enter such characters in the table entry line, but you can enter them using the Edit Text Cell dialog. To display the dialog, click the icon at the right end of the table entry line. This icon appears only when you are editing an existing text wave.

You can display whitespace characters (tabs, spaces, and terminators) in the Edit Text Cell dialog by right-clicking in the text entry area and choosing Settings→Show Whitespace. This is recommended for editing multi-line text. Tabs and terminators are identified with these symbols:

→	Tab
↵	Carriage-return (CR) For historical reasons, CR is the Igor-standard line terminator. Text for use within Igor should have CR terminators.

¬	Linefeed (LF) LF is the Unix-standard line terminator. Text for use with Unix programs should have LF terminators.
¶	Carriage-return/linefeed (CRLF) CRLF is the Windows-standard line terminator. Text for use with Windows programs should have CRLF terminators.

When the dialog's text entry area includes one or more terminators, buttons appear that allow you to change terminators:


Edit Text Cell

```
One↵
Two¬
```

One
Two

Terminators

Line terminators are inconsistent

You can also enter control characters by executing a command. For example:

```
textWave0[0] = "Hello\tGoodbye"      // Text with tab character
```

Use \t for tab, \r for carriage-return, and \n for linefeed.

You can examine what is in a text wave by printing it from the command line:

```
Print textWave0
```

This displays tabs, carriage-returns and linefeeds using escape sequences.

Editing Invalid Text

Some patterns of bytes are invalid in some text encodings. For example, this command creates a UTF-8 text wave with invalid text:

```
// "\xFE" represents a single byte with value 0xFE
Make/O/T test = {"A", "\xFE", "C"}
```

Point 1 of the wave is invalid because, in UTF-8, any byte outside the range 0x00..0x7F must be part of a multi-byte character. Invalid bytes are displayed in table cells using the Unicode replacement character.

The most likely way for this situation to arise is if you have a text wave containing MacRoman, Windows-1252 or Shift JIS text but the wave's text encoding is mistakenly set to UTF-8. In this case, you can either edit the wave to remove the invalid text or correct Igor's notion of the wave's text encoding using Misc→Text Encodings→Set Wave Text Encoding. In this section we assume that you want to edit the wave.

If you click a cell containing invalid text, Igor displays a yellow warning icon. If you attempt to edit the cell contents, Igor displays a warning dialog.

When the yellow warning icon is visible, the entry line works in a special mode. Each invalid byte in the cell is represented by a `\x` escape code. In the example above, if you click the cell representing point 1 of the wave, Igor displays `\xFE` in the entry area.

While editing the text, in most cases you will want to remove the escape sequences.

If you accept the text in the entry area, Igor converts any remaining `\x` escape sequences back to the bytes which they represent. In the example above, assuming that you started the editing process, made no changes in the entry line, and accepted the text, the contents of the cell would be the same after editing as they were before.

This use of escape sequences in tables applies only when the yellow warning icon is visible, indicating that the cell contained invalid text when you selected it. It applies only to `\x` escape sequences, not to other escape sequences such as `\t` (tab), `\r` (carriage-return) or `\n` (linefeed). If the yellow warning icon is not visible, there is no special treatment of escape sequences.

For expert debugging purposes, you can get a hex dump of text wave and dimension label cell contents by double-clicking the yellow warning icon. You can also get a hex dump whether the warning icon is visible or not by pressing Cmd (*Macintosh*) or Ctrl (*Windows*) and double-clicking a cell. The hex dump is printed in the history area of the command window.

See also Chapter III-16, **Text Encodings**, **Wave Text Encodings** on page III-472.

Editing Control Characters

Control characters are special non-printing characters created in olden times for controlling teletype machines and similar equipment. The ASCII codes for control characters, expressed as hexadecimal, fall in the range 0x00 to 0x1F, except for 0x7F which is the delete control character.

Other than tab (0x09), CR (carriage-return-0x0D) and LF (linefeed-0x0A), control characters have little use and rarely appear in modern computer text. When they do appear, it is often the result of an error or bug. Null (0x00) is used internally in many programs to mean “end-of-string” but should not appear in text documents.

In nearly all computer fonts, control characters are displayed as blanks. This makes it difficult to recognize and edit them, if they happen to appear in text whether on purpose or by error.

When Igor displays a control character in a table cell, it displays a “stand-in” symbol so that you can see what is there instead of seeing a blank. Here are some examples:

Character	ASCII Code	Stand-in Symbol (Unicode)
Tab	0x09	→ (U+2192)
CR	0x0D	↵ (U+21B5)
LF	0x0A	¬ (U+00AC)
CRLF	0x0D, 0x0A	¶ (U+00B6)
Null	0x00	␀ (U+2400)
Escape	0x1B	␛ (U+241B)

If you click a cell containing a control character, Igor displays a yellow warning icon. If you attempt to edit the cell contents, Igor displays a warning dialog.

When the yellow warning icon is visible, the entry line works in a special mode. If you edit and accept the text in the entry line, Igor converts any remaining stand-in symbols back to the control characters which they represent.

This use of stand-in symbols in tables applies only when the yellow warning icon is visible. If the yellow warning icon is not visible, there is no special treatment of these symbols.

You can disable use of stand-in symbols by unchecking the Use Special Symbols for Control Characters checkbox in the Tables section of the Miscellaneous Settings dialog. Except in rare cases, you should leave that setting checked.

Entering Special Characters

While editing a cell, you can enter special characters, such as Greek letters and math symbols, by choosing Edit→Characters to enter commonly-used characters such as Greek letters and math symbols or Edit→Special Characters to enter other characters.

If the wave whose element you are editing uses the UTF-8 text encoding, which is the default for waves created in Igor7 or later, then you can enter any character.

If the wave uses another text encoding then it is not possible to represent all characters. For example, the triple integral character, U+222D, can not be represented in MacRoman, Windows-1252, Shift JIS, or any other non-Unicode text encoding.

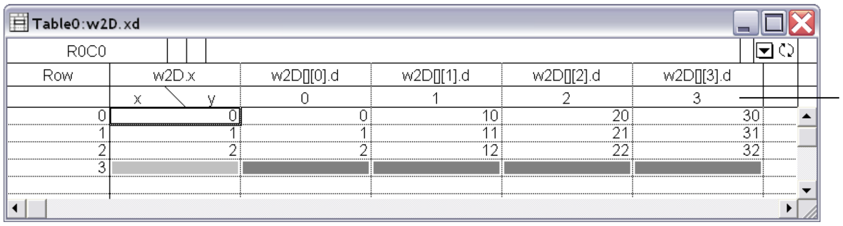
If you attempt to enter a character that can not be represented in the wave's text encoding, Igor displays an alert informing you of the problem. Your options are to omit that character or to convert the wave to UTF-8, using Misc→Text Encoding→Set Wave Text Encoding.

See also Chapter III-16, **Text Encodings**, **Wave Text Encodings** on page III-472.

Editing Multidimensional Waves

If you view a multidimensional wave in a table, Igor adds some items to the table that are not present for 1D waves. To see this, execute the following commands which create and display a 2D wave:

```
Make/O/N=(3,4) w2D = p + 10*q; Edit w2D.id
```



Row	w2D.x	w2D[][0].d	w2D[][1].d	w2D[][2].d	w2D[][3].d
	x y	0	1	2	3
0	0	0	10	20	30
1	1	1	11	21	31
2	2	2	12	22	32
3					

Horizontal index row.

The first column in the table is labeled Row, indicating that it shows row numbers. The second column contains the scaled row indices, which in this case are the same as the wave row numbers. The remaining columns show the wave data. Notice the name at the top of the first column of wave data: “w2D[][0].d”. The “w2D” identifies the wave. The “.d” specifies that the column shows wave data rather than wave indices. The “[][0]” identifies the part of the wave shown by this column. The “[]” means “all rows” and the “[0]” means column 0. This is derived from the syntax that you would use from Igor’s command line to store values into all rows of column 0 of the wave:

```
w2D[][0] = 123        // Set all rows of column 0 to 123
```

When displaying a multidimensional wave in a table, Igor adds a row to the table below the row of names. This row is called the horizontal index row. It can display either the scaled indices or the dimension labels for the wave elements shown in the columns below.

By default, if you view a 2D wave in a table and append the wave’s index column, Igor displays the wave’s row indices in a column to the left of the wave data and displays the wave’s column indices in the horizontal index row, above the wave data.

If you append the wave's dimension label column, Igor displays the wave's row labels in a column to the left of the wave data and displays the wave's column labels in the horizontal index row, above the wave data.

If you display neither index columns nor dimension labels, Igor still displays the wave's column indices in the horizontal index row.

If you want to show numeric indices horizontally and dimension labels vertically or vice versa, you can use the Table→Horizontal Index submenu to override the default behavior. For example, if you want dimension labels vertically and numeric indices horizontally, append the wave's dimension label column to the table and then choose Table→Horizontal Index→Numeric Indices.

In the example above, the row and column indices are equal to the row and column element numbers. You can set the row and column scaling of your 2D data to reflect the nature of your data. For example, if the data is an image with 5 mm resolution in both dimensions, you should use the following commands:

```
SetScale/P x 0, .005, "m", w2D; SetScale/P y 0, .005, "m", w2D
```

When you do this, you will notice that the row and column indices in the table reflect the scaling that you have specified.

1D waves have no column indices. Therefore, if you have no multidimensional waves in the table, Igor does not display the horizontal index row. If you have a mix of 1D and multidimensional waves in the table, Igor does display the horizontal index row but displays nothing in that row for the 1D waves.

When showing 1D data, Igor displays a column of point numbers at the left side of the table. This is called the Point column. When showing a 2D wave, Igor titles the column Row or Column depending on how you are viewing the 2D wave. If you have 3D or 4D waves in the table, Igor titles the column Row, Column, Layer or Chunk, depending on how you are viewing the waves.

It is possible to display a mix of 1D waves and multidimensional waves such that none of these titles is appropriate. For example, you could display two 2D waves, one with the wave rows shown vertically (the normal case) and one with the wave rows shown horizontally. In this case, Igor will title the Point column "Element".

You can edit dimension labels in the main body of the table by merely clicking in the cell and typing. However, you can't edit dimension labels in the horizontal index row this way. Instead you must double-click a label in this row. Igor then displays a dialog into which you can enter a dimension label. You can also set dimension labels using the SetDimLabel operation from the command line.

Changing the View of the Data

A table can display waves of dimension 1 through 4. A one dimensional wave appears as a simple column of numbers, or, if the wave is complex, as two columns, one real and one imaginary. A two dimensional wave appears as a matrix. In the one and two dimensional cases, you can see all of the wave data at once.

If you display a three dimensional wave in a table, you can view and edit only one slice at a time. To see this, execute the following commands:

```
Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D.id
```

Initially you see the slice of the wave whose layer index is 0 — layer zero of the wave. You can change which layer you are viewing using the next layer and previous layer icons that appear in the top/right corner of the table or using keyboard shortcuts.

By analogy with the up-arrow and down-arrow keys as applied to the row dimension of a 1D wave, "up" means "previous layer" (lower index) and "down" means "next layer" (higher index).

To change the layer from the keyboard, press Option-Up Arrow (*Macintosh*) or Alt+Up Arrow (*Windows*) to view the previous layer and Option-Down Arrow (*Macintosh*) or Alt+Down Arrow (*Windows*) to view the next layer.

Pressing the Shift key reverses the direction for both the icons and the arrow keys.

If you know the index of the layer you want to view you can enter it directly in a dialog by pressing Command (*Macintosh*) or Ctrl (*Windows*) while clicking the next or previous layer icon.

If you display a four dimensional wave in a table, you can still view and edit only one layer at a time. You can change which layer you are viewing by using the icons or keyboard shortcuts described above. To view the previous chunk in the 4D wave, press Option (*Macintosh*) or Alt (*Windows*) while clicking the up icon or press Command-Option-Up Arrow (*Macintosh*) or Ctrl+Alt+Up Arrow (*Windows*). To view the next chunk, press Option or Alt while clicking the down icon or press Command-Option-Down Arrow or Ctrl+Alt+Down Arrow.

In addition to using the keyboard shortcuts, you can specify the layer and chunk that you want to use using the ModifyTable elements operation.

Changing the Viewed Dimensions

When you initially view a 3D wave in a table, Igor shows a slice of the wave in the rows-columns plane. The wave's rows dimension is mapped to the table's vertical dimension and the wave's columns dimension is mapped to the table's horizontal dimension.

Using the Choose Dimensions icon, which appears to the right of the layer icons, you can instruct Igor to map any wave dimension to the vertical table dimension and any other wave dimension to the horizontal table dimension. This is primarily of interest if you work with waves of dimension three or higher because you can view and edit any orthogonal plane in the wave. You can, for example, create a new wave that contains a slice of data from the 3D wave.

When you click the Choose Dimensions icon, Igor displays the Choose Dimensions dialog. This dialog allows you to specify how the dimensions in the wave are to be displayed in the table.

ModifyTable Elements Command

This section discusses choosing viewed dimensions using the ModifyTable "elements" keyword. You do not need to know about this unless you want a thorough understand of editing 3D and 4D waves and viewing them from different perspectives.

The best way to understand this section is to execute all of the commands shown.

Igor needs to know which wave dimension to map to the vertical table dimension and which wave dimension to map to the horizontal table dimension. In addition, for waves of dimension three or higher, Igor needs to know which element of the remaining dimensions to display.

The form of the command is:

```
ModifyTable elements(<wave name>) = (<row>,<column>,<layer>,<chunk>)
```

The parameters specify which element of the wave's rows, columns, layers and chunks dimensions you want to view. The value of each parameter may be an element number (0 or greater) or it may be a special value. There are three special values that you can use for any of the parameters:

-1	Means no change from the current value.
-2	Means map this dimension to the table's vertical dimension.
-3	Means map this dimension to the table's horizontal dimension.

In reading the following discussion, remember that the first parameter specifies how you want to view the wave's rows, the second parameter specifies how you want to view the wave's columns, the third parameter specifies how you want to view the wave's layers and the fourth parameter specifies how you want to view the wave's chunks.

If you omit a parameter, it takes the default value of -1 (no change). Thus, if you are dealing with a 2D wave, you can supply only the first two parameters and omit the last two.

To get a feel for this command, let’s start with the example of a simple matrix, which is a 2D wave.

```
Make/O/N=(3,3) wave0 = p + 10*q
Edit wave0.id
```

As you look down in the table, you see the rows of the matrix and as you look across the table, you see its columns. Thus, initially, the rows dimension is mapped to the vertical table dimension in the and the columns dimension is mapped to the horizontal table dimension. This is the default mapping. You can change this with the following command:

```
ModifyTable elements(wave0) = (-3, -2)
```

The first parameter specifies how you want to view the wave’s rows and the second parameter specifies how you want to view the wave’s columns. Since the wave has only two dimensions, the third and fourth parameters can be omitted.

The -3 in this example maps the wave’s rows to the table’s horizontal dimension. The -2 maps the wave’s columns to the table’s vertical dimension.

You can return the wave to its default view using:

```
ModifyTable elements(wave0) = (-2, -3)
```

When you consider a 3D wave, things get a bit more complex. In addition to the rows and columns dimensions, there is a third dimension — the layers dimension. When you initially create a table containing a 3D wave, it shows all of the rows and columns of layer 0 of the wave. Thus, as with the 2D wave, the rows dimension is mapped to the vertical table dimension and the columns dimension is mapped to the horizontal table dimension. You can control which layer of the 3D wave is displayed in the table using the icons and keyboard shortcuts described above, or using the ModifyTable elements keyword.

For example:

```
Make/O/N=(5,4,3) wave0 = p + 10*q + 100*r
ModifyTable elements(wave0)=(-2, -3, 1)   //Shows layer 1 of 3D wave
ModifyTable elements(wave0)=(-2, -3, 2)   //Shows layer 2 of 3D wave
```

In these examples, the wave’s layers dimension is fixed to a specific value whereas the wave’s rows and columns dimensions change as you look down or across the table. The term “free dimension” refers to a wave dimension that is mapped to either of the table’s dimensions. The term “fixed dimension” refers to a wave dimension for which you have chosen a fixed value.

In the preceding example, we viewed a slice of the 3D wave in the rows-columns plane. We can view any orthogonal plane. For example, this command shows us the data in the layers-rows plane:

```
ModifyTable elements=(-3, 0, -2)    // Shows column 0 of 3D wave
```

The first parameter says that we want to map the wave’s rows dimension to the table’s horizontal dimension. The second parameter says that we want to see column 0 of the wave. The third parameter says that we want to map the wave’s layers dimension to the table’s vertical dimension.

Dealing with a 4D wave is similar to the 3D case, except that, in addition to the two free dimensions, you have two fixed dimension.

```
Make/O/N=(5,4,3,2) wave0 = p + 10*q + 100*r + 1000*s
ModifyTable elements(wave0)=(-2, -3, 1, 0)   //Shows layer 1/chunk 0
ModifyTable elements(wave0)=(-2, -3, 2, 1)   //Shows layer 2/chunk 1
```

If you change a wave (using Make/O or Redimension) such that one or both of the free dimensions has zero points, Igor automatically resets the view to the default — the wave’s rows dimension mapped to the table’s vertical dimension and the wave’s column dimension mapped to the table’s horizontal dimension. Here is an example:

```
Make/O/N=(5,4,3) wave0 = p + 10*q + 100*r
Edit wave0.id
```

```
Modify elements(wave0) = (0, -2, -3)// Map layers to horizontal dim
Redimension/N=(5,4) wave0           // Eliminate layers dimension!
```

This last command has eliminated the wave dimension that is mapped to the table horizontal dimension. Thus, Igor will automatically reset the table view.

If you use a dimension with zero points as a free dimension, Igor will also reset the view to the default:

```
Make/O/N=(3,3) wave0 = p + 10*q
Edit wave0.id
Modify elements(wave0) = (0, -2, -3)   // Map layers to horizontal dim
```

This last command maps the wave's layers dimension to the table's horizontal dimension. However, the wave has no layers dimension, so Igor will reset the view to the default.

The initial discussion of changing the view using keyboard shortcuts was incomplete for the sake of simplicity. It said that Option-Down Arrow (*Macintosh*) or Alt+Down Arrow (*Windows*) displayed the next layer and that Command-Option-Down Arrow or Ctrl+Alt+Down Arrow displayed the next chunk. This is true if rows and columns are the free dimensions. A more general statement is that Option-Down Arrow or Alt+Down Arrow changes the viewed element of the first fixed dimension and Command-Option-Down Arrow or Ctrl+Alt+Down Arrow changes the viewed element of the second fixed dimension. Here is an example using a 4D wave:

```
Make/O/N=(5,4,3,2) wave0 = p + 10*q + 100*r + 1000*s
Edit wave0.id
ModifyTable elements(wave0)=(0, -2, 0, -3)
```

The ModifyTable command specifies that the columns and chunks dimensions are the free dimensions. The rows and layers dimensions are fixed at 0. If you now press Option-Down Arrow or Alt+Down Arrow, you change the element of the first fixed dimension — the rows dimension in this case. If you press Command-Down Arrow or Ctrl+Alt+Down Arrow, you change the element of the second fixed dimension — the layers dimension in this case.

Multidimensional Copy/Cut/Paste/Clear

The material in this section is primarily of interest if you intend to edit 3D and 4D data in a table. There are also a few items that deal with 2D data.

When you copy table cells, Igor copies the data in two formats:

- Igor binary, for pasting into another Igor wave
- Plain text, for pasting into another program

For 1D and 2D waves, the subset that you copy and paste is obvious from the table selection. In the case of 3D and 4D waves, you can only see two dimensions in the table at one time and you can select a subset of those two dimensions.

When copying as plain text, Igor always copies just the visible cells to the clipboard.

When copying as Igor binary, if you press Option (*Macintosh*) or Alt (*Windows*), Igor copies cells from all dimensions.

Consider the following example. We make a wave with 5 rows, 4 columns and 3 layers and display it in a table:

```
Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D.id
```

The table now displays all rows and all columns of layer 0 of the wave. If you select all of the visible data cells and do a copy, Igor copies all of layer 0 to the clipboard. However, if you do an Option-copy or Alt-copy, Igor copies all three layers to the clipboard — layer 0 plus the two layers that are currently not visible — in Igor binary format.

This table shows the effect of the Option (*Macintosh*) or Alt (*Windows*) key.

	Copy	**Paste**	**Clear**
No modifiers	Copies visible	Replace-pastes visible	Clears visible
Option or Alt	Copies all	Replace-pastes all	Clears all

The middle column of the preceding table mentions “replace-pasting”. When you do a paste, Igor normally replaces the selection in the table with the data in the clipboard. However, if you press Shift while pasting, Igor inserts the data as new cells in the table. This is called an “insert-paste”. This table shows the effect of the Shift and Option (*Macintosh*) or Alt (*Windows*) keys on a paste.

	Paste
No modifiers	Replace-pastes visible
Option	Replace-pastes all
Shift	Insert-pastes visible
Shift and Option or Alt	Insert-pastes all

Replace-Paste of Multidimensional Data

If you copy data in a table, and then select cells in the table or in another table, and do a paste, you are doing a replace-paste. The copied data replaces the selection when you do the paste.

For 1D and 2D waves, the subset that you copy and paste is obvious from the table selection. In the case of 3D and 4D waves, you can only see two dimensions in the table at one time and you can select a subset of those two dimensions.

When you do a normal replace-paste involving waves of dimension 3 or higher, the data in the clipboard replaces the data in the currently visible slice of the selected wave.

Copying a Layer of a 3D Wave to Another Layer

Here is an example that illustrates replace-paste.

1. Make a wave with 5 rows, 4 columns and 3 layers and display it in a table:

   ```
   Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D
   ```

   The table now displays all rows and all columns of layer 0 of the wave. Let’s look at layer 1 of the wave.
2. Press Option-Down Arrow (*Macintosh*) or Alt+Down Arrow (*Windows*) while the table is active.

   This changes the view to show layer 1 instead of layer 0.
3. Select all of the visible cells and choose Edit→Copy.

   This copies all of layer 1 to the clipboard.
4. Press Option-Down Arrow or Alt+Down Arrow again to view layer 2 of the wave.
5. With all of the cells still selected, choose Edit→Paste.

   The data copied from layer 1 replaces the data in layer 2 of the wave.
6. Choose Edit→Undo.

   Layer 2 is restored to its original state.
7. Press Option-Up Arrow or Alt+Up Arrow two times.

   We are now looking at layer 0.
8. Choose Edit→Paste.

The data that we copied from layer 1 replaces the data in layer 0.

9. Choose Edit→Undo to return layer 0 to the original state.

Copying and Pasting All Data of a 3D Wave

Now let's consider an example in which we copy and paste all of the wave data, not just one layer.

1. Make a wave with 5 rows, 4 columns and 3 layers and display it in a table:

   ```
   Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D
   ```

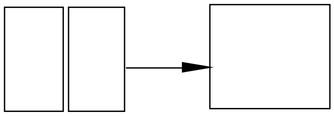
2. Select all of the visible cells.
3. Press Option (*Macintosh*) or Alt (*Window*) and choose Edit→Copy All Layers.

   This copies the entire wave, all three layers, to the clipboard.
4. Press Option or Alt and choose Edit→Clear All Layers.

   This clears all three layers.
5. Use Option-Down Arrow or Alt+Down Arrow and Option-Up Arrow or Alt+Up Arrow to verify that all three layers were cleared.
6. Press Option or Alt and choose Edit→Paste All Layers.

   This pastes all three layers from the clipboard into the selected wave.
7. Use Option-Down Arrow and Option-Up Arrow or Alt+Down Arrow and Alt+Up Arrow to verify that all three layers were pasted.

Making a 2D Wave from Two 1D Waves



In this example, we make a 2D wave from two 1D waves. Execute:

```
Make/O/N=5 w1DA=p, w1DB=100+p; Edit w1DA, w1DB
```

1. Select all of the first 1D wave and choose Edit→Copy.
2. Click in the first unused cell in the table and choose Edit→Paste.

   Because you pasted into the unused cell, Igor created a new wave. This is a "create-paste".
3. Choose Redimension w1DA1 from the Table pop-up menu (gear icon).
4. In the Redimension Waves dialog, enter 2 for the number of columns and click Do It.
5. Right-click the column name of the redimensioned wave and choose Rename.
6. Rename w1DA1 as w2D.
7. Select all of the second 1D wave, w1DB, and choose Edit→Copy.
8. Select all of the second column of the 2D wave, w2D, and choose Edit→Paste.

We now have a 2D wave generated from two 1D waves.

Insert-Paste of Multidimensional Data

If you copy data in a table, and then select in the table or in another table, and do a paste while pressing Shift, you are doing an insert-paste. The copied data is inserted into the selected wave or waves before the selected cells. As in the case of the replace-paste, the insert-paste works on just the visible layer of data if Option (*Macintosh*) or Alt (*Windows*) is not pressed or on all layers if the Option or Alt is pressed. Next is an example that illustrates this.

Inserting New Rows in a 3D Wave

1. Make a wave with 5 rows, 4 columns and 3 layers and display it in a table:

   ```
   Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D
   ```

2. Select all of the cells in rows 1 and 2 of the table.

   An easy way to do this is to click the "1" in the Row column and drag down to the "2".

3. Choose Edit→Copy to copy the selected cells.

   Since you did not press Option or Alt, this copies just the visible layer.
4. Press shift and choose Edit→Insert Paste to insert-paste the copied data.

   Notice that two new rows were inserted, pushing the pre-existing rows down.
5. Press Option-Down Arrow or Alt+Down Arrow to see what was inserted in layers 1 and 2 of the 3D wave.

   Notice that zeros were inserted. This is because the paste stored data only in the visible layer.
6. Press Option-Up Arrow or Alt+Up Arrow to view layer 0 again.
7. Choose Edit→Undo from the Edit menu to undo the paste.
8. Use Option-Down Arrow or Alt+Down Arrow to check the other layers of the wave and then use Option-Up Arrow or Alt+Up Arrow to come back to layer zero.

   The wave is back in its original state.

   Now we will do an insert-paste in all layers.
9. Select rows 1 and 2 of the wave.
10. Press Option or Alt and choose Edit→Copy All Layers.

    This copies data from all three layers to the clipboard.
11. Press Shift-Option or Shift+Alt and choose Edit→Insert Paste All Layers.

    This pastes data from the clipboard into all three layers of the wave. By pressing Shift, we did an insert-paste rather than a replace-paste and by pressing Option or Alt, we pasted into all layers, not just the visible layer.
12. Use Option-Down Arrow or Alt+Down Arrow to verify that we have pasted data into all layers.

Cutting and Pasting Rows Versus Columns

Normally, you cut and paste rows of data. However, there may be cases where you want to cut and paste columns. For example, if you have a 2D wave with 5 rows and 3 columns, you may want to cut the middle column from the wave. Here is how Igor determines if you want to cut rows or columns.

If the selected wave is 2D or higher, *and* if one or more *entire* columns is selected, Igor cuts the selected column or columns. In all other cases Igor cuts rows.

After copying or cutting wave data, you have data in the clipboard. Normally, a paste overwrites the selected rows or inserts new rows (if you press Shift).

To insert columns, you need to do the following:

1. Copy the column that you want to insert.
2. Select exactly one entire column. You can do this quickly by clicking the column name.
3. Press Shift and choose Edit→Insert Paste or press Command-Shift-V (*Macintosh*) or Ctrl+Shift+V (*Windows*).

If the wave data is real (not complex), Igor normally pastes the new column or columns before the selected column. This behavior would provide you with no way to paste columns after the last column of a wave. Therefore, if the selected column is the last column, Igor presents a dialog to ask you if you want to paste the new columns before or after the last column.

If the wave data is complex, Igor pastes the new columns before the selected column if the selected column is real or after the selected column if it is imaginary.

If you select more than one column or if you do not select all of the column, Igor will insert rows instead of columns.

Create-Paste of Multidimensional Data

When you copy data in a table and then select the first unused cell in the table and then do a paste, Igor creates one or more new waves.

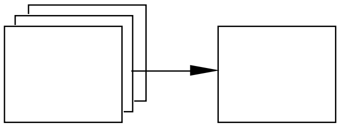
The number of waves created and the number of dimensions in each wave are the same as for the copied data. Igor also copies and pastes the following wave properties:

- Data units and dimension units
- Data full scale and dimension scaling
- Dimension labels
- The wave note

Igor copies and pastes the wave note only if you copy the entire wave. If you copy part of the wave, it does not copy the wave note.

You can use a create-paste to create a copy of a subset of the data displayed in the table. For 1D and 2D waves, the subset that you copy and paste is obvious from the table selection. In the case of 3 and higher dimension waves, you can only see two dimensions in the table at one time and you can choose a subset of those two dimensions. If you do a copy and then a create-paste, Igor creates a new 2D wave containing just the data that was visible when you did the copy. If you do an Option-copy (*Macintosh*) or Alt-copy (*Windows*) to copy all and then a create-paste, Igor creates a new wave with the same number of dimensions and same data as what you copied.

Making a 2D Wave from a Slice of a 3D Wave



1. Make a 3D wave and display it in a table:

```
Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D
```

2. Select all of the cells of the 3D wave and choose Edit→Copy.
3. Click in the first unused cell and choose Edit→Paste.

You now have a 2D wave consisting of the data from layer 0 of the 3D wave.

Making a 3D Wave from a 3D Wave

1. Make a 3D wave and display it in a table:

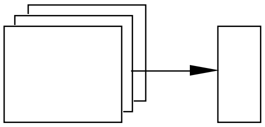
```
Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D
```

2. Select all of the cells of the 3D wave.
3. Press Option or Alt, and choose Edit→Copy All Layers.
4. Click in the first unused cell and choose Edit→Paste.

   You now have a 3D wave consisting of the data from all layer2 of the original 3D wave.

   To confirm this, we will inspect all layers of the new wave.
5. To demonstrate this, view the other layers of the new wave by pressing Option-Down Arrow or Alt+Down Arrow (to go to the next layer) and Option-Up Arrow or Alt+Up Arrow (to go to the previous layer).

Making a 1D Wave from a Column of a 3D Wave



1. Make a 3D wave and display it in a table:

```
Make/O/N=(5,4,3) w3D = p + 10*q + 100*r; Edit w3D
```

2. Select a single column of the 3D wave and choose Edit→Copy.
3. Click the first unused cell in the table and choose Edit→Paste.

   You have created a single-column wave but it is a 2D wave, not a 1D wave.
4. Choose Redimension w3D1 from the Table pop-up menu (gear icon) to display the Redimension Waves dialog.

5. Enter 0 for the number of columns and click Do It.

   You now have a 1D wave created from a column of a 3D wave.

Save Table Copy

You can save the active table as an Igor packed experiment file or as a tab or comma-delimited text file by choosing File→Save Table Copy.

The main uses for saving as a packed experiment are to save an archival copy of data or to prepare to merge data from multiple experiments (see **Merging Experiments** on page II-19). The resulting experiment file preserves the data folder hierarchy of the waves displayed in the table starting from the “top” data folder, which is the data folder that encloses all waves displayed in the table. The top data folder becomes the root data folder of the resulting experiment file. Only the table and its waves are saved in the packed experiment file, not variables or strings or any other objects in the experiment.

Save Table Copy does not know about dependencies. If a table contains a wave, wave0, that is dependent on another wave, wave1 which is not in the table, Save Table Copy will save wave0 but not wave1. When the saved experiment is open, there will be a broken dependency.

The main use for saving as a tab or comma-delimited text file is for exporting data to another program.

The point column is never saved.

When saving as text, the data format matches the format shown in the table. This causes truncation if the underlying data has more precision than shown in the table.

To save data as text with full precision, choose Data→Save Waves→Save Delimited Text or use the **SaveTableCopy** operation with /F=1.

When saving 3D and 4D waves as text, only the visible layer is saved. To save the entirety of a 3D or 4D wave, choose Data→Save Waves→Save Delimited Text.

The **SaveTableCopy** provides options that are not available using the Save Table Copy menu command.

Object Reference Waves in Tables

This topic is for advanced users.

Object reference waves are waves containing wave references (WAVE waves) or data folder references (DFREF waves).

Object reference wave elements are formatted as hexadecimal by default. See **Object Reference Wave Formatting** on page II-258 for details.

Because entering an invalid reference may cause a crash, you can not modify the elements of an object reference wave by editing in the table.

If you display an object reference wave in a table, turn column info tags on via the Table menu, and hover the mouse over an element of the object reference wave, Igor displays in the column info tag information about the wave or data folder referenced by the element. This feature works in the debugger if you have column info tags turned on.

Editing Waves Referenced by WAVE Waves

This is a feature for advanced users.

If you have a WAVE wave displayed in table, you can select one or more cells of that wave, right-click, and choose Edit Waves. Igor creates a new table showing the waves referenced by the selected wave references in the table.

You can do the same using the table popup menu (gear icon in top/right corner of the table). The Edit Waves item does not appear in the Table menu in the main menu bar.

The Edit Waves item appears in the contextual menu or table popup menu only if one or more editable wave references are selected. NULL waves (wave reference=0) and free waves can not be edited in a table.

This feature is not available in the table subwindow in the debugger.

Printing Tables

Before printing a table you should bring the table to the top of the desktop and set the page size and orientation using the Page Setup dialog. Choose the Page Setup for All Tables item from the Files menu.

In each experiment, Igor stores one page setup for all tables. Thus, changing the page setup while a table is active changes the page setup for all tables in the current experiment.

When you invoke the Page Setup dialog you must make sure that the table that you want to print is the top window. Changing the page setup for graphs, page layouts or other windows does not affect the page setup for tables.

You can print all or part of a table. To print the whole table, select just one cell and choose File-Print Table. To print the selection, select multiple cells and choose File→Print Table Selection.

Exporting Tables as Graphics

Although Igor tables are intended primarily for editing data, you can also use them for presentation purposes. You can put a table into an Igor page layout, as discussed in Chapter II-18, **Page Layouts**. This section deals with exporting a table to other applications as a picture.

Typically you would do this if you are preparing a report in a word processor or page layout program or making an illustration in a drawing program. If you are exporting to a program that has strong text formatting features, it may be better to copy the data from the table as text, using the Copy item in the Edit menu. You can paste the text into the other program and then format it as you wish.

Exporting a Table as a Picture

To export a table as a Macintosh picture via the clipboard, choose Export Graphics from the Edit menu. This copies the table to the clipboard as a picture.

The picture that Igor puts into the clipboard contains just the visible cells in the table window. You can scroll, expand or shrink the window to control which cells will appear in the picture.

Igor can write a picture out to a file instead of copying it to the clipboard. To do this, use the Save Graphics submenu of the File menu.

Most word processing, drawing and page layout programs permit you to import a picture via the clipboard or via a file.

Although we have not optimized tables for presentation purposes, we did put two features into tables specifically for presentation. First, you can hide the point column by setting its width to zero. Second, you can replace the automatic column titles with column titles of your own. Use the Modify Columns dialog for both of these.

There are some features lacking that would be nice for presentation. You can’t change the background color of a table. You can’t change or remove the grid lines. If you want to do these things, export the table to a drawing program for sprucing up.

Table Preferences

Table preferences allow you to control what happens when you create a new table or add new columns to an existing table. To set preferences, create a table and set it up to your taste. We call this your *prototype* table. Then choose Capture Table Prefs from the Table menu.

Preferences are normally in effect only for *manual* operations, not for automatic operations from Igor procedures. This is discussed in more detail in **How to Use Preferences** on page III-516.

When you initially install Igor, all preferences are set to the factory defaults. The dialog indicates which preferences you have changed, and which are factory defaults.

The Window Position and Size preference affects the creation of new tables only.

The Column Styles preference affects the formatting of newly created tables and of columns added to an existing table. This preference stores column settings for the point column and for one additional column - the first column after the point column in the prototype table. When you create a new table or add columns to a table, these settings determine the formatting of the columns.

The page setup preference affects what happens when you create a new *experiment*, not when you create a new *table*. Here is why.

Each experiment stores *one* page setup for *all* tables in that experiment. The preferences also store one page setup for tables. When you set the preferred page setup for tables, Igor stores a copy of the current experiment's page setup for tables in the preferences file. When you create a new experiment, Igor stores a copy of the preferred page setup for tables in the experiment.

Table Style Macros

The purpose of a table style macro is to allow you to create a number of tables with the same stylistic properties. Using the Window Control dialog, you can instruct Igor to automatically generate a style macro from a prototype table. You can then apply the macro to other tables.

Igor can generate style macros for graphs, tables and page layouts. However, their usefulness is mainly for graphs. See **Graph Style Macros** on page II-350. The principles explained there apply to table style macros also.

Table Shortcuts

To view table keyboard navigation shortcuts, see **Table Keyboard Navigation** on page II-237.

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To add columns for existing waves	Choose Append to Table from the Table menu.	Choose Append to Table from the Table menu.
To create new numeric waves	Click in the first unused column and enter a number or paste numeric data from the clipboard.	Click in the first unused column and enter a number or paste numeric data from the clipboard.
	Copy all or part of an existing numeric wave and paste in the first unused column.	Copy all or part of an existing numeric wave and paste in the first unused column.
To create new text waves	Click in the first unused column and enter nonnumeric text or paste text data from the clipboard.	Click in the first unused column and enter nonnumeric text or paste text data from the clipboard.
	Copy all or part of an existing text wave and paste in the first unused column.	Copy all or part of an existing text wave and paste in the first unused column.
To kill one or more waves	Select the waves in the table and choose Kill Waves from the Table pop-up menu.	Select the waves in the table and choose Kill Waves from the Table pop-up menu.
To rename a wave	Select one or more cells from the wave and choose Rename from the Table pop-up menu.	Select one or more cells from the wave and choose Rename from the Table pop-up menu.
To redimension a wave	Select one or more cells from the wave and choose Redimension from the Table pop-up menu.	Select one or more cells from the wave and choose Redimension from the Table pop-up menu.
To add cells to a column	Click in a row and choose Insert Points from the Table pop-up.	Click in a row and choose Insert Points from the Table pop-up.
	Press Command-Shift-V. This does an insert-paste.	Press Ctrl+Shift+V. This does an insert-paste.
	Click in the insertion cell at the end of the column and enter a value or paste.	Click in the insertion cell at the end of the column and enter a value or paste.
To remove cells from a column	Select the cells to be removed and choose Delete Points from the Table pop-up.	Select the cells to be removed and choose Delete Points from the Table pop-up.
	Press Command-X (Cut).	Press Ctrl+X (Cut).
To identify a particular column	Press Command-Option-Control and click in the column.	Press Shift-F1 to summon context-sensitive help and then click in the column.
To modify column styles	Double-click a column title (goes to Modify Columns dialog).	Double-click a column title (goes to Modify Columns dialog).
	Select one or more columns and click the Table pop-up menu.	Select one or more columns and click the Table pop-up menu.
	Press Control and click the column title.	Right-click the column title.
To modify the style of all columns at once	Press Shift while using the Table pop-up menu.	Press Shift while using the Table pop-up menu.
	Press Shift-Option to modify all but the Point column.	Press Alt+Shift to modify all but the Point column.

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To modify the style of a subset of the data columns of multidimensional wave	Select the columns to modify, press Command, then click the Table menu or the Table pop-up menu, then make a selection.	Select the columns to modify, press Ctrl, then click the Table menu or the Table pop-up menu, then make a selection.
To select entire rows	Drag in the point number column.	Drag in the point number column.
To select entire columns	Drag on the column titles.	Drag on the column titles.
To select all of the columns of a given wave	Press Command and click the column title of a data column from the wave. This selects all data columns from the wave. Press Command and click the column title of an index or label column from a wave to select that column plus all data columns.	Press Ctrl and click the column title of a data column from the wave. This selects all data columns from the wave. Press Ctrl and click the column title of an index or label column from a wave to select that column plus all data columns.
To change a wave's position	Press Option and drag the column title to the new position.	Press Alt and drag the column title to the new position.
To adjust the width of a column	Drag the border at the right of the column title. Press Shift to resize all columns except the point column. Press Command while clicking to set the width of a single data column of a multidimensional wave.	Drag the border at the right of the column title. Press Shift to resize all columns except the point column. Press Ctrl while clicking to set the width of a single data column of a multidimensional wave.
To bring the target cell into view	Click in the cell ID in the top left corner of the table.	Click in the cell ID in the top left corner of the table.

Chapter II-13

Graphs

Overview

Igor graphs are simultaneously:

- Publication quality presentations of data.
- Dynamic windows for exploratory data analysis

This chapter describes how to create and modify graphs, how to adjust graph features to your liking, and how to use graphs for data exploration. It deals mostly with general graph window properties and with waveform and XY plots.

These other chapters discuss material related to graphs:

Category Plots on page II-355, **Contour Plots** on page II-365, **Image Plots** on page II-385
3D Graphics on page II-405, **Drawing** on page III-61, **Annotations** on page III-33
Exporting Graphics (Macintosh) on page III-95, **Exporting Graphics (Windows)** on page III-101
Graphics Technology on page III-506

A single graph window can contain one or more of the following:

Waveform plots	Wave data versus X values (scaled point number)
XY plots	Y wave data versus X wave data
Category plots	Numeric wave data versus text wave data
Image plots	Display of a matrix of data
Contour plots	Contour of a matrix or an XYZ triple
Axes	Any number of axes positioned anywhere
Annotations	Textboxes, legends and dynamic tags
Cursors	To read out XY coordinates
Drawing elements	Arrows, lines, boxes, polygons, pictures …
Controls	Buttons, pop-up menus, readouts …

The various kinds of plots can be overlaid in the same plot area or displayed in separate regions of the graph. Igor also provides extensive control over stylistic factors such as font, color, line thickness, dash pattern, etc.

Graph Features

Igor graphs are smart. If you expand a graph to fill a large screen, Igor will adjust all aspects of the graph to optimize the presentation for the larger graph size. The font sizes will be scaled to sizes that look good for the large format and the graph margins will be optimized to maximize the data area without fouling up the axis labeling. If you shrink a graph down to a small size, Igor will automatically adjust axis ticking to prevent tick mark labels from running into one another. If Igor’s automatic adjustment of parameters does not give the desired effect, you can override the default behavior by providing explicit parameters.

Igor graphs are dynamic. When you zoom in on a detail in your data, or when your data changes, perhaps due to data transformation operations, Igor will automatically adjust both the tick mark labels and the axis labels. For example, before zooming in, an axis might be labeled in milli-Hertz and later in micro-Hertz. No matter what the axis range you select, Igor always maintains intelligent tick mark and axis labels.

If you change the values in a wave, any and all graphs containing that wave will automatically change to reflect the new values.

You can zoom in on a region of interest (see **Manual Scaling**), expand or shrink horizontally or vertically, and you can pan through your data with a hand tool (see **Panning**). You can offset graph traces by simply dragging them around on the screen (see **Trace Offsets**). You can attach cursors to your traces and view data readouts as you glide the cursors through your data (see **Info Panel and Cursors**). You can edit your data graphically (see **Drawing and Editing Waves** on page III-73).

Igor graphs are fast. They are updated almost instantly when you make a change to your data or to the graph. In fact, Igor graphs can be made to update in a nearly continuous fashion to provide a real-time oscilloscope-like display during data acquisition (see **Live Graphs and Oscilloscope Displays**)

You can also control virtually every detail of a graph. When you have the graph just the way you like it, you can create a template called a "style macro" to make it easy to create more graphs of the same style in the future (see **Graph Style Macros**). You can also set preferences from a reference graph so that new graphs will automatically be created with the settings you prefer (see **Graph Preferences**).

You can print or export graphs directly, or you can combine several graphs in a page layout window prior to printing or exporting. You can export graphs and page layouts in a wide variety of graphics formats.

A graph can exist as a standalone window or as a subwindow of another graph, a page layout, or a control panel (see **Embedding and Subwindows** on page III-79).

The Graph Menu

The Graph menu contains items that apply only to graph windows. The menu appears in the menu bar only when the active or target window is a graph.

When you choose an item from the Graph menu it affects the top-most graph.

Typing in Graphs

If you type on the keyboard while a graph is the top window, Igor brings the command window to the front and your typing goes into the command line. (The only exception to this is when a graph contains a selected SetVariable control.)

Graph Names

Every graph that you create has a window name which you can use to manipulate the graph from the command line or from a procedure. When you create a new graph, Igor assigns it a name of the form "Graph0", "Graph1" and so on. When you close a graph, Igor offers to create a window recreation macro which you can invoke later to recreate the graph. The name of the window recreation macro is the same as the name of the graph.

The graph *name* is not the same as the graph *title* which is the text that appears in the graph's window frame. The name is for use in procedures but the title is for display purposes only. You can change a graph's name and title using the Window Control dialog which you can access by choosing Windows→Control→Window Control.

Creating Graphs

You create a graph by choosing New Graph from the Windows menu.

You can also create a graph by choosing New Category Plot, New Contour Plot or New Image Plot from the New submenu in the Windows menu.

You select the waves to be displayed in the graph from the Y Waves list. The wave is displayed as a **trace** in the graph. A trace is a visual representation of a wave or an XY pair. By default a trace is drawn as a series of lines joining the points of the wave or XY pair.

Each trace has a name so you can refer to it from a procedure. By default, the trace name is he same as the wave name or Y wave in the case of an XY pair. However, there are exceptions. If you display the same wave multiple times in a given graph, the traces will have names like wave0, wave0#1, and wave0#2. wave0 is equivalent to wave0#0. Such names are called *trace instance names*.

You can also programmatically specify a trace's name using the **Display** or **AppendToGraph** operations. This is something an Igor programmer would do, typically to better distinguish multiple traces with the same Y wave.

Often the data values of the waves that you select in the Y Waves list are plotted versus their calculated X values. This is a **waveform trace**. The calculated X values are derived from the wave's X scaling; see **Waveform Model of Data** on page II-62.

If you want to plot the data values of the Y waves versus the data values of another wave, select the other wave in the X Wave list. This is an **XY trace**. In this case, X scaling is ignored; see **XY Model of Data** on page II-63.

If the lengths of the X and Y waves are not equal, then the number of points plotted is determined by the shorter of the waves.

The New Graph dialog has a simple mode and an advanced mode. In the simple mode, you can select multiple Y waves but just one X wave. If you have multiple XY pairs with distinct X waves, click the More Choices button to use the advanced mode. This allows you to select a different X wave for each Y wave.

You can specify a **title** for the new window. The title is not used except to form the title bar of the window. It is *not* used to identify windows and does not appear *in* the graph. If you specify no title, Igor will choose an appropriate title based on the traces in the graph and the graph name. Igor automatically assigns graph names of the form "Graph0". The name of a window is important because it is used to identify windows in commands. The title is for display purposes only and is not used in commands.

If you have created style macros for the current experiment they will appear in the Style pop-up menu. See **Graph Style Macros** on page II-350 for details.

Normally, the new graph is created using left and bottom axes. You can select other axes using the pop-up menus under the X and Y wave lists. Picking L=VertCrossing automatically selects B=HorizCrossing and vice versa. These **free axes** are used when you want to create a Cartesian type plot where the axes cross at (0,0).

You can create additional free axes by choosing New from the pop-up menu. This displays the New Free Axis dialog. Axes created this way are called "free axes" because they can be freely positioned nearly anywhere in the graph window. The standard left, bottom, right, and top axes always remain at the edge of the plot area.

You should give the new axis a name that describes its intended use. The name must be unique within the current graph and can't contain spaces or other nonalphanumeric characters. The Left and Right radio buttons determine the side of the axis on which tick mark labels will be placed. They also define the edge of the graph from which axis positioning is based.

You can create a blank graph window containing no traces or axes by clicking the Do It button without selecting any Y waves to plot. Blank graph windows are mostly used in programming when traces are appended later using **AppendToGraph**.

The New Graph dialog comes in two versions. The simpler version shown above is suitable for most purposes. If, however, you have multiple pairs of XY data or when you will be using more than one pair of axes, you can click the More Choices button to get a more complex version of the dialog.

Using the advanced mode of the New Graph dialog, you can create complex graphs in one step. You select a wave or an XY pair using the Y Waves and X Wave lists, and then click the Add button. This moves your selection to the trace specification list below. You can then add more trace specifications using the Add button. When you click Do It, your graph is created with all of the specified traces.

The advanced version of the dialog includes two-dimensional waves in the Y Waves and X Wave lists. You can edit the range values for waves in the holding pen to specify individual rows or columns of a matrix or to specify other subsets of data. See **Subrange Display** on page II-321 for details.

Waves and Axes

Axes are dependent upon waves for their existence. If you remove from a graph the last wave that uses a particular axis then that axis will also be removed.

In addition, the first wave plotted against a given axis is called the **controlling wave** for the axis. There is only one thing special about the controlling wave: its units define the units that will be used in creating the axis label and occasionally the tick mark labels. This is normally not a problem since all waves plotted against a given axis will likely have the same units. You can determine which wave controls an axis with the AxisInfo function.

Types of Axes

The four axes named left, right, bottom and top are termed **standard axes**. They are the only axes that many people will ever need.

Each of the four standard axes is always attached to the corresponding edge of the **plot area**. The plot area is the central rectangle in a graph window where traces are plotted. Axis and tick mark labels are plotted outside of this rectangle.

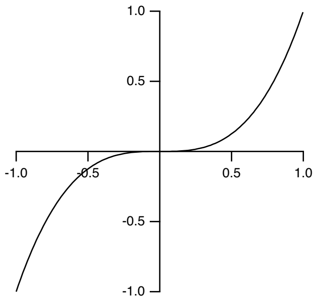
You can also add unlimited numbers of additional user-named axes termed **free axes**. Free axes are so named because you can position them nearly anywhere within the graph window. In particular, vertical free axes can be positioned anywhere in the horizontal dimension while horizontal axes can be positioned anywhere in the vertical dimension.

The Axis pop-up menu entries L=VertCrossing and B=HorizCrossing in the New Graph dialog create free axes that are each preset to cross at the numerical zero location of the other. They are also set to suppress the tick mark and label at zero. For example, create this data:

```
Make yWave; SetScale/I x,-1,1,yWave; yWave= x^3
```

Now, using the New Graph dialog, select yWave from the Y list and then L=VertCrossing from the Y axis pop-up menu. This generates the following command and the resulting graph:

```
Display/L=VertCrossing/B=HorizCrossing yWave
```



You could remove the tick mark and label at -0.5 by double-clicking the axis to reach the Modify Axis dialog, choosing the Tick Options tab, and finally typing -0.5 in one of the unused Inhibit Ticks boxes.

The free axis types described above all require that there be at least one trace that uses the free axis. For special purposes Igor programmers can also create a free axis that does not rely on any traces by using the **NewFreeAxis** operation (page V-679). Such an axis will not use any scaling or units information from any associated waves if they exist. You can specify the properties of a free axis using the **SetAxis** operation (page V-835) or the **ModifyFreeAxis** operation (page V-609), and you can remove them using the **KillFreeAxis** operation (page V-470).

Appending Traces

You can append waves to a graph as a waveform or XY plot by choosing Append Traces to Graph from the Graph menu. This presents a dialog identical to the New Graph dialog except that the title and style macro items are not present. Like the New Graph dialog, this dialog provides a way to create new axes and pairs of XY data. The Append to Graph submenu in the Graph menu allows you to append category plots, contour plots and image plots.

Appending Traces by Drag and Drop

You can append waves to a graph by dragging them from the Data Browser or from the Waves in Window list of the Window Browser.

The following sections present a guided tour showing how to use drag and drop. You can execute the example commands by selecting the line or lines and pressing Control-Enter (*Macintosh*) or Ctrl+Enter (*Windows*).

Drag and Drop Tour

1. **Choose File->New Experiment.**
2. **Execute this command to create waves used in the tour:**

```
Make/O wave0=sin(x/8), wave1=cos(x/8), wave2=wave0*wave1, xwave=x/8
```

3. **Choose Data->Data Browser.**

   We will drag icons from the Data Browser into a graph window.

4. **Execute this command to create an empty graph:**

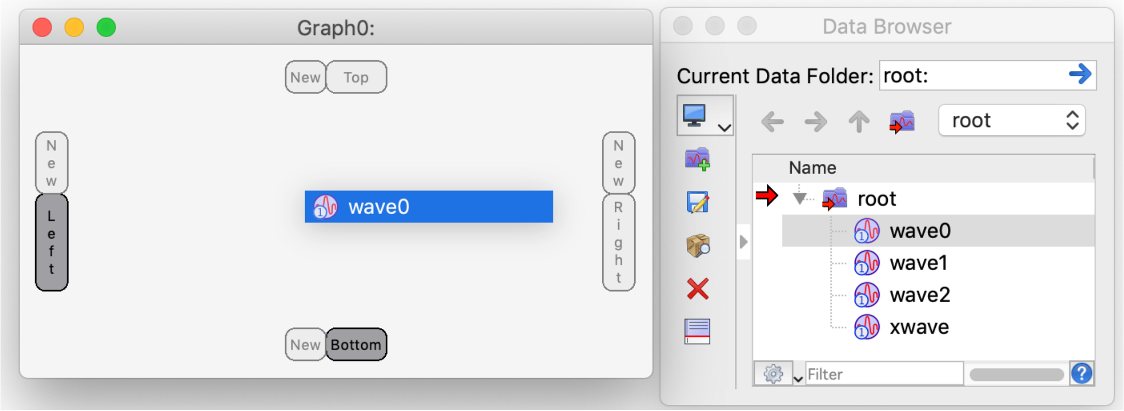
```
Display
```

Appending Traces to Standard Axes by Drag and Drop

You can append a trace using any of the four standard axes by dragging and hovering over the Left, Top, Right, or Bottom drop targets shown below. Dropping a wave in the middle of the graph appends a trace using the left and bottom axes.

1. **Select wave0 in the Data Browser and drag it to the middle of the graph.**

   When the mouse enters the graph, a number of drop targets appear.



2. **Release the mouse to drop the wave in the middle of the graph window.**

   Dropping the wave in the middle of the graph adds a trace to the left and bottom axes. An AppendToGraph command appears in the history area of the command window.

3. **Remove the trace from the graph by executing:**

   ```
   RemoveFromGraph wave0
   ```

   When you remove the last trace plotted against a given axis, Igor removes that axis from the graph.

   Next we will show how to append a trace using any of the four standard axes.

4. **Select wave0 in the Data Browser, drag it to the Top drop target and pause until Top turns blue. Then drag the wave to the Right drop target and release the mouse.**

   A trace is appended to the graph using the standard right and top axes.

5. **Remove the trace from the graph by executing:**

   ```
   RemoveFromGraph wave0
   ```

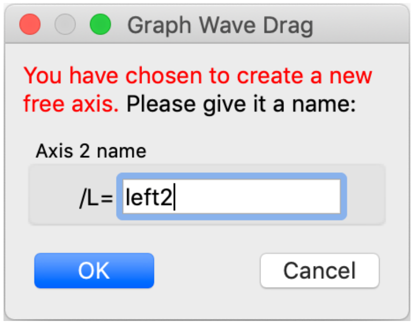
You can cancel a drag and drop operation that you have not yet completed by dragging the wave out of the graph or by pressing the escape key.

Creating Free Axes by Drag and Drop

You can append a trace to a new free axis by dragging and hovering over one of the New drop targets. In this section, we will append a trace to a new free left axis versus the standard bottom axis. For background information on free axes, see Types of Axes.

1. **Select wave0 in the Data Browser, drag it to the New drop target on the left side of the graph, and pause until New turns blue. Then drag the wave to the Bottom drop target and release the mouse.**

   A dialog appears asking for a name for the new free axis:



You have chosen to create a new free axis. Please give it a name:

Axis 2 name

/L= left2

2. **Enter left2 as the name for the new free axis an click OK.**

   The trace is appended versus the left2 and bottom axes.

   In the history area of the command window you can see AppendToGraph and ModifyGraph commands that were executed to create the new free axis and set its position.

3. **Select wave1 in the Data Browser, drag it to the New drop target on the left side of the graph, and pause until New turns blue. Then drag the wave to the Bottom drop target and release the mouse.**

   A dialog appears asking for a name for the new free axis. Enter left3.

   A wave1 trace is appended versus the left3 and bottom axes. The left2 and left3 axes are identical at this point so you can see only one of them. In the next step we will make them distinct.

4. **Stack the left2 and left3 axes by executing:**

   ```
   ModifyGraph axisEnab(left2)={0,0.45}, axisEnab(left3)={0.55,1.0}
   ```

   You now have a stacked graph.

   You can also use the Draw Between settings of the Axis tab of the Modify Graph dialog to generate the axisEnab commands.

Dragging to Existing Axes

Axes that already exist in the graph, whether standard or free, act as drop targets. To append a new trace to an existing axis, you drag onto the existing axis and pause until it turns blue, then drag onto another drop target.

1. **Select wave2 in the Data Browser, drag it to the left3 axis (the upper left axis) in the graph, and pause until New turns blue. Then drag the wave to the Bottom drop target and release the mouse.**

   Wave2 is appended versus the left3 and bottom axes.

2. **Remove the traces from the graph by executing:**

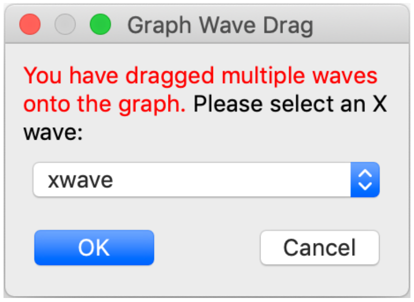
   ```
   RemoveFromGraph wave0, wave1, wave2
   ```

Appending XY Traces by Drag and Drop

To append an XY pair to a graph, you select the X and Y waves in the browser and drag to the graph as described above. When you release the mouse, Igor displays a dialog that allows you to identify one of the waves as the X wave.

1. **Select wave0, wave1, and xwave in the Data Browser, and drag and drop them in the middle of the graph.**

   Igor displays a dialog asking you to select an X wave:



Graph Wave Drag

You have dragged multiple waves onto the graph. Please select an X wave:

xwave

2. **Select xwave from the pop-up menu and click OK.**

   Igor displays wave0 and wave1 versus xwave using the left and bottom axes.

   If you wanted to plot all dragged waves as waveforms rather than as XY pairs, you would choose _calculated_ from the popup menu and click OK.

3. **Remove the traces from the graph by executing:**

   ```
   RemoveFromGraph wave0, wave1
   ```

It is not supported to create multiple XY traces with one drag. For example, if you have waves named wave0 and xwave0 and wave1 and xwave1, you have to do two drags to create two XY pairs. If you have many such XY pairs, use the command line or the advanced mode of the New Graph or Append Traces dialog; see Creating Graphs for details.

Trace Names

Each trace in a graph has a name. Trace names are displayed in graph dialogs and popup menus and used in graphing operations. Some of the operations that take trace names as parameters are **ModifyGraph (traces)**, **RemoveFromGraph**, **ReorderTraces**, **ErrorBars** and **Tag**.

The name of the trace is usually the same as the name of the wave displayed in the trace, but not always. If two traces display the same wave, or waves with the same name from different data folders, **Instance Notation** such as #1and #2, is used to distinguish between the traces. For example:

```
Make wave0 = sin(x/8)

// Create first trace displaying wave0. The trace name is wave0.
Display wave0

// Create second trace displaying of wave0. The trace name is wave0#1.
AppendToGraph wave0
```

Trace names like wave0#1 are automatically generated by Igor based on the name of the wave and the position of the trace in the graph.

It is also possible to assign a user-defined name to a trace that is not derived from the name of the wave being displayed:

```
// Create third instance of wave0. The trace name is thirdInstance.
AppendToGraph wave0/TN=thirdInstance
```

This creates a graph with three traces named wave0, wave0#1 and thirdInstance. The trace name wave0 is equivalent to wave0#0.

As of Igor Pro 9.00, you can change the name of an existing trace using ModifyGraph with the traceName keyword.

User-defined trace names are helpful in graphs that display same-named waves from different data folders. They are also useful for assigning names to be used programmatically which indicate the purpose of the trace rather than the wave supplying the data for it. User-defined trace names can be set only via the Display and AppendToGraph operations.

User-defined trace names are particularly useful for box plots and violin plots, as these plot types often take a list of waves instead of just one wave. Unless you specify a user-defined trace name, the name is taken from the first wave in the list.

See also **User-defined Trace Names** on page IV-89.

Automatically-generated trace names can create surprises. Imagine that three waves from different data folders all having the name "wave0" are added to a graph:

```
Make :df1:wave0
Make :df2:wave0
Make :df3:wave0
Display :df1:wave0,:df2:wave0,:df3:wave0
```

Now the list of traces is wave0, wave0#1, wave0#2. Now remove the first trace:

```
RemoveFromGraph wave0
```

This is equivalent to removing wave0#0. Because the automatically-generated instance number is the based on position in the list of traces, the name of trace wave0#1 changes to wave0 and wave0#2 changes to wave0#1.

For information on programming with trace names, see **Programming With Trace Names** on page IV-87.

Moving Traces

You can move a trace from the left axis to the right axis or vice versa by right-clicking the trace and choosing Move to Opposite Axis. You can do other types of moves using the **ReorderTraces** operation with the /L or /R flags.

Removing Traces

You can remove traces, as well as image plots and contour plots, from a graph by choosing Remove from Graph from the Graph menu. Select the type of item that you want to remove from the pop-up menu above the list.

A contour plot has traces that you can remove, but they will come back when the contour plot is updated. Rather than removing the contour traces, use the pop-up to select Contours, and remove the contour plot itself, which automatically removes all of the contour-related traces. See **Removing Contour Traces from a Graph** on page II-374.

If you remove the last item associated with a given axis then that axis will also be removed.

Replacing Traces

You can “replace” a trace in the sense of changing the wave that the trace is displaying in a graph. All the other characteristics of the trace, such as mode, line size, color, and style, remain unchanged. You can use this to update a graph with data from a wave other than the one originally used to create the trace.

To replace a trace, use the Replace Wave item in the Graph menu to display the Replace Wave in Graph dialog:

A special mode allows you to browse through groups of data sets composed of identically-named waves residing in different data folders. For instance, you might take the same type of data during multiple runs on different experimental subjects. If you store the data for each run in a separate data folder, and you give the same names to the waves that result from each run, you can select the Replace All in Data Folder checkbox and then select one of the data folders containing data from a single run. All the waves in the chosen data folder whose names match the names of waves displayed in the graph will replace the same-named waves in the graph.

You can also replace waves one at a time with any other wave. With the Replace All in Data Folder checkbox unchecked, choose a trace from the list below the menu. To replace the Y wave used by the trace, check the Y checkbox; to replace the X wave check the X checkbox. You can replace both if you wish. Select the waves to use as replacements from the menus to the right of the checkboxes. You can select _calculated_ from the X wave menu to remove the X wave of an XY pair, converting it to a waveform display.

The menus allow you to select waves having lengths that don’t match the length of the corresponding X or Y wave. In that case, use the edit boxes to the right to select a sub-range of the wave’s points. You can also use these boxes to select a single row or column from a two-dimensional wave.

The dialog creates command lines using the **ReplaceWave** operation (page V-801).

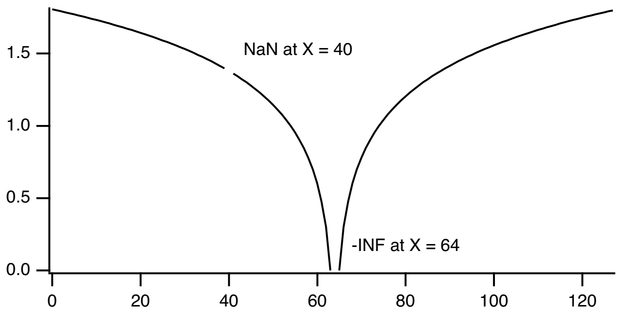
Plotting NaNs and INFs

The data value of a wave is normally a finite number but can also be a NaN or an INF. NaN means “Not a Number”, and INF means “infinity”. An expression returns the value NaN when it makes no sense mathematically. For example, `log(-1)` returns the value NaN. You can also set a point to NaN, using a table or a wave assignment, to represent a missing value. An expression returns the value INF when it makes sense mathematically but has no finite value. `log(0)` returns the value -INF.

Igor ignores NaNs and INFs when scaling a graph. If a wave in a graph is set to lines between points mode then Igor draws lines toward an INF. By default, it draws no line to or from a NaN so a NaN acts like a missing value. You can override the default, instructing Igor to draw lines through NaNs using the Gaps checkbox in the Modify Trace Appearance dialog.

The following graph illustrate these points. It was created with these commands:

```
Make wave1= log(abs(x-64))
wave1(40)=log(-1)
Display wave1
```



You can override the default, instructing Igor to draw lines through NaNs. See **Gaps** on page II-303 for details.

Scaling Graphs

Igor provides several ways of scaling waves in graphs. All of them allow you to control what sections of your waves are displayed by setting the range of the graph's axes. Each axis is either autoscaled or manually scaled.

Autoscaling

When you first create a graph all of its axes are in **autoscaling** mode. This means that Igor automatically adjusts the extent of the axes of the graph so that all of each wave in the graph is fully in view. If the data in the waves changes, the axes are automatically rescaled so that all waves remain fully in view.

If you manually scale any axis, that axis changes to **manual scaling** mode. The methods of manual scaling are described in the next section. Axes in manual scaling mode are never automatically scaled.

If you choose Autoscale Axes from the Graph menu all of the axes in the graph are autoscaled and returned to autoscaling mode. You can set individual axes to autoscaling mode and can control properties of autoscaling mode using the Axis Range tab of the Modify Axis dialog described in **Setting the Range of an Axis** on page II-286.

Manual Scaling

You can manually scale one or more axes of a graph using the mouse. Start by clicking the mouse and dragging it diagonally to frame the region of interest. Igor displays a dashed outline around the region. This outline is called a marquee. A marquee has handles and edges that allow you to refine its size and position.

To refine the size of the marquee move the cursor over one of the handles. The cursor changes to a double arrow which shows you the direction in which the handle moves the edge of the marquee. To move the edge click the mouse and drag.

To refine the position of the marquee move the cursor over one of the edges away from the handles. The cursor changes to a hand. To move the marquee click the mouse and drag.

When you click inside the region of interest Igor presents a pop-up menu from which you can choose the scaling operation.

Choose the operation you want and release the mouse. These operations can be undone and redone; just press Command-Z (*Macintosh*) or Ctrl+Z (*Windows*).

The **Expand** operation scales all axes so that the region inside the marquee fills the graph (zoom in). It sets the scaling mode for all axes to manual.

The **Horiz Expand** operation scales only the horizontal axes so that the region inside the marquee fills the graph horizontally. It has no effect on the vertical axes. It sets the scaling mode for the horizontal axes to manual.

The **Vert Expand** operation scales only the vertical axes so that the region inside the marquee fills the graph vertically. It has no effect on the horizontal axes. It sets the scaling mode for the vertical axes to manual.

The **Shrink** operation scales all axes so that the waves in the graph appear smaller (zoom out). The factor by which the waves shrink is equal to the ratio of the size of the marquee to the size of the entire graph. For example, if the marquee is one half the size of the graph then the waves shrink to one half their former size. The point at the center of the marquee becomes the new center of the graph. The shrink operation sets the scaling mode for all axes to manual.

The **Horiz Shrink** operation is like the shrink operation but affects the horizontal axes only. It sets the scaling mode for the horizontal axes to manual.

The **Vert Shrink** operation is like the shrink operation but affects the vertical axes only. It sets the scaling mode for the vertical axes to manual.

Another way to manually scale axes is to use the Axis Range tab of the Modify Axis dialog (see **Manual Axis Ranges** on page II-287), or the **SetAxis** operation (page V-835).

Scaling Using the Mouse Wheel

You can manually scale a graph axis using the mouse wheel. The scaling performed is determined by where along the axis you position the mouse. Start by positioning the mouse over the axis until the cursor changes to a double-ended arrow.

If you position the mouse over the middle of the axis and rotate the wheel, both ends of the axis are scaled equally.

If you position the mouse over one end of the axis and rotate the wheel, that end remains fixed and the other end of the axis is scaled.

If you position the mouse between the middle and one end of the axis and rotate the wheel, more of the scaling is applied to the other end.

Panning

After zooming in on a region of interest, you may want to view data that is just off screen. To do this, press Option (*Macintosh*) or Alt (*Windows*) and move the mouse to the graph interior where the cursor changes to a hand. Now drag the body of the graph. Pressing Shift will constrain movement to the horizontal or vertical directions.

This operation is undoable.

Fling Mode

If, while panning, you release the mouse button while the mouse is still moving, the panning will automatically continue. While panning, release the Option or Alt key and change the force or direction of the mouse-click gesture to change the panning speed or direction. Click the mouse button once to stop.

Panning Using the Mouse Wheel

You can pan a graph using the mouse wheel. Start by positioning the mouse over the axis until the cursor changes to a double-ended arrow.

Press the shift key and rotate the wheel. The axis range is changed to pan the graph. The axis value over which you position the mouse remains fixed while the axis range is changed.

Panning is triggered by "horizontal scrolling" with the mouse wheel. With most mice, you trigger horizontally scrolling by pressing the shift key and rotating the scroll wheel, but some mice may behave differently.

Setting the Range of an Axis

You can set the range and other scaling parameters for individual axes using the Axis Range tab in the Modify Axis dialog. You can display the dialog with this tab selected by choosing Set Axis Range from the

Graph menu or by double-clicking a tick mark label of the axis you wish to modify. Information on the other tabs in this dialog is available in **Modifying Axes** on page II-306.

Start by choosing the axis that you want to adjust from the Axis pop-up menu. You can adjust each axis in turn, or a selection of axes, without leaving this dialog.

Manual Axis Ranges

When a graph is first created, it is in autoscaling mode. In this mode, the axis limits automatically adjust to just include all the data. The controls in the Autoscale Settings section provide autoscaling options.

You can set the axis limits to fixed values by editing the minimum and maximum parameters in the Manual Range Settings section. You can return the minimum or maximum axis range to autoscaling mode by unchecking the corresponding checkbox. These settings are independent, so you can fix one end of the axis and autoscale the other end.

There are a number of other ways to set the minimum and maximum parameters. Clicking the Expand 5% button expands the range by 5 percent. This has the effect of shrinking the traces plotted on the axis by 5%.

Clicking the Swap button exchanges the minimum and maximum parameters. This has the effect of reversing the ends of the axis, allowing you to plot waves upside-down or backwards with fixed limits.

An additional way to set the minimum and maximum parameters is to select a wave from the list and use the Quick Set buttons. If you click the X Min/Max quick set button then the minimum and maximum X values of the selected wave are transferred to the parameter boxes. If you click the Y Min/Max quick set button then the minimum and maximum Y values of the selected wave are transferred to the parameter boxes. If you specified the full scale Y values for the wave then you can click the Full Scale quick set button. This transfers the wave’s Y full scale values to the parameter boxes. The full scale Y values can be set using the Change Wave Scaling item in the Data menu.

Automatic Axis Ranges

When the manual minimum and maximum checkboxes are unchecked, the axis is in autoscaling mode. In this mode the axis limits are determined by the data values in the waves displayed using the selected axis. The items in the Autoscale Settings section control the method used to determine the axis range:

The Reverse Axis checkbox swaps the minimum and maximum axis range values, plotting the trace upside-down or backwards.

The top pop-up menu controls adjustments to the minimum and maximum axis range values.

The default mode is “Use data limits”. The axis range is set to the minimum and maximum data values of all waves plotted against the axis.

The “Round to nice values” mode extends the axis range to include the next major tick mark.

The “Nice + inset data” mode extends the axis range to include the next major tick mark and also ensures that traces are inset from both ends of the axis.

The bottom pop-up menu controls the treatment f the value zero.

The default mode is “Zero isn’t special”. The axis range is set to the minimum and maximum data values.

The “Autoscale from zero” mode forces the end of the axis that is closest to zero to be exactly zero.

The “Symmetric about zero” mode forces zero to be in the middle of the axis range.

The “Autoscale from zero if not bipolar” mode behaves like “Autoscale from zero” if the data is unipolar (all positive or all negative) and like “Zero isn’t special” if the data is bipolar.

Autoscaling mode usually sets the axis limits using all the data in waves associated with the traces that use the axis. This can be undesirable if the associated horizontal axis is set to display only a portion of the total

X range. Select the Autoscale Only Visible Data checkbox to have Igor use only the data included within the horizontal range for autoscaling. This checkbox is available only if the selected axis is a vertical axis.

Overall Graph Properties

You can specify certain overall properties of a graph by choosing Modify Graph from the Graph menu. This brings up the Modify Graph dialog. You can also get to this dialog by double-clicking a blank area outside the plot rectangle.

Normally, X axes are plotted horizontally and Y axes vertically. You can reverse this behavior by checking the “Swap X & Y Axes” checkbox. This is commonly used when the independent variable is depth or height. This method swaps X and Y for all traces in the graph. You can cause individual traces to be plotted vertically by selecting the “Swap X & Y Axes” checkbox in the New Graph and Append Traces dialogs as you are creating your graph.

Initially, the graph font is determined by the default font which you can set using the Default Font item in the Misc menu. The graph font size is initially automatically calculated based on the size of the graph. You can override these initial settings using the “Graph font” and “Font size” settings. Igor uses the font and size you specify in annotations and axis labels unless you explicitly set the font or size for an individual annotation or label.

Initially, the graph marker size is automatically calculated based on the size of the graph. You can override this using the “Marker size” setting. You can set it to “auto” (or 0 which is equivalent) or to a number from -1 to 99. Use -1 to make a graph subwindow get is default font size from its parent. Igor uses the marker size you specify unless you explicitly set the marker size for an individual wave in the graph.

The “Use comma as decimal separator” checkbox determines whether dot or comma is used as the decimal separator in tick mark labels.

Graph Margins

The margin is the distance from an outside edge of the graph to the edge of the plot area of the graph. The plot area, roughly speaking, is the area inside the axes. See **Graph Dimensions** on page II-288 for a definition. Initially, Igor automatically sets each margin to accommodate axis and tick mark labels and exterior textboxes, if any. You can override the automatic setting of the margin using the Margins settings. You would do this, for example, to force the left margins of two graphs to be identical so that they align properly when stacked vertically in a page layout. The Units pop-up menu determines the units in which you enter the margin values.

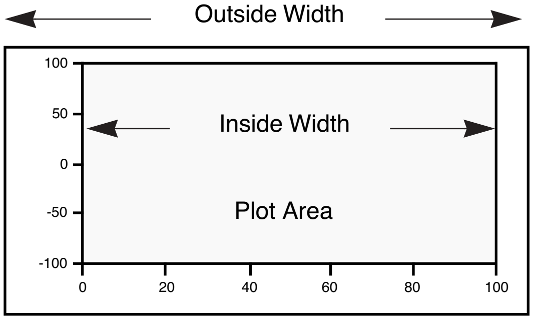
You can also set graph margins interactively. If you press Option (*Macintosh*) or Alt (*Windows*) and position the cursor over one of the four edges of the plot area rectangle, you will see the cursor change to this shape: ⇹ . Use this cursor to drag the margin. You can cause a margin to revert to automatic mode by dragging the margin all the way to the edge of the graph window or beyond. If you drag to within a few pixels of the edge, the margin will be eliminated entirely. If you double click with this cursor showing, Igor displays the Modify Graph dialog with the corresponding margin setting selected.

If you specify a margin for a given axis, the value you specify solely determines where the axis appears. Normally, dragging an axis will adjust its offset relative to the nominal automatic location. If, however, a fixed margin has been specified then dragging the axis will drag the margin.

Graph Dimensions

The Modify Graph dialog provides several ways of controlling the width and height of a graph. Usually you don’t need to use these. They are intended for certain specialized applications.

These techniques are powerful but can be confusing unless you understand the algorithms, described below, that Igor uses to determine graph dimensions.



The graph can be in one of five modes with respect to each dimension: auto, absolute, per unit, aspect, or plan. These modes control the width and height of the **plot area** of the graph. The plot area is the shaded area in the illustration. The width mode and height mode are independent.

In this graph, the axis standoff feature, described in the **Modifying Axes** section on page II-306, is off so the plot area extends to the center of the axis lines. If it were on, the plot area would extend only to the inside edge of the axis lines.

Auto mode automatically determines the width or height of the plot area based on the outside dimensions of the graph and other factors that you specify using Igor's dialogs. This is the normal default mode which is appropriate for most graphing jobs. The remaining modes are useful for special purposes such as matching the axis lengths of two or more graphs or replicating a standard graph or a graph from a journal.

If you select any mode other than auto, you are putting a constraint on the width or height of the plot area which also affects the outside dimensions of the graph. If you adjust the outside size of the graph, by dragging the window's size box, by tiling, by stacking or by using the MoveWindow operation, Igor first determines the outside dimensions as you have specified them and then applies the constraints implied by the width/height mode that you have selected. A graph dimension can be changed by dragging with the mouse only if it is auto mode.

With **Absolute** mode, you specify the width or height of the plot area in absolute units; in inches, centimeters or points. For example, if you know that you want your *plot area* to be exactly 5 inches wide and 3.5 inches high, you should use those numbers with an absolute mode for both the width and height.

If you want the *outside* width and height to be an exact size, you must also specify a fixed value for all four margins. For instance, setting all margins to 0.5 inches in conjunction with an absolute width of 5 inches and a height of 3.5 inches yields a graph whose outside dimensions will be 6 inches wide by 4.5 inches high.

The **Aspect** mode maintains a constant aspect ratio for the plot area. For example, if you want the width to be 1.5 times longer than the height, you would set the width mode to aspect and specify an aspect ratio of 1.5.

The remaining modes, per unit and plan, are quite powerful and convenient for certain specialized types of graphs, but are more difficult to understand. You should expect that some experimentation will be required to get the desired results.

In **Per unit** mode, you specify the width or height of the plot area in units of length per axis unit. For example, suppose you want the plot width to be one inch per 20 axis units. You would specify 1/20 = 0.05 inches per unit of the bottom axis. If your axis spanned 60 units, the plot width would be three inches.

Igor allows you to select a horizontal axis to control the vertical dimension or a vertical axis to control the horizontal direction, but it is very unlikely that you would want to do that.

In **Plan** mode, you specify the length of a unit in the horizontal dimension as a scaling factor times the length of a unit in the vertical dimension, or vice versa. The simplest use of plan scaling is to force a unit in one dimension to be the same as in the other, such as would be appropriate for a map. To do this, you select plan scaling for one dimension and set the scaling factor to 1.

Until you learn how to use the per unit and plan modes, it is easy to create a graph that is ridiculously small or large. Since the size of the graph is tied to the range of the axes, expanding, shrinking or autoscaling the graph makes its size change.

Applying plan or aspect mode to both the X and Y dimensions of a graph is a bad idea. Interactions between the dimensions cause huge or tiny graphs, or other bizarre results. The Modify Graph dialog does not allow both dimensions to be plan or aspect, or a combination of the two. However, the ModifyGraph operation permits it and it is left to you to refrain from doing this.

Sometimes you can end up with a graph whose size makes it difficult to move or resize the window. Use the Graph menu's Modify Graph dialog to reset the size of the graph to something more manageable.

You can change a graph dimension by dragging with the mouse only if it is in auto mode. If you want to resize a graph but can't, use the Modify Graph dialog to check the width and height modes.

If you want to fully understand how Igor arrives at the final size of a graph when the width or height is constrained, you need to understand the algorithm Igor uses:

1. The initial width and height are calculated. When you adjust a window by dragging, the initial width and height are based on the width and height to which you drag the window.
2. If you are exporting graphics, the width and height are as specified in the Export Graphics dialog or in the **SavePICT** command.
3. If you are printing, the width and height are modified by the effects of the printing mode, as set by the **PrintSettings** graphMode keyword.
4. The width modes absolute and per unit are applied which may generate a new width.
5. The height mode is applied which may generate a new height.
6. The width modes aspect and plan are applied which may generate a new width.

Because there are many interactions, it is possible to get a graph so big that you can't adjust it manually. If this occurs, use the Modify Graph dialog to set the width and height to a manageable size, using absolute mode.

Modifying Traces

You can specify each trace's appearance in a graph by choosing Modify Trace Appearance from the Graph menu or by double-clicking a trace in the graph. This brings up the following dialog:

For image plots, choose Modify Image Appearance from the Graph menu, rather than Modify Trace Appearance.

For contour plots, you normally should choose Modify Contour Appearance. Use this to control the appearance of all of the contour lines in one step. However, if you want to make one contour line stand out, use the Modify Trace Appearance dialog.

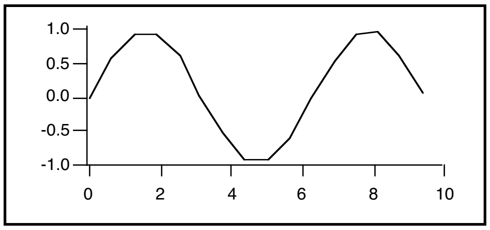
Selecting Traces to be Modified

Select the trace or traces whose appearance you want to modify from the Trace list. If you got to this dialog by double-clicking a trace in the graph then that trace will automatically be selected. If you select more than one trace, the items in the dialog will show the settings for the *first* selected trace.

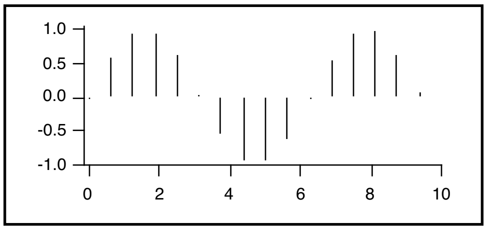
Once you have made your selection, you can change settings for the selected traces. After doing this, you can then select a different trace or set of traces and make more changes. Igor remembers all of the changes you make, allowing you to do everything in one trip to the dialog.

Display Modes

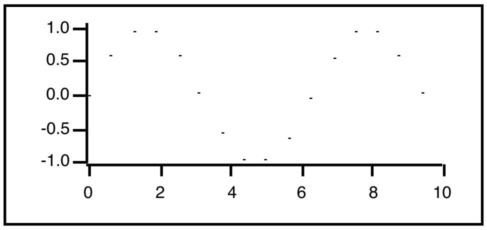
Choose the **mode** of presentation for the selected trace from the Mode pop-up menu. The commonly-used modes are Lines Between Points, Dots, Markers, Lines and Markers, and Bars. Other modes allow you to draw sticks, bars and fills to Y=0 or to another trace.



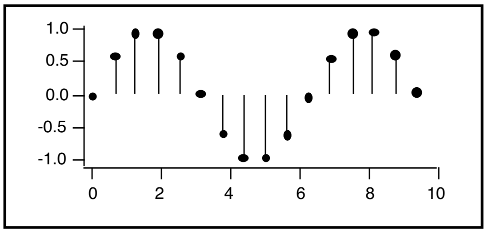
Lines between points mode



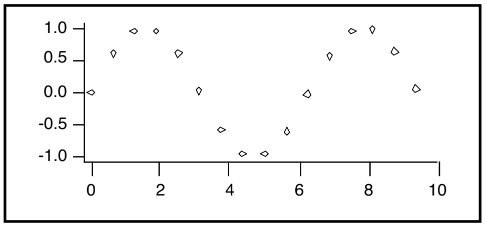
Sticks to zero mode



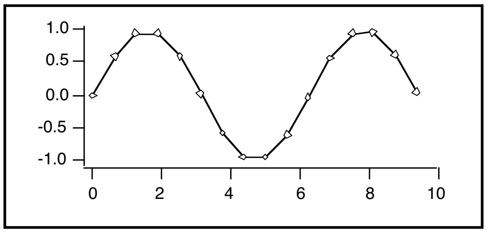
Dots mode



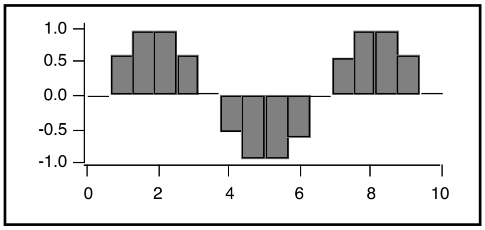
Sticks and markers mode



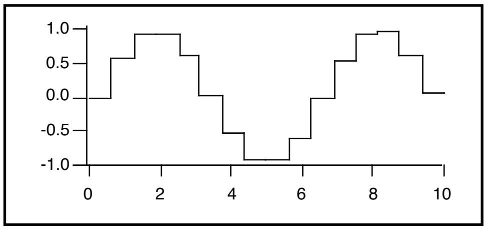
Markers mode



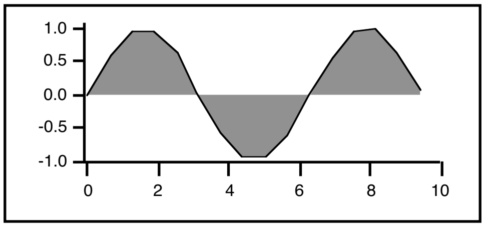
Lines and markers mode



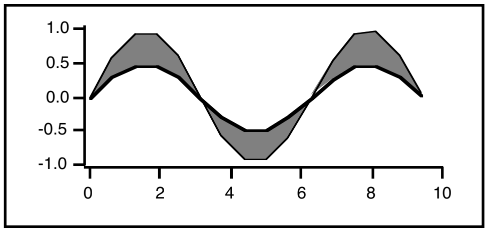
Bars mode



Cityscape mode



Fill to zero mode



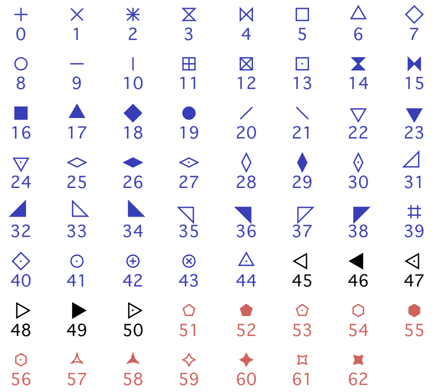
Fill to next mode

Markers

If you choose the Markers or Lines and Markers mode you also can choose the marker, marker size, marker thickness, and whether the marker is transparent or filled with color. The marker size is a fractional number from 0 to 200. 0 means “auto”, which chooses a marker size appropriate to the size of the graph. The marker thickness is in a fractional number from 0 to 50 points. Fractional points may not be evident on screen but can be seen in exported and printed graphics. Setting the marker thickness to 0 makes the markers disappear.

There is an interaction between marker size and marker thickness: Igor will adjust the marker size if this is needed to make the marker symmetrical. The unadjusted width and height of the marker is 2*s+1 points where s is the marker size setting.

Here is a table of the markers and the corresponding marker codes:



You can also create custom markers. See the **SetWindow** operation's markerHook keyword.

Marker Colors

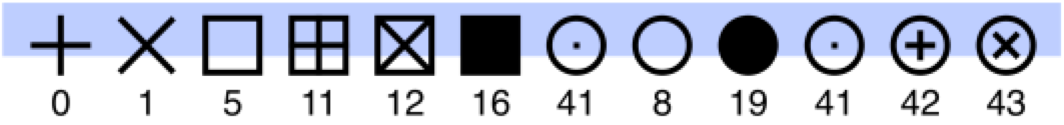
Igor provides control of three colors for graph markers: the trace color, the stroke color and the fill color.

The trace color is simply the color selected for the trace overall and is the same for any trace mode.

The stroke color is the color of the lines making the outlines of the markers. By default the stroke color is the same as the trace color.

The fill color is used to fill the background space of hollow markers. By default there is no fill color so that you see background objects through the interiors of hollow markers.

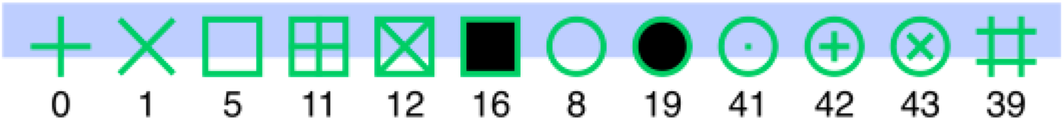
Here is a sample of some of Igor's markers with the trace color set to black, the marker size set to 10 and the marker stroke thickness set to 2 points. A blue rectangle is drawn beneath the markers:



The blue rectangle shows through the interior of the hollow markers (5, 11, 12, 8, 41, 42, 43). The solid markers (16, 19) are solid black because the trace color is black.

You can set the stroke color using the Modify Trace Appearance dialog: turn on the Stroke checkbox and select a color. You can also use the command ModifyGraph mrkStrokeRGB. In the next figure, the stroke color was set to green using the command

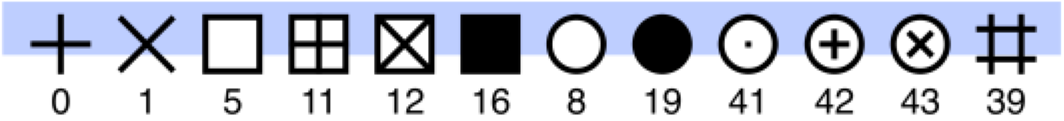
```
ModifyGraph mrkStrokeRGB=(1,52428,26586)
```



The stroke color overrides the trace color, so the outlines of all the markers are now green. The solid markers are black with green outlines.

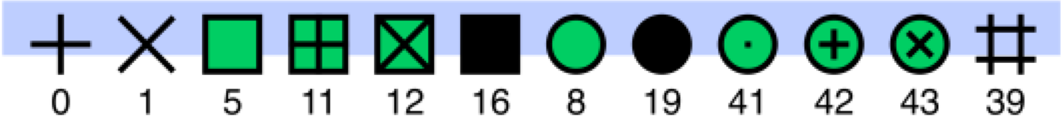
You can choose to make the interiors of hollow markers opaque. In the Modify Trace Appearance dialog, turn on the Fill checkbox. The command is

```
ModifyGraph opaque=1
```



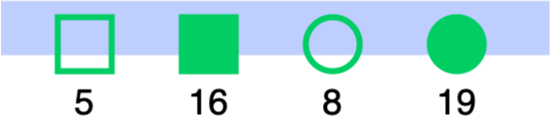
By default, the interior color of opaque hollow markers is white. You can change the interior color by selecting a fill color in the Modify Trace Appearance dialog or with the ModifyGraph mrkFillRGB keyword. Here the fill is set using the command

```
ModifyGraph mrkFillRGB=(1,52428,26586)
```

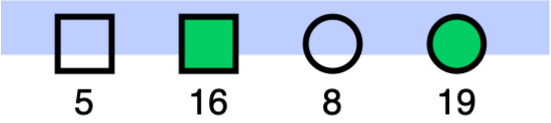


It is possible to achieve identical results by setting the stroke color of a solid marker or the fill color of a hollow marker, as in the following example.

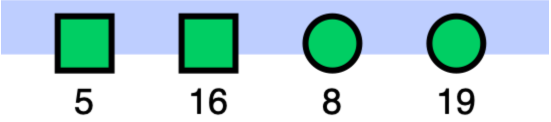
Trace color set to green:



Now set the stroke color set to black, resulting in solid markers having a black outline and green interior:



Finally set the fill color to green resulting in hollow markers being filled with green:



Now the solid and hollow markers look the same. This is a historical oddity - the fill color could not be changed from white before Igor Pro 9.00. Before Igor9 the hollow markers with interior markings such as 11 or 42 could not be filled with a color.

Text Markers

In addition to the built-in drawn markers, you can also instruct Igor to use one of the following as text markers:

- A single character from a font
- The contents of a text wave
- The contents of a numeric wave

A single character from a font is mainly of interest if you want to use a special symbol that is available in a font but is not included among Igor’s built-in markers. The specified character is used for all data points.

The remaining options provide a way to display a third value in an XY plot. For example, a plot of earthquake magnitude versus date could also show the location of the earthquake using a text wave to supply text markers. Or, a plot of earthquake location versus date could also show the magnitude using a numeric wave to supply text markers. For each data point in the XY plot, the corresponding point of the text or numeric wave supplies the text for the marker. The marker wave must have the same number of points as the X and Y waves.

To create a text marker, choose the Markers or Lines and Markers display mode. Then click the Markers pop-up menu and choose the Text button. This leads to the Text Markers subdialog in which you can specify the source of the text as well as the font, style, rotation and other properties of the markers.

You can offset and rotate all the text markers by the same amount but you can not set the offset and rotation for individual data points — use tags for that. You may find it necessary to experimentally adjust the X and Y offsets to get character markers exactly centered on the data points. For example, to center the text just above each data point, choose Middle bottom from the Anchor pop-up menu and set the Y offset to 5-10 points. If you need to offset some items differently from others, you will have to use tags (see **Tags** on page III-43).

Igor determines the font size to use for text markers from the marker size, which you set in the Modify Trace Appearance dialog. The font size used is 3 times the marker size.

You may want to show a text marker *and* a regular drawn marker. For this, you will need to display the wave twice in the graph. After creating the graph and setting the trace to use a drawn marker, choose Graph→Append Traces to Graph to append a second copy of the wave. Set this copy to use text markers.

Arrow Markers

Arrow markers can be used to create vector plots illustrating flow and gradient fields, for example. Arrow markers are fairly special purpose and require quite a bit of advance preparation.

Here is a very simple example:

```
// Make XY data
Make/O xData = {1, 2, 3}, yData = {1, 2, 3}
Display yData vs xData              // Make graph
ModifyGraph mode(yData) = 3         // Marker mode

// Make an arrow data wave to control the length and angle for each point.
Make/O/N=(3,2) arrowData            // Controls arrow length and angle
Edit /W=(439,47,820,240) arrowData

// Put some data in arrowData
arrowData[0][0]= {20,25,30}                  // Col 0: arrow lengths in points
arrowData[0][1]= {0.523599,0.785398,1.0472}  // Col 1: arrow angle in radians

// Set trace to arrow mode to turn arrows on
ModifyGraph arrowMarker(yData) = {arrowData, 1, 10, 1, 1}

// Make an RGB color wave
Make/O/N=(3,3) arrowColor
Edit /W=(440,272,820,439) arrowColor

// Store some colors in the color wave
arrowColor[0][0]= {65535,0,0}       // Red
arrowColor[0][1]= {0,65535,0}       // Green
arrowColor[0][2]= {0,0,65535}       // Blue

// Turn on color as f(z) mode
ModifyGraph zColor(yData)={arrowColor,*,*,directRGB,0}
```

To see a demo of arrow markers choose File→Example Experiments→Graphing Techniques→Arrow Plot.

See the reference for a description of the arrowMarker keyword under the **ModifyGraph (traces)** operation on page V-613 for further details.

Line Styles and Sizes

If you choose the “Lines between points”, “Lines and markers”, or Cityscape mode you can also choose the line style. You can change the dash patterns using the Dashed Lines item in the Line section.

For any mode except the Markers mode you can set the line size. The line size is in points and can be fractional. If the line size is zero, the line disappears.

For more information see **Dashed Lines** on page III-496.

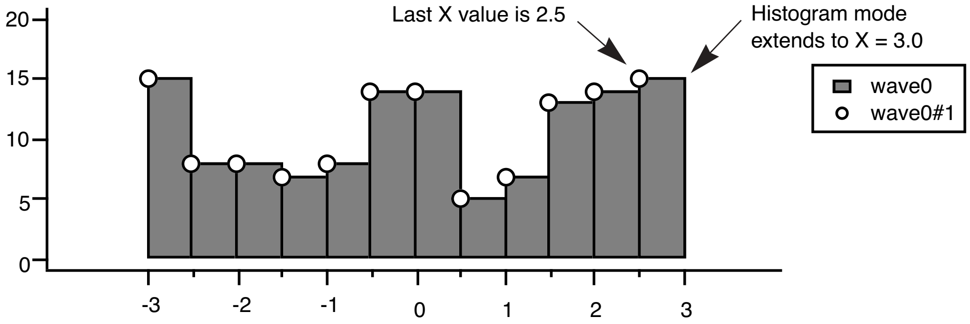
Fills

For traces in the Bars and “Fill to zero” modes, Igor presents a choice of fill type. The fill type can be None, which means the fill is transparent, Erase, which means the fill is white and opaque, Solid, or three patterns of gray. You can also choose a pattern from a palette and can choose the fill types and colors for positive going regions and negative going regions independently.

For more information see **Fill Patterns** on page III-498 and **Gradient Fills** on page III-498.

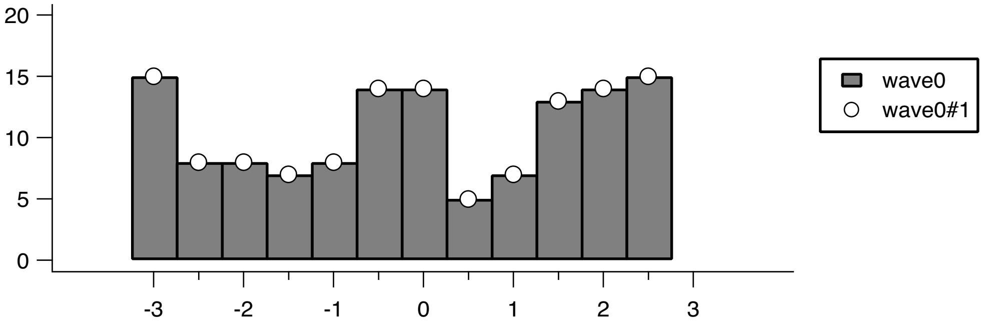
Bars With Waveforms

When Bars mode is used for a waveform plotted on a normal continuous X axis (rather than a category axis, see Chapter II-14, **Category Plots**), the X values are computed from the wave’s X scaling. The bars are drawn from the X value for a given point up to *but not including* the X value for the next point. Such bars are commonly called “histogram bars” because they are usually used to show the number of counts in a histogram that fall between the two X values.



If you want your bars centered on their X values, then you should create a Category Plot, which is more suited for traditional bar charts (see Chapter II-14, **Category Plots**). You can, however, adjust the X values for the wave so that the flat areas appear centered about its original X value as for a traditional bar chart. One way to do this without actually modifying any data is to offset the trace in the graph by one half the bar width. You can just drag it, or use the Modify Trace Appearance dialog to generate a more precise offset command. In our example, the bars are 0.5 X units wide:

```
ModifyGraph offset(wave0)={-0.25,0}
```



Bars With XY Pairs

If your X axis is controlled by an XY pair, the width of each bar is determined by two X values. One X value provides the location of the left edge of the bar and the next X value provides the location of the right edge of the bar.

If you imagine a bar plot with one bar, this requires an XY pair with two points - one X value to set the left edge of the bar and the next X value to set the right edge.

Generalizing, for a bar plot with N bars, you need N+1 points in your X and Y waves. Here is an example using 3 XY points to produce two bars:

```
Make/O barX={0,1,3}, barY = {1,2,2}
Display barY vs barX
ModifyGraph mode=5, hbFill=4
SetAxis left 0,*
```

Grouping, Stacking and Adding Modes

For the four modes that normally draw to y=0 (“Sticks to zero”, “Bars”, “Fill to zero”, and “Sticks and markers”) you can choose variants that draw to the Y values of the next trace. The four variant modes are: “Sticks to next”, “Bars to next”, “Fill to next” and “Sticks&markers to next”. *Next* in this context refers to the trace listed after (below) the selected trace in the list of traces in the Modify Trace Appearance and the Reorder Traces dialogs.

If you choose one of these four modes, Igor automatically selects “Draw to next” from the Grouping pop-up menu. You can also choose “Add to next” and “Stack on next” modes.

The Grouping pop-up menu the Modify Trace Appearance dialog is used to create special effects such as layer graphs and stacked bar charts. The available modes are “Keep with next”, None, “Draw to next”, “Add to next”, and “Stack on next.

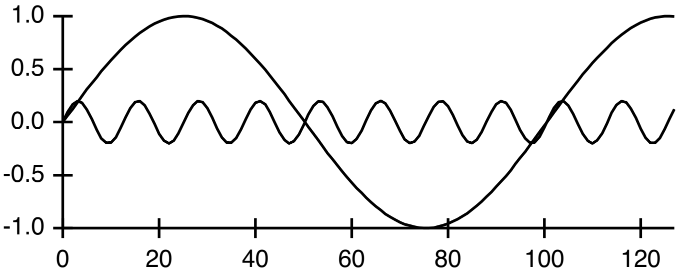
“Keep with next” is used only with category plots and is described in Chapter II-14, **Category Plots**.

“Draw to next” modifies the action of those modes that normally draw to y=0 so that they draw to the Y values of the next trace that is plotted against the same pair of axes as the current trace. The X values for the next trace should be the same as the X values for the current trace. If not, the next trace will not line up with the bottom of the current trace.

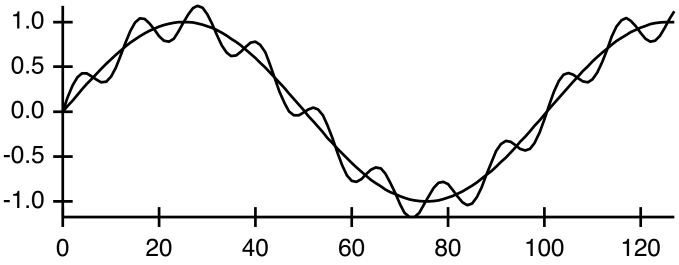
“Add to next” adds the Y values of the current trace to the Y values of the next trace before plotting. If the next trace is also using “Add to next” then that addition is performed first and so on. When used with one of the four modes that normally draw to y=0, this mode also acts like “Draw to next”.

“Stack on next” works just like “Add to next” except Y values are not allowed to backtrack. On other words, negative values act like zero when the Y value of the next trace is positive and positive values act like zero with a negative next trace.

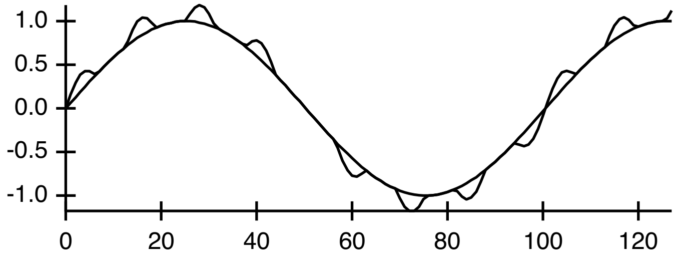
Here is a normal plot of a small sine wave and a bigger sine wave:



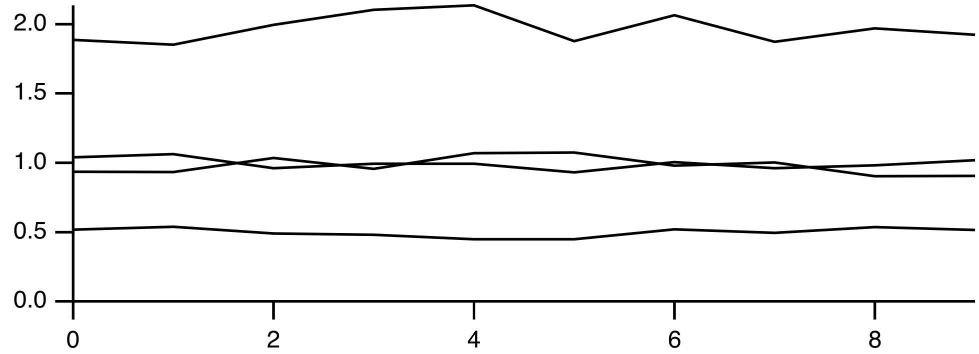
In this version, the small sine wave is set to “Add to next” mode:



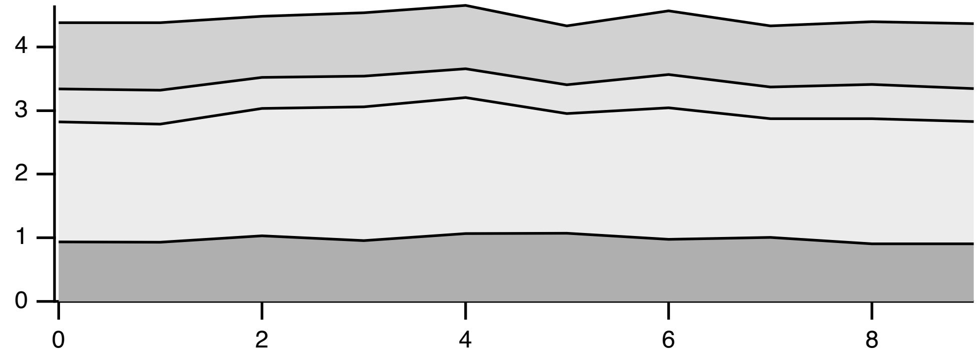
And here we use “Stack on next”:



You can create layer graphs by plotting a number of waves in a graph using the fill to next mode. Depending on your data you may also want to use the add to next grouping mode. For example, in the following normal graph, each trace might represent the thickness of a geologic layer:



We can show the layers in a more understandable way by using fill to next and add to next modes:



Because the grouping modes depend on the identity of the next trace, you may need to adjust the order of traces in the graph. You can do this using the Reorder Traces dialog. Choose Graph→Reorder Traces. Select the traces you want to move. Adjust the order by dragging the selected traces up and down in the list, dropping them in the appropriate spot.

Note: All of the waves you use for the various grouping, adding, and stacking modes should have the same numbers of points, X scaling, and should all be displayed using the same axes.

Trace Color

You can choose a color for the selected trace from the color pop-up palette of colors.

In addition to color, you can specify opacity using the color pop-up via the “alpha” property. An alpha of 1.0 makes the trace fully opaque. An alpha of 0.0 makes it fully transparent.

Setting Trace Properties from an Auxiliary (Z) Wave

You can set the color, marker number, marker size, and pattern number of a trace on a point-by-point basis based on the values of an auxiliary wave. The auxiliary wave is called the “Z wave” because other waves control the X and Y position of each point on the trace while the Z wave controls a third property.

For example, you could position markers at the location of earthquakes and vary their size to show the magnitude of each quake. You could show the depth of the quake using marker color and show different types of quakes as different marker shapes.

To set a trace property to be a function of a Z wave, click the “Set as f(z)” button in the Modify Trace Appearance dialog to display the “Set as f(z)” subdialog.

Color as f(z)

Color as f(z) has four modes: Color Table, Color Table Wave, Color Index Wave, and Three or Four Column Color Wave. You select the mode from the Color Mode menu.

In Color Table mode, the color of each data point on the trace is determined by mapping the corresponding Z wave value into a built-in color table. The mapping is logarithmic if the Log Colors checkbox is checked and linear otherwise. The Log Colors option is useful when the zWave spans many decades and you want to show more detailed changes of the smaller values.

The zMin and zMax settings define the range of values in your Z wave to map onto the color table. Values outside the range take on the color at the end of the range. If you choose Auto for zMin or zMax, Igor uses the smallest or largest value it finds in your Z wave. If any of your Z values are NaN, Igor treats those data points in the same way it does if your X or Y data is NaN. This depends on the Gaps setting in the main dialog.

Color Table Wave mode is the same as Color Table mode except that the colors are determined by a color table wave that you provide instead of a built-in color table. See **Color Table Waves** on page II-399 for details.

In Color Index Wave mode, the color of data points on the trace is derived from the Z wave you choose by mapping its values into the X scaling of the selected 3-column color index wave. This is similar to the way **ModifyImage** cindex maps image values (in place of the Z wave values) to a color in a 3-column color index matrix. See **Indexed Color Details** on page II-400.

In Three or Four Column Color Wave mode, data points are colored according to Red, Green and Blue values in the first three columns of the selected wave. Each row of the three column wave controls the color of a data point on the trace. This mode gives absolute control over the colors of each data point on a trace. If the wave has a fourth column, it controls opacity. See **Direct Color Details** on page II-401 for further information.

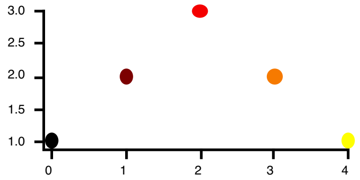
Color as f(z) Example

Create a graph:

```
Make/N=5 yWave = {1,2,3,2,1}
Display yWave
ModifyGraph mode=3, marker=19, msize=5
```

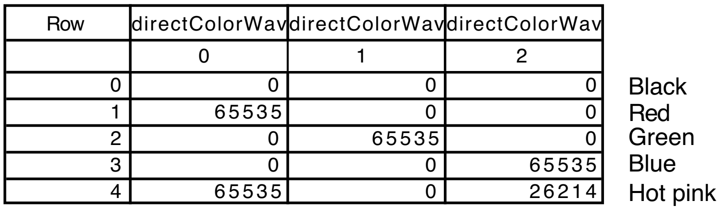
```
Make/N=5 zWave = {1,2,3,4,5}
ModifyGraph zColor(yWave)={zWave,*,*,YellowHot}
```

The commands generate this graph:



If instead you create a three column wave and edit it to enter RGB values:

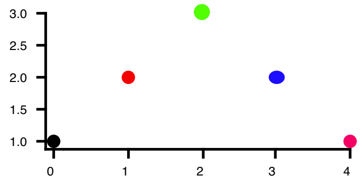
```
Make/N=(5,3) directColorWave
```



Row	directColorWav	directColorWav	directColorWav	
	0	1	2	
0	0	0	0	Black
1	65535	0	0	Red
2	0	65535	0	Green
3	0	0	65535	Blue
4	65535	0	26214	Hot pink

You can use this wave to directly control the marker colors:

```
ModifyGraph zColor(yWave)={directColorWave,*,*,directRGB}
```

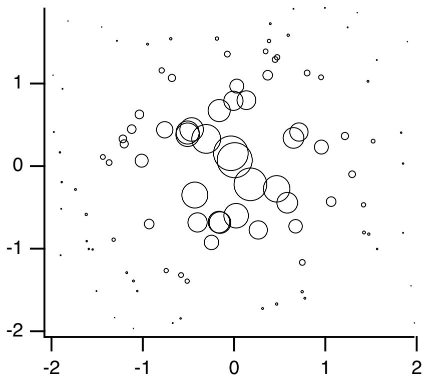


Marker Size as f(z)

“Marker size as f(z)” works just like “Color as f(z)” in Color Table mode except the Z values map into the range of marker sizes that you define using the min and max marker settings.

This example presents a third value as a function of marker size:

```
Make/N=100 xData,yData,zData
xData=enoise(2); yData=enoise(2); zData=exp(-(xData^2+yData^2))
Display yData vs xData; ModifyGraph mode=3,marker=8
ModifyGraph zmrkSize(yData)={zData,*,*,1,10}
```

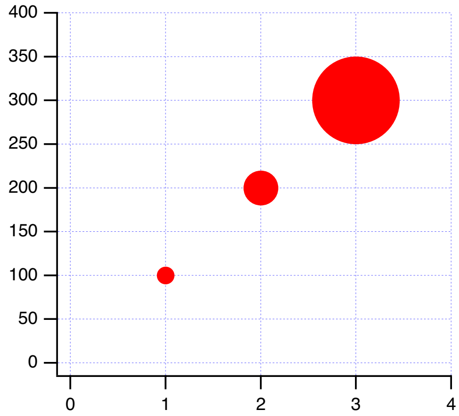


Marker Size as f(z) in Axis Units

In Igor Pro 9.00 or later, you can specify the marker size in units of an axis. For example:

```
Make/O xData2 = {1,2,3}
Make/O yData2 = {100,200,300}
Make/O zData2 = {10, 20, 50}
Display/W=(400,50,700,300) yData2 vs xData2
ModifyGraph mode=3, marker=19, msize=6 // Markers mode, filled circle marker
ModifyGraph grid=1
ModifyGraph nticks(left)=10
SetAxis left,0,400
SetAxis bottom,0,4
ModifyGraph zmrkSize(yData2)={zData2,*,*,6,10,left}
```

The last command sets the marker size to be controlled by the wave zData2 whose values are scaled to the left axis. For example, zData2[2] is 50 and sets the radius of the marker for point 2 of yData2 to be 50 units on the left axis. This produces this graph:



The axis specified can be any axis on the graph. If that axis is later removed, the marker size, the marker size goes back to the "no axis" state as if you never specified an axis.

When scaling marker sizes in axis units, the values that you specify for the zMin, zMax, mrkMin, and mrkMax parameters of the zmrkSize keyword have no impact on the marker size unless you remove the specified axis in which case those parameters control the trace's marker sizes.

If the marker size scale is independent of the scales of the axes against which the trace is plotted, you can use a free axis to show the scale. You do this by creating a free axis with the desired range and setting it as the axis for the zmrkSize keyword. See **Types of Axes** on page II-279 for a discussion of free axes.

If the axis that you specify is a log axis, the interpretation of Z values changes. Think of a conceptual axis of the same length in points as the log axis. Conceptually set the min and max of the conceptual axis to the log of the min and log of the max of the log axis. The marker size for a given Z value is the length of Z units on the conceptual axis. To help visualize this, you can create a free axis representing this conceptual axis.

For a demo, choose File→Example Experiments→Graphing Techniques→Marker Size in Axis Units Demo.

Marker Number as f(z)

In"Marker Number as f(z)" mode, you must create a Z wave that contains the actual marker numbers for each data point. See **Markers** on page II-291 for the marker number codes.

Pattern Number as f(z)

In “Pattern Number as f(z)” mode, you must create a Z wave that contains the actual pattern numbers for each data point. See **Fill Patterns** on page III-498 for a list of pattern numbers.

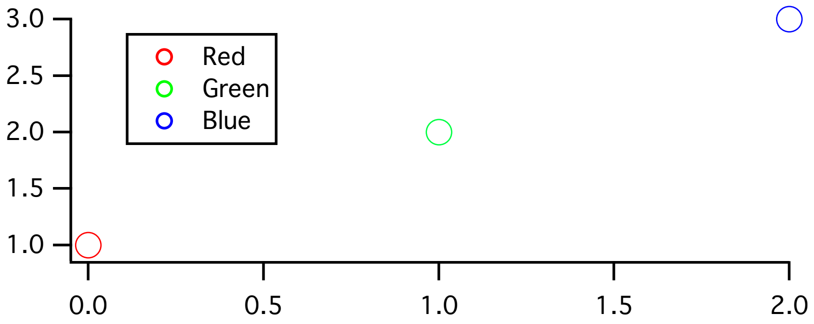
Color as f(z) Legend Example

If you have a graph that uses the color as f(z) mode, you may want to create a legend that identifies what the colors correspond to. This section demonstrates using the features of the Legend operation for this purpose.

Execute these commands, one-at-a-time:

```
// Make test data
Make /O testData = {1, 2, 3}

// Display in a graph in markers mode
Display testData
ModifyGraph mode=3,marker=8,msize=5

// Create a normal legend where the symbol comes from the trace
Legend/C/N=legend0/J/A=LT "\\s(testData) First\r\\s(testData)
Second\r\\s(testData) Third"

// Make a color index wave to control the marker color
Make /O testColorIndex = {0, 127, 225}

// Change the graph trace to use color as f(z) mode.
// Rainbow256 is the name of a built-in color table.
// The numbers 0 and 255 set the color index values that correspond to the
// first and last entries in the color table.
ModifyGraph zColor(testData)={testColorIndex,0,255,Rainbow256,0}

// Change the legend so that it shows the colors
Legend/C/N=legend0/J/A=LT "\\k(65535,0,0)\\W608 Red\r\\k(0,65535,0)\\W608
Green\r\\k(0,0,65535)\\W608 Blue"
```

The result is this graph:



The last command used the \W escape sequence to specify which marker to use in the legend (08 for the circle marker in this case) and the marker thickness (6 means 1.0 points).

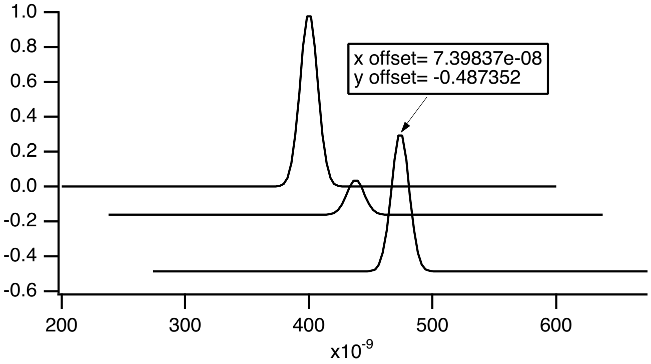
The \k escape sequence specifies the color to use for stroking the marker specified by \W. You would use \K to specify the marker fill color. Colors are specified in RGB format where each component falls in the range 0 to 65535.

This example uses double-backslashes because a single backslash is an escape character in Igor literal strings. Since we want a backslash in the final text, because that is what Igor requires for \k and \W, we need to use a double-backslash in the literal strings.

If you were to enter the legend text in the Add Annotation dialog, you would use just a single backslash and the dialog would generate the requires command, with double-backslashes.

Trace Offsets

You can offset a trace in a graph in the horizontal or vertical direction without changing the data in the associated wave. This is primarily of use for comparing the shapes of traces or for spreading overlapping traces out for better viewing.



Each trace has an X and a Y offset, both of which are initially zero. If you check the Offset checkbox in the Modify Trace Appearance dialog, you can enter an X and Y offset for the trace.

You can also set the offsets by clicking and dragging in the graph. To do this, click the trace you want to offset. Hold the mouse down for about a second. You will see a readout box appear in the lower left corner of the graph. The readout shows the X and Y offsets as you drag the trace. If it doesn't take too long to display the given trace, you will be able to view the trace as you drag it around on the screen.

If you press Shift while offsetting a wave, Igor constrains the offset to the horizontal or vertical dimension.

You can disable trace dragging by pressing Caps Lock, which may be useful for trackball users.

Offsetting is undoable, so if you accidently drag a trace where you don't want it, choose Edit →Undo.

It is possible to attach a tag to a trace that will show its current offset values. See **Dynamic Escape Codes for Tags** on page III-38, for details.

If autoscaling is in effect for the graph, Igor tries to take trace offsets into account. If you want to set a trace's offset without affecting axis scaling, use the Set Axis Range item in the Graphs menu to disable autoscaling.

When offsetting a trace that uses log axes, the trace offsets by the same distance it does when the axis is not log. The shape of the trace is not changed — it is simply moved. If you were to try to offset a trace by adding a constant to the wave's data, it would distort the trace.

Trace Multipliers

In addition to offsetting a trace, you can also provide a multiplier to scale a trace. The effective value used for plotting is then *multiplier*data+offset*. The default value of zero means that no multiplier is provided, not that the data should be multiplied by zero.

With normal (not log) axes, you can interactively scale a trace using the same click and hold technique described for trace offsets. First place Cursor A somewhere on the trace to act as a reference point. Then, after entering offset mode by clicking and holding, press Option (*Macintosh*) or Alt (*Windows*) to adjust the multiplier instead of the offset. You can press and release the key as desired to alternate between scaling and offsetting.

With log axes, the trace multiplier provides an alternative method of offsetting a trace on a log axis (remember: log(*a***b*)=log(*a*)+log(*b*)). For compatibility reasons and because the trace offset method better handles switching between log and linear axis modes, Igor uses the multiplier method when you drag a trace only if the offset is zero and the multiplier is not zero (the default meaning "not set"). Consequently, to use the multiplier technique, you must use the command line or the Offset controls in the Modify Trace Appearance dialog to set a nonzero multiplier. 1 is a good setting for this purpose.

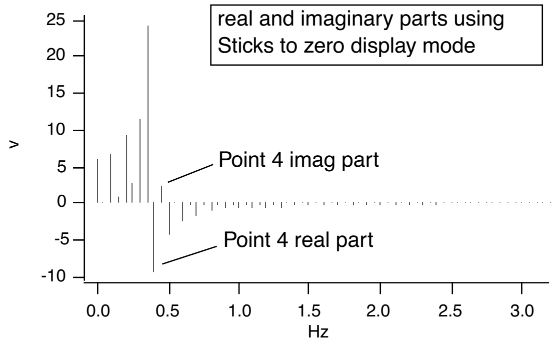
Hiding Traces

You can hide a trace in a graph by checking the Hide Trace checkbox in the Modify Trace Appearance dialog. When you hide a trace, you can use the Include in Autoscale checkbox to control whether or not the data of the hidden trace should be used when autoscaling the graph.

Complex Display Modes

When displaying traces for complex data you can use the Complex Mode pop-up menu in the Modify Trace Appearance dialog to control how the data are displayed. You can display the complex and real components together or individually, and you can also display the magnitude or phase.

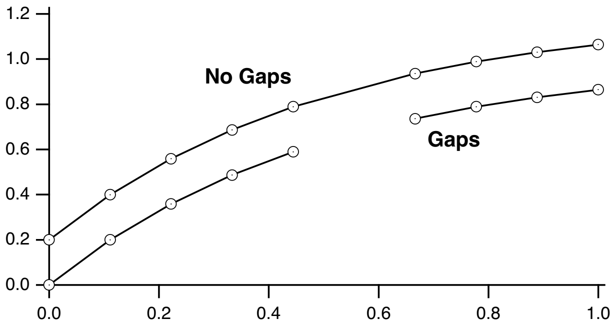
The default display mode is Lines between points. To display a wave's real and imaginary parts side-by-side on a point-for-point basis, use the Sticks to zero mode.



Gaps

In Igor, a missing or undefined value in a wave is stored as the floating point value NaN ("Not a Number"). Normally, Igor shows a NaN in a graph as a gap, indicating that there is no data at that point. In some circumstances, it is preferable to treat a missing value by connecting the points on either side of it.

You can control this using the Gaps checkbox in the Modify Trace Appearance dialog. If this checkbox is checked (the default), Igor shows missing values as gaps in the data. If you uncheck the checkbox, Igor ignores missing values, connecting the available data points on either side of the missing value.



Error Bars

The Error Bars checkbox in the Modify Trace Appearance dialog adds error bars to the selected trace. When you select this checkbox or click the Options button, Igor presents the Error Bars subdialog.

Error bars are a style that you can add to a trace in a graph. Error values can be a constant number, a fixed percent of the value of the wave at the given point, the square root of the value of the wave at the given point, or they can be arbitrary values taken from other waves. In this last case, the error values can be set independently for the up, down, left and right directions.

Choose the desired mode from the Y Error Bars and X Error Bars pop-up menus.

The dialog changes depending on the selected mode. For the “% of base” mode, you enter the percent of the base wave. For the “sqrt of base” mode, you don’t need to enter any further values. This mode is meaningful only when your data is in counts. For the “constant” mode, you enter the constant error value for the X or Y direction. For the “+/- wave” mode, you select the waves to supply the positive and negative error values.

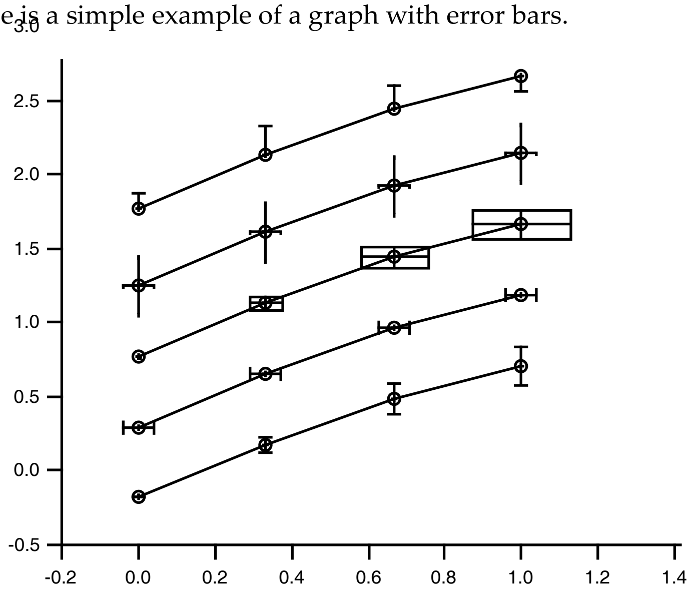
If you select “+/- wave”, pop-up menus appear from which you can choose the waves to supply the upper and lower or left and right error values. These waves are called **error waves**. The values in error waves should normally all be positive since they specify the length of the line from each point to its error bar. This is the only mode that supports single-sided error bars. Error waves do not have to have the same numeric type and length as the base wave. If the value of a point in an error wave is NaN then the error bar corresponding to that point is not drawn.

The Cap Width setting sets the width of the cap on the end of an error bar as an integral number of points. You can also set the cap width to “auto” (or to zero) in which case Igor picks a cap width appropriate for the size of the graph. In this case the cap width is set to twice the size of the marker plus one. For best results the cap width should be an odd number.

You can set the thickness of the cap and the thickness of the error bar. The units for these settings are points. These can be fractional numbers. Nonintegral thicknesses are properly produced when exporting or printing. If you set Cap Thickness to zero no caps are drawn. If you set Bar Thickness to zero no error bars are drawn.

If you enable the “XY Error box” checkbox then a box is drawn rather than an error bar to indicate the region of uncertainty. No box is drawn for points for which one or more of the error values is NaN.

Here is a simple example of a graph with error bars.



The top trace used the “+/- Wave” mode with only a +wave. The last value of the error wave was made negative to reverse the direction of the error bar.

Error Shading

In Igor Pro 7 or later, you can use shading in place of error bars. In the Error Bars subdialog, check the "Use shading instead of bars" checkbox to enable shading.

Shading mode fills between either the +error to -error levels or from +error to data and data to -error depending on the parameters used with the shade keyword.

These commands illustrate multiple and overlapping error shading with transparency:

```
Make/O/N=200 jack=sin(x/8), sam=cos(x/9) + (100-x)/30
Display /W=(64,181,499,520) jack,jack,sam
ModifyGraph lsize(jack)=0,rgb(sam)=(0,0,65535)
ErrorBars jack shade={0,0,(0,65535,0,10000),(0,0,0)},const=2
ErrorBars jack#1 shade={0,0,(0,65535,0,10000),(0,0,0)},const=1
ErrorBars sam shade={0,0,(65535,0,0,10000),(0,0,0)},const=1
```

On Windows shading does not work with the old GDI graphics technology. See **Graphics Technology** on page III-506 for details.

These commands illustrate different +error and -error shading as well as the use of pattern:

```
Make/O jack=sin(x/8)
Display /W=(64,181,499,520) jack
ErrorBars jack shade={0,73,(0,65535,0),(0,0,65535),11,(65535,0,0),(0,65535,0)},const=0.5
```

See the shade keyword for the **ErrorBars** operation for details.

Error Ellipses

In Igor Pro 9.00 and later, you can specify error ellipses instead of bars or boxes. Error ellipses show both X and Y errors along with correlation between X and Y.

You must provide the data for error ellipses via a three-column wave with a row for each data point. This wave is labelled Ellipse Wave in the Error Bars dialog. It is identified by ewave=*ew* in the **ErrorBars** operation.

Each row of *ew* contains information for the error ellipse for one data point. The interpretation of the columns of *ew* depends on the mode, labelled Data Type in the ErrorBars dialog and identified by the mode parameter of the ELLIPSE keyword in the ErrorBars operation.

mode=0: *ew* contains the standard deviation in X, the standard deviation in Y, and the correlation between X and Y.

mode=1: *ew* contains the variance in X, the variance in Y, and the covariance of X and Y.

When specified via the ErrorBars operation ew, may include a subrange specification so long as it results in effectively a 2D wave with three columns and a row for each trace data point. See **Subrange Display Syntax** on page II-321.

Error Ellipse Color

The color of each error ellipse is taken from the color of the data point to which the ellipse is attached. Usually that is simply the trace color. You can change the color of individual data points and their corresponding ellipses using the trace **Color as f(z)**, or by point customization (see **Customize at Point** on page II-306).

You can specify the opacity of all error ellipses for a given trace using the Fill Alpha setting in the Error Bars dialog which ranges from 0 (fully transparent) to 1.0 (fully opaque). This specifies the fill color alpha for all error ellipses, overriding the data points' alpha. In the ErrorBars operation, you specify alpha as a value from 0 (fully transparent) to 65535 (fully opaque) using the ELLIPSE keyword. For background information on alpha, see **RGBA Values** on page IV-13.

Error Ellipse Example

This example illustrates the use of error ellipses:

```
Function ErrorEllipseDemo()
   Make/O/N=5 data = 5*sin(x)
   Display data
   ModifyGraph mode=3, marker=19    // Markers mode, solid circle marker

   Make/O/N=(5,3) ellipseData       // Error ellipse parameters

   // Column 0: Standard deviation in X
   ellipseData[0][0]={0.944527,1.19611,0.32858,1.64494,1.71742}

   // Column 1: Standard deviation in Y
   ellipseData[0][1]={0.357613,0.680198,0.665561,0.581878,0.948245}

   // Column 2: Correlation in X and Y. Must be in the range [-1, 1].
   ellipseData[0][2]={0.562396,0.707809,0.396551,-0.516456,0.554322}

   ErrorBars data, ELLIPSE={0, .95, .3*65535}, ewave=ellipseData

   // Make point 2 blue
   ModifyGraph rgb(data[2])=(0,0,65535)
End
```

Customize at Point

You can customize the appearance of individual points on a trace in a graph displayed in bar, marker, dot and lines to zero modes. To do this interactively, right-click the desired point on a trace and choose Customize at Point→Customize at Point from the contextual menu. The Modify Trace dialog appears with an entry in the trace list shown with the point number in square brackets. When such an entry is selected, only those properties that can be customized are shown in the dialog.

You can remove customizations for a single point by right-clicking on the point and selecting Customize at Point→Remove Customization Here. You can remove all customizations from a trace by right-clicking on the trace and choosing Customize at Point→Remove Customizations from Trace.

Modifying Axes

You can modify the style of presentation of each axis in a graph by choosing Graph→Modify Axis or by double-clicking an axis. This displays the Modify Axis dialog.

The dialog has tabs for various aspects of axis appearance, plus a few controls outside the tabs. These global controls include the standard Igor dialog controls: Do It, To Cmd Line, To Clip, Help and Cancel buttons, plus a box to display the commands generated by the dialog. At the top are the Axis pop-up menu and the Live Update checkbox.

The Axis pop-up menu shows all axes that are in use in the top graph. Choose the axis that you want to change from the Axis menu, or choose Multiple Selection if you want to affect more than one axis. Below the Multiple Selection item are items that provide shortcuts for selecting certain classes of axes.

You select the appropriate tab for the types of changes you want to make. The dialog provides these tabs:

Axis Tab

Tick Options Tab

Axis Range Tab

Auto/Man Ticks Tab

Axis Label Tab

Ticks and Grids Tab

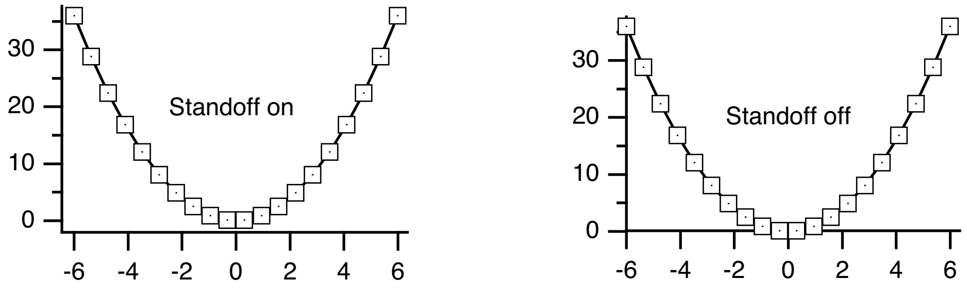
Label Options Tab

Axis Tab

You can set the axis Mode for the selected axis to linear, log base 10, log base 2, or Date/Time.

The Date/Time mode is special. When drawing an axis, Igor looks at the controlling wave's units to decide if it should be a date/time axis. Consequently, if you select Date/Time axis, the dialog immediately changes the units of the controlling wave. See **Date/Time Axes** on page II-315 for details on how date/time axes work.

The Standoff checkbox controls offsetting of axes. When standoff is on, Igor offsets axes so that traces do not cover them:

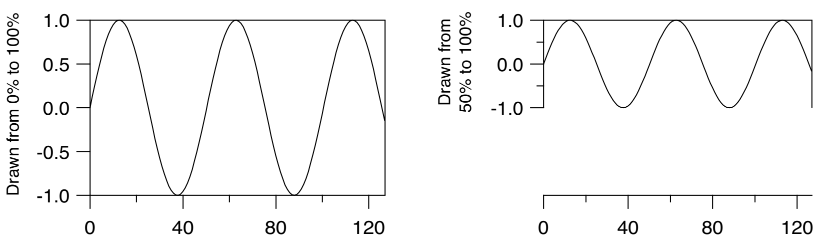


If a free axis is attached to the same edge of the plot rectangle as a normal axis then the standoff setting for the normal axis is ignored. This is to make it easy to create stacked plots.

Use the Mirror Axis pop-up menu to enable the mirror axis feature. A mirror axis is an axis that is the mirror image of the opposite axis. You can mirror the left axis to the right or the bottom axis to the top. The normal state is Off in which case there is no mirror axis. If you choose On from the pop-up, you get a mirror axis with tick marks but no tick mark labels. If you choose No Ticks, you get a mirror axis with no tick marks. If you choose Labels you get a mirror axis with tick marks and tick mark labels. Mirror axes may not do exactly what you want when using free axes, or when you shorten an axis using Draw Between. An embedded graph may be a better solution if free axes don't do what you need; see Chapter III-4, **Embedding and Subwindows**.

Free axes can also have mirror axes. Unlike the free axis itself, the mirror for a given free axis can not be moved — it is always attached to the opposite side of the plot area. This feature can create stacked plots; see **Creating Stacked Plots** on page II-324.

The "Draw between" items are used to create stacked graphs. You will usually leave these at 0 and 100%, which draws the axis along the entire length or width of the plot area. You could use 50% and 100% to draw the left axis over only the top half of the plot area (mirror axes are on in this example to indicate the plot area):



For additional examples using "Draw between", see **Creating Stacked Plots** on page II-324 and **Creating Split Axes** on page II-347.

The Offset setting provides a way to control the distance between the edge of the graph and the axis. It specifies the distance from the default axis position to the actual axis position. This setting is in units of the size of a zero character in a tick mark label. Because of this, the axis offset adjusts reasonably well when you change the size of a graph window. The default axis offset is zero. You can restore the axis offset to zero by dragging the axis to or beyond the edge of the graph. If you enter a graph margin (see **Overall Graph Properties** on page II-288), the margin overrides the axis offset.

Normally you will adjust the axis offset by dragging the axis in the graph. If the mouse is over an axis, the cursor changes to a double-ended arrow indicating that you can drag the axis. If the axis is a mirror axis you will not be able to drag it and the cursor will not change to the double-ended arrow.

The Offset setting does not affect a free axis. To adjust the position of a free axis, use the settings in the Free Position section.

The Thickness setting sets the thickness of an axis and associated tick marks in points. The thickness can be fractional and if you set it to zero the axis and ticks disappear.

The free position can be adjusted by dragging the axis interactively. This is the recommended way to adjust the position when using the absolute distance mode but it is not recommended when using the “crossing at” mode. This is because the crossing value as set interactively will not be exact. You should use the Free Position controls to specify an exact crossing value.

The Font section of the Axis tab specifies the font, font size, and typeface used for the tick labels and the axis label. You should leave this setting at “default” unless you want this particular axis to use a font different from the rest of the graph. You can set the default font for all graphs using the Default Font item in the Misc menu. You can set the default font for a particular graph using the Modify Graph item in the Graph menu. The axis label font can be controlled by escape codes within the axis label text. See **Axis Labels** on page II-318.

Colors of axis components are controlled by items in the Color area.

Auto/Man Ticks Tab

The items in the Auto/Man Ticks tab control the placement of tick marks along the axis. You can choose one of three methods for controlling tick mark placement from the pop-up menu at the top of the tab.

Choose Auto Ticks to have Igor compute nice tick mark intervals using some hints from you.

Choose Computed Manual Ticks to take complete control over the origin and interval for placing tick marks. See **Computed Manual Ticks** on page II-312 for details.

Choose User Ticks from Waves to take complete control over tick mark placement and labelling. See **User Ticks from Waves** on page II-313 for details.

In Auto Ticks mode, you can specify a *suggested* number of major ticks for the selected axis by entering that number in the Approximately parameter box. The actual number of ticks on the axis may vary from the suggested number because Igor juggles several factors to get round number tick labels with reasonable spacing in a common numeric sequence (e.g., 1, 2, 5). In most cases, this automatically produces a correct and attractive graph. The Approximately parameter is not available if the selected axis is a log axis.

You can turn minor ticks on or off for the selected axis using the Minor Ticks checkbox.

The Minimum Sep setting controls the display of minor ticks if minor ticks are enabled. If the distance between minor ticks would be less than the specified minimum tick separation, measured in points, then Igor picks a less dense ticking scheme. For log axes Minor Ticks and Tick Separation affect the drawing of subminor ticks.

Ticks and Grids Tab

The Ticks and Grids tab provides control over tick marks, tick mark labels, and grid lines.

Exponential Labels

When numbers that would be used to label tick marks become very large or very small, Igor switches to exponential notation, displaying small numbers in the tick mark labels and a power of 10 in the axis label. The use of the power of 10 in the axis label is covered under **Axis Labels** on page II-318. In the case of log axes, the tick marks include the power.

With the Low Trip and High Trip settings, you can control the point at which tick mark labels switch from normal notation to exponential notation. If the absolute value of the larger end of the axis is between the low trip and the high trip, then normal notation is used. Otherwise, exponential is used. However, if the exponent would be zero, normal notation is always used.

There are actually two independent sets of low trip and high trip parameters: one for normal axes and one for log axes. The low trip point can be from 1e-38 to 1 and defaults to 0.1 for normal axes and to 1e-4 for log axes. The high trip point can be from 1 to 1e38 and defaults to 1e4.

Under some circumstances, Igor may not honor your setting of these trip points. If there is no room for normal tick mark labels, Igor will use exponential notation, even if you have requested normal notation.

The Engineering and Scientific radio buttons allow you to specify whether tick mark labels should use engineering or scientific notation when exponential notation is used. It does not affect log axes. Engineering notation is just exponential notation where the exponent is always a multiple of three.

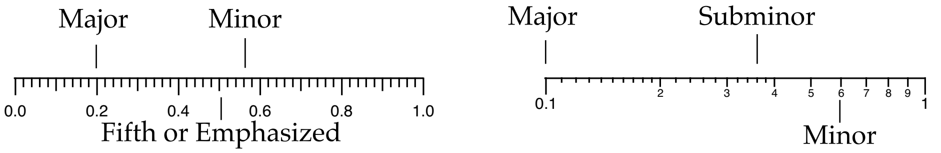
With the Exponent Prescale setting, you can force the tick and axis label scaling to values different from what Igor would pick. For example, if you have data whose X scaling ranges from, say, 9pA to 120pA and you display this on a log axis, Igor will label the tick marks with 10pA and 100pA. But if you really want the tick marks labeled 10 and 100 with pA in the axis label, you can set the Exponent Prescale to 12. For details, see **Axis Labels** on page II-318.

Date/Time Tick Labels

The Date/Time Tick Labels area of the Ticks and Grids tab is explained under **Date/Time Axes** on page II-315.

Tick Dimensions

You can control the length and thickness of each type of tick mark and the location of tick marks relative to the axis line using items in the Tick Dimensions area. Igor distinguishes four types of tick marks: major, minor, “fifth”, and subminor:



Normal Axis **Log Axis**

The tick mark thicknesses normally follow the axis thickness. You can override the thickness of individual tick types by replacing the word “Auto” with your desired thickness specified in fractional points. A value of zero is equivalent to “Auto”.

The tick length is normally calculated based on the font and font size that will be used to label the tick marks. You can enter your own values in fractional points. For example you might enter a value of 6 for the major tick mark, 3 for the minor tick mark and 4.5 for the 5th or emphasized minor tick marks. The subminor tick mark only applies to log axes.

Use the Location pop-up menu to specify that tick marks for the selected axis be outside the axis, crossing the axis or inside the axis or you can specify no tick marks for the axis at all.

Grid

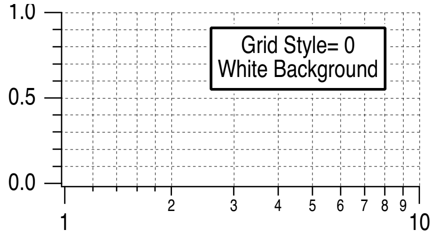
Choose Off from the Grid pop-up menu if you do not want grid lines. Choose On for grid lines on major and minor tick marks. Choose Major Only for grid lines on major tick marks only.

Igor provides five grid styles identified with numbers 1 through 5. Different grid styles have major and minor grid lines that are light, heavy, dotted or solid. If the style is set to zero (the default) and the graph background is white then grid style 2 is used. If the graph background is not white then grid style 5 is used.

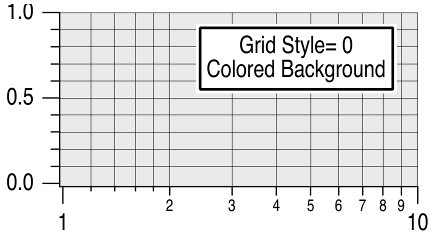
Use the Grid Color palette to set the color of the grid lines. They are by default light blue.

The grid line thickness is specified as a fraction of the axis line thickness. Since the axis line thickness is usually one point, and computer monitors usually have a resolution of about a point, it generally is not possible to see the differences in thickness on your screen. You can see the difference in exported and printed graphs.

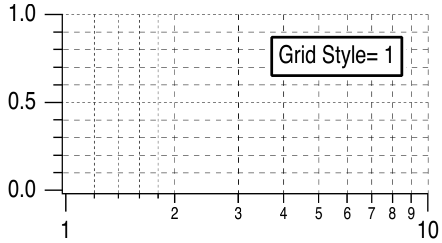
The examples here show graphs with thicker than normal axis lines and the thickest grid lines:

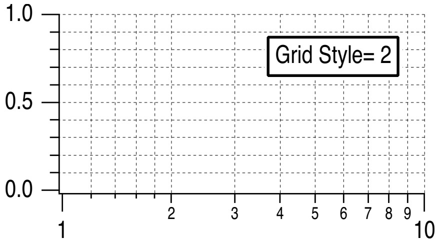


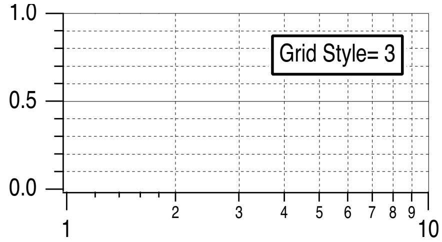
(same as gridStyle=2)

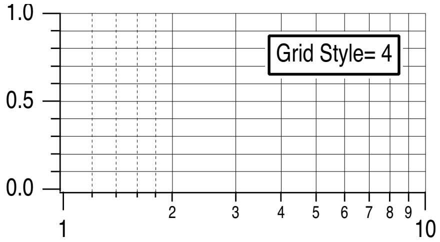


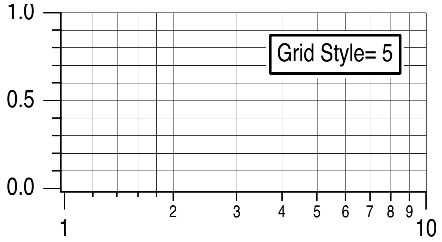
(same as gridStyle=5)











Using the settings "Draw from" and "to" you can restrict the length of the grid lines. This is useful if you have used the similar settings on the Axis tab to shorten one of your axes, and you want grid lines to match.

Zero Line

You can control the zero line for the selected axis using the Zero Line checkbox.

The zero line is a line perpendicular to the axis extending across the graph at the point where the value of the axis is zero. The Zero Line checkbox is not available for log axes.

If you turn the zero line on then you will be able to choose the line style from the Style pop-up menu. The thickness of the line can be set in fractional points from 0 to 5. The zero line has the same color as the axis.

Tick Options Tab

The Tick Options tab provides fine control over tick marks and tick mark labels.

The Enable/Inhibit Ticks section allows you to limit tick marks to a specific range and to suppress specific tick marks.

The Log Ticks section provides control over minor tick marks and labels on log axes.

The Tick Label Tweaks section provides the following settings:

Checkbox	Result
Thousands separator	Tick labels like 10000 are drawn as 10,000.
Zero is '0'	Select this to force the zero tick mark to be drawn as 0 where it would ordinarily be drawn as 0.0 or 0.00.
No trailing zeroes	Tick labels that would normally be drawn as 1.50 or 2.00 are drawn as 1.5 or 2.
No leading zero	Select if you want tick labels such as 0.5 to be drawn as .5
Tick Unit Prefix is Exponent	If tick mark would have prefix and units (µTorr), force to exponential notation (10^{-6} Torr).
No Units in Tick Labels	If tick mark would have units, suppress them.
Units in Every Tick Label	If normal axis, force exponent or prefix and units into each label.

Axis Label Tab

See **Axis Labels** on page II-318.

Label Options Tab

The Label Options tab provides control of the placement and orientation of axis and tick mark labels. You can also hide these labels completely.

Normally, you will adjust the position of the axis label by simply dragging it around on the graph. The “Axis label position” or “Axis label margin” and the “Axis label lateral offset” settings are useful when you want precise numeric control over the position.

The calculations used to position the axis label depend on the setting in the Label Position Mode menu. By default this is set to Compatibility, which will work with older versions of Igor. The other modes may allow you to line up labels on multiple axes more accurately. The choice of positioning mode affects the meaning of the three settings below the menu.

In Compatibility mode, the method Igor uses to position the axis label depends on whether or not a free axis is attached to the given plot rectangle edge. If no free axis is attached then the label position is measured from the corresponding window edge. We call this the axis label margin. Thus if you reposition an axis the axis label will not move. On the other hand, if a free axis is attached to the given plot rectangle edge then the label position is measured from the axis and when you move the axis, the label will move with it.

Because the method used to set the axis label varies depending on circumstances, one or the other of the “Axis label margin” or “Axis label position” settings may be unavailable. If you have selected an axis on the same edge as a free axis, the “Axis label position” setting is made available. If you have selected an axis that does not share an edge with a free axis, the “Axis label margin” setting is made available. If you have selected multiple axes it is possible for both settings to be available.

The axis label position is the distance from the axis to the label and is measured in points.

The axis label margin is the distance from the edge of the graph to the label and is measured in points. The default label margin is zero which butts the axis label up against the edge of the graph.

The margin modes measure relative to an edge of the graph while the axis modes measure relative to the position of the axis. Using an axis mode causes the label to follow a free axis when you move the axis. The margin modes are useful for aligning labels on stacked graphs. The “Axis label margin” setting applies to margin modes while the “Axis label position” setting applies to axis modes.

The absolute modes measure distance in points. Scaled modes have similar numerical values but are scaled to respond to changes in the font size.

The Labels pop-up menu controls which labels are drawn. On gives normal axis labeling. Axis Only leaves the axis label in place but removes the tick mark labels. Off removes the axis labels and tick mark labels.

Axis and Tick label rotations can be set to any value between -360 and 360 degrees.

Axis Range Tab

See **Scaling Graphs** on page II-285.

Manual Ticks

If Igor’s automatic selection of ticks does not suit you, and you can’t find any adjustments that make the tick marks just the way you want them, Igor provides two methods for specifying the tick marks yourself. On the Auto/Man Ticks tab of the Modify Axis dialog, you can choose either Computed Manual Ticks or User Ticks from Waves.

Computed Manual Ticks

Use Computed Manual Ticks to enter a numeric specification of the increment between tick marks and the starting point for calculating where the tick marks fall. This style of manual ticking is available for normal axes and date/time axes. It is not available for normal log axes but is available in LogLin mode.

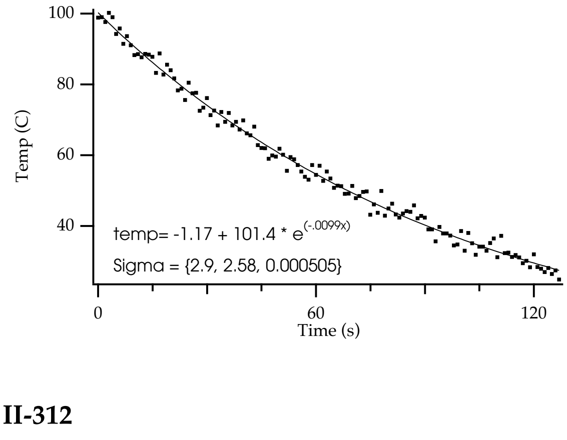
When you choose Computed Manual Ticks, the corresponding settings in theAuto/Man Ticks tab becomes available.

If you click the “Set to auto values” button, Igor sets all of the items in the Compute Manual Ticks section to the values they would have if you let Igor automatically determine the ticking. This is usually a good starting point.

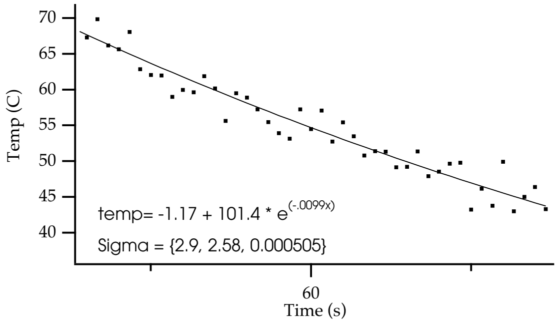
Using the “Canonic tick” setting, you specify the value of any major tick mark on the axis. Using the “Tick increment” setting, you specify the number of axis units per major tick mark. Both of these numbers are specified as a mantissa and an exponent. The canonic tick is not necessarily the first major tick on the axis. Rather, it is a major tick on an infinitely long axis of which the axis in the graph is a subset. That is, it can be *any* major tick whether it shows on the graph or not.

When you use computed manual ticks on a large range logarithmic axis in LogLin mode, the settings in the dialog refer to the exponent of the tick value.

Imagine that you want to show the temperature of an object as it cools off. You want to show time in seconds but you want it to be clear where the integral minutes fall on the axis. You would turn on manual ticking for the bottom axis and set the canonic tick to zero and the tick increment to 60. You could show the half and quarter minute points by specifying three minor ticks per major tick (“Number per major tick” in the Minor Ticks section) with every second minor tick emphasized (“Emphasize every” setting). This produces the following graph:



Now, imagine that you want to zoom in on t = 60 seconds.



The canonic tick, at t = 0, does not appear on the graph but it still controls major tick locations.

User Ticks from Waves

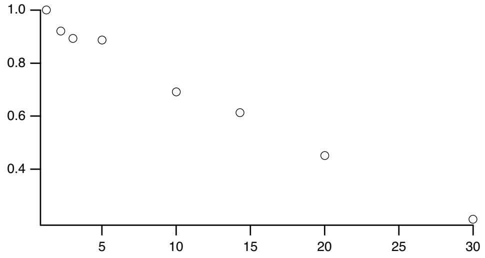
With Computed Manual Ticks you have complete control over ticking as long as you want equally-spaced ticks. If you want to specify your own ticking on a normal log axis, or you want ticks that are not equally spaced, you need User Ticks from Waves.

The first step in setting up User Ticks from Waves is to create two waves: a 1D numeric wave and a text wave. Numbers entered in the numeric wave specify the positions of the tick marks in axis units. The corresponding rows of the text wave give the labels for the tick marks.

Perhaps you want to plot data as a function of Tm/T (melting temperature over temperature, but you want the tick labels to be at nice values of temperature. Starting with this data:

Point	InverseTemp	Mobility
0	30	0.211521
1	20	0.451599
2	14.2857	0.612956
3	10	0.691259
4	5	0.886406
5	3.0303	0.893136
6	2.22222	0.921083
7	1.25	1

you might have this graph:



Create the waves for labelling the axes:

```
Make/N=5 TickPositions
Make/N=5/T TickLabels
```

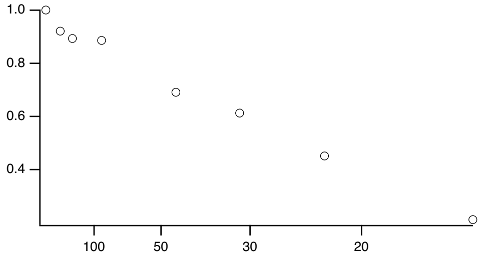
Assuming that Tm is 450 degrees and that you have determined that tick marks at 20, 30, 50, 100, and 400 degrees would look good, you would enter these numbers in the text wave, TickLabels. At this point, a convenient way to enter the tick positions in the TickPositions wave is a wave assignment that embodies the relationship you think is appropriate:

```
TickPositions = 450/str2num(TickLabels)
```

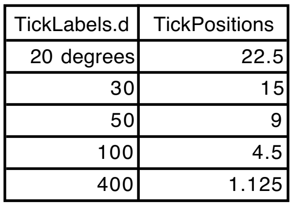
Note that the str2num function was used to interpret the text in the label wave as numeric data. This only works, of course, if the text includes only numbers.

Finally, double-click the bottom axis to bring up the Modify Axis dialog, select the Auto/Man Ticks tab and select User Ticks from Waves. Choose the TickPositions and TickLabels waves:

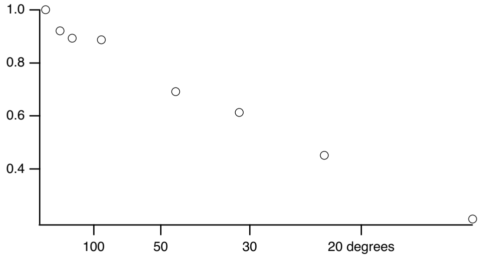
The result is this graph:



You can add other text to the labels, including special marks. For instance:

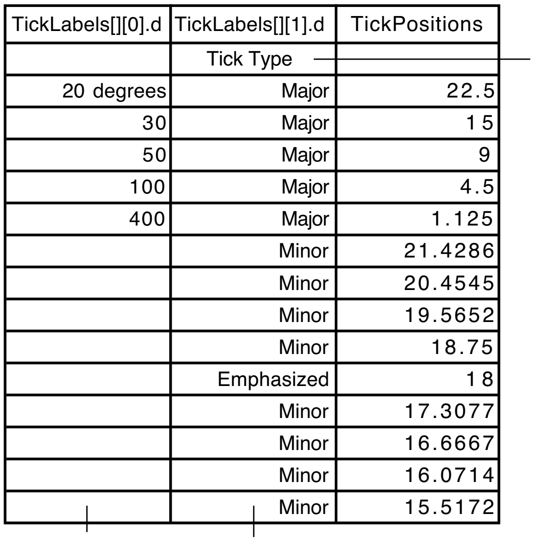


TickLabels.d	TickPositions
20 degrees	22.5
30	15
50	9
100	4.5
400	1.125



Finally, you can add a column to the text wave and add minor, subminor and emphasized ticks by entering appropriate keywords in the other column. To add a column to a wave, select Redimension Waves from the Data menu, select your text wave in the list and click the arrow. Then change the number of columns from 0 to 2.

This extra column must have the column label 'Tick Type'. For instance:



TickLabels[][0].d	TickLabels[][1].d	TickPositions
	Tick Type	
20 degrees	Major	22.5
30	Major	15
50	Major	9
100	Major	4.5
400	Major	1.125
	Minor	21.4286
	Minor	20.4545
	Minor	19.5652
	Minor	18.75
	Emphasized	18
	Minor	17.3077
	Minor	16.6667
	Minor	16.0714
	Minor	15.5172

Dimension label "Tick Type" has keywords to set tick types

Blank entries make ticks with no labels.

Use keyword "Subminor" for subminor ticks such as Igor uses on log axes.

Dimension labels allow you (or Igor) to refer to a row or column of a wave using a name rather than a number. Thus, the Tick Type column doesn't have to be the second column (that is, column 1). For instructions on showing dimension labels in a table, see **Showing Dimension Labels** on page II-235.

An easy way to get started is to let the Modify Axis dialog generate waves for you. Double-click the axis for which you want user ticks. Click the Auto/Man Ticks tab and select User Ticks from Waves from the pop-up menu. Click the New from Auto Ticks button. Igor immediately generates waves with names of the form <graphname>_<axisname>_labels and <graphname>_<axisname>_values and selects them in the Labels and Locations pop-up menus. Click Do It. If you now edit the generated waves, the ticks in the graph will change. You can achieve the same thing programmatically using the **TickWavesFromAxis** operation.

Log Axes

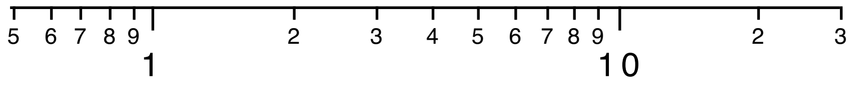
To create a logarithmic axis, set the axis mode to Log in the Axis tab of the Modify Axis dialog.

Computed manual ticks and zero lines are not supported for normal log axes.

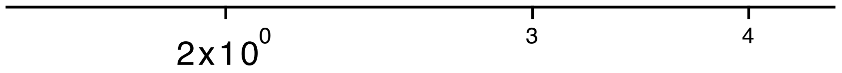
Igor has three ways of ticking a log axis that are used depending on the range (number of decades) of the axis: normal, small range and large range. The normal mode is used when the number of decades lies between about one third to about ten. The exact upper limit depends on the physical size of the axis and the font size.

If the number of decades of range is less than two or greater than five, you can force Igor to use the small/large range methods by checking the LogLin checkbox, which may give better results for log axes with small or very large range. When you do this, all of the settings of a linear axis are enabled including manual ticking.

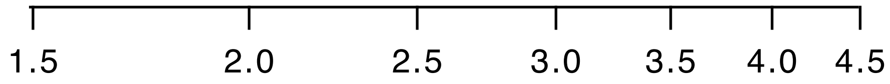
Here is a normal log axis with a range of 0.5 to 30:



If we zoom into a range of 1.5 to 4.5 we get this:



But if we then check the LogLin checkbox, we get better results:



Selecting a log axis makes the Log Ticks box in the Tick Options tab available.

The "Max log cycles with minor ticks" setting controls whether minor ticks appear on a log axis. This setting can range from 0 to 20 and defaults to 0. If it is 0 or "auto", Igor automatically determines if minor ticks are appropriate. Otherwise, if the axis has more decades than this number, then the minor ticks are not displayed. Minor ticks are also not displayed if there is not enough room for them.

Similarly, you can control when Igor puts labels on the minor ticks of a log axis using the "Max log cycles with minor tick labels" item. This is a number from 0 to 8. 0 disables the minor tick labels. As long as the axis has fewer decades than this setting, minor ticks are labeled.

Date/Time Axes

In addition to numeric axes, Igor supports axes labeled with dates, times or dates and times.

Dates and date/times are represented in Igor as the number of seconds since midnight, January 1, 1904.

In Igor, a date can not be accurately represented in a single precision wave. Make sure you use double precision waves to store dates and date/times. (A single precision wave can provide dates and date/times calculated from its X scaling, but not from its data values.)

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-85.

Times without dates can be thought of in two ways: as time-of-day times and as elapsed times.

Time-of-day times are represented in Igor as the number of seconds since midnight.

Elapsed times are represented as a number of seconds in the range -9999:59:59 to +9999:59:59. For integral numbers of seconds, this range of elapsed times can be precisely represented in a signed 32-bit integer wave. A single-precision floating point wave can precisely represent integral-second elapsed times up to about +/-4600 hours.

Igor displays dates or times on an axis if the appropriate units for the wave controlling the axis is “dat”. This is case-sensitive — “Dat” won’t work. You can set the wave’s units using the Change Wave Scaling item in the Data menu, or the **SetScale** operation.

To make a horizontal axis a date or time axis for a waveform graph, you must set the X units of the wave controlling the axis to “dat”. For an XY graph you must set the data units of the wave supplying the X coordinates for the curve to “dat”. To make the vertical axis a date or time axis in either type of graph, you must set the data units of the wave controlling the axis to “dat”.

If you choose Date/Time as the axis mode in the Axis tab of the Modify Axis dialog, the dialog sets the appropriate wave units to “dat”.

For Igor to display an axis as date or time, the following additional restrictions must be met: the axis must span at least 2 seconds and both ends must be within the legal range for a date/time value. If any of these restrictions is not met, Igor displays a single tick mark.

When an axis is in the date/time mode, the Date/Time Tick Labels box in the Ticks and Grids tab of the Modify Axis dialog is available.

From the Time Format pop-up menu, you can choose Normal, Military, or Elapsed. Use Normal or Military for time-of-day times and Elapsed for elapsed times. In normal mode, the minute before midnight is displayed as 11:59:00 PM and midnight is displayed as 12:00:00 AM. In military mode, they are displayed as 23:59:00 and 00:00:00.

Elapsed mode can display times from -9999:59:59 to +9999:59:59. This mode makes sense if the values displayed on the axis are actually elapsed times (e.g., 23:59:00). It makes no sense and will display no tick labels if the values are actually date/times (e.g., 7/28/93 23:59:00).

Custom Date Formats

In the short, long and abbreviated modes, dates are displayed according to system date/time settings. If you choose Other from the Date Format pop-up, a dialog is displayed giving you almost complete control over the format of the tick labels. The dialog allows you to choose from a variety of built-in formats or to create a fully custom format.

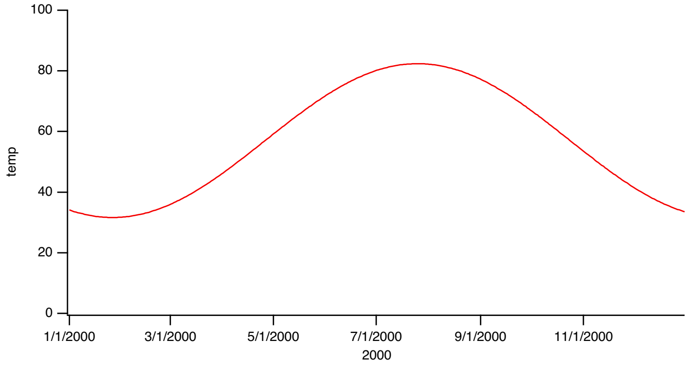
Depending on the extent of the axis, the tick mark labels may show date or date and time. You can suppress the display of the date when both the date and time are showing by selecting the Suppress Date checkbox. This checkbox is irrelevant when you choose the elapsed time mode in which dates are never displayed.

Date/Time Example

The following example shows how you can create a date/time graph of a waveform whose Y values are temperature and whose X values, as set via the SetScale operation, are dates:

```
// Make a wave to contain temperatures for the year
Make /N=365 temperature             // single precision data values
```

```
// Set its scaling so X values are dates
Variable t0, t1
t0 = Date2Secs(2000,1,1); t1 = Date2Secs(2001,1,1)
SetScale x t0, t1, "dat", temperature  // double-precision X scaling

// Enter the temperature data in the wave's Y values
t0 = Date2Secs(2000,1,1); t1 = Date2Secs(2000,3,31)    // winter
temperature(t0, t1) = 32                               // it's cold
t0 = Date2Secs(2000,4,1); t1 = Date2Secs(2000,6,30)    // spring
temperature(t0, t1) = 65                               // it's nice
t0 = Date2Secs(2000,7,1); t1 = Date2Secs(2000,9,31)    // summer
temperature(t0, t1) = 85                               // it's hot
t0 = Date2Secs(2000,10,1); t1 = Date2Secs(2000,12,31) // fall
temperature(t0, t1) = 45                               // cold again

// Smooth the data out
CurveFit sin temperature
temperature= K0+K1*sin(K2*x+K3)

// Graph the wave
Display temperature
SetAxis left, 0, 100;Label left "temp"
Label bottom "2000"
```



The SetScale operation sets the temperature wave so that its X values span the year 2000. In this example, the date/time information is in the *X values* of the wave. X values are always double precision. The wave itself is not declared double precision because we are storing temperature information, not date/time information in the Y values.

Manual Ticks for Date/Time Axes

Just as with regular axes, there are times when Igor’s automatic choices of ticks for date/time axes simply are not what you want. For these cases, you can use computed manual ticks with date/time axes.

To use computed manual ticks, display the Modify Axis dialog by double-clicking the axis, or by choosing Graph→Modify Axis. Select the Auto/Man Ticks tab, and choose Computed Manual Ticks from the menu in that tab.

The first step is to click the Set to Auto Values button. Choose Date, Time, or Date&Time from the pop-up menu below the Canonic Tick setting. This will depend on the range of the data. Choose the units for the Tick Increment setting and enter an increment value.

“Fake” Axes

It is sometimes necessary to create an axis that is not related to the data in a simple way. One method uses free axes that are not associated with a wave (see **NewFreeAxis**). The Transform Axis package uses this technique to make a mirror axis reflecting a different view of the data. An example would be a mirror axis showing wave number to go with a main axis showing wavelength. For an example, choose File→Example Experiments→Graphing Techniques→Transform Axis Demo.

Another technique is to use Igor's drawing tools to create fake axes. For an example, choose File→Example Experiments→Graphing Techniques→New Polar Graph Demo or File→Example Experiments→Graphing Techniques→Ternary Diagram Demo.

Axis Labels

The text for an axis label in a graph can come from one of two places. If you specify units for the wave which controls an axis, using the Change Wave Scaling dialog, Igor uses these units to label the axis. You can override this labeling by explicitly entering axis label text using the Axis Label tab of the Modify Axis dialog.

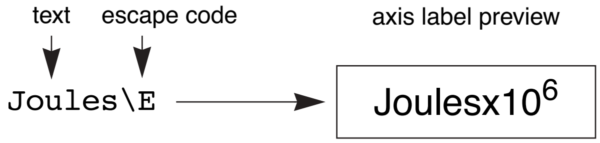
To display the dialog, choose Graph→Label Axis or double-click an axis label. Select the axis that you want to label from the Axis pop-up menu and then enter the text for the axis label in the Axis Label area. Further label formatting options are available in the **Label Options Tab**.

There are two parts to an axis label: the text for the label and the special effects such as font, font size, superscript or subscript. You specify the text by typing in the Axis Label area. At any point in entering the text, you can choose a special effect from a pop-up menu in the Insert area.

The Label Preview area shows what the axis label will look like, taking the text and special effects into account. You can not enter text in the preview. You can also see your label on the graph if you check the Live Update checkbox.

Axis Label Escape Codes

When you choose a special effect, Igor inserts an **escape code** in the text. An escape code consists of a backslash character followed by one or more characters. It represents the special effect you chose. The escape codes are cryptic but you can see their effects in the Label Preview box.



You can insert special affects at any point in the text by clicking at that point and choosing the special effect from the Insert pop-ups.

Choosing an item from the Font pop-up menu inserts a code that changes the font for subsequent characters in the label. The font pop-up also has a "Recall font" item. This item is used to make elaborate axis labels. See **Elaborate Annotations** on page III-51.

Choosing an item from the Font Size pop-up menu inserts a code that changes the font size for subsequent characters in the label. The font size pop-up also has a "Recall size" item used to make elaborate axis labels.

Axis Label Special Effects

The **Special** pop-up menu includes items for controlling many features including superscript, subscript, justification, and text color, as well as items for inserting special characters, markers and pictures.

The Store Info, Recall Info, Recall X Position, and Recall Y Position items are used to create elaborate annotations. See **Elaborate Annotations** on page III-51.

The most commonly used items are Superscript, Subscript and Normal. To create a superscript or subscript, use the Special pop-up menu to insert the desired code, type the text of the superscript or subscript and then finish with the Normal code. For example, suppose you want to create an axis label that reads "Phase space density (s^{3}m^{-6})". To do this, type "Phase space density (s", choose the Superscript item from the Special pop-up menu, type "3", choose Normal, type "m", choose Superscript, type "-6", choose Normal and then type ")". See Chapter III-2, **Annotations**, for a complete discussion of these items.

The "Wave controlling axis" item inserts a code that prints the name of the first wave plotted on the given axis.

The Trace Symbol submenu inserts a code that draws the symbol used to plot the selected trace.

The Character submenu presents a palette from which you can select special characters to add to the axis label.

The Marker submenu inserts a code to draw a marker symbol. These symbols are independent of any traces in the graph.

Axis Label Units

The items in the Units pop-up menu insert escape codes that allow you to create an axis label that automatically changes when the extent of the axis changes.

For example, if you specified units for the controlling wave of an axis, you can make those units appear in the axis label by choosing the Units item from the Units pop-up menu. If appropriate Igor will automatically add a prefix (µ for micro, m for milli, etc.) to the label and will change the prefix appropriately if the extent of the axis changes. The extent of the axis changes when you explicitly set the axis or when it is autoscaled.

If you choose the Scaling or Inverse Scaling items from the Units pop-up menu, Igor automatically adds a power of 10 scaling (x10^3, x10^6, etc.) to the axis label if appropriate and changes this scaling if the extent of the axis changes. The Trial Exponent buttons determine what power is used *only* in the label preview so you can see what your label will look like under varying axis scaling conditions. Both of these techniques can be ambiguous — it is never clear if the axis has been multiplied by the scale factor or if the units contain the scale factor.

A less ambiguous method is to use the Exponential Prefix escape code. This is identical to the Scaling code except the “x” is missing. You can then use it in a context where it is clear that it is a multiplier of units. For example, if your axis range is 0 to 3E9 in units of cm/s, typing “Speed, \ucm/s” would create “Speed, 10^9cm/s”.

It is common to parenthesize scaling information in an axis label. For example the label might say “Joules (x10^6)”. You can do this by simply putting parentheses around the Scaling or Inverse Scaling escape codes. If the scaling for the axis turns out to be x10^0 Igor omits it and also omits the parentheses so you get “Joules” instead of “Joules (x10^0)” or “Joules()”.

If you do not specify scaling but the range of the axis requires it, Igor labels one of the tick marks on the axis to indicate the axis scaling. This is an emergency measure to prevent the graph from being misleading. You can prevent this from happening by inserting the Manual Override escape code, \u#2, into your label. No scaling or units information will be added at the location of the escape code or on the tick marks.

The situation with log axes is a bit different. By their nature, log axes never have to be scaled and units/scaling escape codes are not used in axis labels. If the controlling wave for a log axis has units then Igor automatically uses the units along with the appropriate prefix for each major tick mark label.

Annotations in Graphs

You can add text annotation to a graph by choosing Graph→Add Annotation. This displays the Add Annotation dialog. If text annotation is already on the graph you can modify it by double-clicking it. This brings up the Modify Annotation dialog. See Chapter III-2, **Annotations**, for details.

Info Panel and Cursors

You can display an information panel (“info panel” for short) for a graph by choosing Graph→Show Info. An info panel displays a precise readout of values for waves in the graph. To remove the info panel from a graph while the graph is the target window choose Graph→Hide Info.

You can use up to five different pairs of cursors (AB through IJ). To control which pairs are available, click the gear icon and select select cursor pairs from the Show Cursor Pairs submenu. By default, cursors beyond B use the cross and letter style.

Cursors provide a convenient way to specify a range of points on a wave which is of particular interest. For example, if you want to do a curve fit to a particular range of points on a wave you can start by putting cursor A on one end of the range and cursor B on the other. Then you can summon the Curve Fitting dialog from the Analysis menu. In this dialog on the Data Options tab there is a range control. If you click the “cursors” button then the range of the fit will be set to the range from cursor A to cursor B.

Using Cursors

When you first show the info panel, the cursors are at home and not associated with any wave. The slide control is disabled and the readout area shows no values.

To activate a cursor, click it and drag it to the desired point on the wave whose values you want to examine. Now the cursor appears on the graph and the cursor’s home icon turns black indicating that the cursor is active. The name of the wave which the cursor is on appears next to the cursor’s name. You can drag the cursor to any point on a trace or image plot.

To move the cursor by one element, use the arrow keys. If the cursor is on a trace, you can use the left and right arrow keys. If it is on an image, you can use the left, right, up, and down arrow keys. If you press Shift plus an arrow key, the cursor moves 10 times as far.

If the cursor is on a trace, you can drag the slide control left or right. If it is on an image, you can drag it in any direction.

If you have both cursors of a pair on the graph and both are active, then the slide control moves both cursors at once. If you want to move only one cursor you can use the mouse to drag that cursor to its new location. Another way to move just one cursor is to deactivate the cursor that you don’t want to move. You do this by clicking in the cursor’s home icon. This makes the home icon change from black to white indicating that the cursor is not active. Then the slide control moves only the active cursor.

You can also move both cursors of a pair at once by dragging. With both cursors on the graph, press the Shift key before clicking and dragging one of the cursors.

When you use the mouse to drag a cursor to a new location, Igor first searches for the trace the cursor is currently attached to. Only if the new location is not near a point on the current trace are all the other traces searched. You can use this preferential treatment of the current trace to make sure the cursor lands on the desired trace when many traces are overlapping in the destination region.

You can attach a cursor to a particular trace by right-clicking the cursor home icon and choosing a trace name from the pop-up menu.

To remove a cursor from the graph, drag it outside the plot area or right-click the cursor home icon and choose Remove Cursor.

Free Cursors

By default, cursors are attached to a particular point on a trace or to a particular element of an image plot. By contrast, you can move a free cursor anywhere within the plot area of the graph. To make a cursor free, right-click the cursor home icon in the info panel and choose Style→Free from the resulting pop-up menu.

Cursor Styles

By default, cursors A and B are displayed in the graph using a circular icon for A and a square icon for B. For all other cursors, the default style is a cross. You can change the style for any cursor by right-clicking the cursor home icon in the info panel and using the resulting pop-up menu.

You can create a cursor style function which you can invoke later to apply a given set of cursor style settings Right-click the cursor home icon in the info panel and, from the resulting pop-up menu, choose Style→Style Function→Save Style Function. Igor creates a cursor style function in the built-in procedure window. You can edit the function to give it a more meaningful name than the default name that Igor uses.

To apply your style function to a cursor, right-click the cursor home icon in the info panel and choose choose Style→Style Function→<Your Style Function>.

Programming With Cursors

These functions and operations are useful for programming with cursors.

The **ShowInfo** and **HideInfo** operations show and hide the info panel.

The **Cursor** operation sets the position of a cursor. It can also be used to change characteristics of the cursor such as the color, hair style, and number of digits used in the Graph Info Panel display.

The **CsrInfo** function returns information about a cursor.

These functions return the current position of a cursor:

pcsr	qcsr	hcsr	vcsr	xcsr	zcsr

These functions return information about the wave to which a cursor is attached, if any:

CsrWave	CsrWaveRef	CsrXWave	CsrXWaveRef

The **CursorStyle** keyword marks a user-defined function for inclusion in the Style Function submenu of the cursor pop-up menu.

The section **Cursors — Moving Cursor Calls Function** on page IV-339 explains how to trigger a user-defined function when a cursor is moved.

Identifying a Trace

Igor can display a tooltip that identifies a trace when you hover the mouse over it. To enable this mode, choose Graph→Show Trace Info Tags.

Subrange Display

In addition to displaying an entire wave in a graph, you can specify a subrange of the wave to display. This feature is mainly intended to allow the display of columns of a matrix as if they were extracted into individual 1D waves but can also be used to display other subsets or to skip every *n*th data point.

To display a subrange of a graph using the New Graph and Append Traces dialogs, you must be in the more complex version of the dialogs which appears when you click the More Choices button. Select your Y wave and optionally an X wave and click the Add button. This adds the trace to the list below. You can then edit the subrange in the list.

Subrange Display Syntax

The **Display** operation (page V-161), **AppendToGraph** operation (page V-35), and **ReplaceWave** operation (page V-801) support the following subrange syntax for a wave list item:

```
wavename[rdspec][rdspec][rdspec][rdspec]
```

where *rdspec* is a range or dimension specification expressed as dimension indices (point numbers for 1D waves). For an n-dimensional wave, enter n specifications and omit the rest.

Only one *rdspec* can be a range spec. The others must be a single numeric element index or dimension label value.

You can enter `[]` or `[*]` to indicate the entire range of the dimension, or `[start,stop]` for a contiguous subrange, or `[start,stop;inc]` where `start`, `stop`, and `inc` are dimension indices. Entering * for `stop` is the same as enteringthe index of the last element in the dimension.

For example:

```
Make/N=100 w1D = p
Display w1D[0,*;10]                // Display every tenth point
ModifyGraph mode=3, marker=19

Make/N=(10,8) w2D = p + 10*q
Display w2D[0][0,*;2]              // Display every other column of row 0
ModifyGraph mode=3, marker=19
```

The subrange syntax rules can be restated as:

1. Only one dimension specifier can contain the range to be displayed.

   Legal syntax for range is: `[]` or `[*]` for an entire dimension

   `[start,stop]` for a subrange

   `stop` may be *

   `stop` must be >= `start`

   The range is inclusive

   `[start,stop;inc]` for a subrange with a positive increment

2. Other dimensions must contain a single numeric index or dimension label using % syntax.

   Legal syntax for nonrange specifier is: `[index]`

   `[%label]`

3. Unspecified higher dimensions are treated as if [0] was specified.

For non-XY plots, the X-axis label uses the dimension label (if any) for the active dimension (the one with a range).

When cursors or tags are placed on a subranged trace, the point number used is the virtual point number as if the subrange had been extracted into a 1D wave.

Subrange syntax is also supported for waves used with error bars and with color, marker size and marker number as f(Z). These correspond to the **ErrorBars** operation (page V-199) with the wave keyword and to the **ModifyGraph (traces)** operation (page V-613) with the zmrkSize, zmrkNum, and zColor keywords.

Subrange Display Limitations

In category plots, the category wave (the text wave) may not be subranged. Waves used to specify text using ModifyGraph textMarker mode may not be subranged.

Subranged traces may not be edited using the draw tools (such as: option click on the edit poly icon in the tool palette on a graph).

Waterfall plots may not use subranges.

When multiple subranges of the same wave are used in a graph, they are distinguished only using instance notation and not using the subrange syntax. For example, given `display w[][0],w[][1]`, you must use `ModifyGraph mode(w#0)=1,mode(w#1)=2` and not `ModifyGraph mode(w[][0])=1,mode(w[][1])=2` as you might expect.

The trace instance and subrange used to plot given trace is included in trace info information. See **Identifying a Trace** on page II-321.

Printing Graphs

Before printing a graph you should set the page size and orientation using the Page Setup dialog. Choose Page Setup from the Files menu. Often graphs are wider than they are tall and look better when printed using the horizontal orientation.

When you invoke the Page Setup dialog you must make sure that the graph that you want to print is the top window. Igor stores one page setup in each experiment for all graphs and stores other page setups for other types of windows. You can set the default graph page setup for new experiments using the Capture Graph Preferences dialog.

To print a graph, choose File→Print while the graph is the active window. You can also choose File→Print Preview to display a preview.

Graphs by default print at the same size as the graph window unless they do not fit in which case they are scaled down at the same aspect ratio.

Prior to Igor Pro 7, the Print dialog supported scaling modes such as Fill Page and Same Aspect. These are no longer available in the Print dialog. You can use **PrintSettings** with the graphMode and graphSize keywords prior to printing to achieve the same effects.

Printing Poster-Sized Graphs

Using the **PrintGraphs** operation, you can specify a size for a graph that is too big for a single sheet of paper. When you do this, Igor uses multiple sheets of paper to print the graph. Use this to make very large, poster-sized printouts.

To make the multiple sheets into one big poster, you need to trim the edges of the sheets and tape them together. Igor prints tiny alignment marks on the edges so you can line the pages up. You should trim the unneeded borders so that the alignment marks are flush against the edge of the trimmed sheet. Then align the sheets so that the alignment marks butt up against each other. All of the alignment marks should still be visible. Then tape the sheets together.

Other Printing Methods

You can also print graphs by placing them in page layouts. See Chapter II-18, **Page Layouts** for details.

You can print graphs directly from macros using the **PrintGraphs** (see page V-773) operation.

Save Graph Copy

You can save the active graph as an Igor packed experiment file by choosing File→Save Graph Copy. The main uses for saving as a packed experiment are to save an archival copy of data or to prepare to merge data from multiple experiments (see **Merging Experiments** on page II-19). The resulting experiment file preserves the data folder hierarchy of the waves displayed in the graph starting from the “top” data folder, which is the data folder that encloses all waves displayed in the graph. The top data folder becomes the root data folder of the resulting experiment file. Only the graph, its waves, dashed line settings, and any pictures used in the graph are saved in the packed experiment file, not procedures, variables, strings or any other objects in the experiment.

Save Graph Copy does not work well with graphs containing controls. First, the controls may depend on waves, variables or FIFOs (for chart controls) that Save Graph Copy will not save. Second, controls typically rely on procedures which are not saved by Save Graph Copy.

Save Graph Copy does not know about dependencies. If a graph contains a wave, wave0, that is dependent on another wave, wave1 which is not in the graph, Save Graph Copy will save wave0 but not wave1. When the saved experiment is open, there will be a broken dependency.

The **SaveGraphCopy** operation on page V-821 provides options that are not available using the Save Graph Copy menu command.

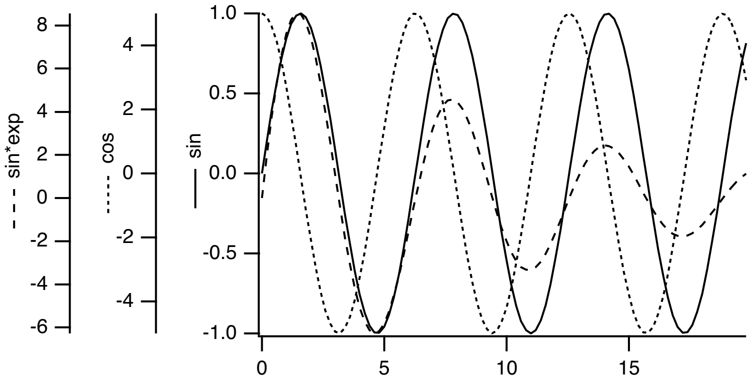
Exporting Graphs

You can export a graph to another application through the clipboard or by creating a file. To export via the clipboard, choose Edit→Export Graphics. To export via a file, choose File→Save Graphics.

The process of exporting graphics from a graph is very similar to exporting graphics from a layout. Because of this, we have put the details in Chapter III-5, **Exporting Graphics (Macintosh)**, and Chapter III-6, **Exporting Graphics (Windows)**. These chapters describe the various export methods you can use and how to choose a method that will give you the best results.

Creating Graphs with Multiple Axes

This section describes how to create a graph that has many axes attached to a given plot edge. For example:



To create this example we first created some data:

```
Make/N=100 wave1,wave2,wave3; SetScale x,0,20,wave1,wave2,wave3
wave1=sin(x); wave2=5*cos(x); wave3=10*sin(x)*exp(-0.1*x)
```

We then followed these steps:

1. Use the New Graph dialog to display the following

   wave1 versus the built-in left axis and the built-in bottom axis

   wave2 versus a free left axis, L1, and the built-in bottom axis

   wave3 versus another free left axis, L2, and the built-in bottom axis

   Use the Axis pop-up menu under the Y Waves list to create the L1 and L2 axes.
2. Use the Modify Graph dialog to set the left margin to 1.5 inches.

   This moves the built-in left axis to the right, creating room for the free axes.
3. Drag the L1 axis to the left of the left axis.
4. Drag the L2 axis to the left of the L1 axis.
5. Use the Modify Trace Appearance dialog to set the trace dash patterns.
6. Use the Axis Label tab of the Modify Axis dialog to set the axis labels.

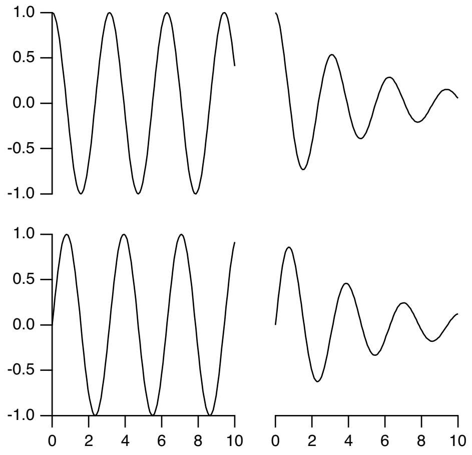
   We used Wave Symbol from the Special pop-up menu to include the line style.
7. Drag the axis labels into place.

Creating Stacked Plots

Igor's ability to use an unlimited number of axes in a graph combined with the ability to shrink the length of an axis makes it easy to create stacked plots. You can even create a matrix of plots and can also create inset plots.

Another way to make a stacked graph is to use subwindows. See **Layout Mode and Guide Tutorial** on page III-86 for an example. It is also possible to do make stacked graphs in page layouts, using either graph subwindows or graph layout objects.

In this section we create stacked plot areas in a single graph window using Igor's ability to limit a plot to a portion of a graph. As an example, we will create the following graph:



First we create some data:

```
Make wave1,wave2,wave3,wave4
SetScale/I x 0,10,wave1,wave2,wave3,wave4
wave1=sin(2*x); wave2=cos(2*x)
wave3=cos(2*x)*exp(-0.2*x)
wave4=sin(2*x)*exp(-0.2*x)
```

We then followed these steps:

1. Use the New Graph dialog to display the following

   wave1 versus the built-in left axis and the built-in bottom axis

   wave2 versus a free left axis, L2, and the built-in bottom axis

   wave3 versus L2 and a free bottom axis, B2

   wave4 versus the built-in left axis and B2

   Use the Axis pop-up menu under the Y Waves list to create the L2 axis.

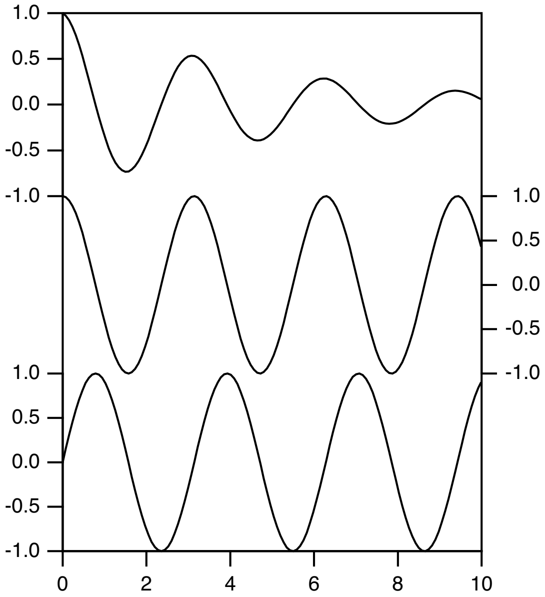
   Use the Axis pop-up menu under the X Waves list to create the B2 axis.

   This creates a graph consisting of a jumble of axes and traces.
2. Choose Graph→Modify Graph and click the Axis tab.

   The next four steps use the "Draw between" settings in the Axis section of the Axis tab.
3. From the Axis pop-up menu, select left and set the left axis to draw between 0% and 45% of normal.
4. From the Axis pop-up menu, select bottom and set the bottom axis to draw between 0% and 45% of normal.
5. From the Axis pop-up menu, select L2 and set the L2 axis to draw between 55% and 100% of normal. Also, in the Free Position section, choose Distance from Margin from the pop-up menu and set the Distance setting to 0.
6. From the Axis pop-up menu, select B2 and set the B2 axis to draw between 55% and 100% of normal. Also, in the Free Position section, choose Distance from Margin from the pop-up menu and set the Distance setting to 0.
7. Click Do It.

Staggered Stacked Plot

Here is a common variant of the stacked plot:



This example was created from three of the waves used in the previous plot. Wave1 was plotted using the left and bottom axes, wave2 used the right and bottom axes and wave3 used L2 and bottom axes. Then the Axis tab of the Modify Axis dialog was used to set the left axis to be drawn from 0% to 33% of normal, the right axis from 33% to 66% and the L2 axis from 66% to 100%. The Axis Standoff checkbox was unchecked for the bottom axis. This was not necessary for the other axes as axis standoff is not used when axes are drawn on a reduced extent.

After returning from the Modify Axis dialog, the graph was resized and the frame around the plot area was drawn using a polygon in plot-relative coordinates.

Waterfall Plots

You can create a graph displaying a sequence of traces in a perspective view. We refer to these types of graphs as waterfall plots, which can be created and modified using the **NewWaterfall** operation or by choosing Windows→New→Packages→Waterfall Plot.

To display a waterfall plot, you must first create or load a matrix wave. (If your data is in 1D waveform or XY pair format, you may find it easier to create a fake waterfall plot - see **Fake Waterfall Plots** on page II-328.) In this 2D matrix, each of the individual matrix columns is displayed as a separate trace in the waterfall plot. Each column from the matrix wave is plotted in, and clipped by, a rectangle defined by the X and Z axes with the plot rectangle displaced along the angled Y axis, which is the right-hand axis, as a function of the Y value.

You can display only one matrix wave per plot.

The traces can be plotted evenly-spaced, in which case their X and Y positions are determined by the X and Y dimension scaling of the matrix. Alternatively they can be plotted unevenly-spaced as determined by separate 1D X and Y waves.

To modify certain properties of a waterfall plot, you must use the ModifyWaterfall operation. For other properties, you will need to use the usual axis and trace dialogs.

Because the traces in the waterfall plot are from a single wave, any changes to the appearance of the waterfall plot using the Modify Trace Appearance dialog or ModifyGraph operation will globally affect all of the waterfall traces. For example, if you change the color in the dialog, then all of the waterfall traces will change to the same color. If you want each of the traces to have a different color, then you will need to use a separate wave to specify (as f(z)) the colors of the traces. See the example in the next section for an illustration of how this can be done.

The X and Z axes of a waterfall are always at the bottom and left while the Y axis runs at a default 45 degrees on the right-hand side. The angle and length of the Y axis can be changed using ModifyWaterfall. Except when hidden lines are active, the traces are drawn in back to front order. Note that hidden lines are active only when the trace mode is lines between points.

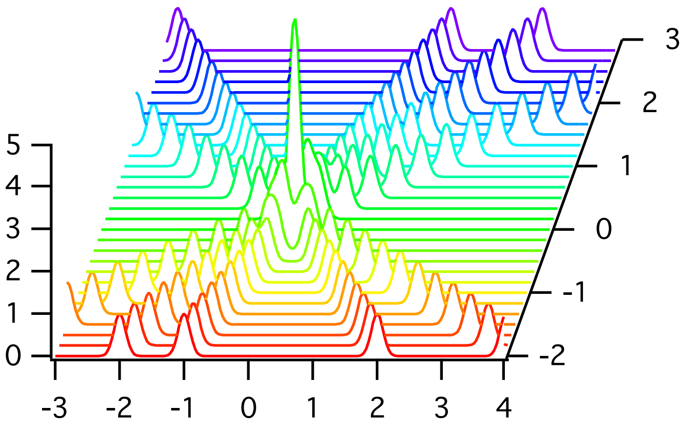
Marquee expansion is based only on the bottom and right (waterfall) axes. The marquee is drawn as a box with the bottom face in the ZY plane at zmin and the top face is drawn in the ZY plane at zmax.

Cursors may be used and the readout panel provides X, Y and Z axis information. The hcsr and xcsr functions are unchanged; the vcsr function returns the Y data value (waterfall) and the zcsr returns the data (Z axis) value.

Evenly-Spaced Waterfall Plot Example

In this example we create a waterfall plot with evenly-spaced X and Y values that come from the X and Y scaling of the matrix being plotted.

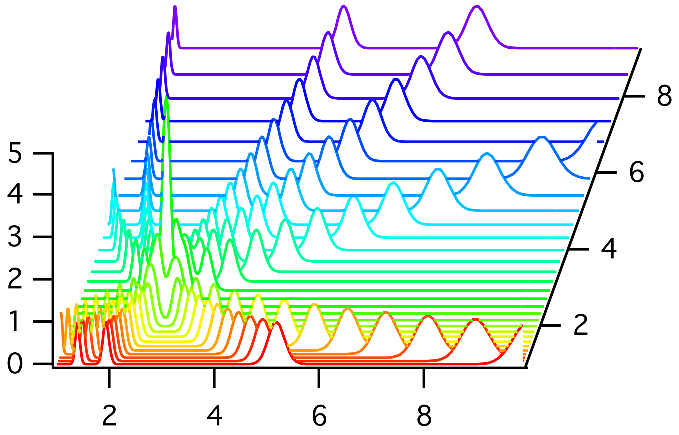
```
Function EvenlySpacedWaterfallPlot()
   // Create matrix for waterfall plot
   Make/O/N=(200,30) mat1
   SetScale x,-3,4,mat1
   SetScale y,-2,3,mat1
   mat1=exp(-((x-y)^2+(x+3+y)^2))
   mat1=exp(-60*(x-1*y)^2)+exp(-60*(x-0.5*y)^2)+exp(-60*(x-2*y)^2)
   mat1+=exp(-60*(x+1*y)^2)+exp(-60*(x+2*y)^2)

   // Create waterfall plot
   NewWaterfall /W=(21,118,434,510) mat1
   ModifyWaterfall angle=70, axlen= 0.6, hidden= 3

   // Apply color as a function of Z
   Duplicate mat1,mat1ColorIndex
   mat1ColorIndex=y
   ModifyGraph zColor(mat1)={mat1ColorIndex,*,*,Rainbow}

End
```



Unevenly-Spaced Waterfall Plot Example

In this example we create a waterfall plot with unevenly-spaced X and Y values that come from separate 1D waves.

```
Function UnvenlySpacedWaterfallPlot()
   // Create matrix for waterfall plot
   Make/O/N=(200,30) mat2
   SetScale x,-3,4,mat2        // Scaling is needed only to generate
   SetScale y,-2,3,mat2        // the fake data
   mat2=exp(-((x-y)^2+(x+3+y)^2))
   mat2=exp(-60*(x-1*y)^2)+exp(-60*(x-0.5*y)^2)+exp(-60*(x-2*y)^2)
   mat2+=exp(-60*(x+1*y)^2)+exp(-60*(x+2*y)^2)
   SetScale x,0,0,mat2         // Scaling no longer needed because we will
   SetScale y,0,0,mat2         // use X and Y waves in waterfall plot

   // Make X and W waves
   Make/O/N=200 xWave = 10^(p/200)
   Make/O/N=30 yWave = 10^(p/30)

   // Create waterfall plot
   NewWaterfall /W=(21,118,434,510) mat2 vs {xWave,yWave}
   ModifyWaterfall angle=70, axlen= 0.6, hidden= 3

   // Apply color as a function of Z
   Duplicate mat2,mat2ColorIndex
   mat2ColorIndex=y
   ModifyGraph zColor(mat2)={mat2ColorIndex,*,*,Rainbow}
End
```

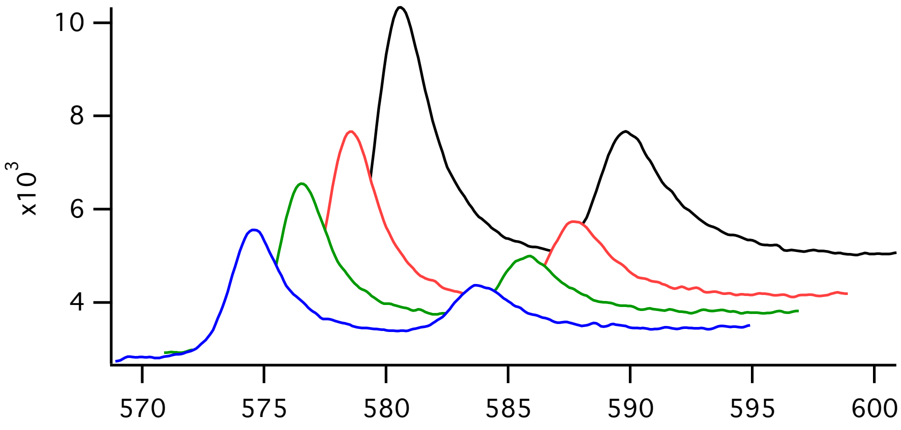


Fake Waterfall Plots

Creating a real waterfall plot requires a 2D wave. If your data is in the form of 1D waveforms or XY pairs, it may be simpler to create a "fake waterfall plot".

In a fake waterfall plot, you plot your waveform or XY data using a regular graph and then create the waterfall effect by offsetting the traces. Since fake waterfall plots use regular Igor traces, you can control their appearance the same as in a regular graph.

The result, with hidden line removal, looks like this:



Because of the offsetting in the X and Y directions, the axis tick mark labels can be misleading.

Igor includes a demo experiment showing how to create a fake waterfall plot. Choose File→Example Experiments→Graphing Techniques→Fake Waterfall Plot.

Wind Barb Plots

You can create a wind barb plot by creating an XY plot and telling Igor to use wind barbs for markers. You turn markers into wind barbs using "ModifyGraph arrowMarker", passing to it a wave that specifies the length, angle and number of barbs for each point.

If you want to color-code the wind barbs, you turn on color as f(z) mode using "ModifyGraph zColor", passing to it a wave that specifies the color for each point.

Here is an example. Execute the commands one section at at time to see how it works.

```
// Make XY data
Make/O xData = {1, 2, 3}, yData = {1, 2, 3}
Display yData vs xData          // Make graph
ModifyGraph mode(yData) = 3     // Marker mode

// Make a barb data wave to control the length, angle
// and number of barbs for each point.
// To control the number of barbs, column 2 must have a column label of WindBarb.
Make/O/N=(3,3) barbData         // Controls barb length, angle and number of barbs
SetDimLabel 1, 2, WindBarb, barbData          // Set column label to WindBarb
Edit /W=(439,47,820,240) barbData

// Put some data in barbData
barbData[0][0]= {20,25,30}      // Column 0: Barb lengths in points
barbData[0][1]= {0.523599,0.785398,1.0472}    // Column 1: Barb angle in radians
barbData[0][2]= {10,20,30}      // Column 2: Wind speed code from 0 to 40

// Set trace to arrow mode to turn barbs on
ModifyGraph arrowMarker(yData) = {barbData, 1, 10, 1, 1}

// Make an RGB color wave
Make/O/N=(3,3) barbColor
Edit /W=(440,272,820,439) barbColor

// Store some colors in the color wave
barbColor[0][0]= {65535,0,0}          // Red
```

```
barbColor[0][1]= {0,65535,0}        // Green
barbColor[0][2]= {0,0,65535}        // Blue

// Turn on color as f(z) mode
ModifyGraph zColor(yData)={barbColor,*,*,directRGB,0}
```

To see a demo of wind barbs choose File→Example Experiments→Feature Demos2→Barbs and Arrows.

See the arrowMarker keyword under **ModifyGraph (traces)** on page V-613 for details on the construction of the barb data wave.

The various color as f(z) modes are explained under **Setting Trace Properties from an Auxiliary (Z) Wave** on page II-298. You can eliminate the barbColor wave by using a color table lookup instead of a color wave.

Box Plots and Violin Plots

When you have multiple measurements that all represent the same conditions, it is useful to know how those measurements are distributed—tightly or loosely clustered, grouped around a central value or more loosely clustered with outliers, and many other possibilities. It is difficult for the eye to comprehend a simple cluster of dots, so plots that summarize the distribution are helpful. Box plots and violin plots are two ways to summarize a distribution of data points.

In Igor, box plots and violin plots are a special kind of graph trace. Each "point" of the trace represents an entire dataset. The data may be stored in individual waves, one wave for each dataset, in which case the input for the trace is a list of waves. Alternately, each dataset may be a column in a single two-dimensional matrix wave.

A normal XY graph trace is named for the wave containing the Y data. Thus, if the Y data is in a wave called "wave0", the trace is also called "wave0". But a box or violin plot may represent data coming from a number of waves. By default the box or violin plot trace is named for the first wave in the list of waves unless you specify a custom trace name. We recommend choosing a custom trace name that describes the nature of the collection of waves.

Box Plot and Violin Plot Terminology

To illustrate this terminology, consider these commands:

```
Make/N=10 wave0, wave1, wave2
Make/T labels = {"Run 1", "Run 2", "Run 3"}
Display; AppendBoxPlot/TN=trace0 wave0, wave1, wave2 vs labels
```

This creates a box plot with one trace named trace0. The trace consists of three datasets named wave0, wave1, and wave2. If we omitted /TN=trace0, the trace would have the default name wave0.

We can achieve the same thing using a 2D three-column 2D wave instead of three 1D waves:

```
Make/N={10,3} mat
Make/T labels = {"Run 1", "Run 2", "Run 3"}
Display; AppendBoxPlot/TN=trace0 mat vs labels
```

This creates a box plot with one trace named trace0. The trace consists of three datasets named mat[0], mat[1], and mat[2]. If we omitted /TN=trace0, the trace would have the default name mat.

Creating Box Plots and Violin Plots

To create a box plot, choose either Windows→New→Box Plot. To create a violin plot, choose Windows→New→Violin Plot. This displays a dialog in which you can choose datasets and set other parameters.

You choose datasets to be used for the plot from the list on the left, and transfer them to the list on the right by clicking the arrow button. Initially, the dialog shows only 1D waves and you need to select one wave for each box or violin to be displayed in the trace. If your data is in a 2D wave, turn on the One Multicolumn Wave checkbox below the list on the left.

If you are using 1D waves, after you have transferred the list of waves to the righthand list, you can reorder the waves by dragging up or down. The order of the waves in the list sets the order of the plots in the trace. If you use a multicolumn wave, the order is set by the columns in the wave.

You may also need to select an X wave. If you select _calculated_, the positions of the plots along the X axis are computed by Igor. For a list of 1D waves, the plots are positioned at 0, 1, 2, ... If your datasets are columns in a multicolumn wave, choosing _calculated_ results in plots positioned according to the Y scaling of the wave, that is, the scaled values of the column dimension indices.

The X Wave menu contains both numeric and text waves. Choosing a numeric wave allows you to position each plot at an arbitrary point on the X axis. Choosing a text wave results in a category X axis (see **Category Plots** on page II-355). The waves shown in the X Wave menu are limited to those waves that have the one point for each selected dataset.

The New Box Plot or New Violin Plot dialog can also make a new text wave for you. Selecting "_new text wave_" from the X Wave menu causes the dialog to generate commands which make a new text wave of the appropriate length, fill it with placeholder text, and display it in a table for editing. The result is a category X axis using the new text wave.

You can give your new trace a custom name by checking the Trace Name checkbox and entering the name in the associated edit box. A custom name is especially useful when you use a list of 1D waves because the default trace name, based on the name of the first data wave, is confusing.

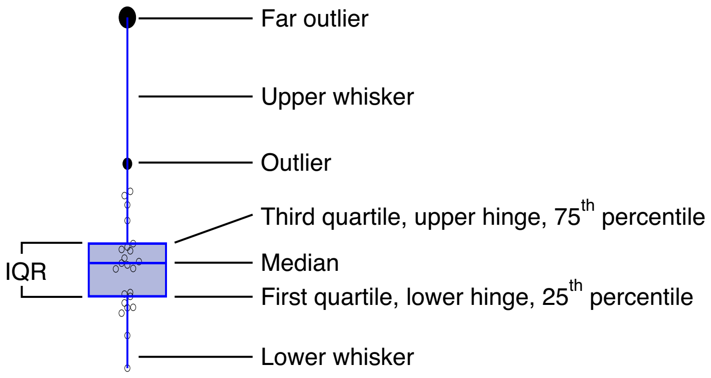
The X Axis, Y Axis and Swap XY Axes controls work the same way as they do in the New Graph dialog (see **Creating Graphs** on page II-277).

A graph can hold more than one box plot or violin plot trace and you can mix the two. To add another box plot or violin plot, choose Graph→Append to Graph→Box Plot or Graph→Append to Graph→Violin Plot.

Box Plots

The box plot, or box and whisker plot, was invented by John W. Tukey to present an easy-to-understand display of the distribution of the data points (see **Box Plot Reference** on page II-337).

A box plot has several parts:


Far outlier
Upper whisker
Outlier
Third quartile, upper hinge, 75th percentile
IQR
Median
First quartile, lower hinge, 25th percentile
Lower whisker

The bottom and top of the box are at the first and third quartiles of the dataset, with a line drawn across the box to represent the median value. Thus, the box gives an indication of the width of the distribution and the median line an indication of the central location of the distribution. The whiskers represent more information about the width of the data distribution such as the length of tails or the symmetry of the distribution. By default, Igor draws the whiskers to the extreme data points, but there are eight different options for whisker length.

The width of the box is not meaningful, but you can control it to make a pleasing display. The width can be expressed as a fraction, in which case it is the fraction of the width of the plot area. If the width is greater than one, it is taken to be an absolute width in points. By default the width is 1/(2*N), where N is the number of datasets or box plots included in the trace.

It is common to show the actual data points when they are far from the center of the distribution. Tukey defines "outliers" and "far outliers". Igor allows you to show all data points, only outliers and far outliers, or only far outliers. You can also select different markers, sizes and colors for each category of data points.

Box Plot Fences

Some options use Tukey's "fences" to define the length of the whiskers and the boundaries defining outliers and far outliers. Tukey also uses the term "hinge" to refer to the the 25th and 75th percentiles.

Inner fences are defined as

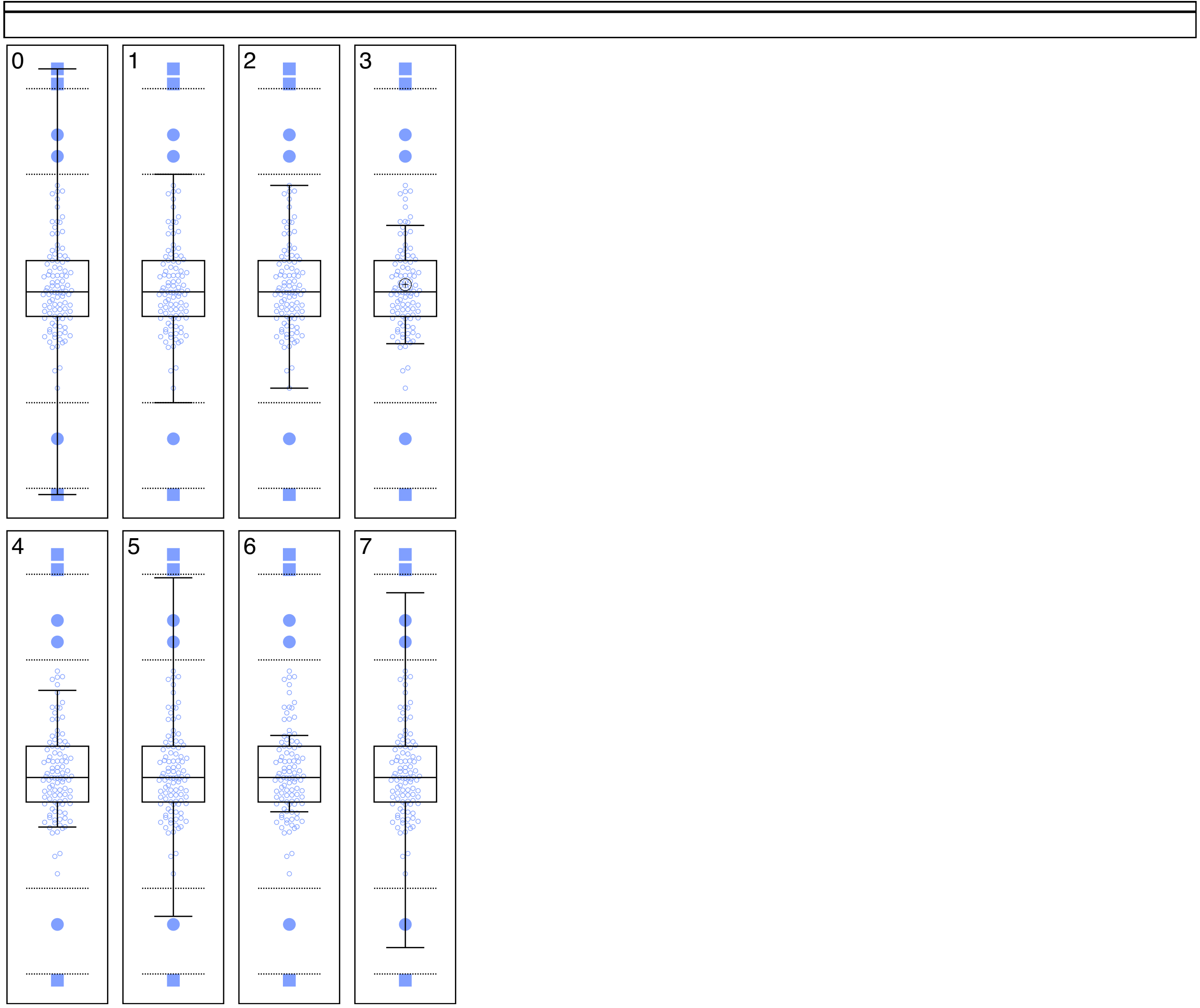
```
inner fence = upper hinge + 1.5*IQR and lower hinge - 1.5*IQR
```

Outer fences are defined as

```
outer fence = upper hinge + 3*IQR and lower hinge - 3*IQR
```

Box Plot Whisker Length

The following figure shows the options for defining the whisker length. The fences are shown for reference to Tukey's definitions, and the outlier method (see **Box Plot Outlier Methods** on page II-334) is option 0:



Option 0: Minimum and maximum data points (the default).

Option 1: Inner fences.

Option 2: "Adjacent points", which Tukey defines as the most extreme data points inside the inner fences. That is, the most extreme data points that are not outliers, if you define outliers the way Tukey does.

Option 3: One standard deviation from the mean of the data. The light circle with plus shows the mean value.

Option 4: The 9th and 91st percentiles.

Option 5: The 2nd and 98th percentiles.

Option 6: Arbitrary percentiles. Here the ends are set to the 20th and 80th percentiles in order to make it look different from the other options.

Option 7: An arbitrary factor times one standard deviation from the mean. In this case, the factor is 3.

If the data are normally distributed, the 2nd, 9th, 25th, 50th, 75th, 91st, and 98th percentiles should be equally spaced.

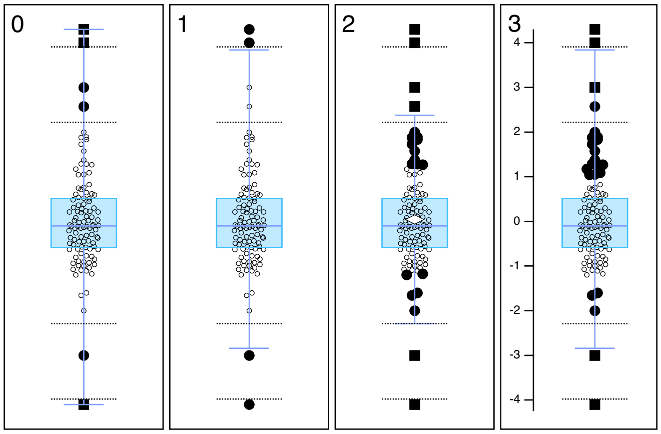
The examples show all the data points but it is common to show only outliers and far outliers. The data points are displayed with "jitter"—data points that would overlap are offset horizontally so that each data point can be seen. You can control the maximum offset by specifying the jitter amount in units of fractions of the box width. If the specified width is insufficient to separate the points, they will overlap as needed. If the width is greater than necessary, only as much offset as necessary is applied.

In usual practice the fences are not shown.

Box Plot Outlier Methods

By default, Igor follow Tukey in defining outliers and far outliers based on the fences, with "outliers" being points outside the inner fences and "far outliers" being points beyond the outer fences. While not usually shown in practice, you can tell Igor to include the fences on the plot using the command ModifyBoxPlot showFences=1. The outliers are shown as filled circles and the far outliers are large filled squares.

Igor offers four options to control which data points are outliers and far outliers:



Option 0: Tukey's definition. Outliers are any data points beyond the inner fences, far outliers are any data points beyond the outer fences.

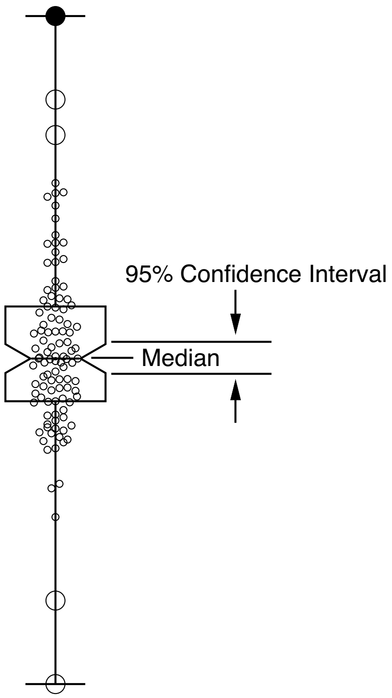
Option 1: Any points beyond the ends of the whiskers are outliers. There are no far outliers. For this option, the whiskers were set to option 6, 2nd and 98th percentiles.

Option 2: Outliers and far outliers are points beyond an arbitrary factor times the standard deviation of the mean. In this case, those factors are 1 and 2. The whisker lengths are set to option 7, arbitrary factor times the standard deviation of the mean. The factor is set to 2. The white diamond shows the mean value.

Option 3: Outliers and far outliers are determined by four arbitrary data values. In this case the values are -2, -1.5, 1 and 2.6.

Notched Box Plots

A notched box plot shows the 95% confidence interval of the median in addition to the various percentiles usually shown:



The notches are at median ± 1.57 * IQR/sqrt(n) where n is the number of data points in the dataset represented by the box plot. Two box plots have a high probability of significantly different median values if the notches don't overlap.

You make a notched box plot by right-clicking on the box plot trace and selecting Modify Box Plot. In the Modify Box Plot dialog, General tab, turn on the Notched checkbox.

Appearance Options for Box Plots

To change the appearance of a box plot, select Modify Box Plot from the Graph menu or right-click on a box plot trace and select Modify Box Plot from the contextual menu.

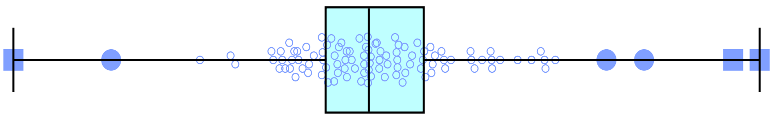
You can change the color, width and dash style for the lines. Those settings can be made individually for each component: the box, the whiskers, the median line and whisker caps.

We regard the fences as an incidental detail. You can show the fences in order to understand the display better, but we have not seen them used for publication purposes. Consequently, the lines for the fences are not modifiable.

You can control the width of the whisker caps—to omit the caps (the default), set the width to zero. A fractional width sets the size of the cap as a fraction of the box width. This is useful to make the cap width consistent when resizing the graph. A width greater than 1 is taken to be an absolute width in points.

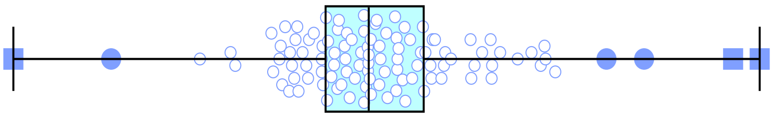
The plots that follow are drawn horizontally to save space. You can achieve that effect by choosing Modify Graph from the Graph menu, and then turning on the Swap XY checkbox. Alternately, you can turn on the Swap X Y Axes checkbox in the New Box Plot or the Append Box Plot dialog to apply the swap to just one box plot trace.

The box can be filled with a color. The box fill color is drawn before the markers so that whatever choices you make for the markers will not be overdrawn by the fill color.

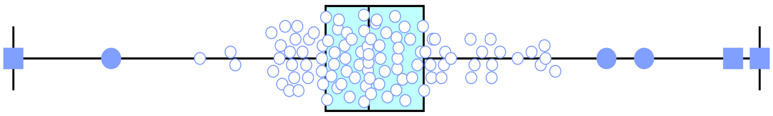


You can select a different marker for non-outlier data, outliers and far outliers. In the plot above, regular data points and outliers are drawn with the hollow circle marker, and outliers have been made larger than the regular data points to emphasize them. The far outlier (there is just one) is drawn with a filled circle marker.

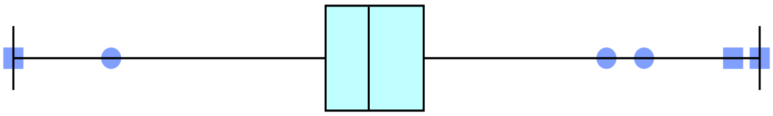
You can achieve some special effects by using a hollow marker and selecting a fill color for the marker For instance, in this plot the hollow circle marker is used for the regular data points, and the fill color is enabled and set to white:



The lines making the box, median and whiskers are on top of the markers so that the markers don't obscure the lines. Sometimes you may wish to have the markers on top. With this dataset, that may obscure the lines too much. To achieve this effect, on the Markers tab turn on the Draw Data Points on Top checkbox. You can separately control drawing the mean marker and median marker:

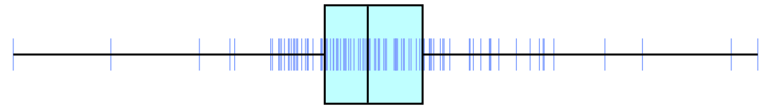


You can choose to show all the raw data, as above, or you can choose to show no raw data points, only outliers and far outliers, or just the far outliers. This plot shows outliers and far outliers:

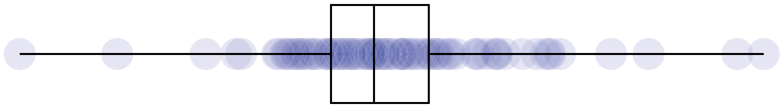


The plots above apply jitter to the data points; that is, if the markers would overlap a lateral offset is applied so that you can see all of them. You can specify a width for the jitter that sets a maximum lateral offset. If the offset is too small to completely separate the markers, they will overlap to some degree. If an offset is not required for a given marker, it is not offset.

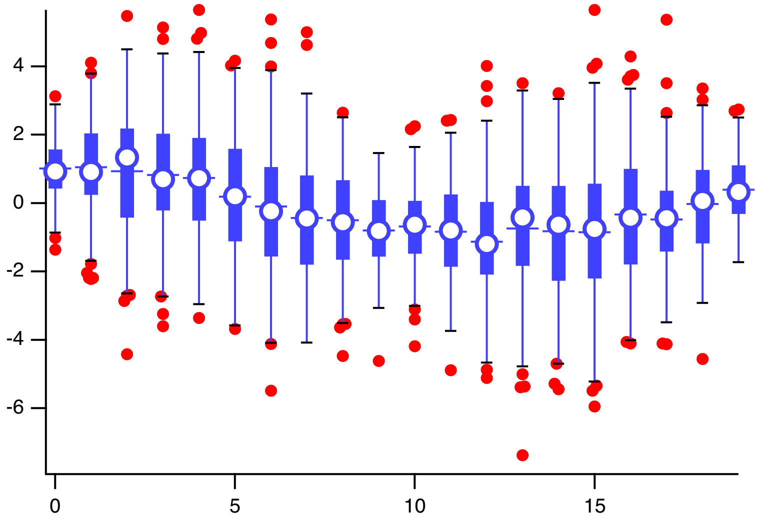
You will sometimes see a box plot in which the data points are shown as a "rug plot", that is, each data point is represented by a thin line. That is another way to allow all the data points to be seen, as long as there aren't too many. To do this, use the line marker and set the jitter to zero. In this plot the marker for normal points, outliers and far outliers is the same vertical line marker (marker 10) with the size set to 8. The caps were set to zero and the extreme data points look like caps.



Large markers with transparency can also give a sense of data density. This plot uses the solid circle marker (number 19) with a blue color with alpha set to 0.1. The fill color for the box is turned off, as the background will show through the transparent markers and affect the color.



Sometimes the number of datasets and data points can be overwhelming. Here is an alternative look that makes a more compact display:



The whisker method is set to Mean ± f*SD, with the factor set to 2, so the whiskers show a span of 2 standard deviations about the mean. The outlier method is set to Whiskers, so only data points beyond the ends of the whiskers are shown. That limits the shear number of data points on the plot.

There are many datasets, so the boxes are quite narrow and filled to make a solid box. Instead of a hard-to-see median line, the median is shown as a white-filled circle marker. To prevent the box outlines from showing on top of the white marker fill, the Draw Median Marker On Top checkbox is turned on in the Markers tab of the Modify Box Plot dialog.

Finally, the mean is shown as a horizontal bar marker. The marker size is set large enough that it shows outside the median circle marker.

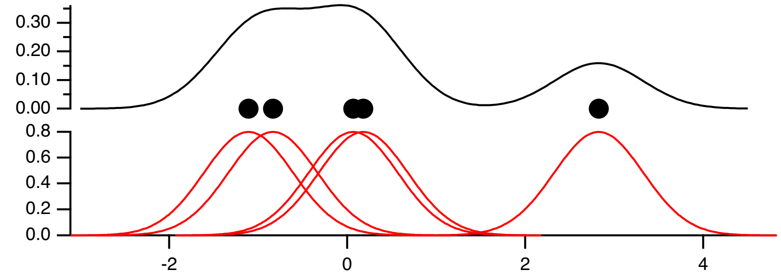
Box Plot Reference

Tukey, John W., Exploratory Data Analysis, Addison-Wesley Publishing Company, 1977.

Violin Plots

A violin plot, also called a bean plot, is a way to summarize the distribution of data. A violin plot shows the distribution of a dataset using a kernel density estimate (KDE). The KDE creates a smooth estimate of the underlying data distribution by summing some kernel function, one function per data point. The summed curve is then normalized to an area of 1.0 so that it is an estimate of the probability distribution function for the dataset.

Suppose you have five points drawn from a Gaussian distribution, such as the points represented by the black dots here:

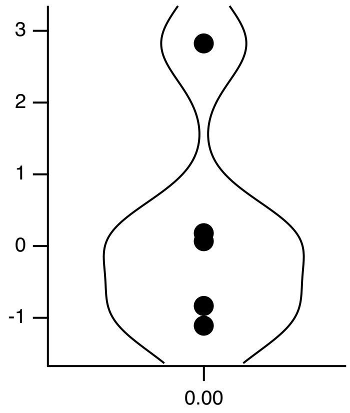


Generate a Gaussian curve for each point (the red curves), then sum the curves and normalize for an area of one (the black curve). In this plot we have arbitrarily selected a standard deviation for the red curves of 0.5; this is referred to as the bandwidth when computing a KDE curve or violin plot.

Now we have a smooth curve that gives a possible representation of the underlying distribution from which the data points were drawn. Quite possibly the kernel bandwidth we used was too small, and unjustified by the small number of points. The choice of kernel function, Gaussian in this example, and the width of the kernel are somewhat arbitrary. The Gaussian kernel is in some sense "smooth" and reflects our bias that most data follow a Gaussian distribution. Others are possible.

You can compute a KDE for your datasets yourself using the **StatsKDE** operation, but to get from there to a violin plot is quite tedious. Igor does a lot of this work for you when you create a violin plot trace by executing the steps described under **Creating Box Plots and Violin Plots** on page II-330.

In a violin plot, the curve is in general plotted vertically and reflected across the midline to give a plot that looks somewhat like a violin or a green bean pod with lumps for each seed. Here is Igor's violin plot of the five points shown above, modified to show the raw data points and to use black coloring:



In this plot, the width of the curves is not meaningful except in a relative way. The centerline indicates zero estimated probability density. Because it is the only dataset included in the trace, and it is not a category plot, it is positioned at zero on the X axis.

For the following illustrations, we made two representative fake datasets:

```
Make/N=(50,3) run1      // 50 points per dataset, three datasets
Make/N=(50,3) run2      // Another with three datasets with 50 points
run1 = gnoise(q+1)      // Gaussian data with standard deviation
run2 = gnoise(1) + q    // Gaussian data with constant width and location
Make/N=3/T categories="Dataset "+num2str(p+1)    // Text wave for category plot
```

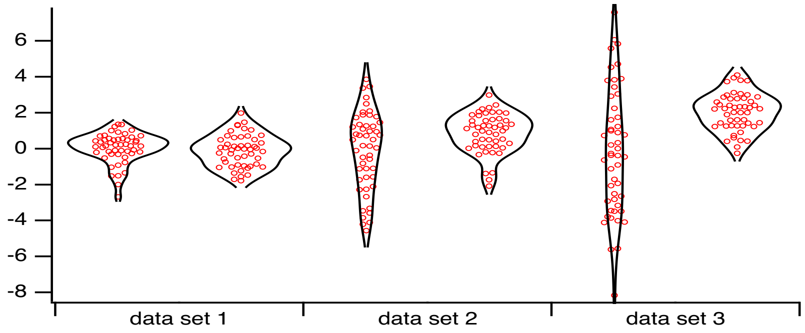
At this point you could select Windows→New Violin Plot, turn on the Multicolumn checkbox, select run1 as the data and categories as the X wave. That would result in these commands:

```
Display;AppendViolinPlot run1 vs categories
```

Now select Graph→Append Violin Plot, turn on the Multicolumn checkbox, select run2 as the data and accept categories as the X wave. That would result in this command:

```
AppendViolinPlot run2
```

The resulting graph looks like this:



Appearance Options for Violin Plots

To change the appearance of a violin plot, select Graph→Modify Violin Plot or right-click and select Modify Violin Plot.

Igor's violin plot has six kernel shapes to choose from. The default is Gaussian and we don't anticipate that other shapes will be used much.

There are three methods for automatically estimating the best bandwidth to use. If you don't like the results of the automatic bandwidths, you can set your own, with a separate bandwidth for each dataset. The automatic estimate assumes a Gaussian kernel.

Each plot of a violin trace occupies a horizontal space equivalent to the box width of a box plot, and that box width can be set in the same way as a box plot—fractions give a width that is a fraction of the plot area, values larger than one are absolute sizes in points. If the X axis is a category axis, Igor sets the box width, overriding whatever you may have chosen.

When Igor makes a violin trace containing more than one dataset, the curves are all normalized to the largest peak amongst all the KDE curves so only one of the violin plots occupies the full box width. You can see this in the plot above, where the third dataset in the Run1 trace (left plots in each category) represents a dataset with broader distribution, which means that to achieve an area of one it must have a smaller amplitude. Compare it with the short, fat distributions in the other categories.

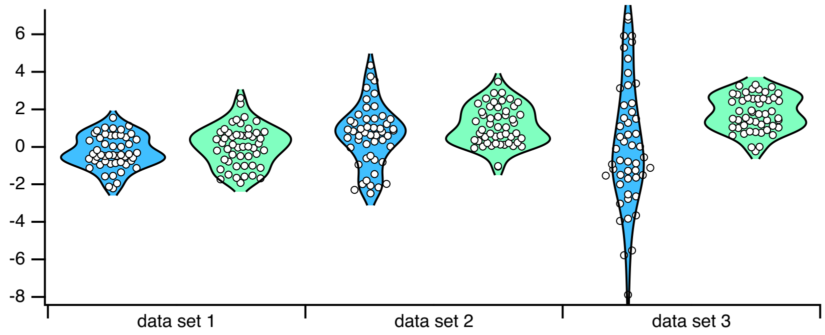
If you have more than one set of violin plots, as in this example where there are two violin traces—one from run1 and one from run2, by default the widest plot in each trace fills the box. The two traces are normalized separately. If you want all the plots in both traces to use the same normalization, you can set the normalization yourself so that the relative amplitudes of all the curves represent the same relative values. This is done using the Distribution Max setting in the General tab of the Modify Violin Plot dialog. The value computed by Igor is shown next to the edit box.

It is common to show the raw data with markers. By default, the raw data points are shown using hollow circles drawn half the size of the default markers for a normal trace. You can choose a marker for the raw data points, plus one for the mean and the median if desired. Violin plots do not distinguish outliers. Jitter can be applied to the data point markers so that they don't overlap.

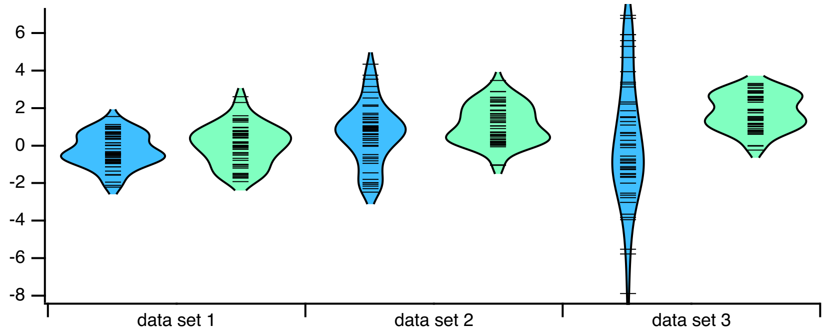
To make a rug plot of the raw data, use the appropriate line marker and set the jitter to 0. Most violin plots are plotted vertically, and the horizontal line marker would be appropriate.

The space between the KDE curves can be filled with color. The fill color is drawn behind the markers so that special effects can be achieved with marker color, marker stroke color and marker fill color.

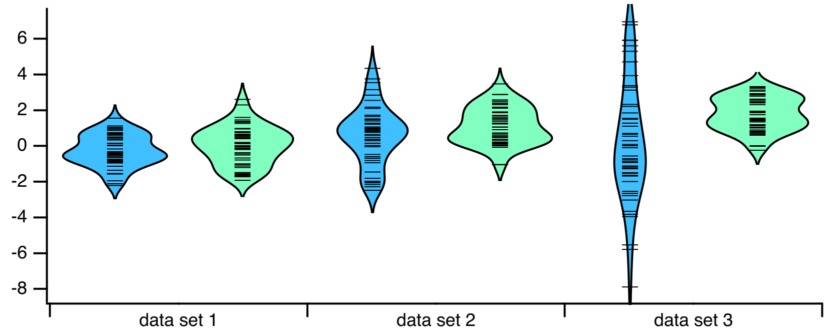
Here is the plot above, re-worked with fill color and hollow round markers with white fill:



And here it is as a rug plot:



You may notice that some of the curves appear to be a bit truncated. That is particularly true of the dataset 3 plot in the right-hand trace. By default, Igor plots the KDE curves one bandwidth beyond the last data point. In this case, it appears that it might be nice to have the curves extend a bit farther. To achieve that, right-click the trace and select Modify Violin Plot. On the General tab, find the Curve Extension edit box and enter a more pleasing amount of extension. The units are kernel bandwidths. Here we have set 2 for the plot above:

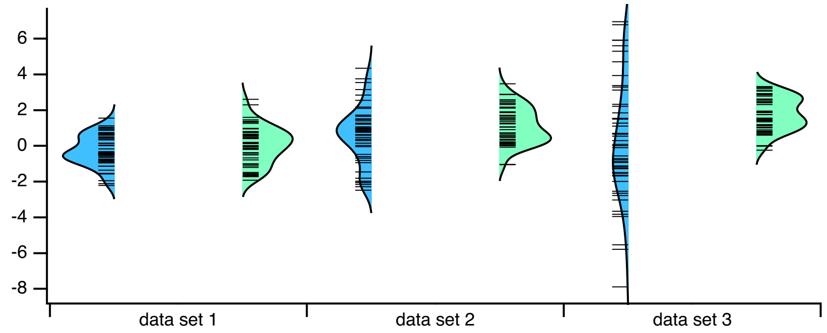


Asymmetric Violin Plots

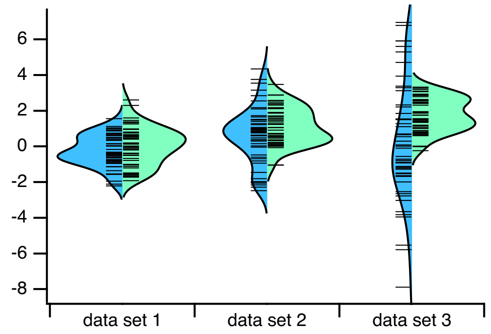
A common variation on a standard violin plot uses just half the violin. This is usually done to compare two conditions or two runs. One condition provides the left half and the other provides the right half.

Let's make the plot above into an asymmetric plot.

The first step is to right-click one of the traces in the plot and choose Modify Violin Plot from the contextual menu. Select the trace that you want for the left half, say the Run1 trace. On the General tab, find the Plot Side menu and select Left Side. Select Run2 and select Right Side in the Plot Side menu. Click Do It and you now have a plot that looks like this:

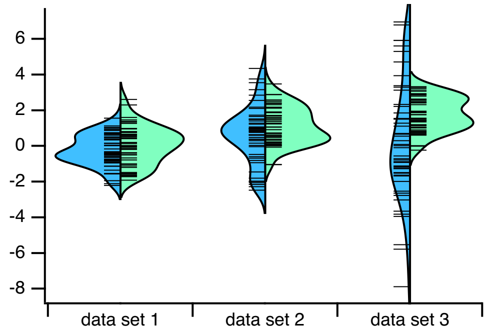


Because the X axis is a category axis, each plot occupies its own slot in the category. To bring them together, double-click the Run1 trace to bring up the Modify Trace Appearance dialog. Make sure the first trace in the trace list is selected. In the Grouping menu, select Keep With Next. We also made the graph window narrower for aesthetic reasons:



If your X axis is a numeric axis, you don't need the last step of setting the Grouping mode. The two violin plot traces draw the plots at the numeric value set by the X wave, or at 0, 1, 2, and so on, if you chose Calculated.

Finally, you may decide that the plot looks better with a line enclosing the entire color-filled area. On the General tab of the Modify Violin Plot dialog, turn on the Close Outline checkbox:



Violin Plot References

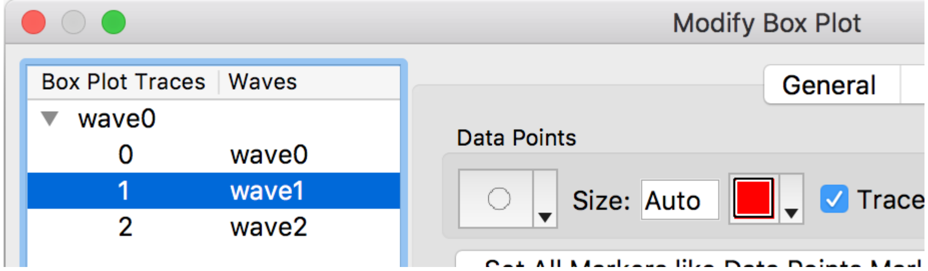
Wand M.P. and Jones M.C. (1995) Monographs on Statistics and Applied Probability, London: Chapman and Hall

Bowman, A.W., and Azzalini, A. (1997), Applied Smoothing Techniques for Data Analysis, London: Oxford University Press.

Making Boxes and Violins Look Different

At times you may want one of the boxes or violins in a trace to have a different appearance to distinguish the corresponding dataset from others in the trace. The Modify Box Plot and Modify Violin Plot dialogs support this by allowing you to select a single dataset from a trace.

This graphic shows the Modify Box Plot dialog with a box plot trace made with a list of three waves. The first wave in the list was wave0, so the name of the trace is "wave0". Clicking the triangle icon for the wave0 trace discloses a list of the three datasets, one for each wave.

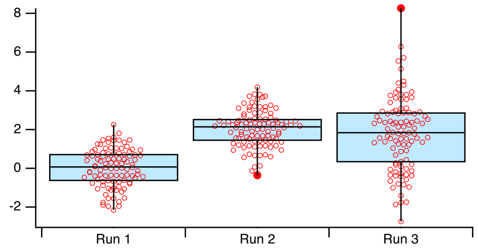


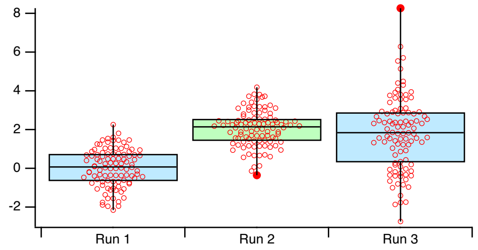
Box Plot Traces	Waves
▼ wave0	
0	wave0
1	wave1
2	wave2

○ Size: Auto 🟥 ☑ Trace

If the top wave0 item were selected, changes made in the dialog would apply to all datasets in the trace. Here we have selected the second dataset corresponding to wave1 so changes made in the dialog apply only to the dataset and consequently to the second box in the box plot.

These graphs show before and after setting the box fill color for the wave1 dataset only to light blue:



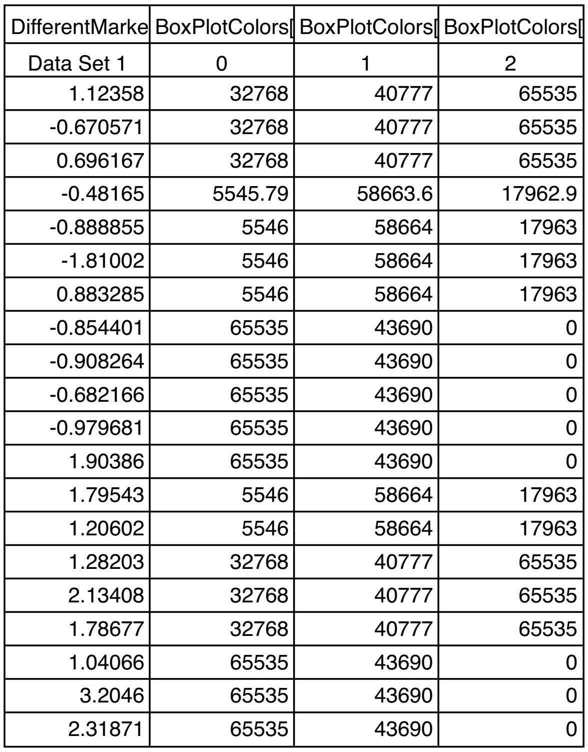


The order in which you make these settings is important. If you change an overall setting, any corresponding values for individual datasets are reset to the overall value. So to make the box plot above with light blue overall, and light green for the second dataset, you need to first set the light blue color for the entire trace followed by setting the light green color for the second dataset only. You do not need to close the dialog between these two changes.

Making Each Data Point Look Different

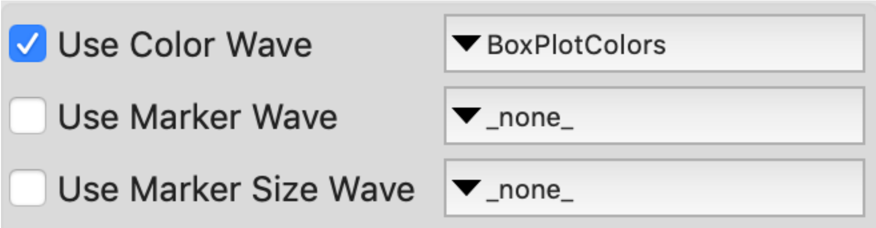
Sometimes the data points in a single dataset come from different sources, or represent different conditions, and you would like to show that in the graph. Using an auxiliary wave you can set the marker color, marker size or marker style for each data point. Controls for these settings appear in the Markers tab of the Modify Box Plot dialog.

For instance, this table shows a fake dataset in the first column and a corresponding three-column wave to specify the marker color for each data point:



DifferentMarke	BoxPlotColors[	BoxPlotColors[	BoxPlotColors[
Data Set 1	0	1	2
1.12358	32768	40777	65535
-0.670571	32768	40777	65535
0.696167	32768	40777	65535
-0.48165	5545.79	58663.6	17962.9
-0.888855	5546	58664	17963
-1.81002	5546	58664	17963
0.883285	5546	58664	17963
-0.854401	65535	43690	0
-0.908264	65535	43690	0
-0.682166	65535	43690	0
-0.979681	65535	43690	0
1.90386	65535	43690	0
1.79543	5546	58664	17963
1.20602	5546	58664	17963
1.28203	32768	40777	65535
2.13408	32768	40777	65535
1.78677	32768	40777	65535
1.04066	65535	43690	0
3.2046	65535	43690	0
2.31871	65535	43690	0

We made a box plot with that dataset and selected the wave BoxPlotColors to set the marker colors for the data points:

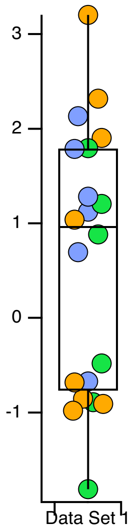


☑ Use Color Wave ▼ BoxPlotColors

☐ Use Marker Wave ▼ _none_

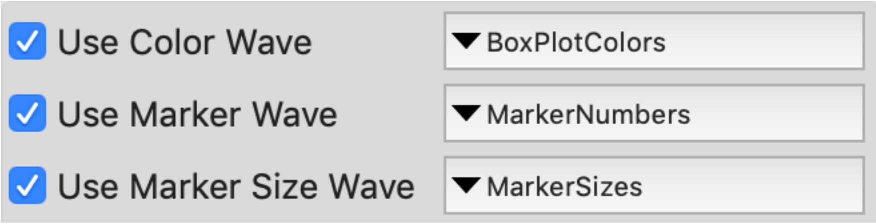
☐ Use Marker Size Wave ▼ _none_

The result is this box plot:



The color wave for this figure was created using the Color Wave Editor package, accessed by choosing Data→Packages→Color Wave Editor.

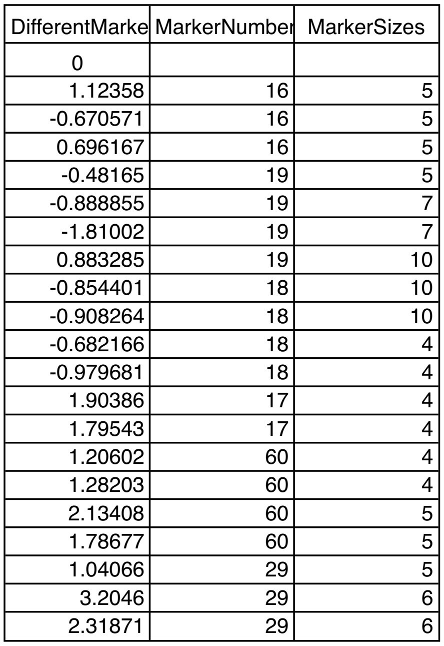
Similarly, you can set each data point marker to a different size or to a different marker style using a 1D wave with a size or marker number for each data point. For instance:



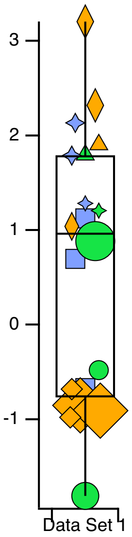
☑ Use Color Wave ▼ BoxPlotColors

☑ Use Marker Wave ▼ MarkerNumbers

☑ Use Marker Size Wave ▼ MarkerSizes



DifferentMarke	MarkerNumber	MarkerSizes
0		
1.12358	16	5
-0.670571	16	5
0.696167	16	5
-0.48165	19	5
-0.888855	19	7
-1.81002	19	7
0.883285	19	10
-0.854401	18	10
-0.908264	18	10
-0.682166	18	4
-0.979681	18	4
1.90386	17	4
1.79543	17	4
1.20602	60	4
1.28203	60	4
2.13408	60	5
1.78677	60	5
1.04066	29	5
3.2046	29	6
2.31871	29	6



Marker sizes can range from 0 to 200. See **Markers** on page II-291 for a table of marker styles and the associated marker numbers.

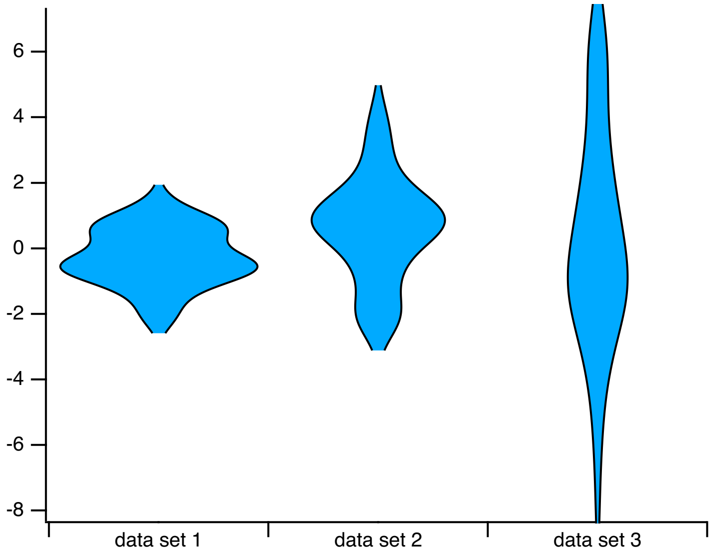
The marker colors, marker sizes and marker styles set by these waves override whatever other settings have been made. So if you have selected a particular marker for box plot outliers, for instance, but an outlier data point has a marker set via a marker wave, the marker is taken from the wave.

The marker color, marker size and marker style waves are not required to have the same number of rows as the dataset wave. If there are extra rows the extras are ignored. If there are too few rows, the extra data points take their color, size, color and style from the normal color, size and style settings.

Combining Box Plots and Violin Plots

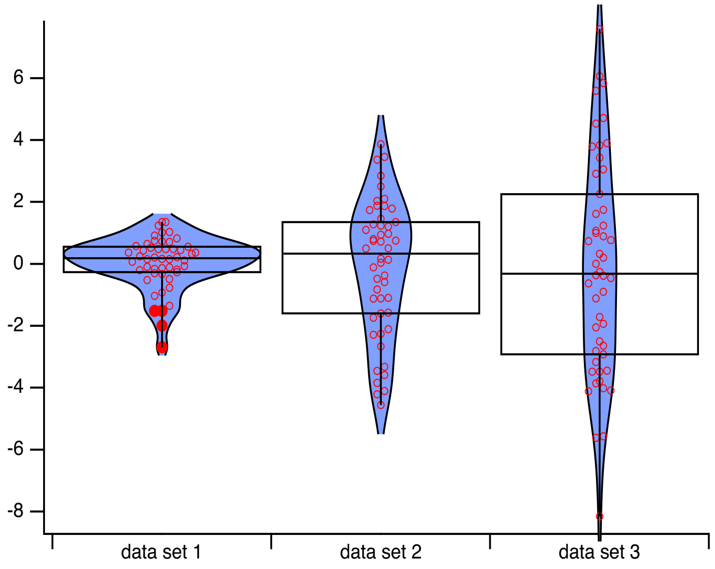
One problem with a box plot is that it hides the true distribution of your data. If your data is bimodal, it still shows you only a box with a median line and whiskers. But a violin plot lacks the statistical information contained in a box plot. A common solution is to combine the two—put a box plot in the middle of a violin plot.

To do this in Igor, first create a violin plot using Windows→New→New Violin Plot. We did this with the Run1 dataset from the violin plot example:

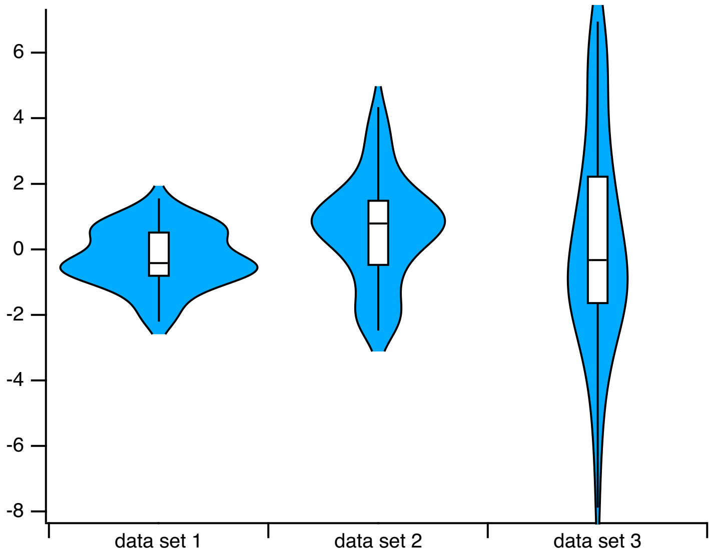


We have chosen a fill color and we are not showing the data points.

Now add a box plot by choosing Graph→Append→Box Plot. Because we are using a category X axis, the plots are side-by-side. Double-click the violin plot trace and select Keep With Next as the grouping mode:



For best appearance, you should make the box plot narrower, and most likely fill the box with contrasting color. Here we set the box width to 0.1 and chose to fill the box with white. We don't show the data points, but if you wish to, it is best to show the data points using the box plot trace so that fills don't obscure the data points.



Creating Split Axes

You can create split axes using the same techniques described above for creating stacked plots. Simply plot your data twice using different axes and then adjust the axes so they are stacked. You can then adjust the range of the axes independently. You can use draw tools to add cut marks.

WaveMetrics supplies a procedure package to automate all aspects of creating split axes except setting the range and adjusting the ticking details of the axes. To use the package, choose Graph→Packages→Split Axes. For an example, choose File→Example Experiments→Graphing Techniques→Split Axes.

Before using the package, you should create the graph in near final form using just the main axes. For best results, especially if you will be using cut marks, you should work with the graph at actual size before adding the split axes. It is recommended that you create a recreation macro just before executing the split axis macros. This is so you can easily revert in case you need to change the pre-split setup.

After creating the split, you can execute the AddSplitAxisMarks procedure to add cut marks between the two axes. You can then use the drawing tools to duplicate the cut marks if you want marks on the traces as well as the axes. Of course, you can also draw your own cut marks. You should use the default Plot Relative coordinate system for cut marks so they will remain in the correct location should you resize the graph.

Some programs draw straight lines between data points on either side of the split. While such lines provide the benefit of connecting traces for the viewer, they also are misleading and inaccurate. This package accurately plots both sections and does not attempt to provide a bridge between them. If you feel it is necessary, you can use drawing tools to add a connecting bridge.

Live Graphs and Oscilloscope Displays

This section will be of interest mainly if you use Igor for data acquisition.

Normally, when the data in a wave is modified, all graphs containing traces derived from that wave are redrawn from scratch. Although fast compared to other programs, this process may noticeably limit the graph update rate.

Live Mode

If you specify one or more traces in a graph as being “live” then Igor takes some shortcuts, resulting in faster than normal updates. Fast update is obtained when certain conditions are observed.

Note: When graphs are redrawn in live mode, *autoscaling is not done.*

To specify a trace in a graph as being live you must use the **live** keyword with the ModifyGraph command. There is no dialog support for this setting.

```
ModifyGraph live(traceName)= mode
```

Mode can be 0 or 1. Zero turns live mode off for the given trace.

WaveMetrics provides a demo experiment that generates and displays synthetic data. You should use this experiment to get a feel for the performance you might expect on your particular computer as a function of the window size, number of points in the live wave, and the live modes. To run the demo, choose File→Example Experiments→Feature Demos→Live Mode.

Although live mode 1 is not restricted to unity thickness solid lines or dots modes, you will get the best performance if you do use these settings.

Quick Append

Another feature that may be of use is the quick append mode. It is intended for applications in which a data acquisition task creates new waves periodically. It permits you to add the new waves to a graph very quickly. To invoke a quick append, use the /Q flag in an AppendToGraph command. There is no dialog support for this setting.

A side effect of quick append is that it marks the wave as not being modified since the last update of graphs and therefore prevents other graphs containing the same wave, if any, from being updated. For a demo, choose File→Example Experiments→Feature Demos→Quick Append.

Graph Preferences

Graph preferences allow you to control what happens when you create a new graph or add new traces to an existing graph. To set preferences, create a graph and set it up to your taste. We call this your *prototype* graph. Then choose Capture Graph Prefs from the Graph menu.

Preferences are normally in effect only for *manual* operations, not for automatic operations from Igor procedures. This is discussed in more detail in Chapter III-18, **Preferences**.

When you initially install Igor, all preferences are set to the factory defaults. The dialog indicates which preferences you have not changed by displaying “default” next to them.

The Window Position and Size preference affects the creation of new graphs only. New graphs will have the same size and position as the prototype graph.

The Page Setup preference is somewhat unusual because all graphs share the same page setup settings, as shown in the Page Setup dialog. The captured page setup is already in use by all other graphs. The utility of this category is that new *experiments* will use the captured page setup for graphs.

The “XY Plots:Wave Styles” preference category refers to the various wave-specific settings in the graph, such as the line type, markers and line size, set with the Modify Trace Appearance dialog. This category also includes settings for waveform plots. Each captured wave style is associated with the index of the wave it was captured from. The index of the first wave displayed or appended to a graph is 0, the second appended wave has an index of 1, and so on.These indices are the same as are used in style macros. See **Graph Style Macros** on page II-350.

If preferences are on when a new graph with waves is created or when a wave is appended to an existing graph, the wave style assigned to each is based on its index. The wave with an index of 2 is given the captured style associated with index 2 (the third wave appended to the captured graph).

You might wonder what style is applied to the fifth and sixth waves if only four waves appeared in the graph from which wave style preferences were captured. You have two choices; either the factory default style is used, or the styles repeat with the first wave style and then the second style. You make this choice in the Miscellaneous Settings dialog, with the Repeat Wave Style Prefs in Graphs checkbox. With that box selected, the fifth and sixth waves would get the first and second captured styles; if deselected, they would both get the factory default style, as would any other waves subsequently appended to the graph.

The XY Plots:Axes and Axis Labels preferences category captures all of the axis-related settings for axes in the graph. Only axes used by XY or waveform plots have their settings captured. Axes used solely for a category, image, or contour plot are ignored. The settings for each axis are associated with the name of the axis it was captured from.

Even if preferences are on when a new graph with waves is created or when a wave is newly appended to an existing graph, the wave is still displayed using the usual default left and bottom axes unless you explicitly specify another named axis. The preferred axes are not automatically applied, but they are listed by name in the New Graph, and the various Append to Graph dialogs, in the two Axis pop-up menus so that you may select them.

For example, suppose you capture preferences for an XY plot using axes named MyRightAxis and MyTopAxis. These names will appear in the X Axis and Y Axis pop-up menus in the New Graph and Append Traces to Graph dialogs.

- If you choose them in the New Graph dialog and click Do It, a graph will be created containing *newly-created* axes named MyRightAxis and MyTopAxis and having the axis settings you captured.
- If you have a graph which already uses axes named MyRightAxis and MyTopAxis and choose these axes in the Append Traces to Graph dialog, the traces will be appended to those axes, as usual, but no captured axis settings will be applied to these *already-existing* axes.

Captured axes may also be specified by name on the command line or in a procedure, provided preferences are on:

```
Function AppendWithCapturedAxis(wav)
   Wave wav
   Variable oldPrefState
   Preferences 1; oldPrefState = V_Flag   // Turn preferences on
   Append/L=MyCapturedAxis wav      // Note: MyCapturedAxis must
                                    //       be vertical to use /L
   Preferences oldPrefState         // Restore old prefs setting
End
```

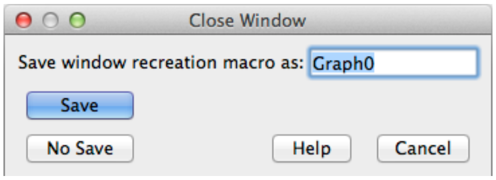
The Category Plots:Axes and Axis Labels and Category Plots:Wave Styles are analogous to the corresponding settings for XY plots. Since they are separate preference categories, you have can independent preferences for category plots and for XY plots. Similarly, preferences for image and contour plots are independent of preferences for other types. See Chapter II-14, **Category Plots**, Chapter II-15, **Contour Plots**, and Chapter II-16, **Image Plots**.

How to use Graph Preferences

Here is our recommended strategy for using graph preferences:

1. Create a new graph containing a single trace. Use the axes you will normally use.
2. Make the graph appear as you prefer using the Modify Graph dialog, Modify Trace Appearance dialog, the Modify Axis dialog, etc. Move the graph to where you prefer it be positioned.
3. Choose the Graph→Capture Graph Prefs menu to display the Capture Graph Preferences dialog. Check the checkboxes corresponding to the categories you want to capture, and click Capture Prefs.
4. Choose Misc→Miscellaneous Settings to display the Miscellaneous Settings dialog. In the Graphs section, check the Repeat Wave Style Prefs checkbox, and click Save Settings.

Saving and Recreating Graphs



If you click in the close button of a graph window, Igor asks you if you want to save a **window recreation macro**.

Igor presents the graph's name as the proposed name for the macro. You can replace the proposed name with any valid macro name.

If you want to make a macro so you can recreate the graph later, click Save. Igor then creates a macro which, when invoked, will recreate the graph with its size, position and presentation intact. Igor saves the recreation macro is placed in the procedure window where you can inspect, modify or delete it as you like.

The macro name appears in the Graph Macros submenu of the Windows menu. You can invoke the macro by choosing it from that submenu or by executing the macro from the command line. The window name of the recreated graph will be the same as the name of the macro that recreated it.

If you are sure that you never want to recreate the graph, you can press Option (*Macintosh*) or Alt (*Windows*) while you click the close button of the graph window. This closes the graph without presenting the dialog and without saving a recreation macro.

For a general discussion of saving, recreating, closing windows, see Chapter II-4, **Windows**.

Graph Style Macros

The purpose of a graph style macro is to allow you to create a number of graphs with the same stylistic properties. Igor can automatically generate a style macro from a prototype graph. You can manually tweak the macro if necessary. Later, you can apply the style macro to a new graph.

For example, you might frequently want to make a graph with a certain sequence of markers and colors and other such properties. You could use preferences to accomplish this. The style macro offers another way and has the advantage that you can have any number of style macros while there is only one set of preferences.

You create a graph style macro by making a prototype graph, setting each of the elements to your taste and then, using the Window Control dialog, instructing Igor to generate a style macro for the window.

You can apply the style macro when you create a graph using the New Graph dialog. You can also apply it to an existing graph by choosing the macro from the Graph Macros submenu of the Windows menu.

Example of Creating a Style Macro

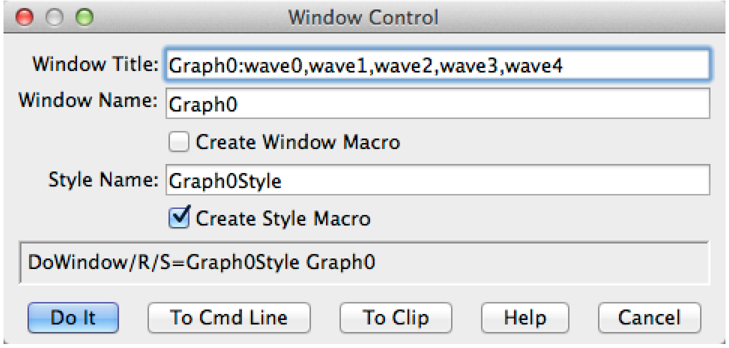
As an example, we will create a style macro that defines the color and line type of five traces.

Since we want our style macro to define a style for five traces, we start by making a graph with five waves:

```
Make wave0=p, wave1=10+p, wave2=20+p, wave3=30+p, wave4=40+p
Display wave0, wave1, wave2, wave3, wave4
```

Now, using the Modify Trace Appearance dialog, we set the color and line style for each of the waves to our liking.

Now we're ready to generate the style macro. With the graph the active window, we choose Windows→Control→Window Control to display the Window Control dialog in which we check the Create Style Macro checkbox.



Window Title: Graph0:wave0,wave1,wave2,wave3,wave4
Window Name: Graph0
□ Create Window Macro
Style Name: Graph0Style
☑ Create Style Macro

```
DoWindow/R/S=Graph0Style Graph0
```

When we click Do It, Igor generates a graph style macro and saves it in the procedure window.

The graph style macro for this example is:

```
Proc Graph0Style() : GraphStyle
   PauseUpdate; Silent 1      // modifying window...
   ModifyGraph/Z lStyle[1]=1,lStyle[2]=2,lStyle[3]=3,lStyle[4]=4
   ModifyGraph/Z rgb[0]=(0,0,0)
   ModifyGraph/Z rgb[1]=(3,52428,1)
   ModifyGraph/Z rgb[2]=(1,12815,52428)
   ModifyGraph/Z rgb[3]=(52428,1,41942)
   ModifyGraph/Z rgb[4]=(65535,21845,0)
EndMacro
```

Notice that the graph style macro does not refer to wave0, wave1, wave2, wave3 or wave4. Instead, it refers to traces by index. For example,

```
ModifyGraph rgb[0]=(0,0,0)
```

sets the color for the trace whose index is 0 to black. A trace's index is determined by the order in which the traces were displayed or appended to the graph. In the Modify Trace Appearance dialog, the trace whose index is zero appears at the top of the list.

The /Z flag used in the graph style macro tells Igor not to worry if the command tries to modify a trace that is not actually in the graph. For example, if you make a graph with three traces (indices from 0 to 2) and apply this style macro to it, there will be no trace whose index is 3 at the time you run the macro. The command:

```
ModifyGraph rgb[3]=(52428,1,41942)
```

would generate an error in this case. Adding the /Z flag continues macro execution and ignores the error.

Style Macros and Preferences

When Igor generates a graph style macro, it generates commands to modify the target graph according to the prototype graph. It assumes that the objects in the target will be in their factory default states at the time the style macro is applied to the target. Therefore, it generates commands only for the objects in the prototype which have been modified. If Igor did not make this assumption, it would have to generate commands for every possible setting for every object in the prototype and style macros would be very large.

Because of this, you should create the new graph with preferences off and then apply the style macro.

Applying the Style Macro

To use this macro, you would perform the following steps.

1. Turn preferences off by choosing Preferences Off from the Misc menu.
2. Create a new graph, using the New Graph dialog and optionally the Append Traces to Graph dialog.
3. Choose Graph0Style from the Graph Macros submenu in the Windows menu.
4. Turn preferences back on by choosing Preferences On from the Misc menu.

If you use only the New Graph dialog, you can use the shorter method:

1. Open the New Graph dialog, select the waves to be displayed in the graph, and choose Graph0Style from the Style pop-up menu in the dialog. Click Do It.

Igor automatically generates the Preferences Off and Preferences On commands to apply the style to the new graph without being affected by preferences.

Limitations of Style Macros

Igor automatically generates style macro commands to set all of the properties of a graph that you set via the ModifyGraph, Label and SetAxis operations. These are the properties that you set using the Modify Trace Appearance, Modify Graph, and Modify Axis dialogs.

It does not generate commands to recreate annotations or draw elements. Igor's assumption is that these things will be unique from one graph to the next. If you want to include commands to create annotations and draw elements in a graph, you must add the appropriate commands to the macro.

Where to Store Style Macros

If you want a style macro to be accessible from a single experiment only, you should leave them in the main procedure window of that experiment. If you want a style macro to be accessible from any experiment then you should store it in an auxiliary procedure file. See Chapter III-13, **Procedure Windows** for details.

Graph Pop-Up Menus

There are a number of contextual pop-up menus that you can use to quickly set colors and other graph properties. To display a contextual menu on Macintosh, press the Control key and click. On Windows, click using the right mouse button.

Different contextual menus are available for clicks on traces, the interior of a graph (but not on a trace) and axes. If you press the Shift key before a contextual click on a trace or axis, the menu will apply to all traces or axes in the graph.

Sometimes it is difficult to contextual click in the plot area of a graph and not hit a trace. In this case, try clicking outside the plot area, but not on an axis.

Graph Expansion

Normally, graphs are shown actual size but sometimes, when working with very small or very large graphs, it is easier to work with an expanded or contracted screen representation. You can set an expansion or contraction factor for a graph using the Expansion submenu in the Graph menu or using the contextual menu for the graph body, away from traces or axes.

The expansion setting affects only the screen representation. It does not affect printing or exporting.

Graph Shortcuts

Action	Shortcut (*Macintosh*)	Shortcut (*Windows*)
To autoscale a graph	Press Command-A.	Press Ctrl+A.
To modify the appearance or front-to-back drawing order of a trace	Press Control and click the trace to get a contextual menu. Press Shift-Control to modify all traces. Double-click the trace to summon a dialog.	Right-click the trace to get a contextual menu. Press Shift while right-clicking to modify all traces. Double-click the trace to summon a dialog.
To modify the appearance of an axis, axis labels, grid lines, tick marks	Press Control and click the axis. Press Shift-Control to modify all axes. Double-click an axis to summon a dialog.	Right-click the axis. Press Shift and right-click to modify all axes. Double-click an axis to summon a dialog.
To modify the appearance of a contour plot	Control-click and choose Modify Contour from the contextual menu or press Shift and double-click the contour plot. See also Chapter II-15, **Contour Plots**.	Right-click and choose Modify Contour from the contextual menu or press Shift and double-click the contour plot. See also Chapter II-15, **Contour Plots**.
To modify the appearance of an image plot	Control-click and choose Modify Image from the contextual menu. See also Chapter II-16, **Image Plots**.	Right-click and choose Modify Image from the contextual menu. See also Chapter II-16, **Image Plots**.
To set background colors	Press Control and click in the graph body, away from any traces.	Press Ctrl and click in the graph body, away from any traces.
To set the range of an axis	Double-click tick mark labels to summon a dialog.	Double-click tick mark labels to summon a dialog.
To pan the graph	Press Option and drag the body of the graph. Press Shift also to constrain the direction of panning.	Press Alt and drag the body of the graph. Press Shift also to constrain the direction of panning.
To offset a trace	Click and hold the trace for about a second, then drag. You can avoid inadvertently triggering this feature by pressing Caps Lock before clicking a trace.	Click and hold the trace for about a second, then drag. You can avoid inadvertently triggering this feature by pressing Caps Lock before clicking a trace.
To adjust the position of an axis or axis label	Drag the axis or label.	Drag the axis or label.
To change a graph margin	Press Option and click the axis and drag. Drag beyond edge of graph to return to default position. Press Option and double-click an axis or just double-click outside of the graph body and axes to summon a dialog.	Press Alt and click the axis and drag. Drag beyond edge of graph to return to default position. Press Alt and double-click an axis or just double-click outside of the graph body and axes to summon a dialog.
To change an axis label	Double-click the label to summon a dialog.	Double-click the label to summon a dialog.

Action	Shortcut (*Macintosh*)	Shortcut (*Windows*)
To change an annotation	Double-click the annotation to summon a dialog.	Double-click the annotation to summon a dialog.
To connect a tag to a different point	Press Option and drag the tag to the new point. Don't drag the arrow or line; drag the tag body. Drag it off the graph window to remove it.	Press Alt and drag the tag to the new point. Don't drag the arrow or line; drag the tag body. Drag it off the graph window to remove it.
To attach a cursor to a particular trace	Drag the cursor from the info panel to the trace or click the cursor name in the info panel and choose a trace name from the pop-up menu.	Drag the cursor from the info panel to the trace or click the cursor name in the info panel and choose a trace name from the pop-up menu.
To get information about a particular trace	Press Command-Option-I to turn on Trace Info tags.	Press Ctrl+Alt+I to turn on Trace Info tags.
To show or hide a graph's tool palette	Press Command-T.	Press Ctrl+T.
To move or resize a user-defined control without using the tool palette	Press Command-Option and click the control. With Command-Option still pressed, drag or resize it. See also Chapter III-14, **Controls and Control Panels**.	Press Ctrl+Alt and click the control. With Ctrl+Alt still pressed, drag or resize it. See also Chapter III-14, **Controls and Control Panels**.
To modify a user-defined control	Press Command-Option and double-click the control. This displays a dialog that you use to modify all aspects of the control. If the control is already selected, you don't need to press Command-Option.	Press Ctrl+Alt and double-click the control. This displays a dialog that you use to modify all aspects of the control. If the control is already selected, you don't need to press Ctrl+Alt.
To edit a user-defined control's action procedure	With the graph in modify mode (tools showing, second icon from the top selected) press Option and double-click the control. This displays the procedure window containing the action procedure or beeps if there is no action procedure.	With the graph in modify mode (tools showing, second icon from the top selected) press Alt and double-click the control. This displays the procedure window containing the action procedure or beeps if there is no action procedure.
To nudge a user-defined control's position	Select the control and press Arrow keys. Press Shift to nudge faster.	Select the control and press Arrow keys. Press Shift to nudge faster.

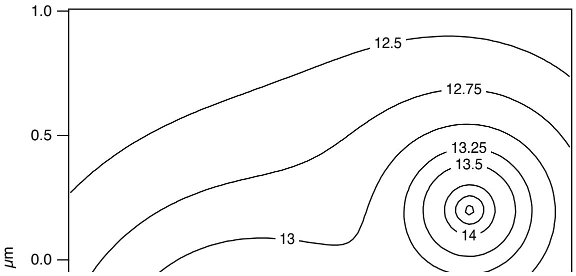
Chapter

II-14

Category Plots

Overview

Category plots are two-dimensional plots with a continuous numeric variable on one axis and a non-numeric (text) category on the other. Most often they are presented as bar charts with one or more bars occupying a category slot either side-by-side or stacked or some combination of the two. You can also combine them with error bars:



Category plots are created in ordinary graphs when you use a text wave or dimension labels as the X data. For more on graphs, see Chapter II-13, **Graphs**.

Creating a Category Plot

To create a category plot, first create your numeric wave(s) and your text category wave.

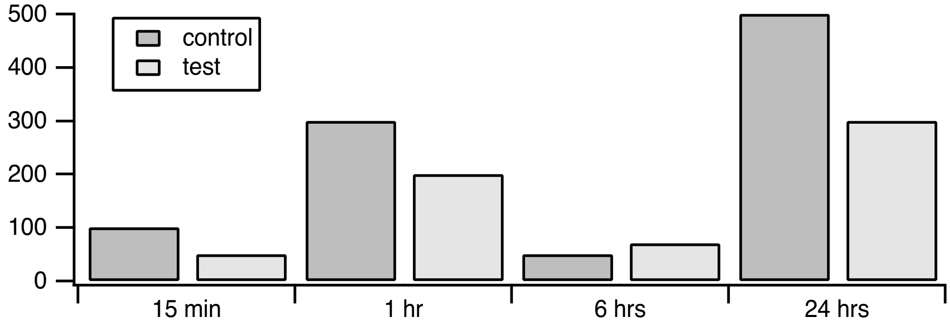
The numeric waves used to create a category plot should have “point scaling” (X scaling with Offset = 0 and Delta = 1). See **Category Plot Pitfalls** on page II-362 for an explanation.

Then invoke the Category Plot dialog by choosing Windows→New→Category Plot. You can append to an existing graph by choosing Graph→Append to Graph→Category Plot.

Select the numeric waves from the Y Waves list, and the category (text) wave from the X Wave list.

You can use also the Display command directly to create a category plot:

```
Make/O control={100,300,50,500},test={50,200,70,300}
Make/O/T sample={"15 min","1 hr","6 hrs","24 hrs"}
Display control,test vs sample   //vs text wave creates category plot
ModifyGraph hbFill(control)=5,hbFill(test )=7
SetAxis/A/E=1 left
Legend
```



Category Plot Commands

The **Display** operation that creates a category plot is the same Display operation that creates an XY plo t. When you use a text wave for the X wave, Igor creates a category plot. When you use a numeric wave for tthe X wave, Igor creates an XY plot. The same applies to the **AppendToGraph** operation.

You can control the gap between categories and the gap between bars within a single category using the ModifyGraph operation with the barGap and catGap keywords. You can create a stacked category plot using the ModifyGraph toMode keyword. See Bar and Category Gaps.

Combining Category Plots and XY Plots

You can have ordinary XY plots and category plots in the same graph window. However, once an axis has been used as either numeric or category, it is not usable as the other type.

For example, if you tried to append an ordinary XY plot to the graph shown above, you would find that the bottom (category) axis was not available in the Axis pop-up menu. If you try to append data to an existing category plot using a different text wave as the category wave, the new category wave is ignored.

The solution to these problems is to create a new axis using the Append Traces to Graph dialog or the Append Category Plot dialog.

Category Plot Using Dimension Labels

An alternative to using a text wave to create a category plot is to use the dimension labels from the Y wave. This feature was added in Igor Pro 8.00.

The easiest way to create the dimension labels is to edit the dimension labels in a table (see **Showing Dimension Labels** on page II-235). This example shows how to programmatically make a category plot using dimension labels:

```
Function DemoCategoryPlotUsingDimensionLabels()
   Make/O control={100,300,50,500}, test={50,200,70,300}
   SetDimLabel 0, 0, '15 min', control
   SetDimLabel 0, 1, '1 hour', control
   SetDimLabel 0, 2, '6 hrs', control
   SetDimLabel 0, 3, '24 hrs', control
   Display /W=(35,45,430,253) control, test vs '_labels_'
   ModifyGraph hbFill(control)=5,hbFill(test)=7
   SetAxis/A/E=1 left
   Legend
End
```

The _labels_ keyword must be enclosed in single quotes because it has the form of a liberal name and it is used in a place where a wave name is expected.

Using the Y wave's dimension labels is convenient for category plots having just one Y wave because it keeps the category labels and the numeric Y data in one place. If you are making the graph manually, you can enter the labels in a table, instead of executing a separate command for each label.

When you have more than one Y wave, the first trace added to a category axis controls the category labels. If you remove the first trace or change the order of traces, the labels may change or become blank. You can prevent this by setting the dimension labels for all the Y waves.

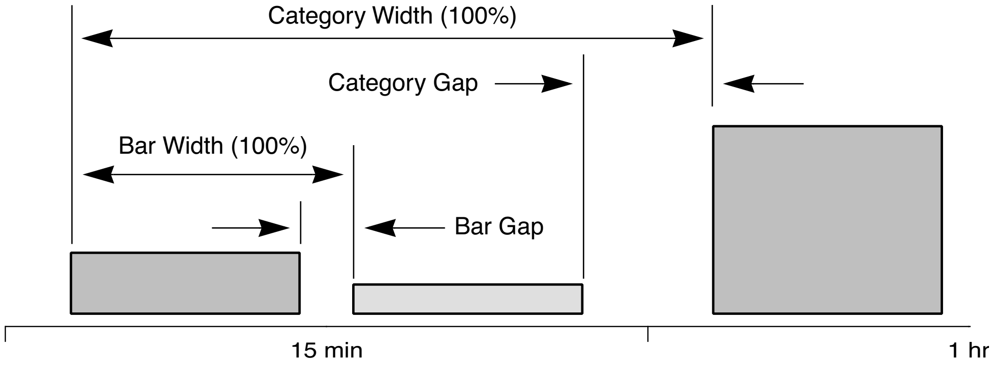
Modifying a Category Plot

Because category plots are created in ordinary graph windows, you can change the appearance of the category plot using the same methods you use for XY plots. For example, you can modify the bar colors and line widths using the Modify Trace Appearance dialog. For information on traces, XY plots and graphs, see **Modifying Traces** on page II-290.

The settings unique to category plots are described in the following sections.

Bar and Category Gaps

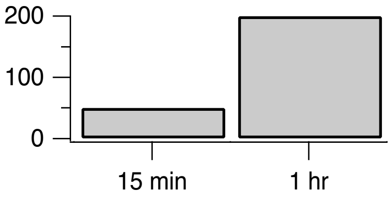
You can control the gap size between bars and between categories.



Generally, the category gap should be larger than the bar gap so that it is clear which bars are in the same category. However, a category gap of 100% leaves no space for bars.

The gap sizes are set in the Modify Axis dialog which you can display by choosing Graph→Modify Axis or by double-clicking the category axis.

Tick Mark Positioning



You can cause the tick marks to appear in the center of each category slot rather than at the edges. Double-click the category axis to display the Modify Axis dialog and check the “Tick in center” checkbox in the “Auto/Man Ticks” pane. This looks best when there is only one bar per category.

Fancy Tick Mark Labels

Tick mark labels on the category axis are drawn using the contents of your category text wave. In addition to simple text, you can insert special escape codes in your category text wave to create multiline labels and to include font changes and other special effects. The escape codes are exactly the same as those used for axis labels and for annotation text boxes — see **Annotation Text Content** on page III-35.

There is no point-and-click way to insert the codes in this version of Igor Pro. You will have to either remember the codes or use the Add Annotation dialog to create a string you can paste into a cell in a table.

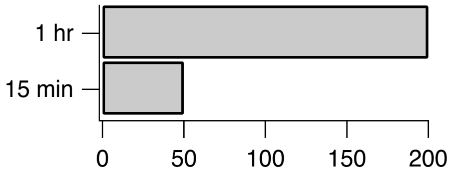
To enter multi-line text in a table cell, click the text editor widget at the right end of the table entry line.

You can also make a multi-line label from the command line, like this:

```
Make/T/N=5 CatWave         // Mostly you won't need this line
CatWave[0]="Line 1\rLine2" // "\r" Makes first label with two lines
```

Multiline labels are displayed center-aligned on a horizontal category axis and right-aligned on a left axis but left-aligned on a right axis. You can override the default alignment using the alignment escape codes as used in the Add Annotation dialog. See the **Annotation Text Escape Codes** operation on page III-35 for a description of the formatting codes.

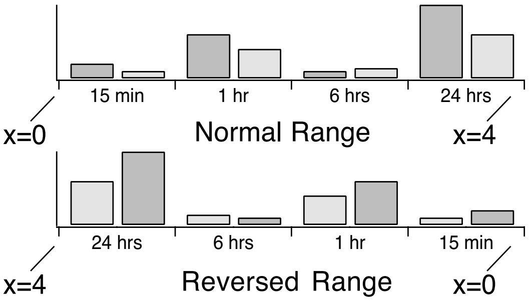
Horizontal Bars



To create a category plot in which the category axis runs vertically and the bars run horizontally, create a normal vertical bar plot and then select the Swap XY checkbox in the Modify Graph dialog.

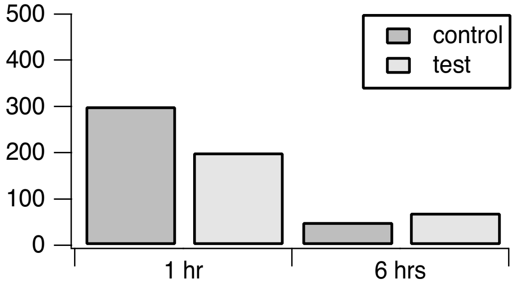
Reversed Category Axis

Although the ordering of the categories is determined by the ordering of the value (numeric) and category (text) waves, you can reverse a category axis just like you can reverse a numeric axis. Double-click one of the category axis tick labels or choose the Set Axis Range from the Graph menu to access the Axis Range pane in the Modify Axes dialog. When the axis is in autoscale mode, select the Reverse Axis checkbox to reverse the axis range.



Category Axis Range

You can also enter numeric values in the min and max value items of the Axis Range pane of the Modify Axes dialog. The X scaling of the numeric waves determine the range of the category axis. We used "point" X scaling for the numeric waves, so the numeric range of the category axis for the 15 min, 1 hr, 6 hrs, 24hrs example is 0 to 4. To display only the second and third categories, set the min to 1 and the max to 3.



Bar Drawing Order

When you plot multiple numeric waves against a single category axis, you have multiple bars within each category group. In the examples so far, there are two bars per category group.

The order in which the bars are initially drawn is the same as the order of the numeric waves in the Display or AppendToGraph command:

```
Display control,test vs elapsed     //control on left, test on right
```

You can change the drawing order in an existing graph using the **Reorder Traces dialog** and the Trace pop-up menu (see **Graph Pop-Up Menus** on page II-352).

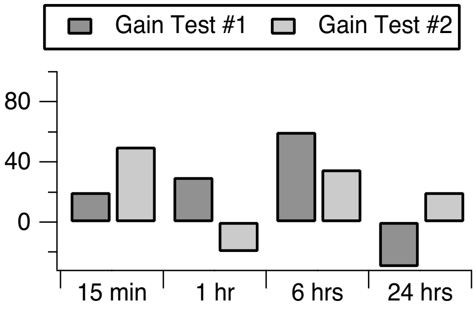
The ordering of the traces is particularly important for stacked bar charts.

Stacked Bar Charts

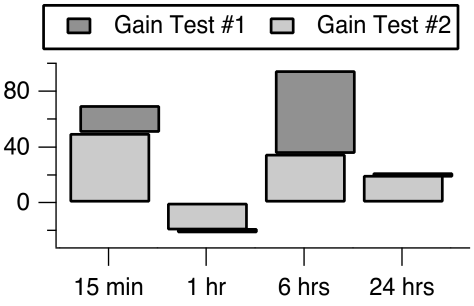
You can stack one bar on top of the next by choosing one of several grouping modes in the Modify Trace Appearance dialog which you can invoke by double-clicking a bar. The Grouping pop-up menu in the dialog shows the available modes. The choices are:

Mode	Mode Name	Purpose
-1	Keep with next	For special effects
0	None	Side-by-side bars (default)
1	Draw to next	Overlapping bars
2	Add to next	Arithmetically combined bars
3	Stack on next	Stacked bars

For most uses, you will use the **None** and “Stack on next” modes which produce the familiar bar and stacked bar chart:



None Mode

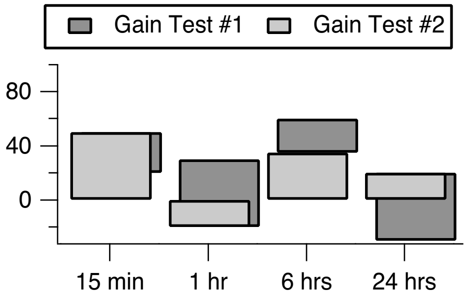


Stack on Next Mode

In all of the Stacked Bar Chart examples that follow, the stacking mode is applied to the Gain Test #1 bar and Gain Test #2 is the “next” bar.

We have offset Gain Test #1 horizontally by 0.1 so that you can see what is being drawn behind Gain Test #2.

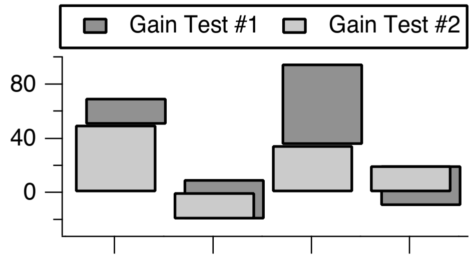
Choosing “**Draw to next**” causes the current bar to be in the same horizontal position as the next bar and to be drawn from the y value of this trace to the Y value of the next trace.



Draw to Next Mode

If the next bar is taller than the current bar then the current bar will not be visible because it will be hidden by the next bar. The result is as if the current bar is drawn behind the next bar, as is done when bars are displayed using a common numeric X axis.

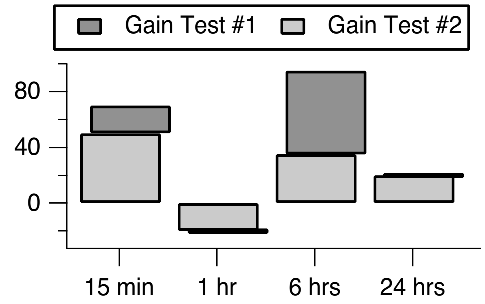
“**Add to next**” is similar to “Draw to next” except the Y values of the current bar are added to the Y values of the next bar(s) before plotting.



Add to Next Mode

If the current Y value is negative and the next is positive then the final position will be shorter than the next bar, as it is here for the 24 hrs bar.

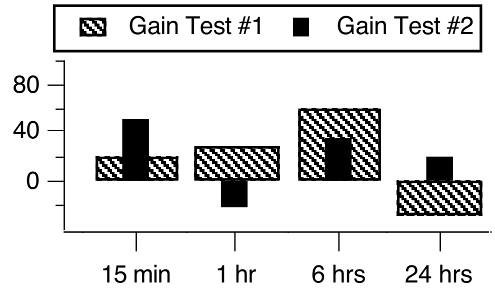
"**Stack on next**" is similar to "Add to next" except bars are allowed only to grow, not shrink.



Stack on Next Mode

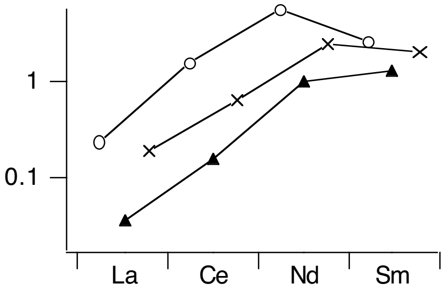
Negative values act like zero when added to a positive next trace (see the 24 hrs bar) and positive values act like zero when added to a negative next trace (see the 1 hr bar). Zero height bars are drawn as a horizontal line. Normally the values are all positive, and the bars stack additively, like the 15 min and 6 hrs bars.

"**Keep with next**" creates special effects in category plots. Use it when you want the current trace to be plotted in the same horizontal slot as the next but you don't want to affect the length of the current bar. For example, if the current trace is a bar and the next is a marker then the marker will be plotted on top of the bar. Here we set the Gain Test #2 wave to Lines from Zero mode, using a line width of 10 points.

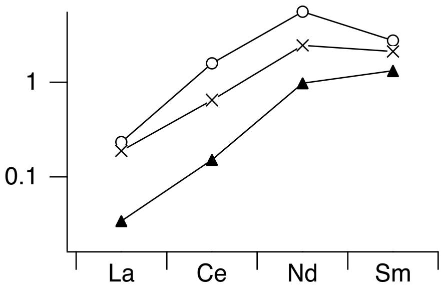


Keep with Next Mode

"Keep with next" mode is also useful for category plots that don't use bars; you can keep markers from different traces vertically aligned within the same category:



None mode



Keep with Next Mode

More details about these modes can be found in **Grouping, Stacking and Adding Modes** on page II-296.

Numeric Categories

You can create category plots with numeric categories by creating a text wave from your numeric category data. Create a text wave containing the numeric values by using the num2str function. For example, if we have years in a numeric wave:

```
Make years={1993,1995,1996,1997}
```

we can create an equivalent text wave:

```
Make/T/N=4 textYears= num2str(years)
```

Then create your category plot using textYears:

```
Display ydata vs textYears       // vs 1993, 1995, 1996, 1997 (as text)
```

Combining Numeric and Category Traces

Normally when you create a category plot, you can append only another category trace (a numeric wave plotted versus a text wave) to that plot. In rare cases, you may want to add a numeric trace to a category plot. You can do this using the /NCAT flag. Here is an example:

```
Make/O/T catx = {"cat0", "cat1", "cat2"}
Make/O caty = {1, 3, 2}
Display caty vs catx
SetAxis/A/E=1 left

// Plot simulated original data for a category
Make/N=10/O cat1over = gnoise(1) + 1.5
SetScale/P x, 1.5, 1e-5, cat1over      // Delta x can not be zero
AppendToGraph/NCAT cat1over
ModifyGraph mode(cat1over)=3, marker(cat1over)=19, rgb(cat1over)=(0,0,65535)
```

The /NCAT flag, used with AppendToGraph, tells Igor to allow adding a numeric trace to a category plot. This flag was added in Igor Pro 6.20.

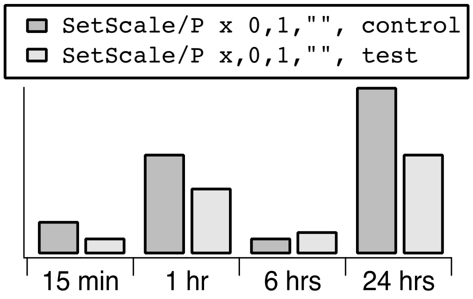
In Igor Pro 6.37 or later, the Display operation also supports the /NCAT flag. This allows you to create a numeric plot and then append a category trace.

Category Plot Pitfalls

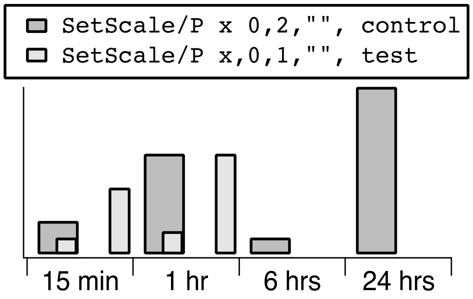
You may encounter situations in which the category plot doesn't look like you expect it to.

X Scaling Moves Bars

Category plots position the bars using the X scaling of the value (numeric) waves. The X scaling of the category (text) wave is completely ignored. It is usually best if you leave the X scaling of the category plot waves at the default “point scaling.” In any event, the X scaling of the value (numeric) waves should be identical. Differing X scaling causes the bars to become separated in category plots containing multiple bars per category. In the graph on the right the numeric waves have different X scaling



Same X scaling



Different X scaling

Changing the Drawing Order Breaks Stacked Bars

Stacked bar charts are heavily dependent on the concept of the “current bar” and the “next bar”. The modes describe how the current bar is connected to the next bar, such as “Stack on next”.

If you change the drawing order of the traces, using the Reorder Traces dialog or the Trace pop-up menu, one or more bars will have a new “next bar” (a different trace than before). Usually this means that a bar will be stacking on a different bar. This is usually a problem only when the stacking modes of the traces differ, or when smaller bars become hidden by larger bars.

After you change the drawing order, you may have to change the stacking modes. Bars hidden by larger bars may have to be moved forward in the drawing order with the Reorder Traces dialog or the Trace pop-up menu.

Bars Disappear with “Draw to next” Mode

In “Draw to next” mode, if the next bar is taller than the current bar then the current bar will not be visible because it will be hidden by the next bar.

You can change the drawing order with the Reorder Traces dialog or the Trace pop-up menu to move the shorter bars forward in the drawing order, so they will be drawn in front of the larger bars.

Category Plot Preferences

You can change the default appearance of category plots by capturing preferences from a prototype graph containing category plots. Create a graph containing a category plot (or plots) having the settings you use most often. Then choose Graph→Capture Graph Prefs. Select the Category Plots categories, and click Capture Prefs.

Preferences are normally in effect only for *manual* operations, not for automatic operations from Igor procedures. This is discussed in more detail in Chapter III-18, **Preferences**.

Category Plot Axes and Axis Labels

When creating category plots with preferences turned on, Igor uses the Category Plot axis settings for the text wave axis and XY plot axis settings for the numeric wave axis.

Only axes used by category plot *text waves* have their settings captured. Axes used solely for an XY plot, image plot, or contour plot are ignored. Usually this means that only the bottom axis settings are captured.

The category plot axis preferences are applied only when axes having the same name as the captured axis are created by a Display or AppendToGraph operation when creating a category plot. If the axes existed before the operation is executed, they are not affected by the category plot axis preferences.

The names of captured category plot axes are listed in the X Axis pop-up menu of the New Category Plot and Append Category Plot dialogs.

For example, suppose you capture preferences for a category plot that was created with the command:

```
AppendToGraph/R=myRightAxis/T=myTopAxis ywave vs textwave
```

Since only the X axis is a category axis, “myTopAxis” appears in the X Axis pop-up menu in the category plot dialogs. The Y Axis pop-up menu is unaffected.

- If you choose “myTopAxis” in the X Axis pop-up menu of the New Category Plot dialog and click Do It, a graph is created containing a *newly-created* X axis named “myTopAxis” and having the axis settings you captured.
- If you have a graph which already uses an axis named “myTopAxis” *as a category axis* and you choose it from the X Axis pop-up menu in the Append Category Plot dialog, the category plot uses the axis, but no captured axis settings are applied to it.

You can capture category plot axis settings for the standard left or bottom axis, and Igor will save the settings separately from left and bottom axis preferences captured for XY, image, and contour plots.

Category Plot Wave Styles

The captured category plot wave styles are automatically applied to a category plot when it is first created provided preferences are turned on — see **How to Use Preferences** on page III-516. “Wave styles” refers to the various trace-specific settings for category plot numeric waves in the graph. The settings include trace mode, line style, stacking mode, fill pattern, colors, etc., as set by the Modify Trace Appearance dialog.

If you capture the category plot preferences from a graph with more than one category plot, the first category plot appended to a graph gets the wave style settings from the category first appended to the prototype graph. The second category plot appended to a graph gets the settings from the second category plot appended to the prototype graph, etc. This is similar to the way XY plot wave styles work.

How to Use Category Plot Preferences

Here is our recommended strategy for using category preferences:

1. Create a new graph containing a single category plot.
2. Use the Modify Trace Appearance dialog and the Modify Axes dialogs to make the category plot appear as you prefer.
3. Choose Capture Graph Prefs from the Graph menu. Select the Category Plot checkboxes, and click Capture Prefs.

Chapter

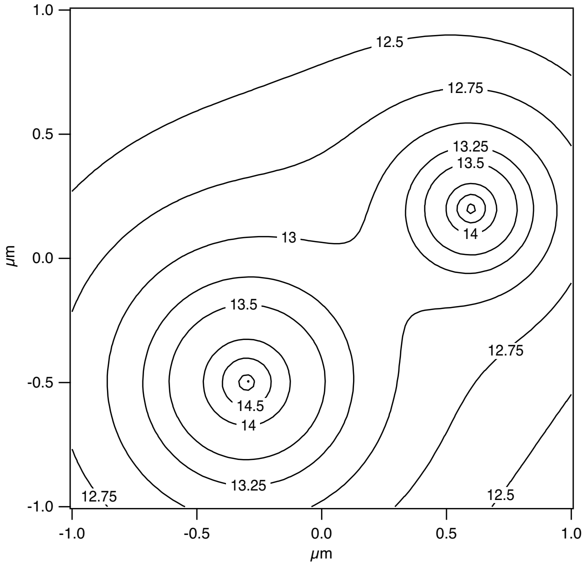
II-15

Contour Plots

Overview

A contour plot is a two-dimensional XY plot of a three-dimensional XYZ surface showing lines where the surface intersects planes of constant elevation (Z).

One common example is a contour map of geographical terrain showing lines of constant altitude, but contour plots are also used to show lines of constant density or brightness, such as in X-ray or CT images, or to show lines of constant gravity or electric potential.



Contour Data

The contour plot is appropriate for data sets of the form:

$$z = f(x,y)$$

meaning there is only one Z value for each XY pair. This rules out 3D shapes such as spheres, for example.

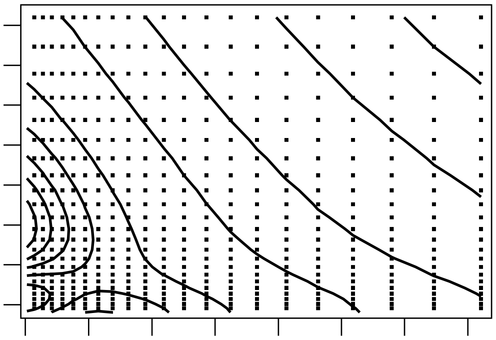
You can create contour plots from two kinds of data:

- Gridded data stored in a matrix
- XYZ triplets

Gridded Data

Gridded data is stored in a 2D wave, or “matrix wave”. By itself, the matrix wave defines a regular XY grid. The X and Y coordinates for the grid lines are set by the matrix wave’s row X scaling and column Y scaling.

You can also provide optional 1D waves that specify coordinates for the X or Y grid lines, producing a non-linear rectangular grid like the one shown here. The dots mark XY coordinates specified by the 1D waves:



Contouring gridded data is computationally much easier than XYZ triplets and consequently much faster.

XYZ Data

XYZ triplets can be stored in a matrix wave of three columns, or in three separate 1D waves each supplying X, Y, or Z values. You must use this format if your Z data does not fall on a rectangular grid. For example, you can use the XYZ format for data on a circular grid or for Z values at random X and Y locations.

XYZ contouring involves the generation of a Delaunay triangulation of your data, which takes more time than is needed for gridded contouring. You can view the triangulation by selecting the Show Triangulation checkbox in the Modify Contour Appearance dialog.

For best results, you should avoid having multiple XYZ triplets with the same X and Y values. If the contour data is stored in separate waves, the waves should be the same length.

If your data is in XYZ form and you want to convert it to gridded data, you can use the **ContourZ** function on an XYZ contour plot of your data to produce a matrix wave. The AppendImageToContour procedure in the WaveMetrics Procedures folder produces such a matrix wave, and appends it to the top graph as an image. Also see the Voronoi parameter of the **ImageInterpolate** operation on page V-382 for generating an interpolated matrix.

Creating a Contour Plot

Contour plots are appended to ordinary graph windows. All the features of graphs apply to contour plots: axes, line styles, drawing tools, controls, etc. See Chapter II-13, **Graphs**.

You can create a contour plot in a new graph window by choosing Windows→New→Contour Plot. This displays the New Contour Plot dialog. This dialog creates a blank graph to which the plot is appended.

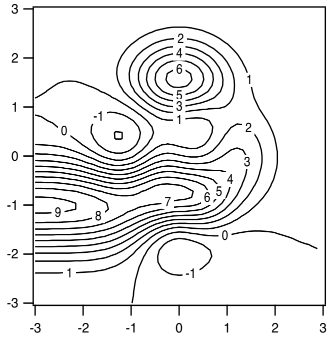
To add a contour plot to an existing graph, choose Graph→Append To Graph→Contour plot. This displays the Append Contour Plot dialog.

You can also use the **AppendMatrixContour** or **AppendXYZContour** operations.

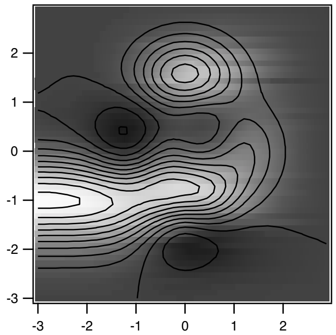
You can add a sense of depth to a gridded contour plot by adding an image plot to the same graph. To do this, choose Graph-Append To Graph-Image Plot. This displays the Append Image Plot dialog. If your data is in XYZ form, use the AppendImageToContour WaveMetrics procedure to create and append an image.

These commands generate a gridded contour plot with an image plot:

```
Make/O/D/N=(50,50) mat2d            // Make some data to plot
SetScale x,-3,3,mat2d
SetScale y,-3,3,mat2d
mat2d =  3*(1-y)^2 * exp(-((x*(x>0))^2.5) - (y+0.5)^2)
mat2d -= 8*(x/5 - x^3 - y^5) * exp(-x^2-y^2)
mat2d -= 1/4 * exp(-(x+.5)^2 - y^2)
Display;AppendMatrixContour mat2d   //This creates the contour plot
AppendImage mat2d
```



AppendMatrixContour mat2d



AppendMatrixContour mat2d
AppendImage mat2d

Instead of displaying an image plot with the contour plot, you can instruct Igor to add a color fill between contour levels. You can see this using these commands:

```
RemoveImage mat2d
ModifyContour mat2d ctabFill={*,*,Grays256,0}
```

Modifying a Contour Plot

You can change the appearance of the contour plot choosing Graph→Modify Contour Appearance. This displays the Modify Contour Appearance dialog. This dialog is also available as a subdialog of the New Contour Plot and Append Contour Plot dialogs.

You can open the Modify Contour Appearance by Shift-double-clicking the contour plot or by right-clicking it but not on a contour and choosing Modify Contour Appearance from the contextual menu.

Use preferences to change the default contour appearance, so you won’t be making the same changes over and over. See **Contour Preferences** on page II-380.

The Modify Contour Appearance Dialog

The following sections describe some of the contour plot parameters you can change using the Modify Contour Appearance dialog.

Contour Data Pop-Up Menu

The Contour Data pop-up menu shows the “contour instance name” of the contour plot being modified. The name of the contour plot is the same as the name of the Z wave containing the contour data.

If the graph contains more than one contour plot, you can use this pop-up menu to change all contour plots in the target graph.

If the graph contains two contour plots *using the same Z wave name*, an instance number is appended to those Z wave names in this pop-up menu. See **Instance Notation** on page IV-20, and **Contour Instance Names** on page II-376.

Contour Levels

Each contour trace draws lines at one constant Z level. The Z levels are assigned automatically or manually as specified in this part of the dialog.

Igor computes automatic levels by subdividing the range of Z values into approximately the number of requested levels. You can instruct Igor to compute the Z range automatically from the minimum and maximum of the Z data, or to use a range that you specify in the dialog. Igor attempts to choose “nice” contour levels that minimize the number of significant digits in the contour labels. To achieve this, Igor may create more or fewer levels than you requested.

You can specify manual levels directly in the dialog in several ways:

- Linearly spaced levels (constant increment) starting with a first level and incrementing by a specified amount.
- A list of arbitrary levels stored in a wave you choose from a pop-up Wave Browser.
- A list of arbitrary levels you enter in the More Contour Levels dialog that appears when you click the More Contour Levels checkbox. These levels are *in addition to* automatic, manual, or from-wave levels.

The More Levels dialog can be used for different purposes:

- To add a contour level to those already defined by the automatic, manual, or from-wave levels. You might do this to indicate a special feature of the data.
- As the only source of arbitrary contour levels, for complete control of the levels. You might do this to slightly change a contour level to avoid problems (see **Contouring Pitfalls** on page II-379). Disable the auto or manual levels by entering 0 for the number of levels. The only contour levels in effect will be those entered in the More Levels dialog.

WaveMetrics provides some utility procedures for dealing with contour levels. See “WaveMetrics Contour Plot Procedures” in the “WM Procedures Index” help file for details.

Update Contours Pop-Up Menu

Igor normally recalculates and redraws the contour plot whenever any change occurs that might alter its appearance. This includes changes to any data waves and the wave supplying contour levels, if any. Since calculating the contour lines can take a long time, you may want to disable this automatic update with the Update Contours pop-up menu.

“Off” completely disables updating of the contours for any reason. Choose “once, now” to update the contour when you click the Do It button. “Always” selects the default behavior of updating whenever the contour levels change.

Contour Labels Pop-Up Menu

Igor normally adds labels to the contour lines, and updates them whenever the contour lines change (see **Update Contours Pop-Up Menu** on page II-369). Since updating plots with many labels can take a long time, you may want to disable or modify this automatic update with the Labels pop-up menu.

“None” removes any existing contour labels, and prevents any more from being generated.

“Don’t update any more” keeps any existing labels, and prevents any more updates. This is useful if you have moved, removed, or modified some labels and you want to keep them that way.

“Update now, once” will update the labels when you click the Do It button, then prevents any more updates. Use this if updating the labels takes too long for you to put up with automatic updates.

“If contours change”, the default, updates the labels whenever Igor recalculates the contour lines.

“Always update” is the most aggressive setting. It updates the labels if the graph is modified in almost any way, such as changing the size of the graph or adjusting an axis. You might use this setting temporarily while making adjustments that might otherwise cause the labels to overlap or be too sparse.

Click Label Tweaks to change the number format and appearance of the contour labels with the Contour Labels dialog. See **Modifying Contour Labels** on page II-378.

For more than you ever wanted to know about contour labels, see **Contour Labels** on page II-377.

Line Colors Button

Click the Line Colors button to assign colors to the contour lines according to their Z level, or to make them all the same color, using the Contour Line Colors dialog.

Autoscaled color mapping assigns the first color in a color table to the minimum Z value of the contour data (not to the minimum contour level), and the last color to the maximum Z value.

See **The Color of Contour Traces** on page II-371 for further discussion of contour colors.

Show Boundary Checkbox

Click this to generate a trace along the perimeter of the contour data in the XY plane. For a matrix contour plot, the perimeter is simply a rectangle enclosing the minimum and maximum X and Y. The perimeter of XYZ triplet contours connects the outermost XY points. This trace is updated at the same time as the contour level traces.

Show XY Markers Checkbox

Click this to generate a trace that shows the XY locations of the contour data. For a matrix contour plot, the locations are by default marked with dots. For XYZ triplet contours, they are shown using markers. As with any other contour trace, you can change the mode and marker of this trace with the Modify Trace Appearance dialog. This trace is updated at the same time as the contour level traces.

Fill Contours Checkbox

Click this to fill between contour levels with solid colors. Click the Fill Colors button to adjust the colors in the same manner as described under **Line Colors Button** on page II-369.

Warning: Solid fills can sometimes fail.

You can set the fill and color for individual levels using the Modify Trace Appearance dialog even if the Fill Contours checkbox is off.

See **Contour Fills** on page II-373 for more information.

Show Triangulation Checkbox

Click this to generate a trace that shows the Delaunay triangulation of the contour data. This is available only for XYZ triplet contours. This trace is updated at the same time as the contour level traces.

Interpolation Pop-Up Menu

XYZ triplet contours can be interpolated to increase the apparent resolution, resulting in smoother contour lines. The interpolation uses the original Delaunay triangulation. Increasing the resolution requires more time and memory; settings higher than x16 are recommended only to the very patient.

Contour Traces

Igor creates XY pairs of double-precision waves to contain the contour trace data, and displays them as ordinary graph traces. Each trace draws all the curves for one Z level. If a single Z level generates more than one contour line, Igor uses a blank (NaN) at the end of each contour line to create a gap between it and the following line.

The same method is used to display markers at the data’s XY coordinates, the XY domain’s boundary, and, for XYZ triplet contours only, the Delaunay triangulation.

The names of these traces are fabricated from the name of the Z data wave or matrix. See **Contour Trace Names** on page II-371.

One important special property of these waves is that they are private to the graph. These waves do not appear in the Data Browser or in any other dialog, and are not accessible from commands. There is a trick you can use to copy these waves, however. See **Extracting Contour Trace Data** on page II-376.

The contour traces, which are the visible manifestation of these private waves, *do* show up in the Modify Trace Appearance dialog, and *can* be named in commands just like other traces.

There is often no need to bother with the individual traces of a contour plot because the Modify Contour Appearance dialog provides adequate control over the traces for most purposes. However, if you want to distinguish one or more contour levels, to make them dashed lines, for example, you can do this by modifying the traces using the Modify Trace Appearance dialog. For further discussion, see **Overriding the Color of Contour Traces** on page II-373.

Contour Trace Names

The name of a contour trace is usually something like “zwave=2.5”, indicating that the trace is the contour of the z data set “zwave” at the z=2.5 level. A name like this must be enclosed in single quotes when used in a command:

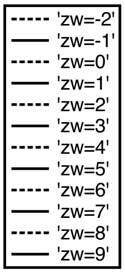
```
ModifyGraph mode('zwave=2.5')=2
```

This trace naming convention is the default, but you can create a different naming convention when you first append a contour to a graph using the /F flag with AppendMatrixContour or AppendXYZContour. The contour dialogs do not generate /F flags and therefore create contours with default names. See the **AppendMatrixContour** operation on page V-33 and the **AppendXYZContour** operation on page V-39 for a more thorough discussion of /F.

Example: Contour Legend with Numeric Trace Names

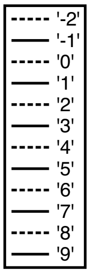
To make a legend contain slightly nicer wording you can omit the “zwave=” portion from trace names with `/F="%g"`:

```
AppendMatrixContour/F="%g" zw    // trace names become just numbers
```



```
----- 'zw=-2'
————— 'zw=-1'
----- 'zw=0'
————— 'zw=1'
----- 'zw=2'
————— 'zw=3'
----- 'zw=4'
————— 'zw=5'
----- 'zw=6'
————— 'zw=7'
----- 'zw=8'
————— 'zw=9'
```

`/F="%s=%g"` (default)



```
----- '-2'
————— '-1'
----- '0'
————— '1'
----- '2'
————— '3'
----- '4'
————— '5'
----- '6'
————— '7'
----- '8'
————— '9'
```

`/F="%g"`

You can manually edit the legend text to remove the single quotes around the trace names. Double-click the legend to bring up the Modify Annotation dialog.

For details on creating contour plot legends, see **Contour Legends** on page II-377.

Contour Trace Programming Notes

The **TraceNameList** function returns a string which contains a list of the names of contour traces in a graph. You can use the name of the trace to retrieve a wave reference for the private contour wave using **TraceNameToWaveRef**. See also **Extracting Contour Trace Data** on page II-376.

If a graph happens to contain two traces with the same name, “instance notation” uniquely identifies them. For example, two traces named “2.5” would show up in the Modify Trace Appearance dialog as “2.5” and “2.5#1”. On the command line, you would use something like:

```
ModifyGraph mode('2.5'#1)=2
```

Notice how the instance notation (the “#1” part) is outside the single quotes. This instance notation is needed when the graph contains two contour plots that generate identically named traces, usually when they use the same /F parameters and draw a contour line at the same level (z=2.5).

See **Instance Notation** on page IV-20. Also see **Contour Instance Names** on page II-376.

The Color of Contour Traces

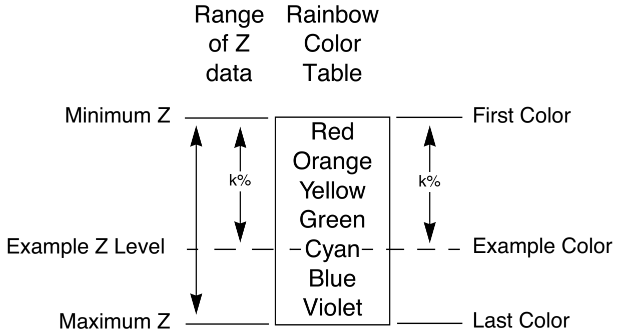
By default, contour traces are assigned a color from the Rainbow color table based on the contour trace’s Z level. You can choose different colors in the Contour Line Colors subdialog of the Modify Contour Appearance dialog.

The Contour Line Colors dialog provides four choices for setting the contour trace colors:

1. All contour traces can be set to the same color.
2. The color for a trace can be selected from a “color index wave” that you created, by matching the trace’s Z level with the color index wave X index.
3. The color for a trace can be computed from a chosen built-in color table or from a color table wave that you created, by matching the trace’s Z level with the range of available colors.

Color Tables

When you use a color table to supply colors, Igor maps the Z level to an entry in the color table. By default, Igor linearly maps the entire range of Z levels to the range of colors in the table by assigning the minimum Z value of the contour data to the first color in the table, and the maximum Z value to the last color:



Automatic Mode Color Table Mapping

With the Modify Contour Appearance dialog, you can assign a specific Z value to the color table’s first and last colors. For example, you can use the first half of the color table by leaving First Color on Auto, and typing a larger number (2*(ZMax -ZMin), to be precise) for the Last Color.

To see what the built-in color tables look like, see **Color Table Details** on page II-395.

You can create your own color table waves that act like custom color tables. See **Color Table Waves** on page II-399 for details.

Color Index Wave

You can create your own range of colors by creating a color index wave. The wave must be a 2D wave with three columns containing red, green, and blue values that range from 0 (zero intensity) to 65535 (full intensity), and a row for each color. Igor finds the color for a particular Z level by choosing the row in the color index wave whose X index most closely matches the Z level.

To choose the row, Igor converts the Z level into a row number as if executing:

```
colorIndexWaveRow= x2pnt(colorIndexWave,Z)
```

which rounds to the nearest row and limits the result to the rows in the color index wave.

When the color index wave has default X scaling (the X index is equal to row number), then row 0 contains the color for z=0, row 1 contains the color for z=1, etc. By setting the X scaling of the wave (Change Wave Scaling dialog), you can control how Igor maps Z level to color. This is similar to setting the First Color and Last Color values for a color table.

Color Index Wave Programming Notes

Looking up a color in a color index wave can be expressed programmatically as:

```
red=   colorIndexWave (Z level)[0]
green= colorIndexWave (Z level)[1]
blue=  colorIndexWave (Z level)[2]
```

where ( ) indexes into rows using X scaling, and [ ] selects a column using Y point number (column number).

Here is a code fragment that creates a color index wave that varies from blue to red:

```
Function CreateBlueRedColorIndexWave(numberOfColors,zMin,zMax)
   Variable numberOfColors
   Variable zMin,zMax              // From min, max of contour or image data

   Make/O/N=(numberOfColors,3) colorIndexWave
   Variable white = 65535          // black is zero
   Variable colorStep = white / (numberOfColors-1)

   colorIndexWave[][0]= colorStep*p // red increases with row number,
   colorIndexWave[][1]= 0          // no green
   colorIndexWave[][2]= colorStep*(numberOfColors-1-p)    // blue decreases

   SetScale/I x,zMin,zMax,colorIndexWave// Match X scaling to Z range
End
```

Log Color for Contour Traces

You can obtain a logarithmic mapping of z level values to colors using the Log Color checkbox in the Contour Line Colors dialog. This generates a `ModifyContour logLines=1` command. In this mode, the colors change more rapidly at smaller contour z level values than at larger values.

For a color index wave, the colors are mapped using the log(color index wave's x scaling) and log(contour z level) values this way:

```
colorIndexWaveRow = (nRows-1)*(log(Z)-log(xMin))/(log(xmax)-log(xMin))
```

where,

```
nRows = DimSize(colorIndexWave,0)
xMin = DimOffset(colorIndexWave,0)
xMax = xMin + (nRows-1) * DimDelta(colorIndexWave,0)
```

The colorIndexWaveRow value is rounded before it is used to select a color from the color index wave.

A similar mapping is performed with color tables, where the xMin and xMax are replaced with the automatically determined or manually provided zMin and zMax values.

Overriding the Color of Contour Traces

You can override the color set by the Line Color subdialog by using the Modify Trace Appearance dialog, the **ModifyGraph** command, or by Control-clicking (*Macintosh*) or right-clicking (*Windows*) the graph's plot area to pop up the **Trace Pop-Up Menu**. The color you choose will continue to be used until either:

1. The trace is removed when the contours are updated, because the levels changed, for instance.
2. You choose a new setting in the Line Color subdialog.

Contour Fills

You can specify colors or patterns to fill contour levels. To fill all contour levels with colors, check the Fill Contours checkbox in the Modify Contour dialog. To fill individual contour levels with colors or patterns, use the Contour Fill controls in the Modify Trace Appearance dialog. You can use the Add Annotation dialog to create a legend for contour fills.

Programatically, to turn contour fills on for all contour levels, execute:

```
ModifyContour <contour instance name>, fill=1
```

To control the color use one of these ModifyContour keywords:

ctabFill	Fills using a color table
cindexFill	Fills using a color index wave
rgbFill	Fills with a specific color

These keywords have the same syntax as `ctabLines`, **cindexLines** and `rgbLines` which control the colors of the contour lines themselves.

To turn on an individual contour level fill, execute:

```
ModifyContour <contour instance name>, fill=0         // Global fill mode off
ModifyGraph usePlusRGB(<contour level trace name>)=1  // Trace fill mode on
ModifyGraph hbFill(<contour level trace name>)=2      // Solid fill for trace
```

For example, in the Contour Demo example experiment, select Macros→Matrix Contour plot to display the Demo Matrix Contour graph. Double-click one of the traces to display the Modify Trace Appearance dialog. Choose Solid for +Fill Type; this automatically checks the Custom Fill checkbox. Select a yellow color from the associated popup menu. This gives the following commands:

```
ModifyGraph usePlusRGB('RealisticData=6')=1
ModifyGraph hbFill('RealisticData=6')=2
ModifyGraph plusRGB('RealisticData=6')=(65535,65532,16385)
```

You can also fill all contour levels, using `ModifyContour fill=1`, and then customize one or more levels using this technique.

You can create a color bar for contour fills using the **ColorScale** operation with the `contourFill` keyword. The syntax is the same as for the ColorScale `contour` keyword.

Solid fills can sometimes fail because Igor can not determine a closed path for a contour line. Be sure to visually inspect the results and turn off fills if they are not correct. The success or failure of a contour fill is highly dependent on the data and is more likely with XYZ data. To see this, choose File→Example Experiments→Sample Graphs→Contour Demo and choose the XYZ Contour Plot from the Macros menu. Turn on Fill Levels and experiment with the number of points and the z-function. Occasionally you may see a warning in the history area saying that a contour level is not closeable. There is not much you can do about this other than trying a different data set or converting your XYZ data to a matrix. Although rare, even matrix data can be sufficiently pathological as to cause the contour fill to fail.

If automatic fills do not work with your data, you can use a background image to provide the fill effect using the WaveMetrics procedure FillBetweenContours. See Image and Contour Plots in the WM Procedures Index for information.

To support fills, Igor needs the boundary trace which it creates and then sets as hidden. When loading an experiment into Igor6, you will encounter an error on the command to hide this trace. You can continue the load by simply commenting out this command in the error dialog.

Removing Contour Traces from a Graph

Removing traces from a contour plot with the RemoveFromGraph operation or the Remove from Graph dialog will work only temporarily. As soon as Igor updates the contour traces, any removed traces may be replaced.

You can prevent this replacement by disabling contour updates with the Modify Contour Appearance dialog. It is better, however, to use the Modify Contour Appearance dialog to control which traces are drawn in the first place.

To permanently remove a particular automatic or manual contour level, you are better off not using manual levels or automatic levels at all. Use the More Contour Levels dialog to explicitly enter all the levels, and enter zero for the number of manual or automatic levels.

Tip: Make a legend in the graph to list the contour traces. The trace names normally contain the contour levels. To do this, select the command window, type `Legend`, and press Return or Enter.

Similarly, if you don't want the triangulation or XY marker traces to be drawn, use the Modify Contour Appearance dialog to turn them off, rather than removing them with the Remove from Graph dialog.

Cursors on Contour Traces

You can attach cursors to a contour trace. Just like any other trace, the X and Y values are shown in the graph info panel (choose Show Info from the Graph menu). When the cursor is attached to a contour line the Z value (contour level) of the trace is also shown in the info panel.

There are several additional methods for displaying the Z value of a contour trace:

- The zcsr function returns the Z value of the trace the cursor is attached to. zcsr returns a NaN if the cursor is attached to a noncontour trace. You can use this in a procedure, or print the result on the command line using: `Print zcsr(A)`.
- If you add an image behind the contour, you can use cursors to display the X, Y, and Z values at any point.
- The name of the trace the cursor is attached to shows up in the info panel, and the name usually contains the Z value.
- Contour labels show the Z value. Contour labels are tags that contain the \OZ escape code or TagVal(3). See **Contour Labels** on page II-377. You can drag these labels around by pressing Option (*Macintosh*) or Alt (*Windows*) while dragging a tag. See **Changing a Tag's Attachment Point** on page III-45.

Contour Trace Updates

Igor normally updates the contour traces whenever the contour data (matrix or XYZ triplets) changes or whenever the contour levels change. Because creating the contour traces can be a lengthy process, you can prevent these automatic updates through a pop-up menu item in the Modify Contour Appearance dialog. See **Update Contours Pop-Up Menu** on page II-369.

Preventing automatic updates can be useful when you are repeatedly editing the contour data in a table or from a procedure. Use the "once, now" pop-up item to manually update the traces.

Contour Trace Update Programming Note

Programmers should be aware of another update issue: contour traces are created in two steps. To understand this, look at this graph recreation macro that appends a contour to a graph and then defines the contour levels, styles and labels:

```
Window Graph1() : Graph
   PauseUpdate; Silent 1      // building window...
   Display /W=(11,42,484,303)
   AppendMatrixContour zw
   ModifyContour zw autoLevels={*,*,5}, moreLevels={0.5}
   ModifyContour zw rgbLines=(65535,0,26214)
   ModifyContour zw labels=0
   ModifyGraph lSize('zw=0')=3
   ModifyGraph lStyle('zw=0')=3
   ModifyGraph rgb('zw=0')=(0,0,65535)
   ModifyGraph mirror=2
EndMacro
```

First, the AppendMatrixContour operation runs and creates stub traces consisting of zero points. The ModifyContour and ModifyGraph operations that follow act on the stub traces. Finally, after all of the commands that define the graph have executed, Igor does an update of the entire graph, when the effect of the PauseUpdate operation expires when the graph recreation macro returns. This is the time when Igor does the actual contour computations which convert the stub traces into fully-formed contour traces.

This delayed computation prevents unnecessary computations from occurring when ModifyContour commands execute in a macro or function. The ModifyContour command often changes default contour level settings, rendering any preceding computations obsolete. For the same reason, the New Contour Plot dialog appends ";DelayUpdate" to the Append Contour commands when a ModifyContour command is also generated.

The DoUpdate operation updates graphs and objects. You can call DoUpdate from a macro or function to force the contouring computations to be done at the desired time.

Drawing Order of Contour Traces

The contour traces are drawn in a fixed order. From back-to-front, that order is:

1. Triangulation (Delaunay Triangulation trace, only for XYZ contours),
2. Boundary
3. XY markers
4. Contour trace of lowest Z level

... intervening contour traces are in order from low-to-high Z level...

N. Contour of highest Z level

You can temporarily override the drawing order with the **Reorder Traces Dialog**, the **Trace Pop-Up Menu**, or the **ReorderTraces** operation. The order you choose will be used until the contour traces are updated. See **Contour Trace Updates** on page II-375.

The order of a contour plot's traces relative to any other traces (the traces belonging to another contour plot for instance) is *not* preserved by the graph's window recreation macro. Any contour trace reordering is lost when the experiment is closed.

Extracting Contour Trace Data

Advanced users may want to create a copy of a private XY wave pair that describes a contour trace. You might do this to extract the Delaunay triangulation, or simply to inspect the X and Y values in a table, for example. To extract contour wave pair(s), include the Extract Contours As Waves procedure file:

```
#include <Extract Contours As Waves>
```

which adds "Extract One Contour Trace" and "Extract All Contour Traces" menu items to the Graph menu.

Another way to copy the traces into a normal wave is to use the Data Browser to browse the saved experiment. The contour traces are saved as waves in a temporary data folder whose name begins with "WM_CTraces_" and ends with the contour's "contour instance name". See **The Browse Expt Button** on page II-117 for details about browsing experiment files.

Contour Instance Names

Igor identifies a contour plot by the name of the wave providing Z values (the matrix wave or the Z wave). This "contour instance name" is used in commands that modify the contour plot.

Contour instance names are not the same as contour trace instance names. Contour instance names refer to an entire contour plot, not to an individual contour trace.

The Modify Contour Appearance dialog generates the correct contour instance name automatically.

Contour instance names work much the same way wave instance names for traces in a graph do. See **Instance Notation** on page IV-20.

Examples

In the first example the contour instance name is "zw":

```
Display; AppendMatrixContour zw                    // New contour plot
ModifyContour zw ctabLines={*,*,BlueHot}           // Change color table
```

In the unusual case that a graph contains two contour plots of the same data, an instance number must be appended to the name to modify the second plot: zw#1 is the contour instance name of the second contour plot:

```
Display
AppendMatrixContour zw; AppendMatrixContour zw     // Two contour plots
ModifyContour zw ctabLines={*,*,RedWhiteBlue}      // Change first plot
ModifyContour zw#1 ctabLines={*,*,BlueHot}         // Change second plot
```

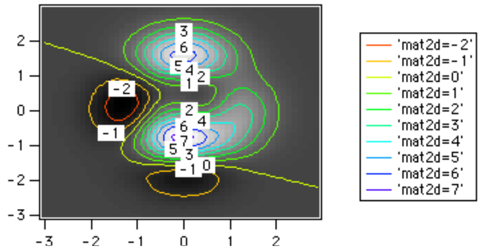
You might have two contour plots of the same data to show different subranges of the data side-by-side. This example uses separate axes for each plot.

The **ContourNameList** function returns a string containing a list of contour instance names. Each name corresponds to one contour plot in the graph. **ContourInfo** (see page V-85) returns information about a particular named contour plot.

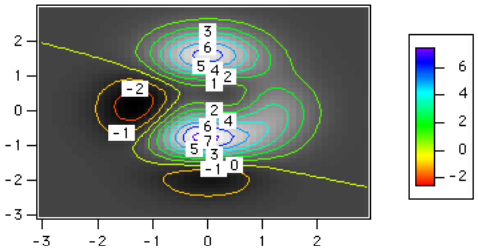
Contour Legends

You can create two kinds of legends appropriate for contour plots using the Add Annotation dialog: a Legend or a ColorScale. For more details about the Add Annotation dialog and creating legends, see Chapter III-2, **Annotations**, and the **Legends** (see page III-42) and **Color Scales** (see page III-47) sections.

A Legend annotation will display the contour traces with their associated color. A ColorScale will display the entire color range as a color bar with an axis that spans the range of colors associated with the contour data.



Legend



Color Scale

Contour Labels

Igor uses specialized tags to create the numerical labels for contour plots. Igor puts one label on every contour curve. Usually there are several contour curves drawn by one contour trace. The tag uses the \OZ escape code or the TagVal(3) function to display the contour level value in the tag instead of displaying the literal value. See **Annotation Text Content** on page III-35 for more about escape codes and tags.

You can select the rotation of contour labels using the Label Tweaks subdialog of the Modify Contour Appearance Dialog. You can request tangent, horizontal, vertical or both orientations. If permitted, Igor will prefer horizontal labels. The "Snap to" alternatives convert horizontal or vertical labels within 2 degrees of horizontal or vertical to exactly horizontal or vertical.

Igor positions the labels so that they don't overlap other annotations and aren't outside the graph's plot area. Contour labels are slightly special in that they are always drawn below all other annotations, so that they will never show up on top of a legend or axis label. Igor chooses label locations and tangent label orientations based on the slope of the contour trace on the screen.

Controlling Contour Label Updates

By default, Igor automatically relabels the graph only when the contour data or contour levels change, but you can control when labels update with the Labels pop-up menu in the Modify Contour Appearance dialog. See **Contour Labels Pop-Up Menu** on page II-369. Be aware that updating a graph containing many labels can be slow.

Repositioning and Removing Contour Labels

Contour labels are “frozen” so that they can’t be dragged, but since they are tags, you *can* Option-drag (*Macintosh*) or Alt-drag (*Windows*) them to a new attachment point. See **Changing a Tag’s Attachment Point** on page III-45. The labels are frozen to make them harder to accidentally move.

You can reposition contour labels, but they will be moved back, moved to a completely new position, or deleted when labels are updated. If you want full manual control of labels, turn off label updating before that happens. See **Controlling Contour Label Updates** on page II-377.

Here’s a recommended strategy for creating contour labels to your liking:

1. Create the contour plot and set the graph window to the desired size.
2. Choose Graph→Modify Contour Appearance, click the Label Tweaks button, and choose the rotation for labels, and any other label formatting options you want.
3. Choose “update now, once” from the Labels pop-up menu, and then click the Do It button.
4. Option-drag or Alt-drag any labels you don’t want completely off the graph.
5. Option-drag or Alt-drag any labels that are in the wrong place to another attachment point. You can drag them to a completely different trace, and the value printed in the label will change to the correct value.

To drag a label away from its attachment point, you must first unfreeze it with Position pop-up menu in the **Annotation Tweaks** dialog. See **Overriding the Contour Labels** on page II-378.

Adding Contour Labels

You can add a contour label with a Tag command like:

```
Tag/Q=ZW#1/F=0/Z=1/B=2/I=1/X=0/Y=0/L=1 'zw#1=2', 0 , "\\OZ"
```

An easier alternative is to use the Modify Annotation dialog to duplicate the annotation and then drag it to a new location.

Modifying Contour Labels

You can change the label font, font size, style, color, and rotation of all labels for a contour plot by clicking Label Tweaks in the Modify Contour Appearance dialog. This brings up the Contour Labels subdialog.

You can choose the rotation of contour labels from tangent, horizontal, vertical or both orientations. If both vertical and horizontal labels are permitted, Igor will choose vertical or horizontal with a preference for horizontal labels. Selecting one of the Tangent choices creates labels that are rotated to follow the contour line. The "Snap to" alternatives convert labels within 2 degrees of horizontal or vertical to exactly horizontal or vertical.

You can choose a specific font, size, and style. The “default” font is the graph’s default font, as set by the Modify Graph dialog.

The background color of contour labels is normally the same as the graph background color (usually white). With the Background pop-up menu, you can select a specific background color for the labels, or choose the window background color or the transparent mode.

You can choose among general, scientific, fixed-point, and integer formats for the contour labels. These correspond to printf conversion specifications, “%g”, “%e”, “%f”, and “%d”, respectively (see the **printf**). These specifications are combined with TagVal(3) into a dynamic text string that is used in each tag.

For example, choosing a Number Format of “###0.0...0” with 3 Digits after Decimal Point in the Contour-Labels dialog results in the contour tags having this as their text: `\{"%.3f",tagVal(3)}`. This format will create contour labels such as “12.345”.

Overriding the Contour Labels

Since Igor implements contour labels using standard tags, you can adjust labels individually by simply double-clicking the label to bring up the Modify Annotation dialog.

However, once you modify a label, Igor no longer considers it a contour label and will not automatically update it any more. When the labels are updated, the modified label will be ignored, which may result in two labels on a contour curve.

You may want to take complete, manual control of contour labels. In this case, set the Labels pop-up menu in the Modify Contour Appearance dialog to “no more updates” so that Igor will no longer update them. You can then make any desired changes without fear that Igor will undo them.

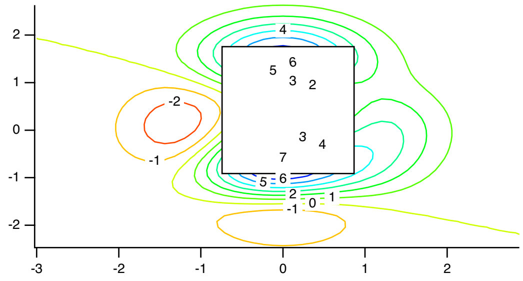
Contour labels are distinguished from other tags by means of the /Q flag. Tag/Q=*contourInstanceName* assigns the tag to the named contour. Igor uses the /Q flag in recreation macros to assign tags to a particular contour plot.

When you edit a contour label with the Modify Annotation dialog, the dialog adds a plain /Q flag (with no =*contourInstanceName* following it) to the Tag command to divorce the annotation from its contour plot.

Add the `/Q=ContourInstanceName` to Tag commands to temporarily assign ownership of the annotation to the contour so that it is deleted when the contour labels are updated.

Contour Labels and Drawing Tools

One problem with Igor’s use of annotations as contour labels is that normal drawing layers are below annotations. If you use the drawing tools to create a rectangle in the same location as some contour labels, you will encounter something like the following window.



You can solve this by putting the drawing in the overlay layer. See **Drawing Layers** on page III-68 for details.

Another solution is to remove the offending labels as described under **Repositioning and Removing Contour Labels** on page II-378.

Contouring Pitfalls

You may encounter situations in which the contour plot doesn’t look as you expect. This section discusses these pitfalls.

Insufficient Resolution

Contour curves are generally closed curves, or they intersect the data boundary. Under certain conditions, typically when using XYZ triplet data, the contouring algorithm may generate what appears to be an open curve (a line rather than a closed shape). This open curve typically corresponds to a peak ridge or a valley trough in the surface. At times, an open curve may also correspond to a line that intersects a nonobvious boundary.

The line may actually be a very narrow closed curve: zoom in by dragging out a marquee, clicking inside, and choosing “expand” from the pop-up menu.

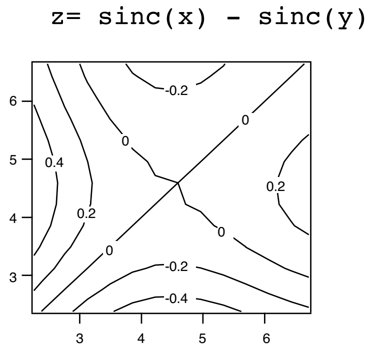
If it really is a line, increasing the resolution of the data in that region, by adding more X, Y, Z triplets, may result in a closed curve. Selecting a higher interpolation setting using the Modify Contour Appearance dialog may help.

Another solution is to shift the contour level slightly down from a peak or up from a valley. Or you could choose a new set of levels that don’t include the level exhibiting the problem. See **Contour Levels** on page II-368.

Crossing Contour Lines

Contour lines corresponding to different levels will not cross each other, but contour lines of the same level may appear to intersect. This typically happens when a contour level is equal to a “saddle point” of the surface. An example of this is a contour level of zero for the function:

```
z= sinc(x) - sinc(y)
```



You should shift the contour level away from the level of the saddle point. See **Contour Levels** on page II-368.

Flat Areas in the Contour Data

Patches of constant Z values in XYZ triplet data don’t contour well at those levels. If the data has flat areas equal to 2.0, for example, a contour level at Z=2.0 may produce ambiguous results. Gridded contour data does not suffer from this problem.

You should shift the contour level above or below the level of the flat area. See **Contour Levels** on page II-368.

Contour Preferences

You can change the default appearance of contour plots by capturing preferences from a prototype graph containing contour plots.

Create a graph containing one or more contour plots having the settings you use most often. Then choose Capture Graph Prefs from the Graph menu. Select the Contour Plots category, and click Capture Prefs.

Preferences are normally in effect only for *manual* operations, not for automatic operations from Igor procedures. This is discussed in more detail in Chapter III-18, **Preferences**.

The Contour Plots category includes both contour appearance settings and axis settings.

Contour Appearance Preferences

The captured contour appearance settings are automatically applied to a contour plot when it is first created, provided preferences are turned on. They are also used to preset the Modify Contour Appearance dialog.

If you capture the contour plot preferences from a graph with more than one contour plot, the first contour plot appended to a graph gets the settings from the contour first appended to the prototype graph. The second contour plot appended to a graph gets the settings from the second contour plot appended to the prototype graph, etc. This is similar to the way XY plot wave styles work.

Contour Axis Preferences

Only settings for axes used by the contour plot are captured. Axes used solely for an XY, category, or image plot are not captured when the Contour Plots category is selected.

The contour axis preferences are applied only when axes having the same name as the captured axis are created by AppendMatrixContour or AppendXYZContour commands. If the axes existed before those commands are executed, they are not affected by the axis preferences. The names of captured contour axes are listed in the X Axis and Y Axis pop-up menus of the New Contour Plot and Append Contour Plot dialogs. This is similar to the way XY plot axis preferences work.

You can capture contour axis settings for the standard left and bottom axes, and Igor will save these separately from left and bottom axis preferences captured for XY, category, and image plots. Igor will use the contour axis settings for AppendMatrixContour or AppendXYZContour commands only.

How to Use Contour Plot Preferences

Here is our recommended strategy for using contour preferences:

1. Create a new graph containing a single contour plot. If you want to capture the triangulation and interpolation settings, you must make an XYZ contour plot. Use the axes you will want for a contour plot.
2. Use the Modify Contour Appearance dialog and the Modify Axis dialog to make the contour plot appear as you prefer.
3. Choose Graph→Capture Graph Prefs, select the Contour Plots category, and click Capture Prefs.

Contour Plot Shortcuts

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To modify the appearance of the contour plot as a whole	Control-click and choose Modify Contour from the pop-up menu or press Shift and double-click the plot area of the graph, away from any labels or traces. This brings up the Modify Contour Appearance dialog.	Right-click and choose Modify Contour from the pop-up menu or press Shift and double-click the plot area of the graph, away from any labels or traces. This brings up the Modify Contour Appearance dialog.
To modify a contour label	Press Shift and double-click the plot area, and click Label Tweaks in the Modify Contour Appearance dialog. Don't double-click the label to use the Modify Annotation dialog unless you intend to maintain the label yourself. See **Overriding the Contour Labels** on page II-378.	Press Shift and double-click the plot area, and click Label Tweaks in the Modify Contour Appearance dialog. Don't double-click the label to use the Modify Annotation dialog unless you intend to maintain the label yourself. See **Overriding the Contour Labels** on page II-378.
To remove a contour label	Press Option, click on the label, and drag it completely off the graph.	Press Alt, click on the label, and drag it completely off the graph.
To move a contour label to another contour trace	Press Option, click on the label and drag it to another contour trace.	Press Alt, click on the label and drag it to another contour trace.
To duplicate a contour label	Double-click any label, click the Duplicate button, and click Do It.	Double-click any label, click the Duplicate button, and click Do It.
To modify the appearance or front-to-back drawing order of a contour trace	Control-click the trace to get a contextual menu. Press Shift-Control to modify all traces. Double-click the trace to summon the Modify Trace Appearance dialog.	Right-click the trace to get a contextual menu. Press Shift while right-clicking to modify all traces. Double-click the trace to summon the Modify Trace Appearance dialog.

Contour References

Watson, David F., *Contouring: A Guide to the Analysis and Display of Spatial Data*, 340 pp., Pergamon Press, New York, 1992.

Contour Plot Shortcuts

Because contour plots are drawn in a normal graph, all of the **Graph Shortcuts** (see page II-353) apply. Here we list those which apply specifically to contour plots.

Action	**Shortcut** (Macintosh)	**Shortcut** (Windows)
To modify the appearance of the contour plot as a whole	Control-click and choose Modify Contour from the pop-up menu **or** Press Shift and double-click the plot area of the graph, away from any labels or traces. This brings up the Modify Contour Appearance dialog.	Right-click and choose Modify Contour from the pop-up menu **or** Press Shift and double-click the plot area of the graph, away from any labels or traces. This brings up the Modify Contour Appearance dialog.
To modify a contour label	Press Shift and double-click the plot area, and click Label Tweaks in the Modify Contour Appearance dialog. Don't double-click the label to use the Modify Annotation dialog unless you intend to maintain the label yourself. See **Overriding the Contour Labels** on page II-378.	Press Shift and double-click the plot area, and click Label Tweaks in the Modify Contour Appearance dialog. Don't double-click the label to use the Modify Annotation dialog unless you intend to maintain the label yourself. See **Overriding the Contour Labels** on page II-378.
To remove a contour label	Press Option, click in the label, and drag it completely off the graph.	Press Alt, click in the label, and drag it completely off the graph.
To move a contour label to another contour trace	Press Option, click in the label and drag it to another contour trace.	Press Alt, click in the label and drag it to another contour trace.
To duplicate a contour label	Double-click any label, click the Duplicate button, and click the Do It button.	Double-click any label, click the Duplicate button, and click the Do It button.
To modify the appearance or front-to-back drawing order of a contour trace	Press Control and click the trace to get a pop-up menu. Press Command-Shift to modify all traces. Double-click the trace to summon the Modify Trace Appearance dialog. Also see **Overriding the Color of Contour Traces** on page II-373 and **Drawing Order of Contour Traces** on page II-376.	Right-click the trace to get a contextual menu. Press Shift while right-clicking to modify all traces. Double-click the trace to summon the Modify Trace Appearance dialog. Also see **Overriding the Color of Contour Traces** on page II-373 and **Drawing Order of Contour Traces** on page II-376.

Chapter

II-16

Image Plots

Overview

You can display image data as an image plot in a graph window. The image data can be a 2D wave, a layer of a 3D or 4D wave, a set of three layers containing RGB values, or a set of four layers containing RGBA values where A is “alpha” which represents opacity.

When discussing image plots, we use the term *pixel* to refer to an element of the underlying image data and *rectangle* to refer to the representation of a data element in the image plot.

Each image data value defines a the color of a rectangle in the image plot. The size and position of the rectangles are determined by the range of the graph axes, the graph width and height, and the X and Y coordinates of the pixel edges.

If your image data is a floating point type, you can use NaN to represent missing data. This allows the graph background color to show through.

Images are displayed behind all other objects in a graph except the ProgBack and UserBack drawing layers and the background color.

An image plot can be false color, indexed color or direct color.

False Color Images

In false color images, the data values in the 2D wave or layer of a 3D or 4D wave are mapped to colors using a color table. This is a powerful way to view image data and is often more effective than either surface plots or contour plots. You can superimpose a contour plot on top of a false color image of the same data.

Igor has many built-in color tables as described in **Image Color Tables** on page II-392. You can also define your own color tables using waves as described in **Color Table Waves** on page II-399. You can also create color index waves that define custom color tables as described in **Indexed Color Details** on page II-400.

Indexed Color Images

Indexed color images use the data values stored in a 2D wave or layer of a 3D or 4D wave as indices into an RGB or RGBA wave of color values that you supply. “True color” images, such as those that come from video cameras or scanners generally use indexed color. Indexed color images are more common than direct color because they consume less memory. See **Indexed Color Details** on page II-400.

Direct Color Images

Direct color images use a 3D RGB or RGBA wave. Each layer of the wave represents a color component - red, green, blue, or alpha. A set of component values for a given row and column specifies the color for the corresponding image rectangle. With direct color, you can have a unique color for every rectangle. See **Direct Color Details** on page II-401.

Loading an Image

You can load TIFF, JPEG, PNG, BMP, and Sun Raster image files into matrix waves using the **ImageLoad** or the Load Image dialog via the Data menu.

You can also load images fom plain text files, HDF5 files, GIS files, and from camera hardware.

For details, see **Loading Image Files** on page II-157.

Creating an Image Plot

Image plots are displayed in ordinary graph windows. All the features of graphs apply to image plots: axes, line styles, drawing tools, controls, etc. See Chapter II-13, **Graphs**.

You can create an image plot in a new graph window by choosing Windows→New→Image Plot which displays the New Image Plot dialog. This dialog creates a blank graph to which the plot is appended.

The dialog normally generates two commands — a Display command to make a blank graph window, and an AppendImage command to append a image plot to that graph window. This creates a graph like any other graph but, for most purposes, it is more convenient to use the NewImage operation.

Checking the "Use NewImage command" checkbox replaces Display and AppendImage with NewImage. NewImage automatically sizes the graph window to match the number of pixels in the image and reverses the vertical axis so that pictures are displayed right-side-up.

You can show lines of constant image value by appending a contour plot to a graph containing an image. Igor draws contour plots above image plots. See **Creating a Contour Plot** on page II-367 for an example of combining contour plots and images in a graph.

X, Y, and Z Wave Lists

The Z wave is the wave that contains your image data and defines the color for each rectangle in the image plot.

You can optionally specify an X wave to define rectangle edges in the X dimension and a Y wave to define rectangle edges in the Y dimension. This allows you to create an image plot with rectangles of different widths and heights.

When you select a Z wave, Igor updates the X Wave and Y Wave lists to show only those waves, if any, that are suitable for use with the selected Z wave. Only those waves with the proper length appear in the X Wave and Y Wave lists. See **Image X and Y Coordinates** on page II-388 for details.

Choosing _calculated_ from the X Wave list uses the row scaling (X scaling) of the Z wave selected in the Z Wave list to provide the X coordinates of the image rectangle centers.

Choosing _calculated_ from the Y Wave list uses the column scaling (Y scaling) of the Z wave to provide Y coordinates of the image rectangle centers.

Modifying an Image Plot

You can change the appearance of the image plot by choosing Image-Modify Image Appearance. This displays the Modify Image Appearance dialog, which is also available as a subdialog of the New Image Plot dialog.

Tip: Use the preferences to change the default image appearance, so you won't be making the same changes over and over. See **Image Preferences** on page II-403.

The Modify Image Appearance Dialog

The Modify Image Appearance dialog applies to false color and indexed color images, but not direct color images. See **Direct Color Details** on page II-401.

To use indexed color, click the Color Index Wave radio button and choose a color index wave. For color index wave details, see **Indexed Color Details** on page II-400.

To use false color, click the Color Table radio button and choose a built-in color table or click the Color Table Wave radio button and choose a color table wave. Autoscaled color mapping assigns the first color in a color table to the minimum value of the image data and the last color to the maximum value. The dialog uses "Z" to refer to the values in the image wave. For more information, see **Image Color Tables** on page II-392.

Indexed and color table colors are distributed between the minimum and maximum Z values either linearly or logarithmically, based on the `ModifyImage log` parameter, which is set by the Log Colors checkbox.

Use Explicit Mode to select specific colors for specific Z values in the image. If an image element is exactly equal to the number entered in the dialog, it is displayed using the assigned color. This is not very useful for images made with floating-point data; it is intended for integer data. It is almost impossible to enter exact matches for floating-point data.

When you select Explicit Mode for the first time, two entries are made for you assigning white to 0 and black to 255. A third blank line is added for you to enter a new value. If you put something into the blank line, another blank line is added.

To remove an entry, click in the blank areas of a line in the list to select it and press Delete (*Macintosh*) or Backspace (*Windows*).

Image X and Y Coordinates

Images display wave data elements as rectangles. They are displayed versus axes just like XY plots.

The intensity or color of each image rectangle is controlled by the corresponding data element of a matrix (2D) wave, or by a layer of a 3D or 4D wave, or by a set of layers of a 3D RGB or RGBA wave.

When discussing image plots, we use the term *pixel* to refer to an element of the underlying image data and *rectangle* to refer to the representation of a data element in the image plot.

For each of the spatial dimensions, X and Y, the edges of each image rectangle are defined by one of the following:

- The dimension scaling of the wave containing the image data or
- A 1D auxiliary X or Y wave

In the simplest case, all pixels have the same width and height so the pixels are squares of the same size. Another common case consists of rectangular but not square pixels all having the same width and the same height. Both of these are instances of evenly-spaced data. In these cases, you specify the rectangle centers using dimension (X and Y) scaling. This is discussed further under **Image X and Y Coordinates - Evenly Spaced** on page II-389.

Less commonly, you may have pixels of unequal widths and/or unequal heights. In this case you must supply auxiliary X and/or Y waves that specify the edges of the image rectangles. This is discussed further under **Image X and Y Coordinates - Unevenly Spaced** on page II-389.

It is possible to combine these cases. For example, your pixels may have uniform widths and non-uniform heights. In this case you use one technique for one dimension and the other technique for the other dimension.

Sometimes you may have data that is not really image data, because there is no well-defined pixel width and/or height, but is stored in a matrix (2D) wave. Such data may be more suitable for a scatter plot but can be plotted as an image. This is discussed further under **Plotting a 2D Z Wave With 1D X and Y Center Data** on page II-389.

In other cases you may have 1D X, Y and Z waves. These cases are discussed under **Plotting 1D X, Y and Z Waves With Gridded XY Data** on page II-390 and **Plotting 1D X, Y and Z Waves With Non-Gridded XY Data** on page II-391.

The following sections include example commands. If you want to execute the commands, find the corresponding section in the Igor help files by executing:

```
DisplayHelpTopic "Image X and Y Coordinates"
```

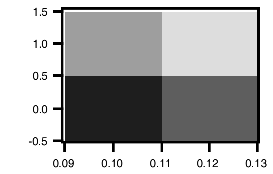
Image X and Y Coordinates - Evenly Spaced

When your data consists of evenly-spaced pixels, you use the image wave's dimension scaling to specify the image rectangle coordinates. You can set the scaling using the Change Wave Scaling dialog (Data menu) or using the **SetScale** operation.

The scaled dimension value for a given pixel specifies the center of the corresponding image rectangle.

Here is an example that uses a 2x2 matrix to exaggerate the effect:

```
Make/O small={{0,1},{2,3}}        // Set X dimension scaling
SetScale/I x 0.1,0.12,"", small
SetScale/P y 0.0,1.0,"", small    // Set Y dimension scaling
Display
AppendImage small    // _calculated_ X & Y
ModifyImage small ctab={-0.5,3.5,Grays}
```



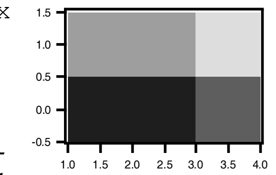
Note that on the X axis the rectangles are centered on 0.10 and 0.12, the matrix wave's X (row) indices as defined by its X scaling. On the Y axis the rectangles are centered on 0.0 and 1.0, the matrix wave's Y (column) indices as defined by its Y scaling. In both cases, the rectangle edges are one half-pixel width from the corresponding index value.

Image X and Y Coordinates - Unevenly Spaced

If your pixel data is unevenly-spaced in the X and/or Y dimension, you must supply X and/or Y waves to define the coordinates of the image rectangle edges. *These waves must contain one more data point than the X (row) or Y (column) dimension of the image wave in order to define the edges of each rectangle.*

In this example, the matrix wave is evenly-spaced in the Y dimension but unevenly-spaced in the X dimension:

```
Make/O small={{0,1},{2,3}}
SetScale/P y 0.0,1.0,"", small    // Set Y dimension scaling
Make smallx={1,3,4}               // Define X edges with smallx
Display
AppendImage small vs {smallx,*}
ModifyImage small ctab={-0.5,3.5,Grays,0}
```



The X coordinate wave (smallx) now controls the vertical edges of each image rectangle. smallx consists of three data points which are necessary to define the vertical edges of the two rectangles in the image plot. The values of smallx are interpreted as follows:

Point 0: 1.0	Sets left edge of first rectangle
Point 1: 2.75	Sets right edge of first rectangle and left edge of second rectangle
Point 2: 4.0	Sets right edge of last rectangle

The 1D edge wave must be either strictly increasing or strictly decreasing.

If you have X and/or Y waves that specify edges but they do not have an extra point, you may be able to proceed by simply adding an extra point. You can do this by editing the waves in a table or using the **InsertPoints** operation. If this is not appropriate, see the next section for another approach.

Plotting a 2D Z Wave With 1D X and Y Center Data

In an image, each pixel has a well-defined width and height. If your data is sampled at specific X and Y points and there is no well-defined pixel width and height, or if you don't know the width and height of each pixel, you don't really have a proper image.

However, because this kind of data is often stored in a matrix wave with associated X and Y waves, it is sometimes convenient to display it as an image, treating the X and Y waves as containing the center coordinates of the pixels.

To do this, you must create new X and Y waves to specify the image rectangle edges. The new X wave must have one more point than the matrix wave has rows and the new Y wave must have one more point than the matrix wave has columns.

A set of image rectangle centers does not uniquely determine the rectangle edges. To see this, think of a 1x1 image centered at (0,0). Where are the edges? They could be anywhere.

Without additional information, the best you can do is to generate a set of plausible edges, as we do with this function:

```
Function MakeEdgesWave(centers, edgesWave)
   Wave centers            // Input
   Wave edgesWave          // Receives output

   Variable N=numpnts(centers)
   Redimension/N=(N+1) edgesWave

   edgesWave[0]=centers[0]-0.5*(centers[1]-centers[0])
   edgesWave[N]=centers[N-1]+0.5*(centers[N-1]-centers[N-2])
   edgesWave[1,N-1]=centers[p]-0.5*(centers[p]-centers[p-1])
End
```

This function demonstrates the use of MakeEdgesWave:

```
Function DemoPlotXYZAsImage()
   Make/O mat={{0,1,2},{2,3,4},{3,4,5}}   // Matrix containing Z values
   Make/O centersX = {1, 2.5, 5}          // X centers wave
   Make/O centersY = {300, 400, 600}      // Y centers wave
   Make/O edgesX; MakeEdgesWave(centersX, edgesX)  // Create X edges wave
   Make/O edgesY; MakeEdgesWave(centersY, edgesY)  // Create Y edges wave
   Display; AppendImage mat vs {edgesX,edgesY}
End
```

If you have additional information that allows you to create edge waves you should do so. Otherwise you can use the MakeEdgesWave function above to create plausible edge waves.

Plotting 1D X, Y and Z Waves With Gridded XY Data

In this case we have 1D X, Y and Z waves of equal length that define a set of points in XYZ space. The X and Y waves constitute an evenly-spaced sampling grid though the spacing in X may be different from the spacing in Y.

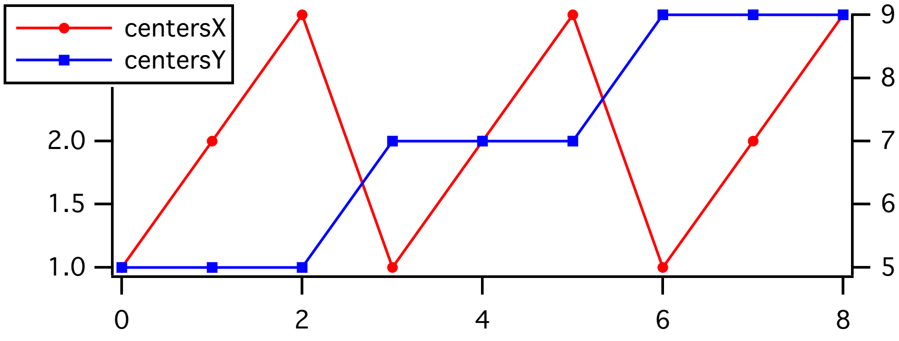
A good way to display such data is to create a scatter plot with color set as a function of the Z data. See **Setting Trace Properties from an Auxiliary (Z) Wave** on page II-298.

It is also possible to transform your data so it can be plotted as an image, as described under **Plotting a 2D Z Wave With 1D X and Y Center Data**. To do this you must convert your 1D Z wave into a 2D matrix wave and then convert your X and Y waves to contain the horizontal an vertical centers of your pixels.

For example, we start with this X, Y and Z data:

```
Make/O centersX = {1,2,3,1,2,3,1,2,3}
Make/O centersY = {5,5,5,7,7,7,9,9,9}
Make/O zData = {1,2,3,4,5,6,7,8,9}
```

If we display the X and Y data in a graph we can see that the X and Y waves exhibit repeating patterns:



To display this as an image, we transform the data so that the Z wave becomes a 2D matrix representing pixel values and the X and Y waves describe the centers of the rows and columns of pixels:

```
Redimension/N=(3,3) zData
Make/O/N=3 xCenterLocs = centersX[p]      // 1, 2, 3
Make/O/N=3 yCenterLocs = centersY[p*3]    // 5, 7, 9
```

We now have data as described under **Plotting a 2D Z Wave With 1D X and Y Center Data** on page II-389.

Plotting 1D X, Y and Z Waves With Non-Gridded XY Data

In this case you have 1D X, Y and Z waves of equal length that define a set of points in XYZ space. The X and Y waves do not constitute a grid, so the method of the previous section will not work.

A 2D scatter plot is a good way to graphically represent such data:

```
Make/O/N=20 xWave=enoise(4),yWave=enoise(5),zWave=enoise(6)  // Random points
Display yWave vs xWave
ModifyGraph mode=3,marker=19
ModifyGraph zColor(yWave)={zWave,*,*,Rainbow,0}
```

Although the data does not represent a proper image, you may want to display it as an image instead of a scatter plot. You can use the **ImageFromXYZ** operation to create a matrix wave corresponding to your XYZ data. The matrix wave can then be plotted as a simple image plot.

You can also Voronoi interpolation to create a matrix wave from the XYZ data:

```
Concatenate/O {xWave,yWave,zWave}, tripletWave
ImageInterpolate/S={-5,0.1,5,-5,0.1,5} voronoi tripletWave
AppendImage M_InterpolatedImage
```

Note that the algorithm for Voronoi interpolation is computationally expensive so it may not be practical for very large waves. See also **Loess** on page V-515 and **ImageInterpolate** on page V-382 kriging as alternative approaches for generating a smooth surface from unordered scatter data.

Additional options for displaying this type of data as a 3D surface are described under "Scatter Plots" in the "Visualization.ihf" help file and in the video tutorial "Creating a Surface Plot from Scatter Data" at http://www.youtube.com/watch?v=kggo0B43n_c.

Image Orientation

By default, the AppendImage operation draws increasing Y values (matrix column indices) upward, and increasing X (matrix row indices) to the right. Most image formats expect Y to increase downward. As a result, if you create an image plot using

```
Display; AppendImage <image wave>
```

your plot appears upside down.

You can flip an image vertically by reversing the Y axis, and horizontally by reversing the X axis, using the Axis Range tab in the Modify Axes dialog:

You can also flip the image vertically by reversing the Y scaling of the image wave.

A simpler alternative is to use NewImage instead of AppendImage. You can do this in the New Image Plot dialog by checking the “Use NewImage command” checkbox. NewImage automatically reverses the left axes.

Image Rectangle Aspect Ratio

By default, Igor does not make the image rectangles square. Use the Modify Graph dialog (in the Graph menu) to correct this by choosing Plan as the graph’s width mode. You can use the Plan height mode to accomplish the same result.

If DimDelta(*imageWave*,0) does not equal DimDelta(*imageWave*,1), you will need to enter the ratio (or inverse ratio) of these two values in the Plan width or height:

```
SetScale/P x 0,3,"", mat2dImage
SetScale/P y 0,1,"", mat2dImage
ModifyGraph width=0, height={Plan,3,left,bottom}
// or
ModifyGraph height=0, width={Plan,1/3,bottom,left}
```

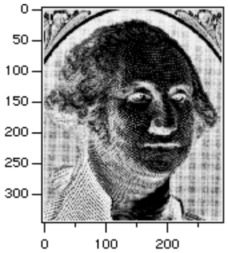
Do not use the Aspect width or height modes; they make the entire image plot square even if it shouldn’t be.

Plan mode ensures the image rectangles are square, but it allows them to be of any size. If you want each image rectangle to be a single point in width and height, use the per Unit width and per Unit height modes. With point X and Y scaling of an image matrix, use one point per unit:

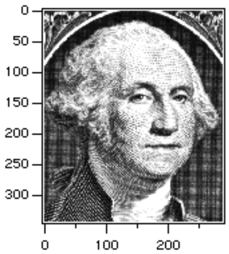
You can also flip an image along its diagonal by setting the Swap XY checkbox.

Image Polarity

Sometimes the image’s pixel values are inverted, too. False color images can be inverted by reversing the color table. Select the Reverse Colors checkbox in the Modify Image Appearance dialog. See **Image Color Tables** on page II-392. To reverse the colors in an index color plot is harder: the rows of the color index wave must be reversed.



After SetAxis/A/R left
ModifyGraph width={Plan,1,bottom,left}



After reversing
the Grays color table

Image Color Tables

In a false color plot, the data values in the 2D image wave are normally linearly mapped into a table of colors containing a set of colors that lets the viewer easily identify the data values. The data values can be logarithmically mapped by using the `ModifyImage log=1` option, which is useful when they span multiple orders of magnitude.

There are many built-in color tables you can use with false color images. Also, you can create your own color table waves - see **Color Table Waves** on page II-399.

The **CTabList** returns a list of all built-in color table names. You can create a color index wave or a color table wave from any built-in color table using **ColorTab2Wave**.

The ColorsMarkersLinesPatterns example Igor experiment, in “Igor Pro Folder:Examples:Feature Demos 2”, demonstrates all built-in color tables. These color tables are summarized in the section **Color Table Details** on page II-395.

Image Color Table Ranges

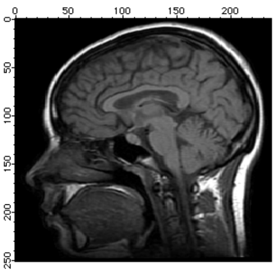
The range of data values that maps into the range of colors in the table can be set either manually or automatically using the Modify Image Appearance dialog.

When you choose to autoscale the first or last color, Igor examines the data in your image wave and uses the minimum or maximum data value found.

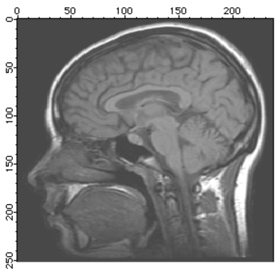
By changing the “First Color at Z=” and “Last Color at Z=” values you can examine subtle features in your data.

For example, when using the Grays color table, you can lighten the image by assigning the First Color (which is black) to a number lower than the image minimum value. This maps a lighter color to the minimum image value. To darken the maximum image values, assign the Last Color to a number higher than the image maximum value, mapping a darker color to the maximum image value.

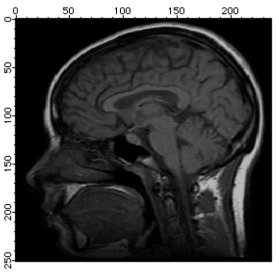
You can adjust these settings interactively by choosing Image→Image Range Adjustment.



`ctab = {0,255,Grays}`



`ctab = {-100,255,Grays}`



`ctab = {0,355,Grays}`

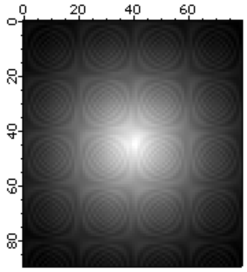
Data values greater than the range maximum are given the last color in the color table, or they can all be assigned to a single color or made transparent. Similarly, data values less than the range minimum are given the first color in the color table, or they can all be assigned to a single color (possibly different from the max color), or made transparent.

Example: Overlaying Data on a Background Image

By setting the image range to render small values transparent, you can see the underlying image in those locations, which helps visualize where the nontransparent values are located with reference to a background image. Here’s a fake weather radar example.

First, we create some “land” to serve as a background image:

```
Make/O/N=(80,90) landWave
landWave = 1-sqrt((x-40)*(x-40)+(y-45)*(y-45))/sqrt(40*40+45*45)
landWave = 7000*landWave*landWave
landWave += 200*sin((x-60)*(y-60)*pi/10)
landWave += 40*(sin((x-60)*pi/5)+sin((y-60)*pi/5))
NewImage landWave
```

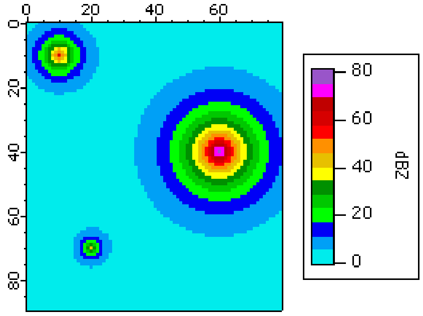


Then we create some “weather” radar data ranging from about 0 to 80 dBZ:

```
Duplicate/O landWave overlayWeather                        // "weather" radar values
overlayWeather=60*exp(-(sqrt((x-10)*(x-10)+(y-10)*(y-10))/5))      // storm 1
overlayWeather+=80*exp(-(sqrt((x-60)*(x-60)+(y-40)*(y-40)))/10)    // storm 2
overlayWeather+=40*exp(-(sqrt((x-20)*(x-20)+(y-70)*(y-70)))/3)     // storm 3
SetScale d, 0, 0, "dBZ", overlayWeather
```

We append the overlayWeather wave using the same axes as the landWave to overlay the images. With the default color table range, the landWave is totally obscured:

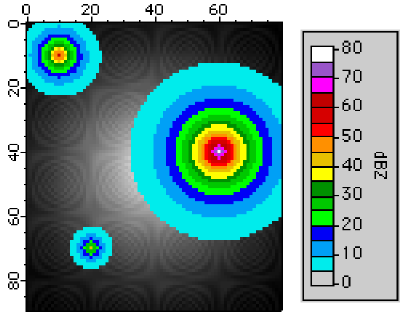
```
AppendImage/T overlayWeather
ModifyImage overlayWeather ctab= {*,*,dBZ14,0}
// Show the image's data range with a ColorScale
ModifyGraph width={Plan,1,top,left}, margin(right)=100
ColorScale/N=text0/X=107.50/Y=0.00 image=overlayWeather
```



We calibrate the image plot colors to National Weather Service values for precipitation mode by selecting the dBZ14color table for data values ranging from 5 to 75, where values below 5 are transparent and values above 75 are white:

We modify the ColorScale to show a range larger than the color table values (0-80):

```
ColorScale/C/N=text0 colorBoxesFrame=1,heightPct=90,nticks=10
ColorScale/C/N=text0/B=(52428,52428,52428) axisRange={0,80},tickLen=3.00
```



Color Table Ranges - Lookup Table (Gamma)

Normally the range of data values and the range of colors are linearly related or logarithmically related if the `ModifyImage log` parameter is set to 1. You can also cause the mapping to be nonlinear by specifying a lookup (or "gamma") wave, as described in the next example.

Example: Using a Lookup for Advanced Color/Contrast Effects

The **ModifyImage** operation (see page V-635) with the lookup parameter specifies a 1D wave that modifies the mapping of scaled Z values into the current color table. Values in the lookup wave should range from 0.0 to 1.0. A linear ramp from 0 to 1 would have no effect while a ramp from 1 to 0 would reverse the colormap. Used to apply gamma correction to grayscale images or for special effects.

```
Make luWave=0.5*(1+sin(x/30))
Make /n=(50,50) simpleImage=x*y
NewImage simpleImage
ModifyImage simpleImage ctab= {*,*,Rainbow,0}

// After inspecting the simple image, apply the lookup:
ModifyImage simpleImage lookup=luWave
```

Specialized Color Tables

Some of the color tables are designed for specific uses and specific numeric ranges.

The BlackBody color table shows the color of a heated "black body", though not the brightness of that body, over the temperature range of 1,000 to 10,000 K.

The Spectrum color table is designed to show the color corresponding to the wavelength of visible light as measured in nanometers over the range of 380 to 780 nm.

The SpectrumBlack color table does the same thing, but over the range of 355 to 830 nm. The fading to black is an attempt to indicate that the human eye loses the ability to perceive colors at the range extremities.

The GreenMagenta16, EOSOrangeBlue11, EOSSpectral11, dBZ14, and dBZ21 tables are designed to represent discrete levels in weather-related images, such as radar reflectivity measures of precipitation and wind velocity and discrete levels for geophysics applications.

The LandAndSea, Relief, PastelsMap, and SeaLandAndFire color tables all have a sharp color transition which is intended to denote sea level. The LandAndSea and Relief tables have this transition at 50% of the range. You can put this transition at a value of 0 by setting the minimum value to the negative of the maximum value:

```
ModifyImage imageName, ctab={-1000,1000,LandAndSea,0}     // image plot
ColorScale/C/N=scale0 ctab={-1000,1000,LandAndSea,0}      // colorscale
```

The PastelsMap table has this transition at 2/3 of the range. You can put this transition at a value of 0 by setting the minimum value to twice the negative of the maximum value:

```
ModifyImage imageName, ctab={-2000,1000,PastelsMap,0}     // image plot
ColorScale/C/N=scale0  ctab={-2000,1000,PastelsMap,0}     // colorscale
```

This principle can be extended to the other color tables to position a specific color to a desired value. Some trial-and-error is to be expected.

The BlackBody, Spectrum, and SpectrumBlack color tables are based on algorithms from the Color Science web site:

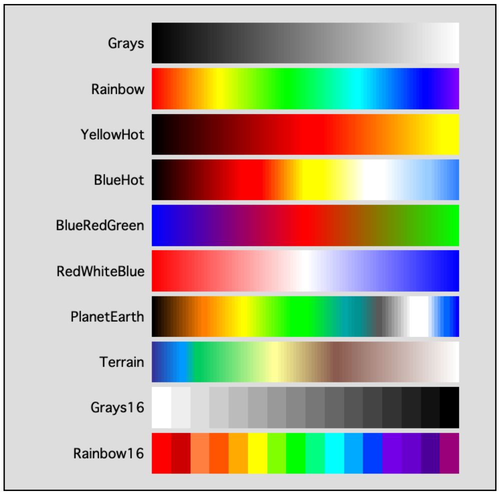
<`http://www.physics.sfasu.edu/astro/color.html`>.

Color Table Details

The built-in color tables can be grouped into several categories.

Igor Pro 4-Compatible Color Tables

Igor Pro 4 supported 10 built-in color tables: Grays, Rainbow, YellowHot, BlueHot, BlueRedGreen, RedWhiteBlue, PlanetEarth, Terrain, Grays16, and Rainbow16. These color tables have 100 color levels except for Grays16 and Rainbow16, which only have 16 levels.



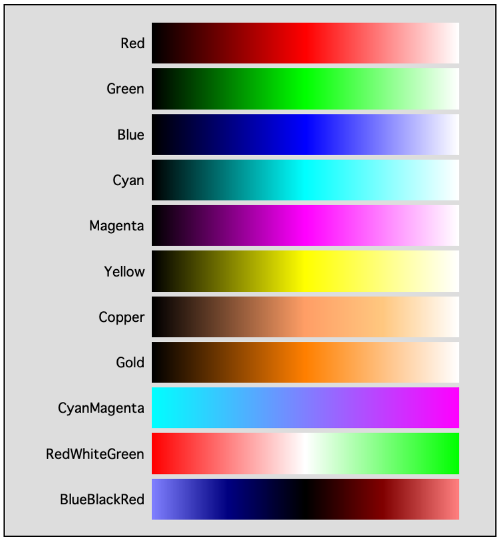
Igor Pro 5-Compatible Color Tables

Igor Pro 5 added 256-color versions of the eight 100-level color tables in Igor Pro 4 (Grays256, Rainbow256, etc.), new gradient color tables, and new special-purpose color tables.

Igor Pro 5 Gradient Color Tables

These are 256-color transitions between two or three colors.

Color Table Name	Colors	Notes
Red	256	Black → red → white.
Green	256	Black → green → white.
Blue	256	Black → blue → white.
Cyan	256	Black → cyan → white.
Magenta	256	Black → magenta → white.
Yellow	256	Black → yellow → white.
Copper	256	Black → copper → white.
Gold	256	Black → gold → white.
CyanMagenta	256	
RedWhiteGreen	256	
BlueBlackRed	256	

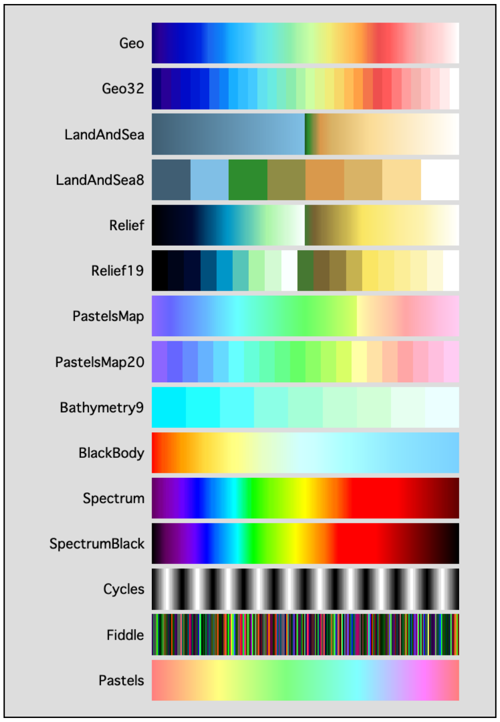


Igor Pro 5 Special-Purpose Color Tables

The special purpose color tables are ones that will find use for particular needs, such as coloring a digital elevation model (DEM) of topography or for spectroscopy. These color tables can have any number of color entries.

The following table summarizes the various special-purpose color tables.

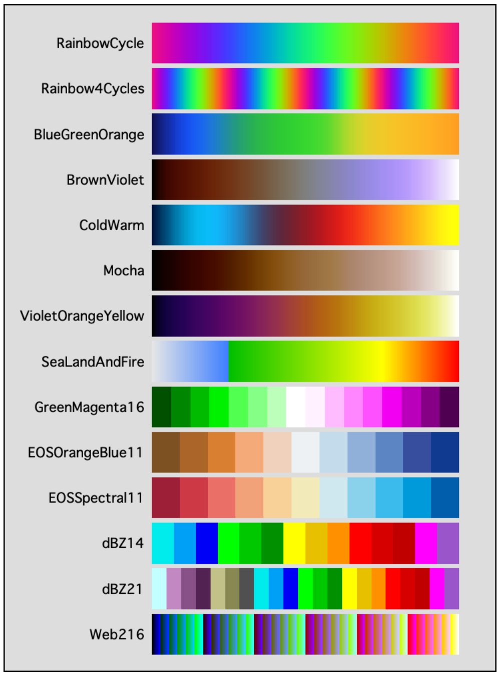
Color Table Name	Colors	Notes
Geo	256	Popular mapping color table for elevations. Sea level is around 50%.
Geo32	32	Quantized to classify elevations. Sea level is around 50%.
LandAndSea	255	Rapid color changes above sea level, which is at exactly 50%. Ocean depths are blue-gray.
LandAndSea8	8	Quantized, sea level is at about 22%.
Relief	255	Slower color changes above sea level, which is at exactly 50%. Ocean depths are black.
Relief19	19	Quantized, sea level is at about 47.5%.
PastelsMap	301	Desaturated rainbow-like colors, having a sharp green→yellow color change at sea level, which is around 66.67%. Ocean depths are faded purple.
PastelsMap20	20	Quantized. Sea level is at about 66.67%.
Bathymetry9	9	Colors for ocean depths. Sea level is at 100%.
BlackBody	181	Red → Yellow → Blue colors calibrated to black body radiation colors (neglecting intensity). The color table range is from 1,000 K to 10,000 K. Each color table entry represents a 50 K interval.
Spectrum	201	Rainbow-like colors calibrated to the visible spectrum when the color table range is set from 380 to 780 nm (wavelength). Each color table entry represents 2nm. Colors do not completely fade to black at the ends of the color table.
SpectrumBlack	476	Rainbow-like colors calibrated to the visible spectrum when the color table range is set from 355 to 830 nm (wavelength). Each color table entry represents 1 nm. Colors fade to black at the ends of the color table.
Cycles	201	Ten grayscale cycles from 0 to 100% to 0%.
Fiddle	254	Some randomized colors for “fiddling” with an image to detect faint details in the image.
Pastels	256	Desaturated Rainbow.



Igor Pro 6-Compatible Color Tables

Igor Pro 6 added 14 new color tables.

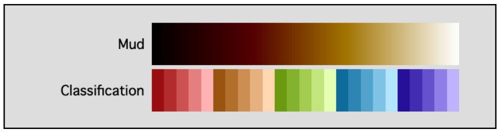
Color Table Name	Colors	Notes
RainbowCycle	360	Red, green, blue vary sinusoidally, each 120 degrees (120 values) out of phase. The first and last colors are identical.
Rainbow4Cycles	360	4 cycles with one quarter of the angular resolution.
BlueGreenOrange	300	Three-color gradient.
BrownViolet	300	Two-color gradient.
ColdWarm	300	Multicolor gradient for temperature.
Mocha	300	Two-color gradient.
VioletOrangeYellow	300	Multicolor gradient for temperature.
SeaLandAndFire	256	Another topographic table. Sea level is at 25%.
GreenMagenta16	16	Similar to the 14-color National Weather Service Motion color tables (base velocity or storm relative values), but friendly to red-green colorblind people.
EOSOrangeBlue11	11	Colors for diverging data (friendly to red-green colorblind people).
EOSSpectral11	11	Modified spectral colors (friendly to red-green colorblind people).
dBZ14	14	National Weather Service Reflectivity (radar) colors for Clear Air (-28 to +24 dBZ) or Precipitation (5 to 70 dBZ) mode.
dBZ21	21	National Weather Service Reflectivity (radar) colors for combined Clear Air and Precipitation mode (-30 to 70 dBZ).
Web216	216	The 216 “web-safe” colors, provides a wide selection of standard colors in a single color table. Intended for trace f(z) coloring using the ModifyGraph zColor parameter.



Igor Pro 6.2-Compatible Color Tables

Igor Pro 6.2 added 2 new color tables:

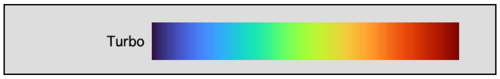
Color Table Name	Colors	Notes
Mud	256	Dark brown to white, without the pink cast of the Mocha color table. For Veeco atomic force microscopes.
Classification	25	5 hues for classification, 5 saturations for variations within each class.



Igor Pro 9-Compatible Color Tables

Igor Pro 9 added one new color table:

Color Table Name	Colors	Notes
Turbo	256	A reversed Rainbow-like color table with smoother transitions to avoid overly emphasizing features only because they map to a certain color. Designed by Anton Mikhailov, Senior Software Engineer, Daydream. See https://ai.googleblog.com/2019/08/turbo-improved-rainbow-colormap-for.html



Color Table Waves

You can use color index waves (see **Indexed Color Details** on page II-400) as if they were color tables.

With a color index wave, the image wave data value is used as an X index into the color index wave to select the color for a given point. The resultant color depends on the data value and the X scaling of the color index wave.

With a color table wave, the image wave's full range of data values, or a range that you explicitly specify, is mapped to the entire color table wave. The resultant color depends on the data value and the operative range only, not on the color table wave's X scaling.

A trivial way to generate a color table wave is to call the ColorTab2Wave operation which creates a 3 column RGB wave named M_Colors, where column 0 is the red component, column 1 is green, and column 2 is blue, and each value is between 0 (dark), and 65535 (bright).

With a 3 column RGB color table wave, all colors are opaque. You can add a fourth column to control transparency, making it an RGBA wave. The fourth column of an RGBA wave represents "alpha", where 0 is fully transparent and 65535 is fully opaque.

Many color table waves are included in the Color Tables folder in the Igor Pro folder, along with a help file that describes them.

The syntaxes for using color table waves for image plots, contour plots, graph traces, and colorscales vary and are detailed in their respective commands. See **ModifyImage** (ctab keyword), **ModifyContour** (ctabFill and ctabLines keywords), **ModifyGraph (traces)** (zColor keyword) and **ColorScale** (ctab keyword).

See the "ColorTableWavesHelp.ihf" help file in the "Color Tables" folder of the Igor Pro folder to preview and load color table waves that ship with Igor Pro.

Indexed Color Details

An indexed color plot uses a 2D image wave, or a layer of a 3D or 4D wave, and a color index wave. The image wave data value is used as an X index into the color index wave to select the color for a given image rectangle. The resulting color depends on the data value and the X scaling of the color index wave.

A color index wave is a 2D RGB or RGBA wave. An RGB wave has three columns and each row contains a set of red, green, and blue values that range from 0 (zero intensity) to 65535 (full intensity). An RGBA wave has three color columns plus an alpha column whose values range from 0 (fully transparent) to 65535 (fully opaque).

Linear Indexed Color

For the normal linear indexed color, Igor finds the color for a particular image data value by choosing the row in the color index wave whose X index corresponds to the image data value. Igor converts the image data value zImageValue into a row number colorIndexWaveRow using the following computation:

```
colorIndexWaveRow = floor(nRows*(zImageValue-xMin)/xRangeInclusive)
```

where,

```
nRows = DimSize(colorIndexWave,0)
xMin = DimOffset(colorIndexWave,0)
xRangeInclusive = (nRows-1) * DimDelta(colorIndexWave,0)
```

If colorIndexWaveRow exceeds the row range, then the Before First Color and After Last Color settings are applied.

By setting the X scaling of the color index wave, you can control how Igor maps the image data value to a color. This is similar to setting the First Color at Z= and Last Color at Z= values for a color table.

Logarithmic Indexed Color

For logarithmic indexed color (the ModifyImage log parameter is set to 1), colors are mapped using the log(x scaling) and log(image z) values this way:

```
colorIndexWaveRow = floor(nRows*(log(zImageValue)-log(xMin))/(log(xmax)-log(xMin)))
```

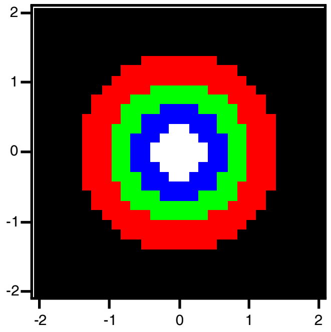
where,

```
nRows = DimSize(colorIndexWave,0)
xMin = DimOffset(colorIndexWave,0)
xMax = xMin + (nRows-1) * DimDelta(colorIndexWave,0)
```

Displaying image data in log mode is slower than in linear mode.

Example: Point-Scaled Color Index Wave

```
// Create a point-scaled, unsigned 16-bit integer color index wave
Make/O/W/U/N=(1,3) shortindex          // initially 1 row; more will be added
shortindex[0][]= {{0},{0},{0}}               // black in first row
shortindex[1][]= {{65535},{0},{0}}           // red in new row
shortindex[2][]= {{0},{65535},{0}}           // green in new row
shortindex[3][]= {{0},{0},{65535}}           // blue in new row
shortindex[4][]= {{65535},{65535},{65535}}   // white in new row

// Generate sample data and display it using the color index wave
Make/O/N=(30,30)/B/U expmat   // /B/U makes unsigned byte image
SetScale/I x,-2,2,"" expmat
SetScale/I y,-2,2,"" expmat
expmat= 4*exp(-(x^2+y^2))      // test image ranges from 0 to 4
Display;AppendImage expmat
ModifyImage expmat cindex=shortindex
```



Direct Color Details

Direct color images use a 3D RGB wave with 3 color planes containing absolute values for red, green and blue or a 3D RGBA wave that adds an alpha plane. Generally, direct color waves are either unsigned 8 bit integers or unsigned 16 bit integers.

For 8-bit integer waves, 0 represents zero intensity and 255 represents full intensity. For alpha, 0 represents fully transparent and 255 represents fully opaque.

For all other number types, 0 represents zero intensity but 65535 represents full intensity. For alpha, 0 represents fully transparent and 65535 represents fully opaque. Out-of-range values are clipped to the limits.

Try the following example, executing each line one at a time:

```
Make/O/B/U/N=(40,40,3) matrgb
NewImage matrgb
matrgb[][][0]= 127*(1+sin(x/8)*sin(y/8))  // Specify red,   0-255
matrgb[][][1]= 127*(1+sin(x/7)*sin(y/6))  // Specify green, 0-255
matrgb[][][2]= 127*(1+sin(x/6)*sin(y/4))  // Specify blue,  0-255
```

```
// Switch to floating point, image turns black
Redimension/S matrgb

// Scale floating point to 0..65535 range
matrgb *= 256
```

Because the appearance of a direct color image is completely determined by the image data, the Modify Image Appearance dialog has no effect on direct color images, and the dialog appears blank.

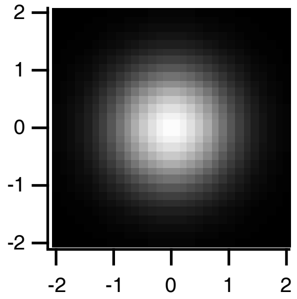
Creating Color Legends

You can create a color legend using a color scale annotation. For background information, see **Legends** on page III-42 and **Color Scales** on page III-47 sections.

We will demonstrate with a simple image plot:

```
Make/O/N=(30,30) expmat
SetScale/I x,-2,2,"" expmat; SetScale/I y,-2,2,"" expmat
expmat= exp(-(x^2+y^2))        // data ranges from 0 to 1
Display;AppendImage expmat     // by default, left and bottom axes
ModifyGraph width={Plan,1,bottom,left},mirror=0
```

This creates the following image, using the autoscaled Grays color table:



Choose Graph→Add Annotation to display the Add Annotation dialog.

Choose “ColorScale” from the Annotation pop-up menu.

Switch to the Frame tab and set the Color Bar Frame Thickness to 0 and the Annotation Frame to None.

Switch to the Position tab, check the Exterior checkbox, and set the Anchor to Right Center.

Click Do It. Igor executes:

```
ColorScale/C/N=text0/F=0/A=RC/E image=expmat,frame=0.00
```

This generates the following image plot:

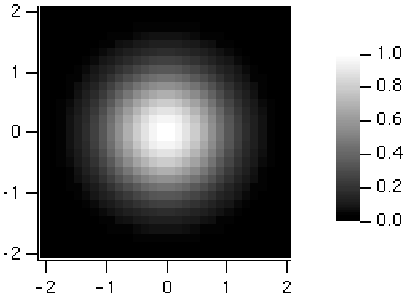


Image Instance Names

Igor identifies an image plot by the name of the wave providing Z values (the wave selected in the Z Wave list of the Image Plot dialogs). This “image instance name” is used in commands that modify the image plot.

In this example the image instance name is “zw”:

```
Display; AppendImage zw             // new image plot
ModifyImage zw ctab={*,*,BlueHot}   // change color table
```

In the unusual case that a graph contains two image plots of the same data, to show different subranges of the data side-by-side,for example, an instance number must be appended to the name to modify the second plot:

```
Display; AppendImage zw; AppendImage/R/T zw  // two image plots
ModifyImage zw ctab={*,*,RedWhiteBlue}       // change first plot
ModifyImage zw#1 ctab={*,*,BlueHot}          // change second plot
```

The Modify Image Appearance dialog generates the correct image instance name automatically. Image instance names work much the same way wave instance names for traces in a graph do. See **Instance Notation** on page IV-20.

The **ImageNameList** function (see page V-397) returns a string list of image instance names. Each name corresponds to one image plot in the graph. The **ImageInfo** function (see page V-380) returns information about a particular named image plot.

ImageNameList returns strings, but ModifyImage uses names. The $ operator turns a string into a name. For example:

```
Function SetFirstImageToRainbow(graphName)
   String graphName
   String imageInstNames = ImageNameList(graphName, ";")
   String firstImageName = StringFromList(0,imageInstNames) // Name in a string
   if (strlen(firstImageName) > 0)
      // $ converts string to name
      ModifyImage/W=$graphName $firstImageName ctab={,,Rainbow}
   endif
End
```

Image Preferences

You can change the default appearance of image plots by capturing preferences from a prototype graph containing image plots. Create a graph containing an image plot with the settings you use most often. Then choose Capture Graph Prefs from the Graph menu. Select the Image Plots category, and click Capture Prefs.

Preferences are normally in effect only for *manual* operations, not for automatic operations from Igor procedures. Preferences are discussed in more detail in Chapter III-18, **Preferences**.

The Image Plots category includes both Image Appearance settings and axis settings.

Image Appearance Preferences

The captured Image Appearance settings are automatically applied to an image plot when it is first created, provided preferences are turned on. They are also used to preset the Modify Image Appearance dialog when it is invoked as a subdialog of the New Image Plot dialog.

If you capture the Image Plot preferences from a graph with more than one image plot, the first image plot appended to a graph gets the settings from the image first appended to the prototype graph. The second image plot appended to a graph gets the settings from the second image plot appended to the prototype graph, etc. This is similar to the way XY plot wave styles work.

Image Axis Preferences

Only axes used by the image plot have their settings captured. Axes used solely for an XY, category, or contour plot are ignored.

The image axis preferences are applied only when axes having the same name as the captured axis are created by an AppendImage command. If the axes existed before AppendImage is executed, they are not affected by the image axis preferences.

The names of captured image axes are listed in the X Axis and Y Axis pop-up menus of the New Image Plot and Append Image Plot dialogs. This is similar to the way XY plot axis preferences work.

For example, suppose you capture preferences for an image plot using axes named “myRightAxis” and “myTopAxis”. These names will appear in the X Axis and Y Axis pop-up menus in image plot dialogs.

If you choose them in the New Image Plot dialog and click Do It, a graph will be created containing *newly-created* axes named “myRightAxis” and “myTopAxis” and having the axis settings you captured.

If you have a graph which already uses axes named “myRightAxis” and “myTopAxis” and choose these axes in the Append Image Plot dialog, the image will be appended to those axes, as usual, but no captured axis settings will be applied to these *already-existing* axes.

You can capture image axis settings for the standard left and bottom axes, and Igor will save these separately from left and bottom axis preferences captured for XY, category, and contour plots. Igor will use the image axis settings for AppendImage commands only.

How to Use Image Preferences

Here is our recommended strategy for using image preferences:

1. Create a new graph containing a single image plot. Use the axes you will normally use, even if they are left and bottom. You can use other axes, too (select New Axis in the New Image Plot and Append Image Plot dialogs).
2. Use the Modify Image Appearance, Modify Graph, and Modify Axis dialogs to make the image plot appear as you prefer.
3. Choose Capture Graph Prefs from the Graph menu. Select the Image Plots category, and click Capture Prefs.

Image Plot Shortcuts

Since image plots are drawn in a normal graph, all of the **Graph Shortcuts** (see page II-353) apply. Here we list those which apply specifically to image plots.

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To modify the appearance of the image plot as a whole	Control-click in the plot area of the graph and choose Modify Image from the pop-up menu.	Right-click in the plot area of the graph and choose Modify Image from the pop-up menu.

References

Light, Adam, and Patrick J. Bartlein, The End of the Rainbow? Color Schemes for Improved Data Graphics, *Eos*, *85*, 385-391, 2004.

Wyszecki, Gunter, and W. S. Stiles, *Color Science: Concepts and Methods, Quantitative Data and Formula*, 628 pp., John Wiley & Sons, 1982.

Chapter

II-17

3D Graphics

Overview

Igor can create various kinds of 3D graphics including:

- **Surface Plots**
- **3D Scatter Plots**
- **3D Bar Plots**
- **Path Plots**
- **Ribbon Plots**
- **Isosurface Plots**
- **Voxelgram Plots**

Image Plots, **Contour Plots** and **Waterfall Plots** are considered 2D graphics and are discussed in other sections of the help.

Igor's 3D graphics tool is called "Gizmo". Most 3D graphics that you produce with Gizmo will be based on data stored in waves. It's important to understand what type of wave data is required for what type of 3D graphic, as explained in the following sections explain.

1D Waves

1D waves can not be used for 3D plots.

If you have three 1D waves that represent X, Y and Z coordinates which you want to display as a 3D plot, you must convert them into a triplet wave. For example:

```
Concatenate {xWave,yWave,zWave}, tripletWave
```

Now you can plot the triplet wave using one of the methods described below.

The conversion of three 1D waves into a triplet wave is appropriate when the data are not sampled on a rectangular grid. If you know that your data are sampled on a rectangular grid you should convert the wave that contains your Z data into a 2D wave using the Redimension operation and then proceed to plot the surface using the 2D wave. You can perform this conversion, for example, using the commands:

```
Duplicate/O zWave, zMatrixWave
Redimension/N=(numRows,numColumns) zMatrixWave
```

2D Waves

A 2D wave, sometimes called a "matrix of Z values", is an M-row by N-column wave where each element represents a scalar Z value. You can apply wave scaling (see **Waveform Model of Data** on page II-62) to associate an X value with each row and a Y value with each column.

2D waves can be displayed as 3D graphics in **Surface Plots** and **3D Bar Plots**.

(2D waves can also be displayed as 2D graphics in **Image Plots**, **Contour Plots**, and **Waterfall Plots**.)

Triplet Waves

A triplet wave is an M-row by 3-column wave containing an XYZ triplet in each row. The X value appears in the first column, the Y value in the second and the Z value in the third. A triplet wave is a 2D wave interpreted as containing X, Y and Z coordinates.

Triplet waves can be displayed as 3D graphics in **3D Scatter Plots**, **Surface Plots**, **Path Plots** and **Ribbon Plots**. In a surface plot the triplet wave defines triangles on a surface.

(Triplet waves can also be displayed as 2D graphics in **Contour Plots**.)

3D Waves

A 3D wave, sometimes called a "volume", is an M-row by N-column by L-layer wave where each element represents a scalar Z value. To be used in 3D graphics, 3D waves must contain at least two elements in each dimension.

3D waves can be displayed as 3D graphics in **Surface Plots**, **Isosurface Plots**, and **Voxelgram Plots**.

(3D waves can also be displayed as 2D graphics in **Image Plots** where a single layer of a 3D wave is displayed as an image.)

Gizmo Overview

Igor's 3D plotting tool is called "Gizmo".

Gizmo is based on OpenGL, an industry standard system for 3D graphics. Gizmo converts Igor data and commands into OpenGL data and instructions and displays the result in an Igor window called a "Gizmo window".

You create a Gizmo window by choosing Windows→New→3D Plot and then appending graphic objects using the Gizmo menu. You can do this without any knowledge of OpenGL. At this level you can create surface plots, scatter plots, path plots, voxelgrams, isosurface plots, 3D bar plots and 3D pie charts. Such objects are called "wave-based objects" or "data objects" to differentiate them from drawing objects, discussed next. After creating the basic plot you can modify various properties using Gizmo's "info" window.

Advanced users can also construct graphics using a set of 3D primitives including lines, triangles, quadrangles, cubes, spheres, cylinders, disks, tetrahedra and pie wedges. Such objects are called "drawing objects" to differentiate them from wave-based objects. If you apply proper scaling you can combine drawing objects and wave-based objects in the same Gizmo window.

Gizmo supports advanced OpenGL features such as lighting effects, shininess and textures. Advanced users who want to create sophisticated 3D graphics will benefit from some familiarity with 3D graphics in general and OpenGL in particular.

System Requirements

Much of Gizmo's operation depends on your computer's graphics hardware and its graphics driver software. We suggest running Gizmo on hardware that includes a dedicated graphics card with at least 512MB of VRAM. Gizmo should also work on computers with onboard graphics and shared memory but you will experience slower performance and may encounter errors when exporting graphics.

Hardware Compatibility

Gizmo graphics may be affected by the version of OpenGL that you are running. This depends on your graphics hardware, graphics driver version and graphics acceleration settings.

Gizmo Guided Tour

The tutorials in the following sections will give you a sense of Gizmo's basic capabilities, how you create basic Gizmo plots, and the type of data that you need for a given type of plot.

At various points in the tour you are instructed to execute Igor commands. The easiest way to do this is to open the “3D Graphics” help file using the Help→Help Windows submenu and do the tour from the help file rather than from this manual. Then you can execute the commands by selecting them in the help file and pressing Ctrl-Enter.

Gizmo 3D Scatter Plot Tour

In this tour we will create a triplet wave containing XYZ data which we will plot as a 3D scatter plot.

1. **Start a new experiment by choosing File→New Experiment.**

2. **To create a triplet wave containing XYX scatter data, execute:**

```
Make/O/N=(20,3) data = gnoise(5)
data[][2] = 2*data[p][0] - 3*data[p][1] + data[p][0]^2 + gnoise(0.05)
```

   This scatter data represents random locations in the XY plane with Z values that are approximately equal to a polynomial function in X and Y.

   As we see next, the quickest way to display this data in Gizmo is by right-click selecting "Gizmo Plot" in the Data Browser.

3. **Choose Data→Data Browser.**

   The Data Browser window appears.

4. **Right-click the data wave icon and choose New Gizmo Plot.**

   Igor created a Gizmo 3D scatter plot from the data wave in a new window named Gizmo0.

   It also created the Gizmo info window entitled "Gizmo0 Info".

   (If you don't see the "Gizmo0 Info" window, choose Gizmo→Show Info.)

   You can rotate the Gizmo scatter plot by dragging the contents of the Gizmo0 window and using the arrow keys. Feel free to play.

   ...

   That was entirely too easy and not very instructive so we will redo it without using the Data Browser shortcut.

5. **Start a new experiment by choosing File→New Experiment.**

6. **To create a triplet wave containing XYX scatter data, execute:**

```
Make/O/N=(20,3) data = gnoise(5)
data[][2] = 2*data[p][0] - 3*data[p][1] + data[p][0]^2 + gnoise(0.05)
```

7. **Choose Windows→New→3D Plot.**

   Notice from the history area of the command window that Igor has executed:

```
NewGizmo
```

   Igor also created an empty Gizmo0 window and the Gizmo0 Info window.

   (If you don't see the "Gizmo0 Info" window, choose Gizmo→Show Info.)

8. **Click the + icon at the bottom of the object list in the Gizmo0 Info window and choose Scatter.**

   Igor displays the Scatter Properties dialog.

9. **Choose data from the Scatter Wave menu.**

   This menu displays only triplet waves. If you want to create a 3D scatter plot, you must have a triplet wave.

   There are several additional options but we will leave them in their default states for now.

10. **Click Do It.**

    Igor created a 3D scatter object named scatter0 and added it to the object list in the info window. It is not yet visible in the Gizmo0 window because we have not yet added it to the display list.

11. **Drag the scatter0 object from the object list to the display list.**

    Spheres representing the XYZ data appear in the Gizmo0 window. Although the plot is by no means complete, you can click and drag the body of the Gizmo0 window to rotate the display.

12. **In the Gizmo0 Info window, click the "+" icon at the bottom of the object list and choose Axes.**

    The Axes Properties dialog appears.

13. **Click the Axis tab if it is not already selected.**

    The Axis Type pop-up menu should be set to Box and all of the axis checkboxes (X0, X1...Z2, Z3) should be checked.

14. **Click Do It.**

    Igor created an axes object named axes0 and added it to the object list in the info window. It is not yet visible in the Gizmo0 window because we have not yet added it to the display list.

15. **Drag the axes0 object from the object list to the display list.**

    You now have box axes around the scatter spheres.

16. **Double-click the axes0 object in either the object list or the display list.**

    The Axes Properties dialog reopens.

17. **Click the Ticks and Labels tab.**

18. **Select X0 from the Axis pop-up menu and check the Show Tick Marks and Show Numerical Labels checkboxes.**

19. **Select Y0 from the Axis pop-up menu and check the Show Tick Marks and Show Numerical Labels checkboxes.**

20. **Select Z0 from the Axis pop-up menu and check the Show Tick Marks and Show Numerical Labels checkboxes.**

21. **Click Do It.**

    You now have labeled tick marks for the X0, Y0 and Z0 axes.

    But which axis is which?

22. **Right-click the body of the Gizmo0 window and select Show Axis Cue.**

    Igor adds an axis cue that shows you which dimension is which.

    Rotate the display a bit to get a sense of the axis cue.

    You can also double click the axes0 object and select the Axis tab. The properties dialog displays the box axes. If you now hover with the mouse cursor over any axis you see a tooltip that identifies the axis.

23. **Click the close box of the Gizmo0 Info window.**

    The Gizmo0 Info window is hidden. It is usually of interest while you are constructing or tweaking a 3D plot and can be hidden when you just want to view the plot. You can make it visible at any time by choosing Gizmo→Show Info or by right-clicking the Gizmo0 window and choosing Show Info Window.

24. **Click the close box of the Gizmo0 window.**

    Igor displays the Close Window dialog asking if you want to save the window as a window recreation macro. This works the same as a graph recreation macro.

25. **Click the Save button to save the window recreation macro.**

    The recreation macro is saved in the main procedure window.

26. **Choose Windows→Procedure Windows→Procedure Window.**

    This displays the main procedure window containing the Gizmo0 recreation macro. You may need to scroll up to see the beginning of it which starts with:

    ```
    Window Gizmo0() : GizmoPlot
    ```

27. **Close the procedure window by clicking its close box.**

28. **Choose Windows→Other Macros→Gizmo0.**

    Igor executes the Gizmo0 recreation macro which recreates the Gizmo0 window.

29. **Choose File→Save Experiment and save the experiment as "Gizmo 3D Scatter Plot Tour.pxp".**

    This is just in case you want to revisit the tour later and is not strictly necessary.

At this point, you have completed the construction of a 3D scatter plot. As you may have noticed, there are many scatter plot and axis options that we did not explore. You can do that now, by double-clicking the scatter0 and axes0 icons in the Gizmo Info window, or you can leave that for later and continue with the next section of the tutorial.

Gizmo Surface Plot Tour

In this tour we will create a 2D wave containing Z values which we will plot as a surface plot.

1. **Start a new experiment by choosing File→New Experiment.**
2. **To create a 2D matrix of Z values, execute:**

   ```
   Make/O/N=(100,100) data2D = Gauss(x,50,10,y,50,15)
   ```

   This matrix data represents Z values on a regular grid.

   As you saw in the first tour, the quickest way to display this data in Gizmo is by right-clicking an appropriate wave in the Data Browser and selecting Gizmo Plot. The same shortcut works with a matrix of Z values. But we will do it the hard way as this will give you a better understanding of Gizmo.
3. **Choose Windows→New→3D Plot.**

   Notice from the history area of the command window that Igor has executed:

   ```
   NewGizmo
   ```

   Igor also created an empty Gizmo0 window and the Gizmo0 Info window.
4. **Click the + icon at the bottom of the object list in the Gizmo0 Info window and choose Surface.**

   Igor displays the Surface Properties dialog.
5. **Choose Matrix from the Source Wave Type pop-up menu and data2D from the Surface Wave pop-up menu.**

   There are several additional options but we will leave them in their default states for now.
6. **Click Do It.**

   Igor created a surface object named surface0 and added it to the object list in the info window. It is not yet visible in the Gizmo0 window because we have not yet added it to the display list.
7. **Drag the surface0 object from the object list to the display list.**

   The surface appears in the Gizmo0 window. You are now looking at it from the top.
8. **Using the mouse, rotate the surface in the Gizmo0 window to reorient it so you can see the side view of the Gaussian peak.**
9. **In the Gizmo info window, click the "+" icon at the bottom of the object list and choose Axes.**

   The Axes Properties dialog appears. Click the Axis tab if it is not already selected.
10. **Click the Axis tab if it is not already selected.**

    The Axis Type pop-up menu should be set to Box and all of the axis checkboxes (X0, X1...Z2, Z3) should be checked.
11. **1Click Do It.**

    Igor created an axis object named axes0 and added it to the object list in the info window. It is not yet visible in the Gizmo0 window because we have not yet added it to the display list.
12. **Drag the axes0 object from the object list to the display list.**

    You now have box axes around the surface plot.
13. **Double-click the axes0 object in the display list. Using the resulting Axes Properties dialog, turn on tick marks and tick mark labels for the X0, Y0 and Z0 axes.**

    This is the same as what we did in the preceding tour.

    Now we will add a colorscale annotation.
14. **Choose Gizmo→Add Annotation.**

    The Add Annotation dialog appears.
15. **From the Annotation pop-up menu in the top/left corner of the dialog, choose ColorScale.**
16. **Click the Position tab and set the Anchor pop-up menu to Right Center.**

17. **Click the ColorScale Main tab and set the following controls as indicated:**

    Color Table: Rainbow

    Axis Range/Top: 100

    Axis Range/Bottom: 0

18. **Click Do It.**

    Igor creates the colorscale annotation.

    Note that its tick mark labels don't agree with the Z axis tick mark labels in the surface plot. We need to set the colorscale range to match the range of the data. We will do this by re-executing the command that created the colorscale.

19. **Click on the last command in the history area of the command window and press Enter to copy it to the command line.**

    The command line should now show this:

    ```
    ColorScale/C/N=text0/M/A=RC ctab={0,100,Rainbow,0}
    ```

20. **Edit the command as shown next and press Enter to execute it:**

    ```
    ColorScale/C/N=text0/M/A=RC ctab={WaveMin(data2D),WaveMax(data2D),Rainbow,0}
    ```

    Now the colorscale tick mark labels agree with the Z axis tick mark labels in the surface plot.

    Next we will add an image plot beneath the surface plot.

21. **In the Gizmo info window, click the "+" icon at the bottom of the object list and choose Image.**

    The Gizmo Image Properties dialog appears.

22. **Set the Source Type to 2D Matrix of Z Values, and select data2D from the Source Wave pop-up menu.**

    We will use the same wave as the source for both the surface plot and the image.

23. **From the Intitial Orientation pop-up menu, choose XY Plane Z=0.**

24. **Uncheck all checkboxes except Translate and set the X, Y, and Z components of the translation to 0, 0, and -1 respectively.**

    The translation in the Z direction moves the image from the center of the display volume to the bottom of the display volume, placing it on the "floor" of the surface plot.

25. **Click Do It.**

    Igor created an image object named image0 and added it to the object list in the info window. It is not yet visible in the Gizmo0 window because we have not yet added it to the display list.

26. **Drag the image0 object from the object list to the display list.**

    You now have an image plot below the surface plot. You may need to rotate the display to see it.

27. **Double-click the image0 object in the object list, set the Z translation to -1.5, then click Do It.**

28. **This separates the image plot from the surface plot, making it easier to see.**

    The translation parameters are in +/-1 display volume units, not in the units of the axes. Display volume units are used in many instances, especially when dealing with drawing objects such as spheres and boxes.

29. **Choose File→Save Experiment and save the experiment as "Gizmo Surface Plot Tour.pxp".**

    This is just in case you want to revisit the tour later and is not strictly necessary.

Gizmo 3D Scatter Plot and Fitted Surface Tour

In this tour we will create a 3D scatter plot from a triplet wave, perform a curve fit, and append a surface showing how the curve fit output relates to the original scatter data.

1. **Start a new experiment by choosing File→New Experiment.**
2. **To create a triplet wave containing XYX scatter data, execute:**

```
Make/O/N=(20,3) data = gnoise(5)
data[][2] = 2*data[p][0] - 3*data[p][1] + data[p][0]^2 + gnoise(0.05)
```

This scatter data represents random locations in the XY plane with Z values that are approximately equal to a polynomial function in X and Y.

3. **Choose Data→Data Browser.**

   The quickest way to display this data in Gizmo is by right-click selecting "Gizmo Plot" in the Data Browser.

4. **Right-click the data wave icon and choose New Gizmo Plot.**

   Igor created a Gizmo 3D scatter plot from the data wave in a new window named Gizmo0.

5. **In the command line, execute:**

   ```
   CurveFit/Q poly2d 2, data[][2]/X=data[][0,1] /D
   ```

   Igor performs a 2D polynomial curve fit and produces output waves and variables. The main output is in the wave fit_data.

6. **Close the Data Browser window.**

   This is just to reduce clutter on your screen.

   Next we will now add a surface to the Gizmo plot.

7. **Click the + icon at the bottom of the object list in the Gizmo0 Info window and choose Surface.**

   Igor displays the Surface Properties dialog.

8. **Choose Matrix from the Source Wave Type pop-up menu and fit_data from the Surface Wave pop-up menu.**

   There are several additional options but we will leave them in their default states for now.

9. **Click Do It.**

   Igor created a surface object named surface0 and added it to the object list in the info window. It is not yet visible in the Gizmo0 window because we have not yet added it to the display list.

10. **Drag the surface0 object from the object list to the display list.**

    The surface appears in the Gizmo0 window and appears to fit the scatter objects pretty well.

11. **Using the mouse, rotate the contents of the Gizmo plot to inspect the fit from various angles.**

12. **Choose File→Save Experiment and save the experiment as "Gizmo 3D Scatter Plot and Fitted Surface Tour.pxp".**

    This is just in case you want to revisit the tour later and is not strictly necessary.

Gizmo Surface Using Voronoi Interpolation Tour

We have already seen how to create a surface plot from a 2D matrix of Z values. In this tour we illustrate the how to plot a 3D surface representation of XYZ scatter data. The process involves triangulation of the XYZ data using Voronoi interpolation.

1. **Start a new experiment by choosing File→New Experiment.**

2. **To create a triplet wave containing XYZ scatter data, execute:**

   ```
   Make/O/N=(20,3) data = enoise(5)
   data[][2] = 2*data[p][0] - 3*data[p][1] + data[p][0]^2 + gnoise(0.05)
   ```

3. **Choose Data→Data Browser.**

4. **Right-click the data wave icon and choose New Gizmo Plot.**

   Igor created a Gizmo 3D scatter plot from the data wave in a new window named Gizmo0.

   Next we will triangulate the scatter data using Voronoi interpolation.

5. **Execute this in the command line:**

```
ImageInterpolate/S={-5,0.1,5,-5,0.1,5}/CMSH Voronoi data
```

The Voronoi interpolation created two waves: M_ScatterMesh and M_InterpolatedImage. M_ScatterMesh consists of a series of XYZ coordinates that define polygons in 3D space which fit the scatter data. We will use M_ScatterMesh to append a surface to the 3D plot.

6. **Click the + icon at the bottom of the object list in the Gizmo0 Info window and choose Surface.**

Igor displays the Surface Properties dialog.

7. **Choose Triangles from the Source Wave Type pop-up menu and M_ScatterMesh from the Surface Wave pop-up menu.**

There are several additional options but we will leave them in their default states for now.

8. **Click Do It.**

Igor created a surface object named surface0 and added it to the object list in the info window. It is not yet visible in the Gizmo0 window because we have not yet added it to the display list.

9. **Drag the surface0 object from the object list to the display list.**

The surface appears in the Gizmo0 window.

10. **Using the mouse, rotate the contents of the Gizmo plot to inspect the fit from various angles.**

The surface fits the scatter objects pretty well.

11. **Double-click the surface0 object in the display list, click the Grid Lines and Points tab, check the Draw Grid Lines checkbox, and click Do It.**

This shows the polygons created by Voronoi interpolation and represented by the M_ScatterMesh wave.

12. **Clean up by executing:**

```
KillWaves M_InterpolatedImage
Rename M_ScatterMesh, VoronoiMesh
```

It's a good idea to rename waves that Igor creates with default wave names so that, if you later execute another command that uses the same default wave name, you will not inadvertently overwrite data. Also we don't need the M_InterpolatedImage wave.

13. **Choose File→Save Experiment and save the experiment as "Gizmo Surface Using Voronoi Interpolation Tour.pxp".**

This is just in case you want to revisit the tour later and is not strictly necessary.

That concludes the Gizmo guided tour. There are more examples below. Also choose File→Example Experiments→Visualization for sample experiments.

Gizmo Windows

For each 3D plot, Gizmo creates a display window and its associated info window. The display window presents a rotatable representation of your 3D objects. You use the info window to control which objects are displayed, the order in which they are drawn, and their properties. You can hide both windows to reduce clutter when you do not need them. You can also kill and recreate Gizmo windows like you kill and recreate graphs.

You can create any number of Gizmo display windows. Keeping multiple Gizmo display windows open has some drawbacks. Even inactive and hidden Gizmo display windows consume graphics resources that could otherwise be used for the active Gizmo display window. Also, in some laptop computers you may be able to reduce power consumption by closing Gizmo display windows that include rotating objects. Depending on your hardware, saving unused Gizmo display windows as recreation macros may be beneficial.

For brevity, we sometimes use the term "Gizmo window" to refer to the Gizmo display window. We use "Gizmo info window" or "info window" to refer to the Gizmo info window.

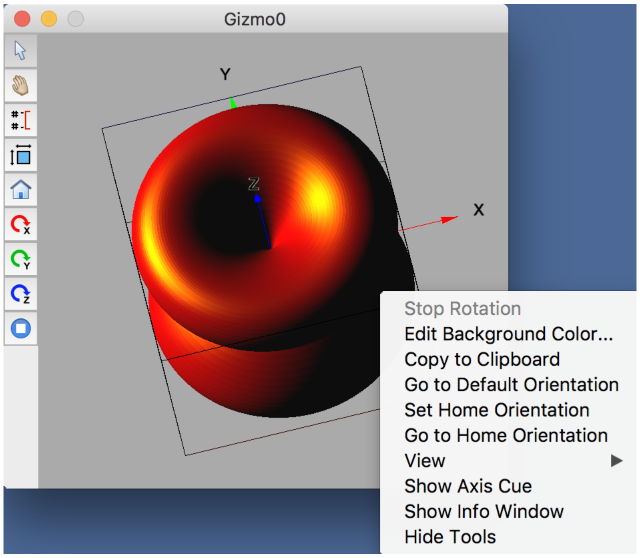
The Gizmo Display Window

The Gizmo display window presents a rotatable 3D display of objects representing waves and 3D drawing primitives as specified by the display list in the associated Gizmo info window.

You can display a palette of tools by choosing Gizmo→Show Tools.

You can access a pop-up menu to modify the appearance of the Gizmo display window by right-clicking (Windows) or Ctrl-clicking (Macintosh) in the display window.

Here is what the display window looks like with the tool palette and pop-up menu showing:



You can rotate the scene in the window by clicking and dragging the mouse as if you were rotating a virtual trackball positioned at the center of the window. You can also use the tool palette, mouse wheel, cursor keys and the x, y, z keys on your keyboard to rotate the scene.

Gizmo Display Window Tool Palette

To display the Gizmo tool palette, choose Gizmo→Show Tools or right-click and choose Show Tools. The tool palette contains the following icons, from top to bottom:

Arrow Tool

When the arrow tool is selected, dragging the body of the display window rotates the 3D scene. The arrow tool and the hand tool are mutually exclusive.

Hand Tool

When the hand tool is selected, dragging the body of the display window pans the 3D scene. The arrow tool and the hand tool are mutually exclusive.

Axis Tool

Clicking the axis tool displays the Axis Range dialog. This is equivalent to choosing Gizmo→Axis Range.

Aspect Ratio Tool

Toggles "aspect ratio" mode.

When aspect ratio mode is off, the length of each axis is the same.

When aspect ratio mode is on, the length of each axis is proportional to the range of data displayed against that axis.

Home Tool

Clicking the home tool sets the X, Y and Z rotation angles to 0 or to some other orientation that you designated as "home".

Rotate About X

Clicking the Rotate About X tool starts the 3D scene rotating about the X axis. To stop it, click the Stop tool or click once in the body of the display window.

Rotate About Y

Clicking the Rotate About Y tool starts the 3D scene rotating about the Y axis. To stop it, click the Stop tool or click once in the body of the display window.

Rotate About Z

Clicking the Rotate About Z tool starts the 3D scene rotating about the Z axis. To stop it, click the Stop tool or click once in the body of the display window.

Stop Tool

Clicking the stop tool stops all rotation.

Gizmo Display Window Contextual Menu

The Gizmo Display contextual menu provides shortcuts for common tasks. It contains the following items, from top to bottom:

Stop Rotation

Stops the rotation of the 3D scene, if any.

Edit Background Color

Sets the background color for the Gizmo window.

Copy to Clipboard

Copies the Gizmo plot to the clipboard using the format set in the Export Graphics dialog (Edit menu).

Go to Default Orientation

Sets the orientation of the 3D space to the orientation in effect when a new Gizmo window is first created.

Set Home Orientation

Stores the current orientation as the orientation to be used when Go to Home Orientation is selected or the Home icon in the tool palette is clicked.

Go to Home Orientation

Rotates the 3D scene to the home orientation.

View

Rotates the 3D scene to one of several preset orientations.

Show Axis Cue

Displays arrows showing the X, Y and Z directions.

Show Info Window

Shows the Gizmo info window associated with the active Gizmo display window.

Show Tools

Shows the Gizmo tool palette.

Match Window Size

Sets another Gizmo display window to the same size as the active Gizmo display window.

This item appears only if you have multiple Gizmo display windows.

Rotate to Match

Rotates another Gizmo display window to the same orientation as the active Gizmo display window.

This item appears only if you have multiple Gizmo display windows.

Sync to Gizmo

Locks the rotation of the current Gizmo plot to the same orientation as the another Gizmo plot. If you rotate the other plot, both rotate to the same orientation.

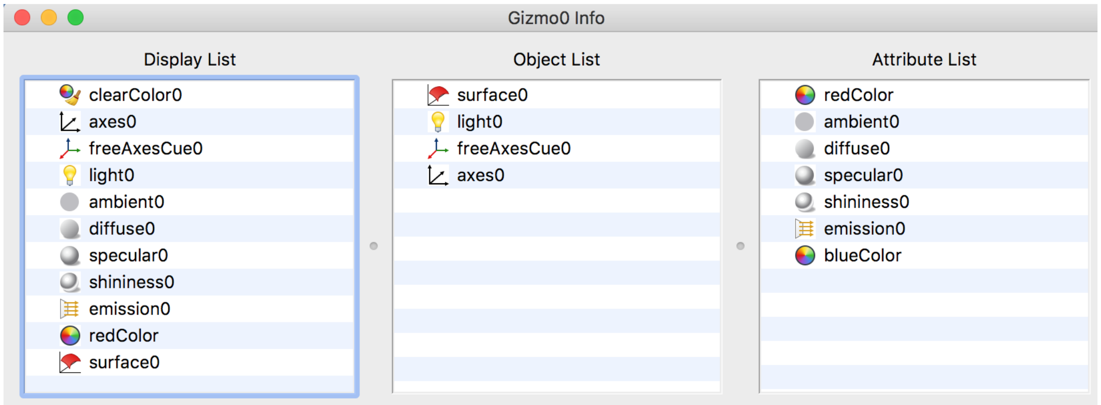
To have mutual syncing you must set each window to be synced to the other.

This item appears only if you have multiple Gizmo display windows.

The Gizmo Info Window

The Gizmo info window is the main user interface for controlling the display of objects in the Gizmo display window. Each info window has an associated Gizmo display window that shows the resulting graphical plot.

The info window contains three lists: the display list, the object list, and the attribute list. You use these lists to add objects and to modify their appearance in the Gizmo display window. Only items that appear in the display list are actually drawn in the Gizmo display window.



Gizmo0 Info

Display List	Object List	Attribute List
clearColor0	surface0	redColor
axes0	light0	ambient0
freeAxesCue0	freeAxesCue0	diffuse0
light0	axes0	specular0
ambient0		shininess0
diffuse0		emission0
specular0		blueColor
shininess0		
emission0		
redColor		
surface0		

You can create objects using the + icon at the bottom of the object list. Clicking the + icon displays a menu from which you choose the type of object to add. The + icon below the attribute list adds attributes while the + icon below the display list adds operations.

The object list contains only objects and the attribute list contains only attributes. The display list can contain objects dragged in from the object list, attributes dragged in from the attribute list, and operations.

You can edit an item's properties by double-clicking its icon or by selecting it and clicking the gear icon below the list. You can remove an item from a list by selecting it and pressing the delete key or by clicking the - icon.

If you have two or more open Gizmo windows in one experiment you can have two or more open info windows. You can then drag an attribute from the attribute list of one Info window and drop it in the attribute list of another info window. You can drag an object from the object list of one info window and drop it in the object list of another info window. If you drag an object from the object list of one info window and drop it in the display list of another info window, Igor also creates the corresponding entry in the object list of the receiving window.

It is not possible to drag and drop objects between info windows belonging to two instances of Igor.

The Gizmo Object List

The middle list in the info window is the object list. It lists all of the objects that you have created which are then available for use in the display list.

Gizmo supports many types of objects including wave-based objects such as surface plots and drawing primitives such as spheres. If you click the + icon under that object list you see a menu of the available object types. See **Gizmo Objects** on page II-418 for details.

For an object to appear in the Gizmo plot, you must drag it to the display list.

The Gizmo Display List

The display list controls what actually appears in the Gizmo display window. Gizmo processes the items in the display list in the order in which they appear.

In addition to objects that you drag in from the object list and attributes that you drag in from the attribute list, you can add the following operations to the display list:

ClearColor, ColorMaterial, Translate, Rotate, Scale, Main Transformation, Enable, Disable and Ortho.

Using the ColorMaterial, Enable and Disable operations requires some familiarity with OpenGL.

The Main Transformation item is used in conjunction with lighting. This is described under **Gizmo Positional Lights** on page II-434.

The Ortho operation controls the projection of the 3D space onto the 2D screen. This is described under **Gizmo Projections** on page II-422.

If you are familiar with OpenGL, you should note that Gizmo automatically generates a small number of OpenGL instructions e.g., viewing transformation, default lighting, etc., that are not visible on the list. If you provide your own alternatives the various defaults are simply not executed. For example, by default Gizmo provides a neutral ambient light to illuminate the scene. However, if you add one or more lights to the display list, the default ambient light is omitted.

Item Ordering in the Gizmo Display List

You can reorder items in the display list by dragging and dropping them in the desired locations. The order of items in the display list is important because it determines the order of execution of OpenGL drawing instructions which govern the appearance of the plot. This becomes obvious when you use operations such as translation, rotation, or scaling.

There are a few items for which the exact position in the list does not make any difference, but in the majority of cases a change in the order of items produces a visible change in the display. For example, if you switch the order of rotation and translation operations you will get a completely different result; see **Gizmo Object Rotation** on page II-424 for an example.

The Gizmo Attribute List

The attribute list appears on the right side of the info window.You create an attribute by clicking the + icon and selecting the type of attribute you want. You then drag that attribute into the display list as a global attribute or on top of an item in the object list as an embedded attribute.

The order of items in the attribute list is unimportant.

Attributes are discussed in detail under **Gizmo Attributes** on page II-419.

Gizmo Objects

There are five main categories of Gizmo objects: wave-based objects, axis objects, drawing primitive objects, lights, and miscellaneous objects.

Wave-based Gizmo objects, also called "data objects", get their data from waves and include the following types:

- **Path Plots**
- **Ribbon Plots**
- **Surface Plots**
- **Isosurface Plots**
- **Voxelgram Plots**
- **3D Scatter Plots**
- **3D Bar Plots**
- **Gizmo Image Plots**

Axis object types include:

- **Axis Objects**
- **Axis Cue Objects**

Drawing primitive object types include:

- **Line Objects**
- **Triangle Objects**
- **Quad Objects**
- **Box Objects**
- **Sphere Objects**
- **Cylinder Objects**
- **Disk Objects**
- **Tetrahedron Objects**
- **Pie Wedge Objects**

There is only one light object type:

- Light Objects (see **Gizmo Colors, Material and Lights** on page II-428)

Miscellaneous object types include:

- **Group Objects**
- **Texture Objects**
- **Matrix4x4 Objects**

You create an object by clicking the + icon below the object list in the info window for a given Gizmo display window.

An object of a given type has a set of internal properties that you can edit when you first create the object. You can edit them later by double-clicking the object in the object or display lists.

Creating a Gizmo object puts it in the object list. It is not displayed until you drag it to the display list. You can drag a given object to the display list multiple times. This creates an new display object each time.

Gizmo Attributes

A Gizmo attribute encapsulates a setting which you can then apply to the Gizmo display list as a global attribute or to a specific Gizmo object as an embedded attribute.

Gizmo supports the following types of attributes:

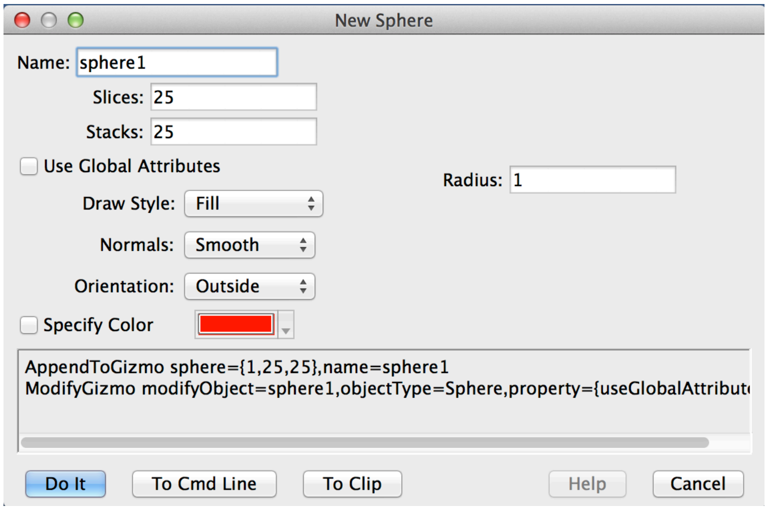
- Color
- Ambient
- Diffuse
- Specular
- Shininess
- Emission
- Blending
- Point size
- Line width
- Alpha test function

You create an attribute using the attribute list in the info window. You can then drag the attribute to the display list as a global attribute or into an object in the object list as an embedded attribute.

In addition to global and embedded attributes, Igor7 added internal attributes, described in the next section.

Internal Attributes

Internal attributes are built into objects. For example, the New Sphere dialog looks like this:



Name: sphere1

Slices: 25

Stacks: 25

□ Use Global Attributes

Radius: 1

Draw Style: Fill

Normals: Smooth

Orientation: Outside

□ Specify Color 🟥

```
AppendToGizmo sphere={1,25,25},name=sphere1
ModifyGizmo modifyObject=sphere1,objectType=Sphere,property={useGlobalAttribut
```

The draw style, normals, orientation and color settings are internal attributes of the sphere object.

The Use Global Attributes checkbox disables the controls under it and enables the use of the respective global attributes for the object in question. It does not affect the use of other global attributes.

The Specify Color checkbox does the same for color. If unchecked, the object has no intrinsic color. In this case you must add a color material operation and a color attribute to the display list before the object. If Specify Color is checked, Gizmo creates a default color material for the object and uses the specified internal color attribute.

Prior to Igor7 Gizmo supported no internal attributes so you had to use global or embedded attributes. We now recommend that you use internal attributes if they are available in preference to global or embedded attributes.

Global Attributes

When you drag an attribute to the display list, it acts as a global attribute that affects all objects later in the display list.

If you place an attribute such as color in the display list, OpenGL draws all subsequent objects that do not have an internal color specification using this global color. You also need a color material operation in order to see the applied color.

Internal attributes and embedded attributes override global attributes.

Embedded Attributes

Embedded attributes are deprecated and are supported mainly for backward compatibility. We don't recommend their use in new projects because primitive obects now have their own internal attributes which should be used instead.

When you drag an attribute on top of an object in the object list, it becomes embedded in that object. You can embed a given attribute in any number of objects and you can embed any number of attributes in a given object.

For example, if you create a sphere object and you want it to appear in blue, you can create a blue color attribute in the attribute list and drop it on top of the sphere object in the object list. The advantage of doing so, as opposed to directly setting the internal color attribute of the sphere object, is in allowing you to reuse the same color attribute with multiple objects. With this approach, by changing a single attribute you can change the color of all associated objects.

Internal attributes override embedded attributes and global attributes.

When an object with embedded attributes is drawn, Gizmo first stores the state of the drawing environment. It then executes the embedded attributes immediately before the object is drawn and finally it restores the state of the drawing environment. As a result, embedded attributes affect only the object in which they are embedded.

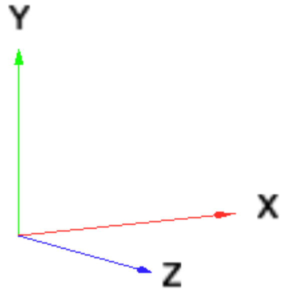
If you apply conflicting attributes to a given object, only the last attribute in the embedded list affects the object appearance. For example, if you have a sphere object with the following embedded attributes: red color, blue color and green color, the sphere is drawn in green.

Gizmo Display Environment

Gizmo constitutes an environment for displaying 3D graphics. This section discusses the main properties of that environment.

Gizmo Coordinate System

Gizmo uses a right-handed, 3D coordinate system. If the positive X axis points to the right and the positive Y axis points up then, by the right-hand rule, the positive Z axis points out of the screen towards you.



Gizmo Dimensions

The default Gizmo viewing volume is a space that is 4 units wide in all three dimensions. The actual display volume is two units in each dimension, centered in the middle of the viewing volume. Each dimension of the display volume extends from -1 to +1 about the origin. The display volume is smaller than the viewing volume to avoid clipping at the corners when the plot is rotated.

All drawing objects, such as spheres and cylinders, are sized in units of the +/-1 display volume. So, for example, if you create a box that is 2 units on a side, it completely fills the display volume. If you create a cylinder that is 3 units high, then the top of the cylinder is clipped because it extends outside the viewing volume boundary.

Superimposed on the display volume and precisely filling it is an axis coordinate system against which wave-based data objects such as scatter and surface plots are plotted. You can set the axis coordinate range for each dimension by choosing Gizmo→Axis Range. The axis coordinate system exists even though, by default, no axes are visible.

The axis coordinate system is autoscaled by default. Consequently, when you initially display a wave-based object, it fills the range of each axis. Since the axis coordinate system fills the display volume, the displayed wave-based object also fills the display volume.

When you display two or more wave-based objects at the same time while the axes are set to autoscale, Gizmo sets the range of each axis based on the minimum and maximum in the respective dimension of all data objects combined.

Once you turn autoscaling off, the axis range that you set determines the extent to which wave-based objects fill the display volume.

When you combine drawing objects and wave-based objects, the dimensions and positions of the drawing objects remain in +/-1 display volume units whereas the wave-based objects are displayed against the axis coordinate system.

Gizmo Clipping

When you set the range of any axis, you may use values that do not include the full range of the data. To display the results correctly in this case, Gizmo creates clipping planes on the relevant sides of the display volume. Once created, these clipping planes affect both wave-based data objects and drawing objects. The clipping planes are not created unless a data object extends beyond the range of the axes.

If you want to do your own clipping, this automatic Gizmo clipping may intefere. To disable automatic clipping, for example for a surface object named surface0, you can execute:

```
ModifyGizmo modifyObject=surface0, objectType=surface, property={Clipped,0}
```

If you are working in advanced mode (see **Advanced Gizmo Techniques** on page II-466), you can create custom clipping planes to create special effects such as gaps in a surface plot. To use clipping planes, make sure that you are not using an axis range that is smaller than the span of the data in any dimension. Current graphics hardware support 6 to 8 clipping planes and axis-range clipping planes have a priority. For an example, open the Clipping Demo experiment.

Gizmo Projections

A number of different projections can be used to control the display of 3D objects in a 2D Gizmo window.

By default Gizmo uses a built-in default orthographic projection which does not appear in the display list. The default orthographic projection is ideal for most applications.

Gizmo calculates the parameters to use with the default orthographic projection by scanning all objects in the display list and computing their largest extent. The default orthographic projection will be 2 units in all directions which allows full rotation of wave-based objects without clipping.

The automatic scanning of objects on the display list does not take into account optional translation, rotation, or scaling operations. If you use any of these in the display list you should also provide a separate ortho operation with the appropriate definition of the projected space.

To add a projection operation, click the + icon under the display list. By default the only projection offered is Ortho and this is sufficient for nearly all purposes. In order to choose another type of projection you must check the Display Advanced Options Menus checkbox in the Gizmo section of the Miscellaneous Settings dialog which you can access via the Misc menu.

You can have any number of projection operations on the display list. When there is more than one projection Gizmo executes only the last one.

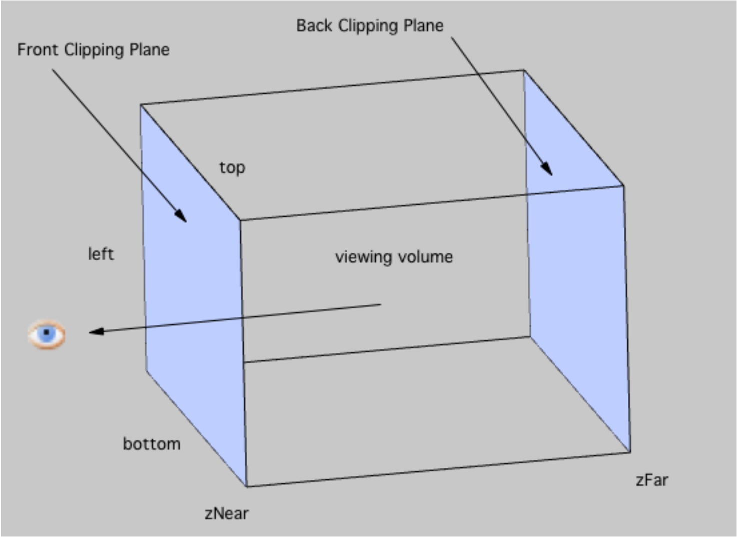
When you are using a projection other than Ortho you must also specify the viewing angle. As a result, the standard mouse rotation and mouse wheel zoom do not apply.

Orthographic

The orthographic projection maintains the geometric orientations and scalings of 3D objects. This is the default projection for the Gizmo display window and is recommended for most purposes.

Unlike perspective projection, discussed in the next section, orthographic projection preserves object parallelism and there is no object foreshortening. Orthographic projection is analogous to an arrangement where dimensions of objects in the displayed scene are small compared to the viewing distance. Another way to think of it is that the image plane is perpendicular to one of the coordinate axes. Only objects contained within the viewing volume are visible. This projection is also faster than perspective.

As shown in this diagram, the ortho projection depends on 6 parameters: left, right, bottom, top, zNear and zFar. The parameters are measured from the center of the display volume and are expressed in +/-1 display volume units.

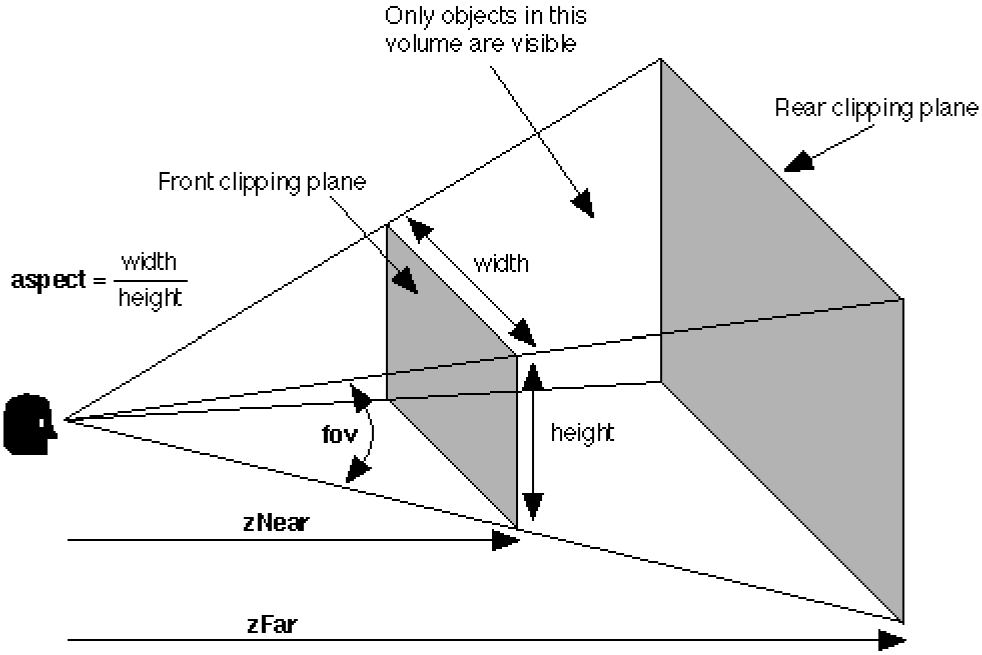


The zoom (using the mouse wheel) and pan tools are implemented by modifying the default ortho projection. If you add your own projection to the display list, the zoom tool is disabled.

Perspective

The perspective projection simulates the way your eye sees 3D objects. Although this is a realistic projection, it does not preserve the exact orientations or shapes of objects; for example, parallel lines may diverge or converge and there is foreshortening of objects. The viewing area is in the shape of a truncated pyramid in which the top has been cut off parallel to the base. Only objects within the viewing volume are visible. This is a symmetric perspective view of the viewing volume; a frustum, discussed in the next section, supports an asymmetric volume.

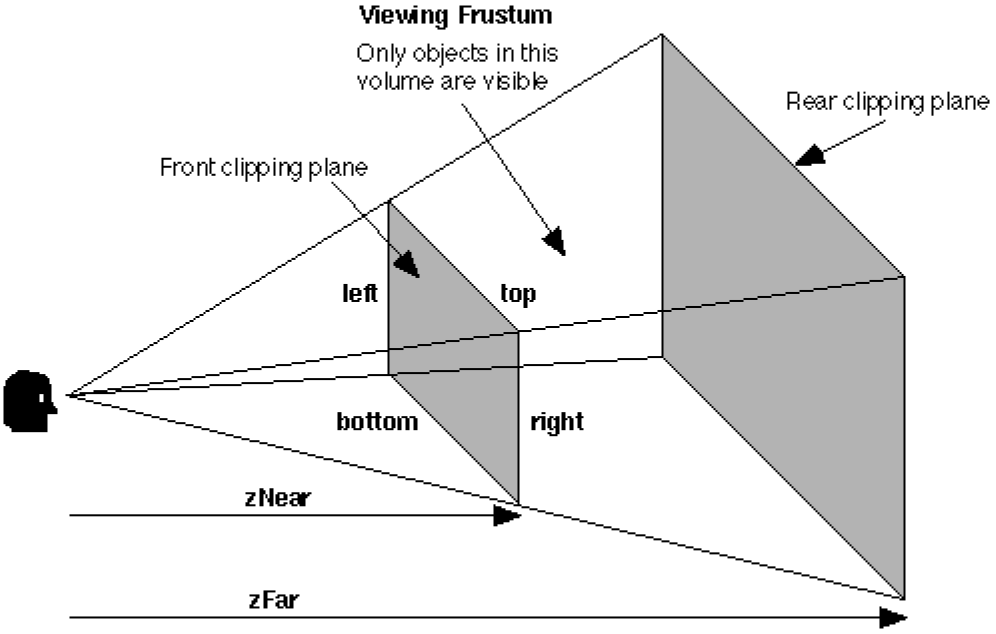
As shown in this diagram, the perspective projection depends on 4 parameters: fov, aspect, zNear and zFar.



Frustum

The frustum projection is essentially the same as the perspective projection, but in this case it has more flexible settings, which means that it does not have to be symmetrical or aligned with the Z axis.

As shown in this diagram, the frustrum projection depends on 6 parameters: left, right, bottom, top, zNear and zFar.



Viewport

A viewport is the rectangular 2D region of the window where the projected scene is drawn. You can use this to scale and distort the scene in the Gizmo display window.

The viewport projection depends on 4 parameters: left, bottom, width and height.

LookAt

A LookAt projection maps the center point to the negative Z axis, the eye point to the center, and the up vector to the Y axis. It may be useful if you want to change the origin of the coordinate system and the basic orientation.

Gizmo Object Rotation

You can click in a Gizmo display and drag the mouse to rotate the plot. The default rotation is implemented as if there is a virtual trackball in the center of the Gizmo window. Rotation through a positive angle is in a counterclockwise direction when viewed along the ray from the origin.

When experimenting with rotation it is helpful to display the axis cue which shows the X, Y and Z directions. Right-click and choose Show Axis Cue. The axis cue shows the orientation of the main Gizmo axes.

You can rotate the plot using the keyboard when a Gizmo display window is active. Press the x, y, or z keys to rotate the display counterclockwise in 1-degree increments about the respective axis. Press the shift key along with x, y, or z to rotate clockwise.

The up arrow and down arrow keys rotate the plot about a horizontal line drawn through the middle of the display area. The left arrow and right arrow keys rotate the plot about a vertical line drawn through the middle of the display area.

By default the mouse scroll wheel zooms the plot. You can change it to rotate the plot using a miscellaneous setting. Choose Misc→Miscellaneous Settings and click Gizmo in the list on the left to see the Gizmo miscellaneous settings. Using the mouse scroll wheel for scrolling behaves the same as using the arrow keys.

You can start the 3D scene rotating continuously by clicking and gently flinging with the mouse. This requires releasing the mouse button before stopping mouse movement. Click the plot once to stop continuous rotation. If the display pans instead of rotates then click the arrow tool in the Gizmo tool palette to enable rotation.

You can also start continuous rotation using the tool palette. Choose Gizmo→Show Tools and then click one of three rotation icons to start rotation about the X, Y or Z axis. Click the plot once or click the stop icon in the tool palette to stop rotation.

The Home icon in the tool palette rotates the scene to the home orientation. By default the home orientation is X=0, Y=0, Z=0. In this case, the positive X axis points to the right, the Y axis points up and the Z axis points out of the plane of the window toward you. You can set the home orientation by right-clicking and choosing Set Home Orientation.

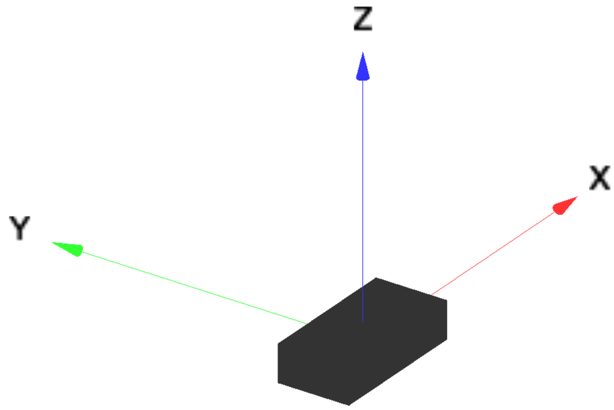
During continuous rotation you can use the x, y and z keys to tweak the orientation.

Gizmo Object Rotation and Translation

In the preceding section we discussed rotation of the entire Gizmo plot about the main Gizmo axes. Each Gizmo object has its own set of axes which, by default, correspond to the main axes. Though this is less frequently needed, it is possible to rotate a Gizmo object's axes independent of the main Gizmo axes. You do this by adding a rotate operation to the display list.

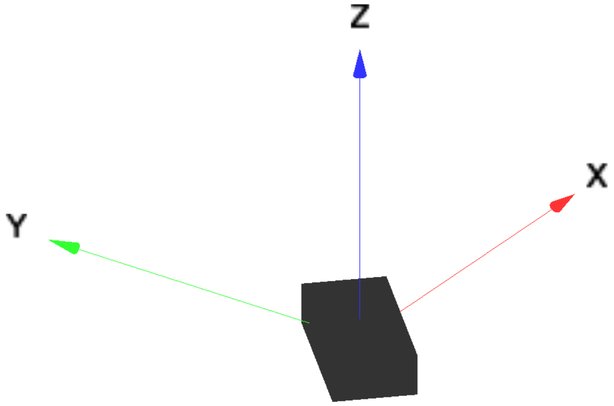
To see this we create a Gizmo with an axis cue and an unrotated box:

```
NewGizmo
ModifyGizmo showAxisCue=1
ModifyGizmo setQuaternion={0.435,-0.227,-0.404,0.777}
AppendToGizmo box={0.5,0.25,0.15}, name=box0
ModifyGizmo setDisplayList=0, object=box0
```



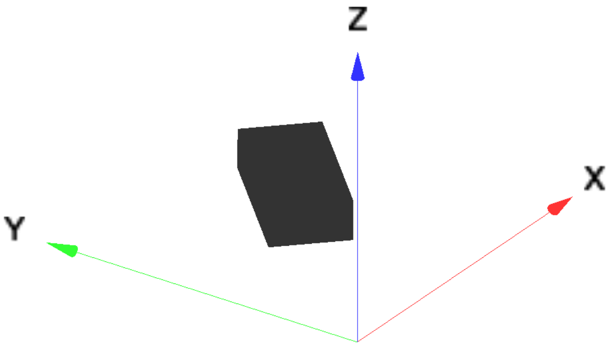
We then add a rotate operation to the display list, before the box object. This rotates the box object's axes by 45 degrees:

```
ModifyGizmo insertDisplayList=0, opName=rotate0, operation=rotate,
      data={45,0,0,1}
```



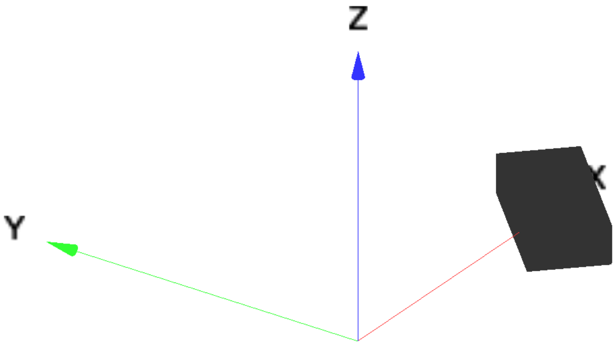
Now we insert a translation along the X axis:

```
ModifyGizmo insertDisplayList=1, opName=translate0, operation=translate,
      data={1,0,0}
```



The translation is along to the object's X axis, not the main X axis. Because of this, rotation followed by translation gives a different result than translation followed by rotation. To illustrate this we switch the order of the operations:

```
ModifyGizmo setDisplayList=0, opName=translate0, operation=translate,
      data={1,0,0}
ModifyGizmo setDisplayList=1, opName=rotate0, operation=rotate, data={45,0,0,1}
```



In this last case, at the time the translation was done, the object's axes were aligned with the main axes. The translation was along the object's X axis which pointed in the same direction as the main X axis.

Translate and rotate operations apply to all objects below them on the display list. They are cumulative. In other words, a translate or rotate operation on a given object starts from the current position or rotation of the object when the operation is applied.

Programming Rotations in Gizmo

The orientation of the Gizmo plot is stored internally as a quaternion. A quaternion is analogous to a complex number but extended to 3 dimensions.

When you manually rotate the plot you are changing the internal quaternion. You can query it using **GetGizmo** curQuaternion.

There are a number of **ModifyGizmo** keywords that programmatically set the orientation. The setQuaternion, setRotationMatrix, and euler keywords set the orientation in absolute terms and take a quaternion parameter, a transformation matrix, or a set of Euler angles, respectively. The appendRotation keyword applies a rotation specified by a quaternion to the current orientation. The goHome keyword goes to the home orientation. The idleEventQuaternion and idleEventRotation keywords change the orientation periodically. The matchRotation keyword sets the orientation to match another Gizmo window. The syncRotation keyword syncs one Gizmo window's rotation to that of another. The stopRotation keyword stops rotation.

No matter how you set the orientation it is stored internally as a quaternion.

If you want to rotate Gizmo's display so that the X axis points to the right, the Y axis points away from you, and the Z axis points up, you need a quaternion for rotation of 90 degrees about the X axis. This can be accomplished using the command

```
ModifyGizmo setQuaternion={sin(pi/4),0,0,cos(pi/4)}
```

If you want to rotate the plot to the orientation specified by an axis of rotation and an angle about that axis, you first need to convert those inputs into a quaternion. For an axis of rotation given by Ax, Ay, Az and an angle theta in radians, the rotation quaternion consists of the four elements:

```
Qx = Ax*sin(theta/2)/N
Qy = Ay*sin(theta/2)/N
Qz = Az*sin(theta/2)/N
Qw = cos(theta/2)
```

where we normalized the rotation vector using N=sqrt(Ax^2+Ay^2+Az^2).

To compute a rotation quaternion that represents two consecutive rotations, i.e., a rotation specified by quaternion q1 followed by a rotation specified by quaternion q2, we need to compute the product quaternion qr=q2*q1 using quaternion multiplication, which is not commutative. This can be computed using the following function:

```
// q1 and q2 are 4 elements waves corresponding to {x,y,z,w} quaternions.
// The function computes a new quaternion in qr which represents quaternion
// product q2*q1.
Function MultiplyQuaternions(q2,q1,qr)
   Wave q2,q1,qr

   Variable w1=q1[3]
   Variable w2=q2[3]
   qr[3]=w1*w2-(q1[0]*q2[0]+q1[1]*q2[1]+q1[2]*q2[2])
   Make/N=4/FREE vcross=0
   vcross[0]=(q2[1]*q1[2])-(q2[2]*q1[1])
   vcross[1]=(q2[2]*q1[0])-(q2[0]*q1[2])
   vcross[2]=(q2[0]*q1[1])-(q2[1]*q1[0])
   MatrixOP/FREE aa=w1*q2+w2*q1+vcross
   qr[0]=aa[0]
   qr[1]=aa[1]
   qr[2]=aa[2]
   Variable NN=norm(qr)
   qr/=NN
End
```

Assume you want to set the Gizmo orientation to the rotation produced by the preceding example (X axis points to the right, Y axis points away from you, and the Z axis points up) followed by a 90-degree rotation about the Z axis, producing the orientation where the X axis points to toward you, the Y axis points to the right, and the Z axis points up. You could execute this:

```
Make/O/N=4 q1={sin(pi/4),0,0,cos(pi/4)}       // Z up, X right, Y away
Make/O/N=4 q2={0,0,sin(pi/4),cos(pi/4)}       // 90 degree rotation about Z
Make/O/N=4 qr// Resultant orientation
MultiplyQuaternions(q2,q1,qr)                 // Compute resultant orientation
Print qr
 qr[0]= {0.5,0.5,0.5,0.5}
ModifyGizmo setQuaternion={0.5,0.5,0.5,0.5}
```

Another way to do this is to use the ModifyGizmo appendRotation command which does the quaternion multiplication for you:

```
ModifyGizmo setQuaternion={sin(pi/4),0,0,cos(pi/4)}    // Z up, X right, Y away
ModifyGizmo appendRotation={0,0,sin(pi/4),cos(pi/4)}
```

The last command rotates 90 degrees about the Z axis starting from the current orientation.

Locking Rotation

In some situations you want to prohibit rotation of the Gizmo plot. You can do this by executing:

```
ModifyGizmo lockMouseRotation = 1
```

This locks the rotation without providing any visible clue that it is locked.

A more complicated approach is to set a fixed viewing transformation on the display list, then add all the objects to be drawn, and end with a MainTransformation operation. All objects in the display list above the main transformation will be drawn with fixed rotation and scaling.

Gizmo Colors, Material and Lights

The rendered color of a Gizmo object depends on its internal color, color attributes, material as specified by a color material operation and lighting. The following sections discuss these topics.

Gizmo Color Specification

Colors of objects and lights are specified using four floating point numbers: RGBA.

The first three are the primary colors red, green, and blue which combine additively to give the final color. The intensity of each component is a number in the range [0.0 to 1.0].

The fourth component is alpha, which determines the opacity of an object. Values are between 1.0, a completely opaque color, and 0.0, a completely transparent or colorless object.

To conserve graphics resources, alpha blending for a Gizmo window is turned off by default. To create objects that have some degree of transparency, in addition to setting the alpha component of their color, you must enable transparency blending by choosing Gizmo Menu→Enable Transparency Blend.

By default the blendFunc0 and enableBlend items are inserted at the top of the display list and therefore affect the drawing of all items that follow them. You may be able to improve drawing speed by moving this pair of entries immediately above the items that require transparency blending. You would then also add a disable operation to disable the blending right after the last entry that uses transparency. For details see **Transparency and Translucency** on page II-432.

Here is an example showing color specification with transparency:

```
// Create a new Gizmo with axis cue and set rotation
NewGizmo
```

```
ModifyGizmo showAxisCue=1
ModifyGizmo setQuaternion={0.435,-0.227,-0.404,0.777}

// Append a red, opaque sphere
AppendToGizmo/D sphere={0.25,25,25}, name=sphere0
ModifyGizmo modifyObject=sphere0, objectType=Sphere, property={colorType,1}
ModifyGizmo modifyObject=sphere0, objectType=Sphere,
      property={color,1.0,0.0,0.0,1.0}

// Append a blue, translucent box
AppendToGizmo/D box={0.5,0.5,0.5}, name=box0
ModifyGizmo modifyobject=box0, objectType=Box, property={colorType,1}
ModifyGizmo modifyobject=box0, objectType=Box,
      property={colorValue,0,0.0000,0.2444,1.0000,0.5000}
```

You now have a translucent blue box surrounding a red sphere but you can not see the red sphere because alpha blending is disabled by default. Now we enable it:

```
// Enable transparency blend
AppendToGizmo attribute blendFunc={770,771}, name=blendFunc0
ModifyGizmo insertDisplayList=0, opName=enableBlend, operation=enable,
      data=3042
ModifyGizmo insertDisplayList=0, attribute=blendFunc0
```

The last set of commands is what is executed when you choose Gizmo Menu→Enable Transparency Blend.

Converting Igor Colors to Gizmo Colors

For historical reasons, Igor represents color components as integer values from 0 to 65535. OpenGL, and consequently Gizmo, represent color components as floating point values from 0.0 to 1.0. To convert from an Igor color component value, such as you might receive from the ColorTab2Wave operation, to a Gizmo color component value for use in a Gizmo command, you need to divide the Igor color component value by 65535.

Here is an example of such a conversion:

```
ColorTab2Wave Rainbow                          // Creates M_colors
MatrixOP/O gizmoRainbowColors = M_colors/65535  // Convert to SP and scale
Redimension/N=(-1,4) gizmoRainbowColors        // Add a column for alpha
gizmoRainbowColors[][3] = 1                    // Set the alpha to 1 (opaque)
```

Colors, Materials and Lighting

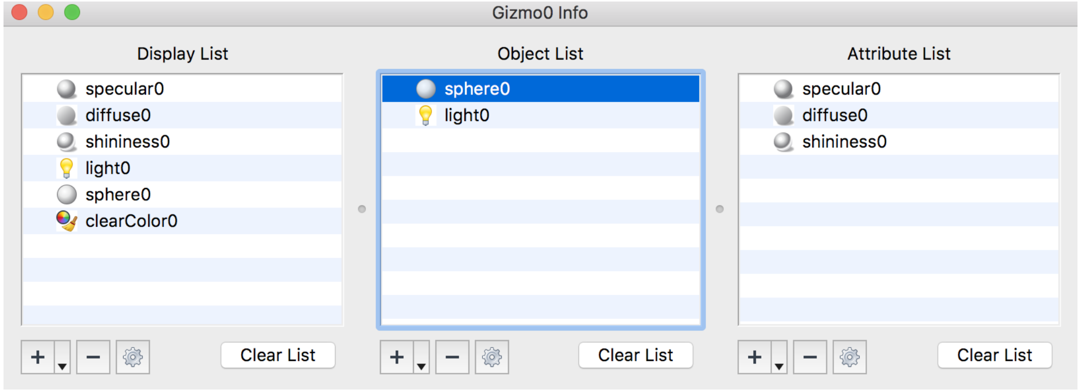
The perceived color of an object depends on the combination of the color, the material and the lighting.

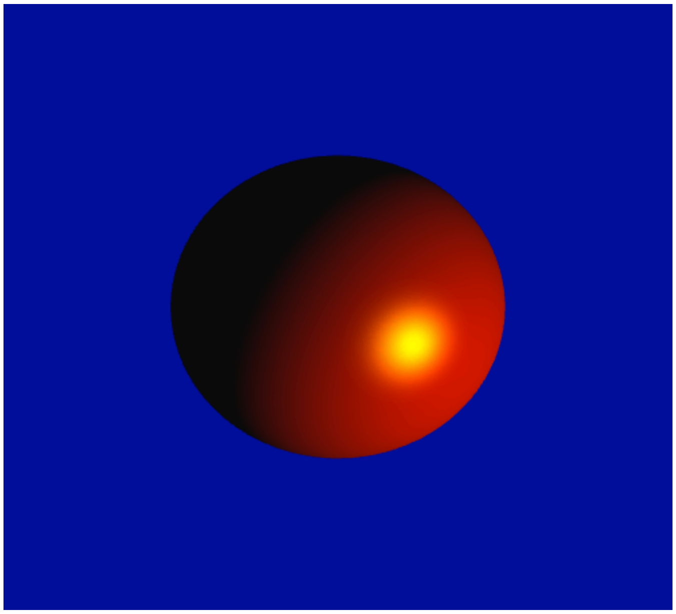
A color material specifies which faces of an object are to be colored by OpenGL and the way the object emits or responds to light. The "face" property can be set to GL_FRONT, GL_BACK or GL_FRONT_AND_BACK. The "mode" property can be set to GL_EMISSION, GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR or GL_AMBIENT_AND_DIFFUSE.

When you create a Gizmo object you have the option to specify a color or to leave it unspecified. If you specify a color, Gizmo creates a default color material for the object. The default color material has the GL_FRONT_AND_BACK and GL_AMBIENT_AND_DIFFUSE settings. If you don't specify a color then Gizmo does not create a default color material and you must create a color material yourself. This color material affects all objects that appear later in the display list if they have no default color material.

Whatever color material is in effect for a given object, you can modify it by adding ambient, diffuse, specular, shininess or emission attributes above it in the display list.

To create a shiny object (e.g., sphere), start with a sphere object, add to it shininess and specular attributes and then add the sphere to the display list following a light object that has matching diffuse and specular components. In this case the info window and display window like this:





For an example, open this demo experiment: Material Attributes.

Color Waves

Wave-based objects can be drawn in fixed colors or using the built-in Igor color tables. You can also use your own color waves to specify the colors of the objects in the Gizmo display window. The format of a color wave is similar to that of the corresponding data wave except that each data node (vertex) has red, green, blue and alpha color components associated with it.

One situation where a color wave is useful is when you want to display a set of scalar values (e.g., temperature measurements) corresponding to points on a 3D surface. In this case you have one wave that describes the shape of the surface and another wave containing the scalar measurements. The application of a color wave gives you complete freedom to represent the scalar data distributed on the surface. In most cases you can create an appropriate color wave using the **ModifyGizmo** makeColorWave and makeTripletColorWave keywords to create a color wave for the data based on one of the built-in tables and then specifying an appropriate alpha in the color wave.

The required color wave format depends on the format of the data wave that describes the object to which the color is to be applied. This table shows some of the various data wave formats with their corresponding color wave formats. Dimensionality is indicated in parenthesis:

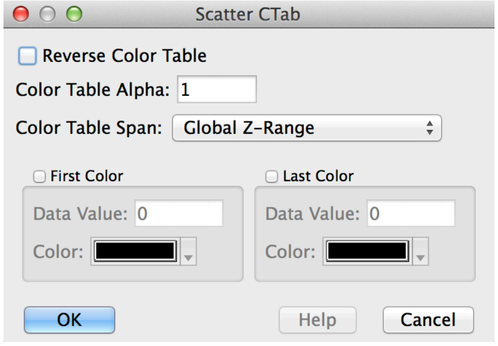
Data Wave	**Color Wave**
Triplet (Mx3) Used in path, ribbon and scatter plots	(Mx4) with RGBA in successive columns
Matrix of Z values (MxN) Used in surface and 3D bar chart plots	(MxNx4) with RGBA in successive layers
Sequential Quads (Mx4x3) Used in parametric surface plots	(Mx4x4) with RGBA in successive layers
Disjoint Quads (Mx12) Used in parametric surface plots	(Mx4x4) with RGBA in successive layers
Triangles (Mx3) Used in parametric surface plots	(Mx4) with RGBA in successive columns
Parametric (MxNx3) Used in parametric surface plots	(MxNx4) with RGBA in successive layers

Color Tables in Gizmo

You can use Igor's built-in color tables to specify the colors of surface, scatter, path and ribbon objects. Isosurface, voxelgram, 3D bar chart, and image objects do not support the use of color tables.

For surface, scatter, path and ribbon objects, you can select the color table in the properties dialog for a given type of object. To see a list of available color tables, see **CTabList** and for more information, see **Color Table Details** on page II-395.

When you choose a color table you can also set related options. In the properties dialog you set these options using the Details button which displays a subdialog that looks like this:


Scatter CTab

□ Reverse Color Table

Color Table Alpha: 1

Color Table Span: Global Z-Range

□ First Color

Data Value: 0

Color: ■

□ Last Color

Data Value: 0

Color: ■

The Color Table Alpha setting applies to all colors in the color table. If you want to apply a variable alpha, set the object color using a color wave where you specify the alpha value for each data point. Color waves are not supported for isosurface objects. Remember that alpha effects require that blending is enabled and that there is a blend function on the display list - see Transparency and Translucency for details.

The Color Table Span setting determines the numeric quantity used to select a color from the color table. For a scatter plot the most common choice is Global Z Range. By default this means that the lowest Z value displayed in the plot is mapped to the first color and the highest Z value is mapped to the last color. The color for a specific scatter element is chosen based on that element's Z value.

This mapping can be tweaked using the First Color and Last Color settings. If you enable First Color and enter a corresponding data value then that data value is mapped to the first color in the color table and any scatter element whose data value is less than the entered value is displayed using the color selected from the color pop-up menu below. If you enable Last Color and enter a corresponding data value then that data value is mapped to the last color in the color table and any scatter element whose data value is greater than the entered value is displayed using the color selected from the color pop-up menu below.

Reflection

Gizmo supports four types of surface object interactions with lights: ambient, diffuse, specular, and shininess.

Lights have ambient, diffuse and specular components.

Materials have ambient, diffuse, specular and shininess attributes.

Ambient

Ambient reflectance determines the overall color of the object. Ambient reflectance is most noticeable in object shadows. The total ambient reflectance is determined by the global ambient light and ambient light from individual light sources. It is unaffected by the viewpoint position.

Diffuse

A diffuse surface reflection scatters light evenly in all directions. This is the most important factor determining the color of an object. It is affected by the incident diffuse light color and by the angle of the incident light relative to the normal direction. It is most intense where the incident light falls perpendicular to the surface. It is unaffected by the viewpoint position.

Specular

Specular reflection governs the appearance of highlights on an object. The amount of specular reflection depends on the location of the viewpoint, being brightest along the direct angle of reflection.

Shininess

Shininess controls the size and brightness of a specular highlight. The shinier the object, the smaller and brighter (more focused) the highlight.

Normals, Lighting and Shading

When you display a scene with lighting effects, make sure to enable the calculation of normals for all objects in the display list. You can do this in the properties dialog for each object. Depending on the type of object, check the Calculate Normals checkbox or choose from the Normals pop-up menu. These settings are off by default to conserve graphics processing resources.

Normals are required because the shading of every pixel depends on the angle between the normal to the surface and the direction of the light source. In the special case of quadric objects (sphere, cylinder and disk) there are internal settings that let you choose between flat, smooth and no normals. All objects draw much slower when normals are calculated.

Transparency and Translucency

Proper implementation of transparency in OpenGL requires that objects be drawn from the back to the front of the scene, starting with the object that is farthest from the viewer and ending with the nearest object. For any fixed viewing transformation it is possible to sort the displayed objects as long as they consist of primitive non-intersecting elements. If you are drawing compound objects such as quadrics you have no control over the order of their constituent segments. Most wave-based objects are transformed into triangle arrays which can be distance-sorted as long as there are no intersecting triangles.

Distance sorting is computationally expensive so most applications avoid it using various tricks. The "poor man's" solution is to use alpha blending. This type of translucency can provide the desired effect for a restricted range of viewing angles and may require re-ordering the objects on the display list.

To use alpha blending, assign colors to two distinct objects on the display list. The translucent object should have an alpha value that corresponds to its opacity. An opaque object has alpha=1, whereas a transparent object has alpha=0. The final steps required for translucency are the addition of the blending function attribute and the enable operation. Select Gizmo→Enable Transparency Blend menu to create the blending function and the enable operation and add both to the display list.

The isosurface is a special case because by construction it consists of non-intersecting triangles. In most applications it is sufficient to sort the triangles in the order of the distance of the viewing point from the centroid of the triangle. You can obtain the triangles corresponding to an isosurface object using **ModifyGizmo** with the saveToWave keyword and establish a sample viewing point. The standard orthographic projection implies infinite distance to viewing point. An example of this type of sorting can be found in the Depth Sorting demo experiment.

Gizmo Lights

Gizmo supports both directional and positional light sources. The type and color of the lights that you add to the display affect the appearance of objects in the display window.

You create lights like any other object by selecting Light from the object list pop-up menu. Using the Light Properties dialog, shown below, you can specify the light type and various light parameters. For the light to have any effect, you must add it to the display list above any object that you want to illuminate.

Lighting effects are defined in terms of their ambient, diffuse, and specular components. The distribution of light intensity is described by the location of the light source, direction, cone angle, and attenuation. The final appearance of an object depends on the combination of the properties of the light and the properties of the object material.

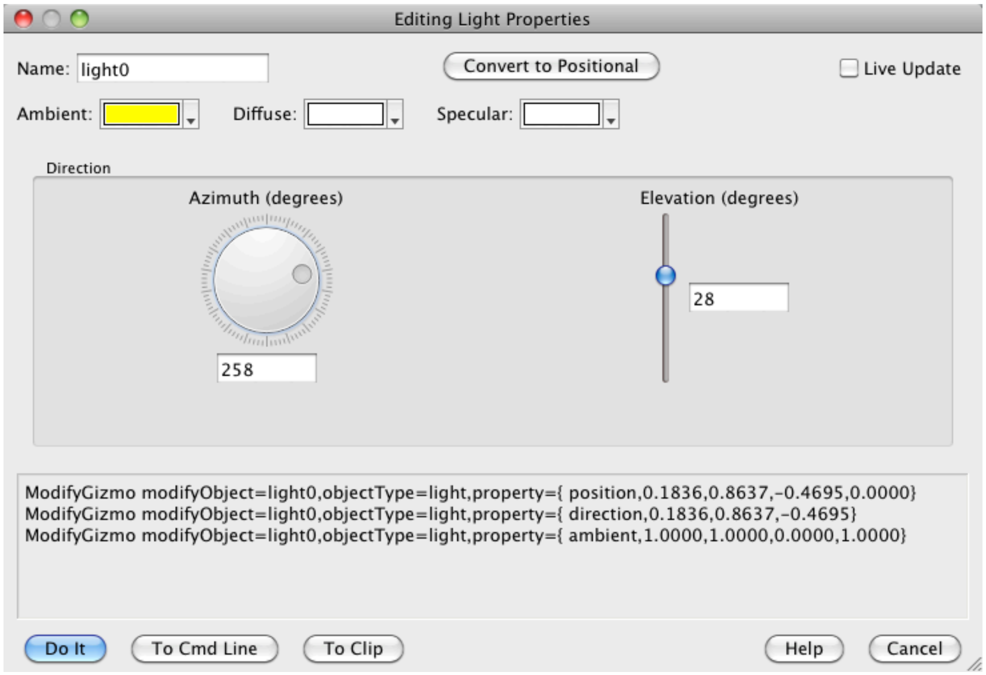
Lighting effects are computed in hardware on a per pixel basis. Therefore, when you want smooth shading, you must describe the object using a sufficiently large number of vertices. For simple objects, such as a single quad (4 vertices), you will likely not see much variation in lighting across the quad. Shading is computed using the dot product between the normal to the surface at each vertex and the direction of the light source. There is no accounting for objects obscuring other objects from the light source or for multiple reflections of light.

If you add no lights to the display list, Gizmo uses default, color-neutral ambient light. If you add a light to the display list, Gizmo removes the default lighting.

Gizmo Directional Lights

You can think of a directional light as a light positioned very far away from the scene so that its rays are essentially parallel within the display volume. The sun is a good example of a directional light. New light objects are directional by default.

The Light Properties dialog contains the controls you need to specify the light's position, color properties and distribution. When editing a light object that is already in the display list, you can click the Live Update checkbox to see how your changes affect the Gizmo Display.



Name: light0

Ambient: 🟨 Diffuse: □ Specular: □

Azimuth (degrees): 258

Elevation (degrees): 28

```
ModifyGizmo modifyObject=light0,objectType=light,property={ position,0.1836,0.8637,-0.4695,0.0000}
ModifyGizmo modifyObject=light0,objectType=light,property={ direction,0.1836,0.8637,-0.4695}
ModifyGizmo modifyObject=light0,objectType=light,property={ ambient,1.0000,1.0000,0.0000,1.0000}
```

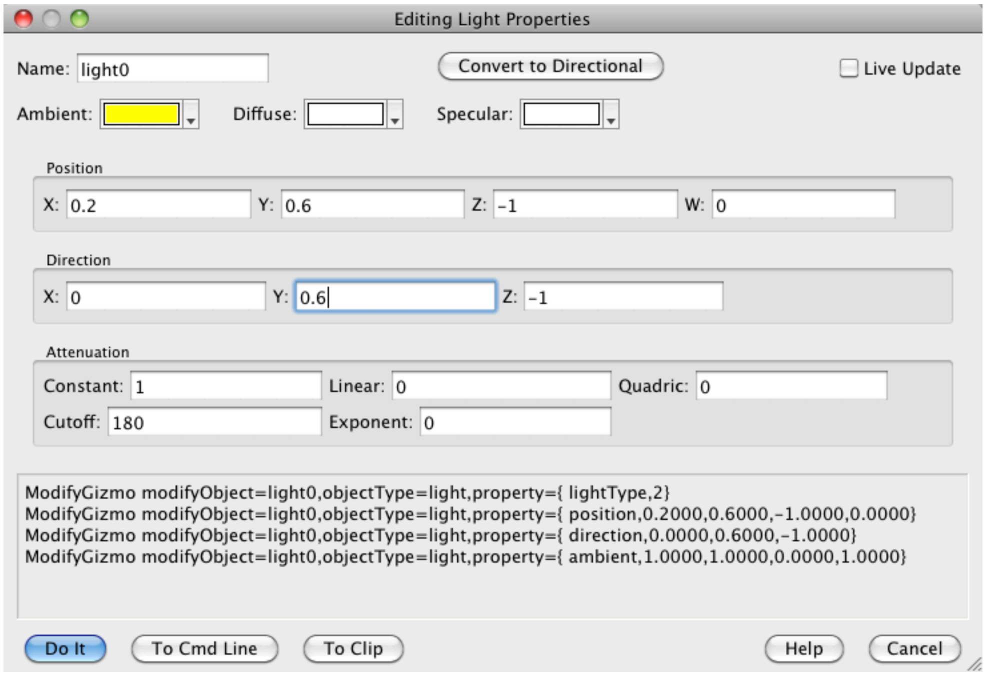
The position of the light is specified via two angles: azimuth and elevation. Elevation is also called "altitude", especially in astronomy. The meaning of these angles is described at http://en.wikipedia.org/wiki/Azimuth. When the elevation is +90 or -90 degrees, the azimuth is undefined.

Ambient, diffuse and specular lighting are described at http://en.wikipedia.org/wiki/Phong_reflection_model. Ambient light illuminates all parts of all objects equally regardless of their orientation and of the position of the light. Diffuse light is reflected off a surface in all directions, as when light hits a rough surface. Specular light is reflected in a specific direction, as when light hits a shiny surface. The illumination created by diffuse and specular light at a given point on an object depends on the angle of the light ray relative to the normal to the surface of that point.

Gizmo Positional Lights

A positional light is a light source that originates at a finite distance from the illuminated scene. A desk lamp is a typical example of a positional light. It can be placed somewhere above the desk and it produces non-uniform illumination as its intensity falls off as a function of distance from the center of the illumination spot.

When you create a light object, it is initialized as a directional light. You can click click the Convert to Positional button in the dialog to get the corresponding positional light settings which you can then adjust as desired.



Editing Light Properties

Name: light0

□ Live Update

Ambient: 🟨 Diffuse: □ Specular: □

Position

X: 0.2 Y: 0.6 Z: -1 W: 0

Direction

X: 0 Y: 0.6 Z: -1

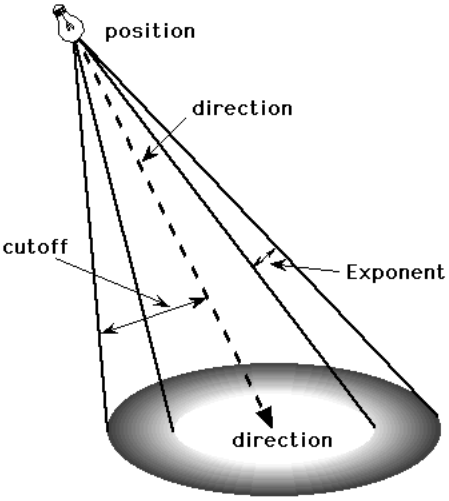
Attenuation

Constant: 1 Linear: 0 Quadric: 0

Cutoff: 180 Exponent: 0

```
ModifyGizmo modifyObject=light0,objectType=light,property={ lightType,2}
ModifyGizmo modifyObject=light0,objectType=light,property={ position,0.2000,0.6000,-1.0000,0.0000}
ModifyGizmo modifyObject=light0,objectType=light,property={ direction,0.0000,0.6000,-1.0000}
ModifyGizmo modifyObject=light0,objectType=light,property={ ambient,1.0000,1.0000,0.0000,1.0000}
```

The parameters for specifying positional lights are illustrated here:



Positional Light Geometry

The top three controls in the directional light dialog specify the RGBA values of each of the ambient, diffuse and specular light components. You should provide some ambient component in at least one of the lights in the display list. This requirement holds even if you are trying to create a predominantly diffuse or specular effect.

When you create a Gizmo object you have the option to specify a color or to leave it unspecified. If you specify a color, Gizmo creates a default color material for the object. The default color material has the GL_FRONT_AND_BACK and GL_AMBIENT_AND_DIFFUSE settings. If you are interested in specular effects you must add specular and shininess attributes (see **Gizmo Colors, Material and Lights** on page II-428).

The Position and Direction controls in the dialog describe the position and direction of the light source. The position is expressed in homogeneous coordinates where the last element (w) is used to normalize the X, Y and Z components. Therefore, if you set w=1, then the X, Y, and Z components specify the absolute position of the light in space. If you set w=0, your light source is infinitely far away and you effectively created a directional light source.

We suggest that you set w=1, and set the direction using the next group of controls in the dialog, which describes the direction of the light as the three components of a vector pointing from the position of the light source to the point that you want the center of the specular spot to illuminate. The typical error here is entering the position of the illuminated spot instead of the direction vector.

The specified position of the light source is subject to the same transformations that apply to any other objects in the display list. In particular, as you rotate display objects, the lights will likewise rotate. If you want to keep the lights stationary so they illuminate different part of the rotating display, you must add a main transformation operation to the display list immediately after the last light object and before all the rotating objects. This will keep all objects listed above the main transformation stationary, and apply the rotation only to all subsequent display list items.

Using the main transformation operation allows you to have some things fixed and other things rotatable. The main transformation operation sets the point in the display list after which coordinate transformations are applied. Coordinate transformations affect translation and scaling in addition to rotation. No transformations are applied to everything above the main transformation in the display list and therefore those items are drawn in their default view, unless you insert explicit translate, rotate or scale operations.

By default, the spot cutoff angle (the light cone half-angle) is 180 degrees which means that the light provides uniform illumination in all directions. If you take the trouble to specify position and direction it will be useful to reduce the cutoff angle to something more realistic - less than 90-degrees.

The constant, linear and quadratic attenuations combine to attenuate positional lights (i.e., for which w is non-zero). The exponent value, in the range 0 to128, determines the intensity falloff from the center of the spot by multiplying the center intensity by the cosine of the angle between the direction of the light and the vertex in question, raised to the power of the exponent value. This can be used to make the light highly specular.

The Positional Light Demo experiment contains a control panel that you can use to explore the interplay between the various positional light parameters and how they affect the lighting on an object.

Gizmo Drawing Objects

Gizmo provides access to a number of drawing primitives. These include **Line Objects**, **Triangle Objects**, **Quad Objects**, **Box Objects**, **Sphere Objects**, **Cylinder Objects**, **Disk Objects**, **Tetrahedron Objects** and **Pie Wedge Objects**. You can use the dialogs associated with the different objects to set the various object properties.

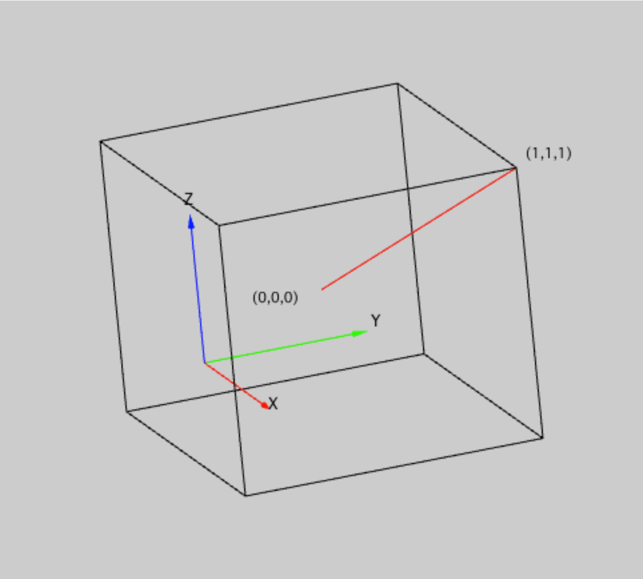
As explained under **Gizmo Dimensions** on page II-421, the size of a drawing object is expressed in display volume units. The display volume extends from the origin to +/- 1 in each dimension. A box of size 1 centered at the origin extends halfway from the origin the edge of the display volume in each dimension.

Unlike drawing objects, wave-based objects such as scatter plots and surface plots are displayed against a separate coordinate system. If you combine drawing objects with wave-based objects, remember that drawing object positions and sizes are always expressed in terms of the +/-1 display volume. If you draw a 2D wave as a surface and you would like to draw a box around it, just add a box whose length, width and height are two units.

Line Objects

A line object is a straight line that connects the two endpoint coordinates. You can add an arrowhead at the start or end of the line or at the mid point. This example shows a line from (1,1,1) to (0,0,0) created by the command

```
AppendToGizmo/D line={1,1,1,0,0,0}
```



You can make the line into a 3D cylinder using:

```
ModifyGizmo modifyObject=line0,objectType=line,property={arrowMode,16}
ModifyGizmo modifyObject=line0,objectType=line,property={cylinderStartRadius,0.05}
ModifyGizmo modifyObject=line0,objectType=line,property={cylinderEndRadius,0.05}
```

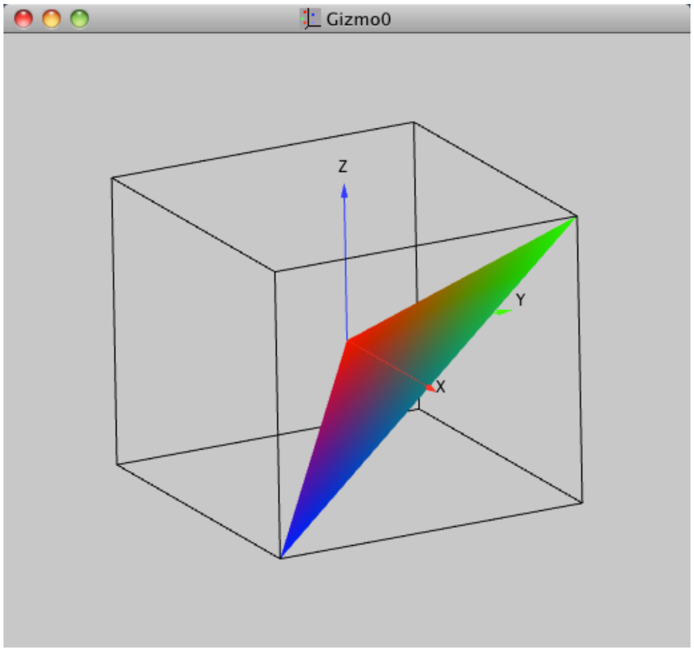
Triangle Objects

A triangle is a planar object bounded by a simple polygon connecting its three vertices. A triangle object is always drawn filled. The fill color is determined by the internal color attribute, an embedded color attribute, or a global color attribute, in that order of precedence. All lighting attributes also apply.

In some situations, it may be more straightforward to create the triangle in a simple orientation and then use translate and rotate operations to position the triangle in its final orientation.

You can set the color of the triangle on a vertex by vertex basis and OpenGL will interpolate the colors between the vertices. Here is an example:

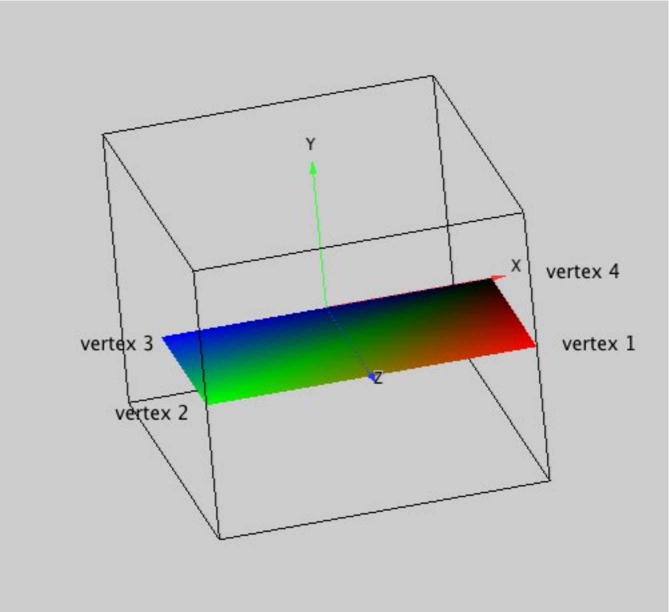
```
AppendToGizmo/D triangle={0,0,0,1,1,1,1,-1,-1}
ModifyGizmo modifyobject=triangle0, objectType=triangle,
      property={colorType,2}
ModifyGizmo modifyobject=triangle0, objectType=triangle,
      property={colorValue,0,1,0,0,1}
ModifyGizmo modifyobject=triangle0, objectType=triangle,
      property={colorValue,1,0,1,0,1}
ModifyGizmo modifyobject=triangle0, objectType=triangle,
      property={colorValue,2,0,0,1,1}
```



Quad Objects

A quad object fills the sheet connecting the four vertices that are positioned sequentially in any direction starting from the first vertex. A quad object is always drawn filled. The fill color is determined by the internal color attribute, an embedded color attribute, or a global color attribute, in that order of precedence. Normals to the quad are calculated on a per-vertex basis. This gives rise to gradual shading when lighting calculations are enabled. For example, here are commands to create a quad:

```
AppendToGizmo/D quad={1,0,1,-1,0,1,-1,0,0,1,0,0}
ModifyGizmo modifyObject=quad0, objectType=quad, property={colorType,2}
ModifyGizmo modifyObject=quad0, objectType=quad,
      property={colorValue,0,1,0,0,1}
ModifyGizmo modifyObject=quad0, objectType=quad,
      property={colorValue,1,1.5259e-05,0.6,0.30425,1}
ModifyGizmo modifyObject=quad0, objectType=quad,
      property={colorValue,2,1.5259e-05,0.244434,1,1}
ModifyGizmo modifyObject=quad0, objectType=quad,
      property={colorValue,3,0,0,0,1}
```

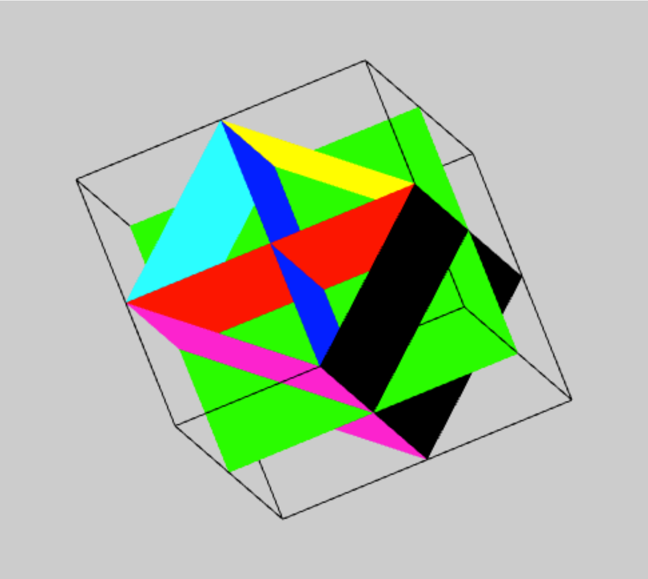


When creating a quad, you may find it easier to create the quad using simple coordinates in either the X, Y, or Z planes and then use translate and rotate operations to position the quad in the desired final orientation.

This table lists some basic quad examples with unit dimensions.

Quad Orientation	Command
XZ Plane	quad={1,0,1,-1,0,1,-1,0,-1,1,0,-1}
XY Plane	quad={1,1,0,-1,1,0,-1,-1,0,1,-1,0}
YZ Plane	quad={0,1,1,0,1,-1,0,-1,-1,0,-1,1}
Oblique Z Plane, +X, +Y	quad={1,0,1,1,0,-1,0,1,-1,0,1,1}
Oblique Z Plane, -X, +Y	quad={-1,0,1,-1,0,-1,0,1,-1,0,1,1}
Oblique Z Plane, -X, -Y	quad={-1,0,1,-1,0,-1,0,-1,-1,0,-1,1}
Oblique Z Plane, +X, -Y	quad={1,0,1,1,0,-1,0,-1,-1,0,-1,1}

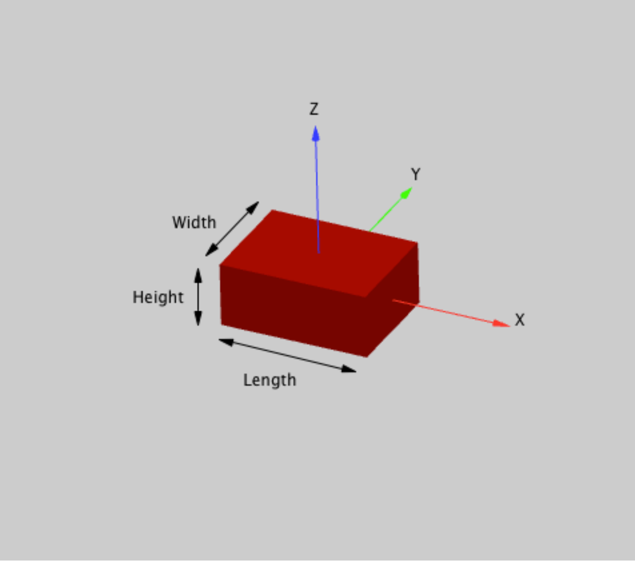
Here is what the seven quads described in the table look like after applying different colors:



Box Objects

A box object is defined by length along the X direction, width along the Y direction, and height along the Z direction. Boxes are centered on the origin and are always drawn filled. The fill color is determined by the internal color attribute, an embedded color attribute, or a global color attribute, in that order of precedence.

This illustration shows a box with length = 1, width = 0.75, height = 0.5:



To position a box in orientations where the sides are not parallel to the axes or not centered on the origin, add rotate and/or translate operations to the display list before the box item.

Sphere Objects

A sphere is a quadric object, meaning that it can be generated by a quadratic polynomial. Internally a quadric object is composed of a list of triangles and quads that approximate the curved surface.

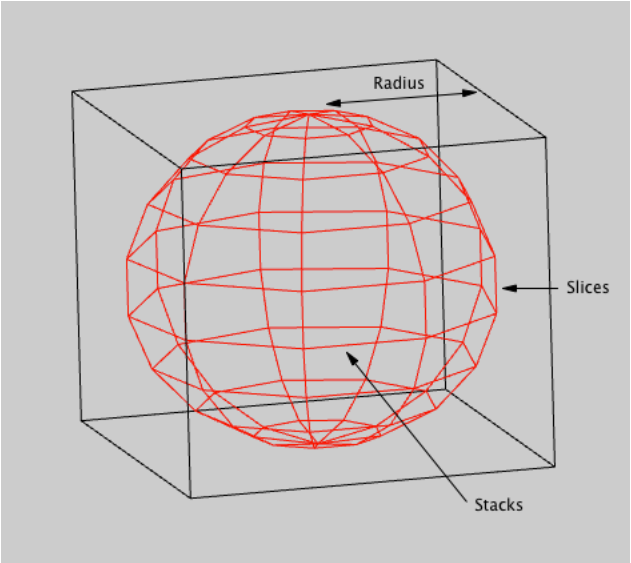
A newly-created sphere is centered on the origin and both poles are on the Z axis. The numbers of slices (subdivisions around the Z axis; divisions of longitude) and stacks (subdivisions along the Z axis; divisions of latitude) determine the smoothness of the sphere. The greater the numbers of subdivisions, the smoother the sphere's appearance. Small values produce other geometric objects (see illustrations below). The more subdivisions you use, the more time it takes to draw the sphere, but in most applications this is not important unless the sphere is replicated many times, such as when used as a marker in a scatter plot.

By default, spheres are initially drawn filled. The fill color is determined by the internal color attribute, an embedded color attribute, or a global color attribute, in that order of precedence.

Use translate and rotate operations to position the sphere in other locations and orientations.

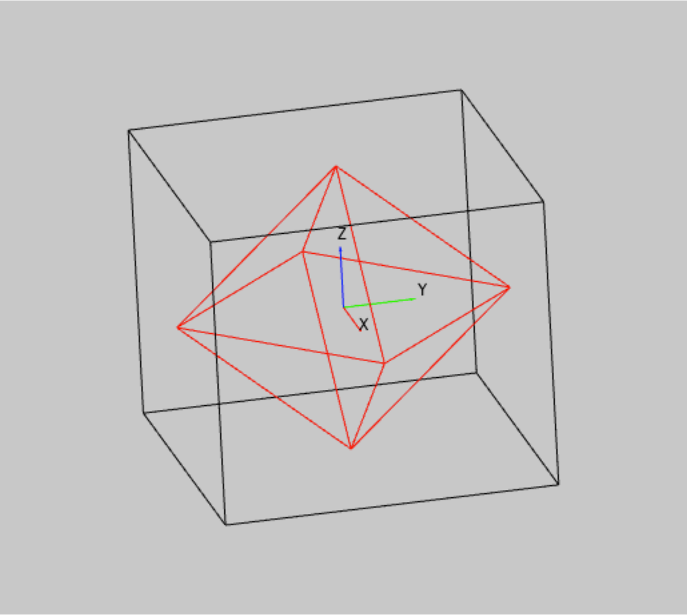
These commands generate a sphere with radius=1, stacks=10, and slices=10 and set its drawing style to lines:

```
AppendToGizmo/D sphere={1,10,10}
ModifyGizmo modifyObject=sphere0, objectType=Sphere,
      property={useGlobalAttributes,0}
ModifyGizmo modifyObject=sphere0, objectType=Sphere,
      property={drawStyle,100011}
```



Create other geometric shapes by using small values for the number of stacks and keeping slices = 2. These commands generate an octahedron with radius=1, stacks=4 and slices=2 and set its drawing style to lines:

```
AppendToGizmo/D sphere={1,4,2}
ModifyGizmo modifyObject=sphere0, objectType=Sphere,
      property={useGlobalAttributes,0}
ModifyGizmo modifyObject=sphere0, objectType=Sphere,
      property={drawStyle,100011}
```



You can create other geometric shapes by increasing the number of stacks and keeping slices=2.

Cylinder Objects

A cylinder is a quadric object, meaning that it can be generated by a quadratic polynomial. Cylinders are constructed from slices and rings. Specifying more slices creates a smoother cylinder. Initially, the cylinder axis is centered on the Z axis, height is in the positive Z direction, and the cylinder base is in the XY-plane.

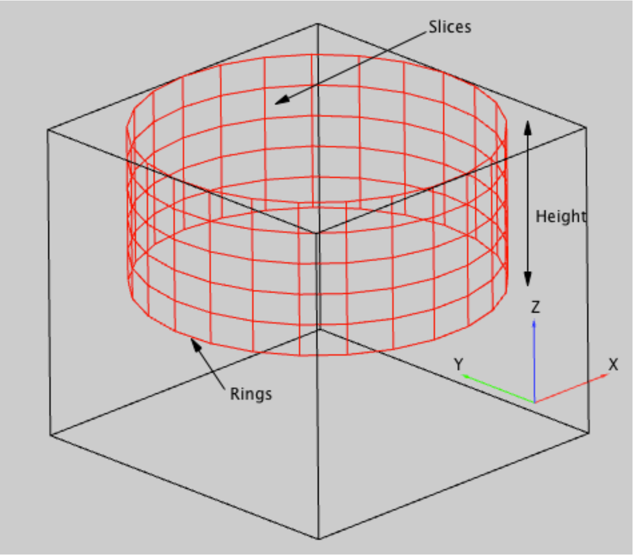
Create conical objects (see illustration below) by specifying different values for the base radius and top radius parameters.

By default, cylinders are initially drawn filled. The fill color is determined by the internal color attribute, an embedded color attribute, or a global color attribute, in that order of precedence.

Use translate and rotate operations to position the cylinder in other locations and orientations.

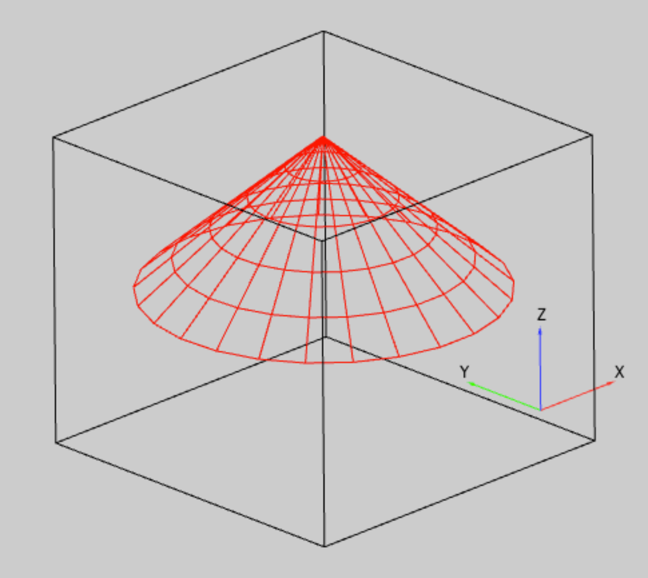
These commands generate a cylinder with baseRadius=1, topRadius=1, height=1, slices=25, and rings=5 and set its drawing style to lines:

```
AppendToGizmo/D cylinder = {1,1,1,25,5}
ModifyGizmo modifyObject=cylinder0, objectType=Cylinder,
      property={useGlobalAttributes,0}
ModifyGizmo modifyObject=cylinder0, objectType=Cylinder,
      property={drawStyle,100011}
```



A cone created using the same command but with topRadius=0:

```
AppendToGizmo/D cylinder = {1,0,1,25,5}, name=cone0
ModifyGizmo modifyObject=cone0, objectType=Cylinder,
      property={useGlobalAttributes,0}
ModifyGizmo modifyObject=cone0, objectType=Cylinder,
      property={drawStyle,100011}
```



You can create other types of cylindrical object shapes by specifying a small number for slices. Create a triangular cylinder with slices = 3; create a square cylinder or open-ended box with slices = 4. Use different values for baseRadius or topRadius to create pyramid shapes.

Disk Objects

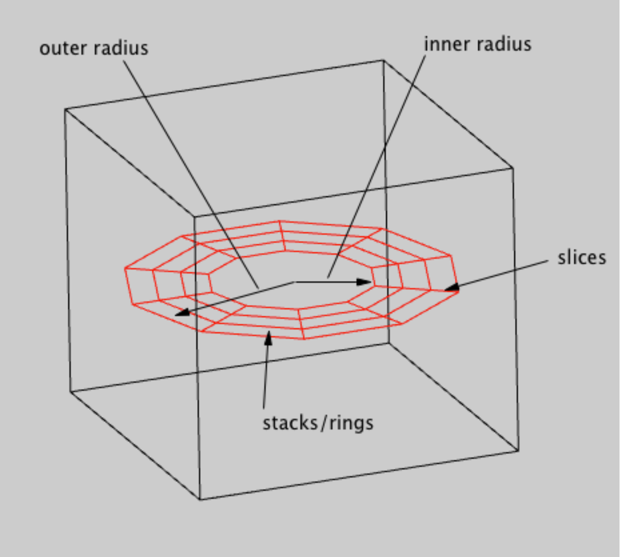
A disk is a quadric object, meaning that it can be generated by a quadratic polynomial. The disk is initially located in the XY plane centered on the origin. Create a smoother disk by specifying a greater number for slices.

By default, disks are initially drawn filled. The fill color is determined by the internal color attribute, an embedded color attribute, or a global color attribute, in that order of precedence.

Use translate and rotate operations to position the disk in other locations and orientations.

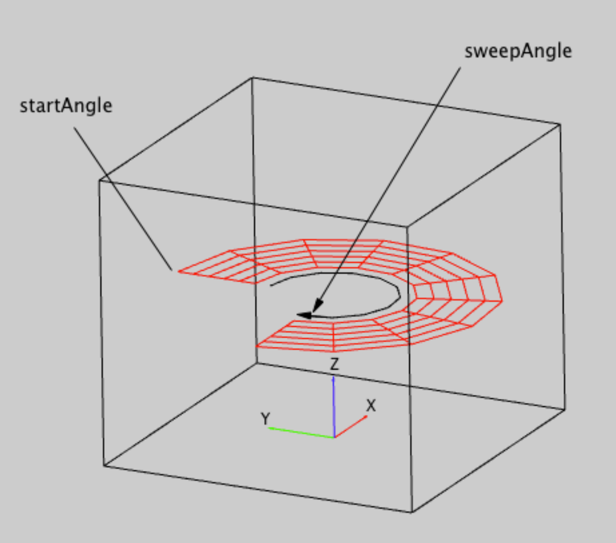
These commands generate a disk with innerRadius=0.5, outerRadius=1, slices=10, stacks=5, startAngle=0, sweepAngle=360 and set its drawing style to lines:

```
AppendToGizmo/D disk={0.5,1,10,5,0,360}
ModifyGizmo modifyObject=disk0, objectType=Disk,
      property={useGlobalAttributes,0}
ModifyGizmo modifyObject=disk0, objectType=Disk, property={drawStyle,100011}
```



This command changes the disk to a partial disk with sweepAngle set to 270 instead of 360. SweepAngle is specified in degrees clockwise from startAngle:

```
ModifyGizmo modifyObject=disk0, objectType=Disk, property={sweepAngle,270}
```



String Objects

You can use standard Igor annotations in Gizmo as you do in graphs.

Annotations are 2D text graphics that lie flat in a plane in front of all 3D graphics. They are well suited for general labeling purposes. To create an annotation choose Gizmo→Add Annotation and choose TextBox from the Annotation pop-up menu. For backward compatibility, Gizmo still supports string objects and they are useful if you want 3D graphics. Annotations are preferrable for general labeling.

For further discussion of annotations versus string objects, see **Changes to Gizmo Text** on page II-471. The rest of this section describes the original Gizmo string object feature.

String objects can be used for short text labels or annotations. You can create a string object using a command such as:

```
AppendToGizmo string="Hello", strFont="Geneva", name=string0
```

Like the primitive objects described above, string objects are 3D objects that are drawn in +/-1 display volume coordinates. You can use translate and rotate operations in order to place the string at the appropriate part of the graph and in the desired orientation.

The key to orienting strings correctly is understanding that they originate at the origin, character advance is in the X direction and text height is in the Y direction.

Multiple lines are not supported so a given string object results in a single line of text only.

ColorScale Objects

Gizmo colorscale objects were used in Igor Pro 6 and before to create 3D color scale graphics. You can now use standard Igor annotations as you do in graphs.

Annotations are 2D graphics that lie flat in a plane in front of all 3D graphics. They are well suited for general labeling purposes. To create a color scale as an annotation choose Gizmo→Add Annotation and choose ColorScale from the Annotation pop-up menu. For backward compatibility, Gizmo still supports colorscale objects and they are useful if you want 3D graphics. Annotations are preferrable for general labeling.

The rest of this section describes the original Gizmo ColorScale object.

Color scale objects are designed to provide an association between a sequence of colors and a numeric scale. You can create a color scale object using a command such as:

```
AppendToGizmo colorScale=colorScale0
```

You would typically following this with ModifyGizmo commands.

Color scales are also drawn in +/-1 display volume coordinates and by default appear planar though you can assign to them a positive depth value to make the color scale into a full 3D object.

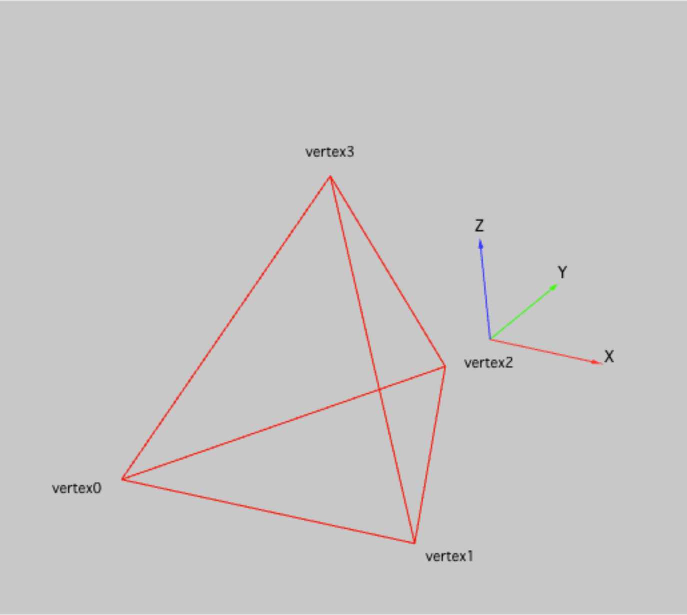
You can choose the sequence of colors to be based on a built-in color table or provide your own sequence using a color wave.

A color scale can be tied to a wave-based data object such as a surface plot but can also be independent on all objects in the plot. You can create multiple color scale objects in the same plot.

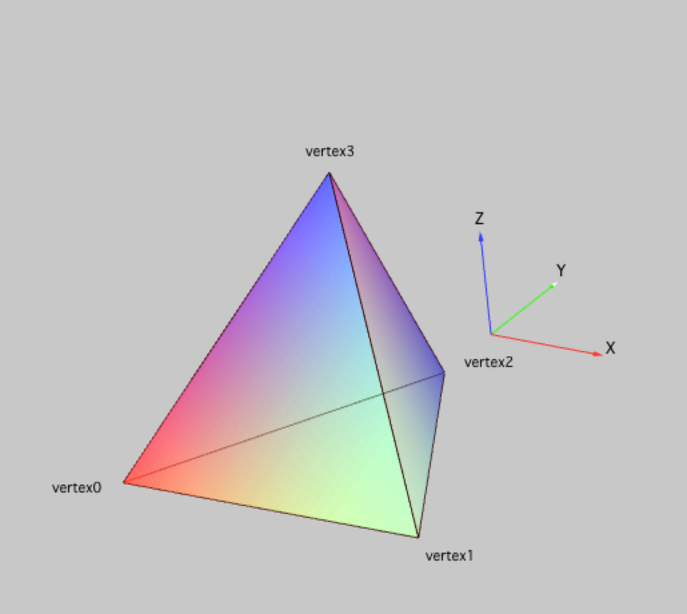
Tetrahedron Objects

A tetrahedron is defined by 4 vertices. These commands generate a tetrahedron and set its drawing style to lines:

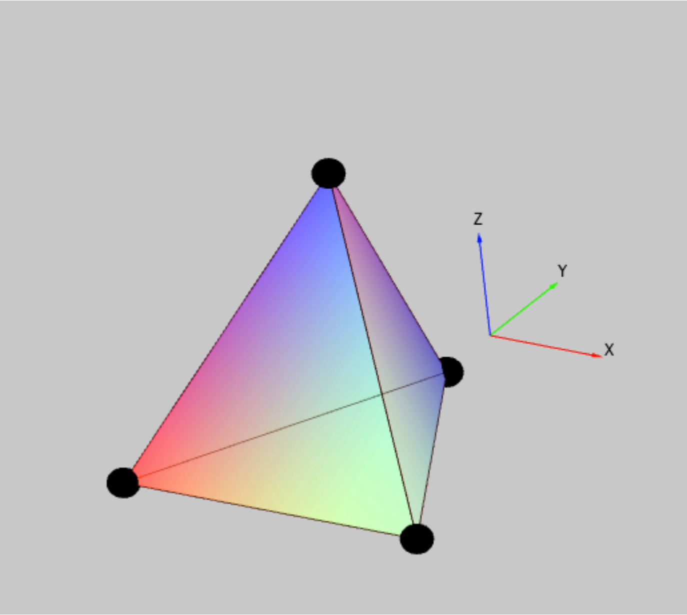
```
AppendToGizmo/D tetrahedron=tetrahedron0
ModifyGizmo ModifyObject=tetrahedron0, objectType=tetrahedron,
      property={vertex0,-1,-1,-1}
ModifyGizmo ModifyObject=tetrahedron0, objectType=tetrahedron,
      property={vertex1,1,-1,-1}
ModifyGizmo ModifyObject=tetrahedron0, objectType=tetrahedron,
      property={vertex2,0,1,-1}
ModifyGizmo ModifyObject=tetrahedron0, objectType=tetrahedron,
      property={vertex3,0,0,1}
```



You can turn on filling and specify a color for each vertex to produce a tetrahedron like this:



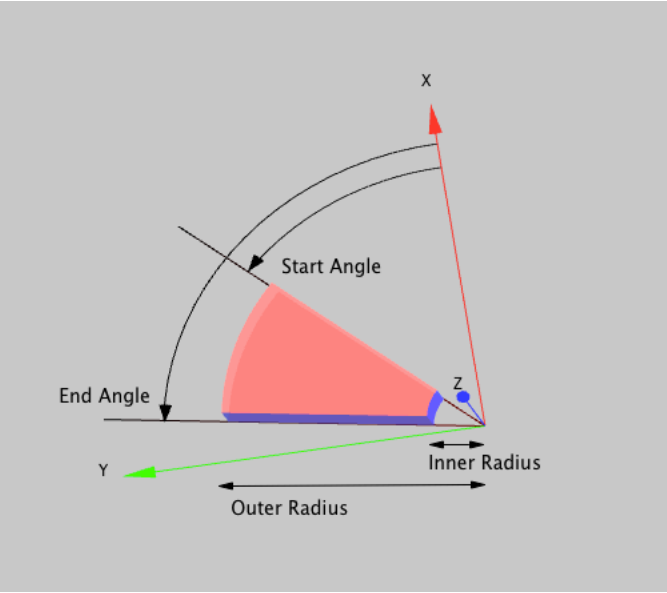
You can also specify that a sphere is to be drawn at each tetrahedron vertex:



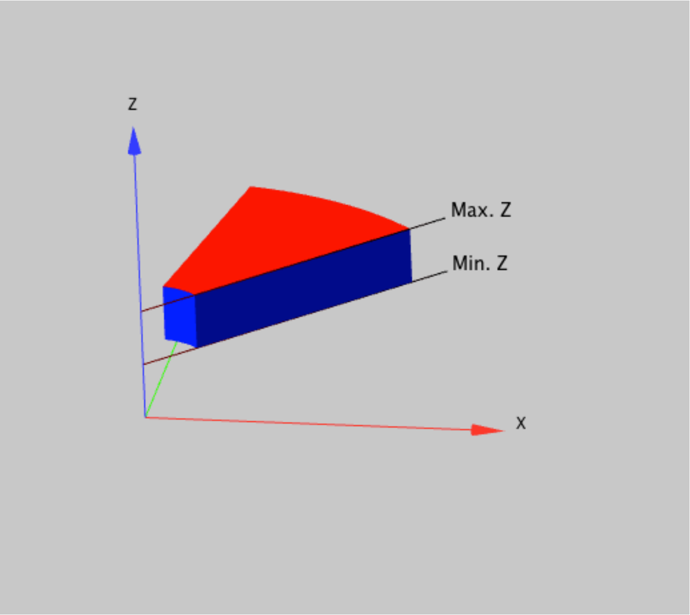
Pie Wedge Objects

A pie wedge object is defined by two angles, two radii and two Z values. You can use multiple pie wedge objects to create many types of 3D pie charts.

This illustration shows a pie wedge from the top:



This illustration shows a pie wedge from the side:



Gizmo Axis and Axis Cue Objects

Gizmo axis objects are used mostly with wave-based data objects such as surface plots and scatter plots. They allow you to indicate the range of data values and to set the scale of data objects.

Axis cue objects show the orientation of the X, Y and Z directions. Axis objects show both orientation and numeric range.

Axis Cue Objects

Gizmo supports two axis cue objects: the default axis cue and the free axis cue. These objects indicate the orientation of the X, Y and Z axes.

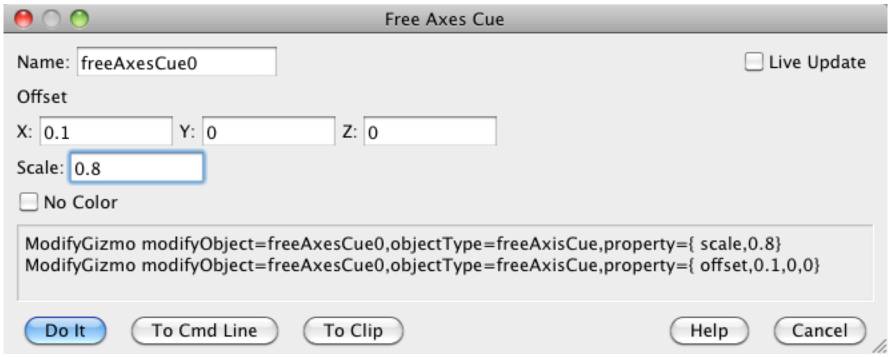
The default axis cue is a triplet axis object that is centered at the display volume origin. The X, Y and Z cue lines extend in the positive X, Y and Z directions of the display volume respectively.

There is just one default axis cue. It is used only to provide an indication of which direction is which. It does not support tick marks, tick mark labels or axis labels. To display the default axis cue, right-click the Gizmo display window and select Show Axis Cue.

The default axis cue object is drawn with the X axis in red, the Y axis in green and the Z axis in blue. The characters labeling the three axes are all drawn in black. Internally, the axes are drawn with emission color material which helps differentiate the axis cue from the background.

A free axis cue object can be drawn at any offset from the origin and at any scale. By default it has the same orientation as the default axis cue. Use a rotate operation if you want to rotate it.

You can create multiple free axis cues and even use them as markers in scatter plots. To add a free axis cue, click the + icon in the object list of the Gizmo info window and choose Free Axis Cue.



Name: freeAxesCue0

□ Live Update

Offset

X: 0.1 Y: 0 Z: 0

Scale: 0.8

□ No Color

```
ModifyGizmo modifyObject=freeAxesCue0,objectType=freeAxisCue,property={ scale,0.8}
ModifyGizmo modifyObject=freeAxesCue0,objectType=freeAxisCue,property={ offset,0.1,0,0}
```

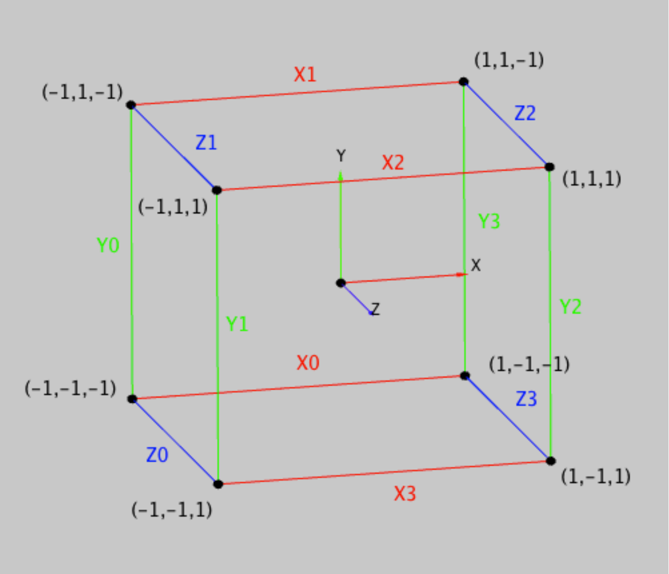
The free axis cue is colored the same as the default axis cue unless you disable its color by checking the No Color checkbox in the Free Axis Cue dialog. In this case it takes on the current color of the drawing environment as set by the previous color attribute in the display list. You also need a color material operation in the display list to apply a color to a free axis.

Axis Objects

As explained under **Gizmo Dimensions** on page II-421, a Gizmo display has an axis coordinate system that is superimposed on the +/-1 display volume. You set the range of this axis coordinate system using the Axis Range dialog, accessible via the Gizmo menu. Wave-based objects, such as scatter plots and surface plots, are displayed against this axis coordinate system. An axis object is a visual representation of it.

Gizmo supports three types of axis objects: box, triplet, and custom. Gizmo treats axis objects just like other objects that you add to the Gizmo display window with one exception: you must have at least one data object or drawing object on the display list for an axis object to be meaningful.

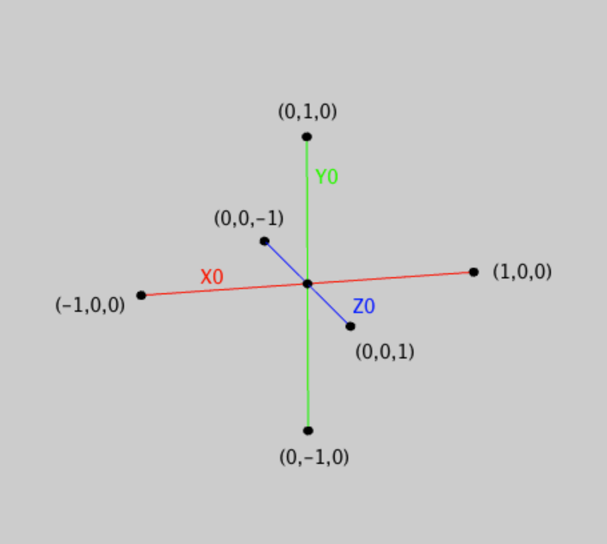
The corners of the box axis object correspond to the corners of the display volume. Thus the location of each corner in display volume space is -1 or +1 in each dimension. In this diagram the coordinates shown are the display volume coordinates of the corners of the box axes.



Box axes consist of 12 axes named X0, X1, X2, X3, Y0, Y1, Y2, Y3, Z0, Z1, Z2 and Z3. You can control each axis individually, assign it a color, range, tick marks, and tick mark labels. You can also add grid lines or paint any of the six sides of the box.

Triplet axes are a system of three orthogonal axes that intersect at the origin and span the +/-1 display volume. You can control each axis individually, assign its tick marks, tick mark labels and color. The origin of the axes is the center of the -/+1 display cube.

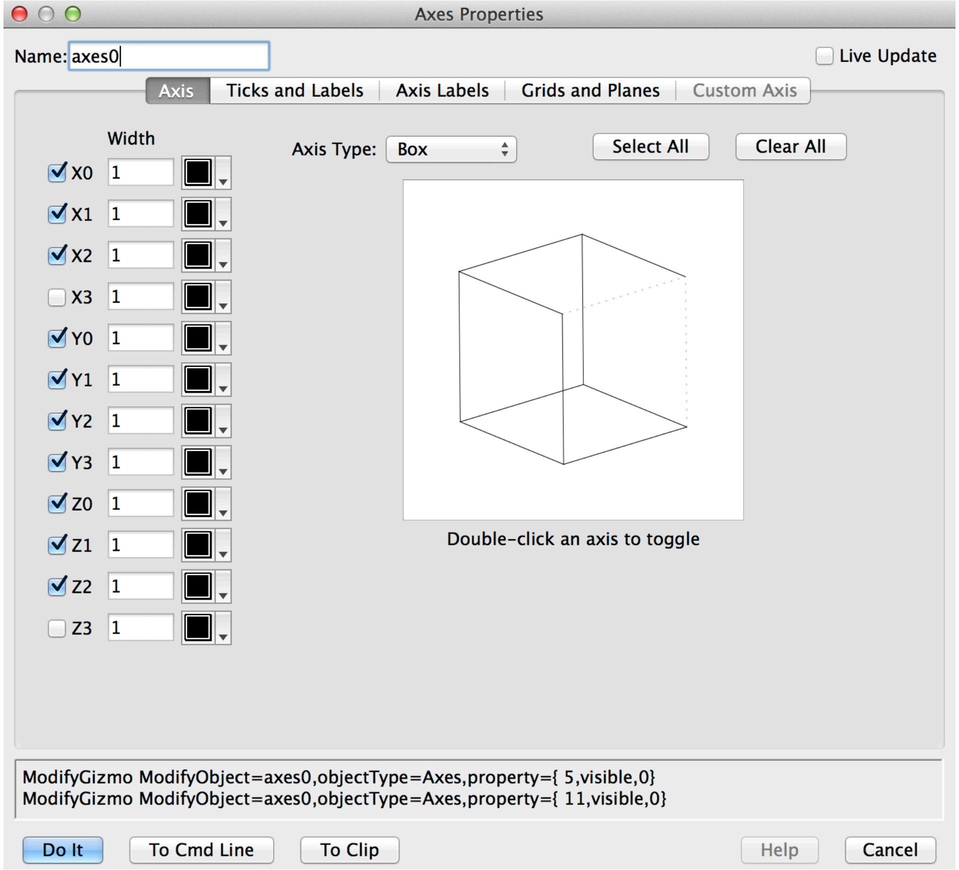
Each triplet axis is identified by name, X0, Y0, and Z0. This diagram shows display volume coordinates for triplet axes and the name of each axis:



A custom axis is a single line in space connecting your chosen start and end points as specified in -1/+1 display volume coordinates. Unlike box and triplet axes which reflect the global axis range as set by Gizmo→Axis Range, a custom axis has its own, independent axis range. You can add tick marks and tick mark labels to the custom axis as to any other axis.

By default, Gizmo attempts to find an appropriate position for tick mark labels. Using the manual position mode, you can take control over their position and orientation.

In most cases you will control the various axis settings using the Axes Properties dialog. When using the dialog it is important to keep track of the different axes that you can select with the checkboxes. To help you identify the various axes in your plot, the Axis tab of the dialog displays an outline of the axes in the same orientation as they appear in the Gizmo window. You can toggle the display of axes by double-clicking them.



```
ModifyGizmo ModifyObject=axes0,objectType=Axes,property={ 5,visible,0}
ModifyGizmo ModifyObject=axes0,objectType=Axes,property={ 11,visible,0}
```

See **AppendToGizmo** and **ModifyGizmo** for programming details.

Gizmo Wave Data Formats

Objects like surface plots, path plots, isosurfaces and voxelgrams are based on the data in a wave and are therefore called "wave-based data objects" or "data objects" for short. The wave supplying the data is called the "data wave" or the "source wave".

The following sections describe the data format requirements for the different data objects.

Scatter, Path, and Ribbon Data Formats

Scatter and path plots require a triplet wave, which is a 2D wave containing 3 columns for the X, Y, and Z coordinates of each vertex. A color wave for a scatter or path plot is a 2D wave in which each row specifies the color of the corresponding vertex in the data wave. The color wave has 4 columns which specify RGBA entries in the range of [0,1].

A ribbon plot also requires a triplet wave. A color wave for ribbon plot is the same as for scatter or path plots with one row of RGBA values per vertex. See also **ModifyGizmo** with the keyword pathToRibbon and **Ribbon Plots** on page II-462.

Surface Object Data Formats

The data wave format depends on the type of surface plot.

A simple surface plot is created from a 2D wave also known as a "matrix" or "matrix of Z values". The color wave for this type of surface plot is a 3D RGBA wave where the four layers in the wave contain the R, G, B and A components in the range of [0,1].

If you are plotting a parametric surface then the data wave is 3D wave containing three layers. At each row/column position, layer 0 contains the X coordinate, layer 1 the Y coordinate, and layer 2 the Z coordinate. The color wave for a parametric surface is a 3D RGBA wave that has the same number of elements in the X and Y dimensions as the data wave and where layer 0 contains the red component, layer 1 the green component, layer 2 the blue component and layer 3 the alpha component.

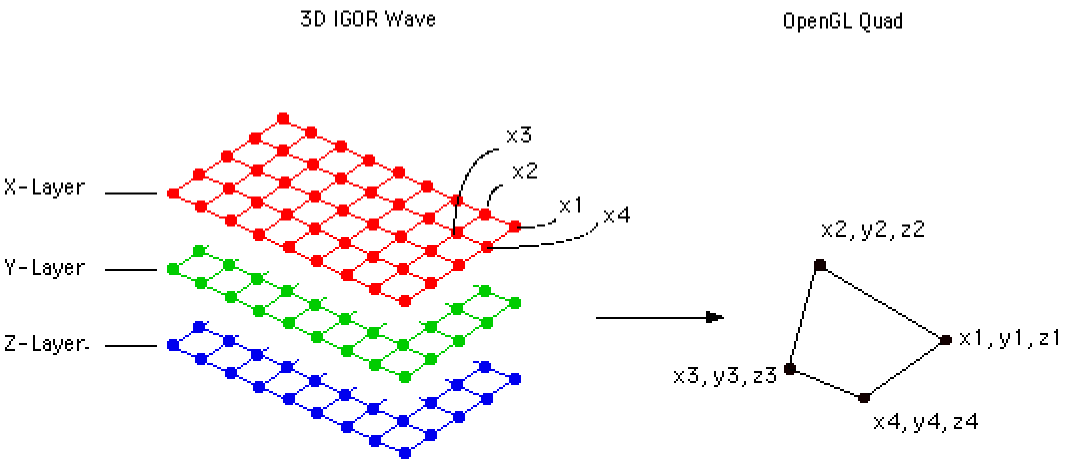
If you are plotting sequential quads (they could be the **ImageTransform** output of a Catmull-Clark B-Spline), the data wave is a 3D wave with four rows and three layers. Each row contains stores the four vertices of a quad and the three layers contain the X, Y, and Z coordinates.

If you are plotting disjoint quads the data wave is 2D with 12 columns corresponding to the X, Y, and Z values of the quad vertices taken in a counterclockwise direction.

The color wave for sequential quads and disjoint quads is a 4 column 2D wave in which the columns correspond to RGBA values in the range [0,1].

Parametric Surface Data Formats

A parametric surface is defined by equations in which the X, Y, and Z coordinates are calculated for, usually, a pair of parameters. To define a parametric surface for Gizmo you need to have a 3D wave containing X, Y, and Z layers. The only restriction on the dimensions of this wave is that it must not have 4 columns. Gizmo constructs the parametric surface from quads. Each quad is defined by four neighboring vertices as shown in the diagram below.



You can find examples of parametric surfaces in these demo experiments:

```
Igor Pro Folder:Examples:Visualization:Mobius Demo
Igor Pro Folder:Examples:Visualization:Spherical Harmonics Demo
Igor Pro Folder:Examples:Visualization:Gizmo Sphere Demo
```

Image Object Data Format

The data wave for an image object can be in one of three formats:

- A 2D matrix of Z values
- A 3D RGB wave of type unsigned byte with 3 layers
- A 3D RGBA wave of type unsigned byte with 4 layers

In the 3D formats, the red component is stored in layer 0, the green component in layer 1, and the blue component in layer 2. The alpha component, if any, is stored in layer 3. Each component value ranges from 0 to 255.

Isosurface and Voxelgram Object Data Formats

The data for an isosurface or a voxelgram object is a 3D volumetric wave.

Isosurfaces and voxelgrams do not use color waves.

NaN Values in Waves

In general, you should avoid using NaNs in data waves plotted in Gizmo. NaNs appears as holes in surfaces or as discontinuities in path plots. A surface object whose data wave contains one or more NaN values is drawn in the usual way except that all constituent triangles for which at least one vertex is a NaN are not drawn. When possible, it is best to display missing data in Gizmo using transparency (see **Transparency and Translucency** on page II-432).

Gizmo Data Objects

Data objects, also called "wave-based objects", are display objects representing data in Igor waves. They include **3D Scatter Plots**, **Path Plots**, **Surface Plots**, **Ribbon Plots**, **Isosurface Plots**, **Voxelgram Plots** and **3D Bar Plots**.

Data Object Scaling

Data objects are displayed against a set of X, Y and Z data axes. These data axes exist whether or not you add an axis object to the plot. The data axes fill the +/-1 display volume.

By default, the data axes are autoscaled to the range of data in all data objects in the plot. You can change the range of the data axes using Gizmo→Set Axis Range.

The following commands illustrate these points. Execute them in a new experiment to follow along:

```
// Create a Gizmo plot with a surface object
NewGizmo
ModifyGizmo showAxisCue=1
Make/O/N=(100,100) data2D = Gauss(x,50,10,y,50,15)
AppendToGizmo/D surface=data2D, name=surface0
ModifyGizmo setQuaternion={0.206113,0.518613,0.772600,0.302713}
```

If you now choose Gizmo→Set Axis Range, you can see that all data axes, which are now invisible, are in autoscale mode (Manual checkboxes are unchecked) and that the X and Y axes range from 0 to 99. These values come from the fact that the default X and Y values of the data2D wave range from 0 to 99 (default wave scaling). The Z axis range is based on the range of Z values in data2D. (If you opened the Axis Range dialog, click Cancel to dismiss it now.)

To help us visualize these data axes we add a box axis object with tick marks and tick mark labels:

```
// Add box axes with tick marks and tick mark labels
AppendToGizmo/D Axes=BoxAxes, name=axes0
ModifyGizmo ModifyObject=axes0, objectType=Axes, property={4,ticks,3}
ModifyGizmo ModifyObject=axes0, objectType=Axes, property={8,ticks,3}
ModifyGizmo ModifyObject=axes0, objectType=Axes, property={9,ticks,3}
```

If we change the values in the data, since the data axes are in autoscaling mode, the data still fills the axes (which always fill the display volume) after the change:

```
// Change the X and Y range of the data
SetScale x, 0, 10, "", data2D; SetScale y, 0, 10, "", data2D
```

Now we set the X and Y data axes to manual scaling mode but leave their ranges unchanged:

```
// Set the X and Y axes to manual scaling mode
ModifyGizmo setOuterBox={0,9.9,0,9.9,1.5286241e-11,0.001061033}
ModifyGizmo scalingOption=48
```

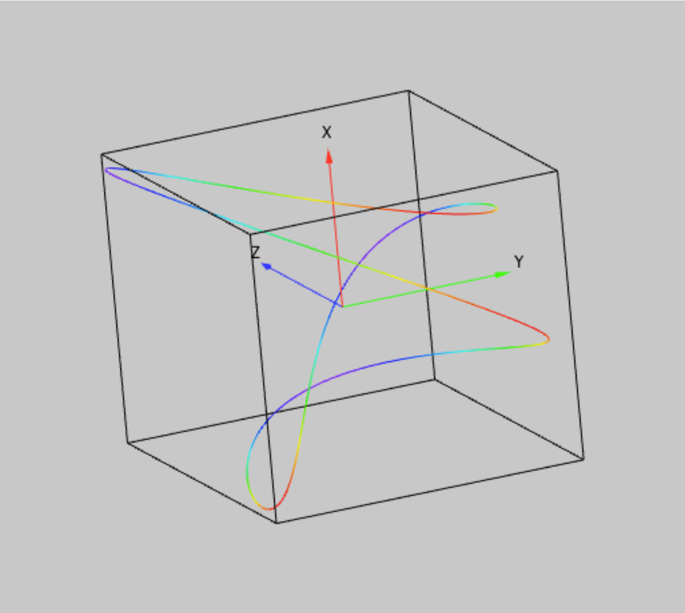
If we now change the range of the data, the data axes remain unchanged and the data no longer fills the axes:

```
// Change the X and Y range of the data
SetScale x, 0, 5, "", data2D; SetScale y, 0, 5, "", data2D
```

To recap, X, Y and Z data axes always exist, whether they are displayed or not. They always fill the +/-1 display volume. Data objects are displayed against the data axes which, by default, are autoscaled. Consequently, by default, data objects fill the display volume. If you change the data axes to manual scaling and change their range or change the range of the data, the data axes still fill the +/-1 display volume but the data objects no longer exactly fit.

Path Plots

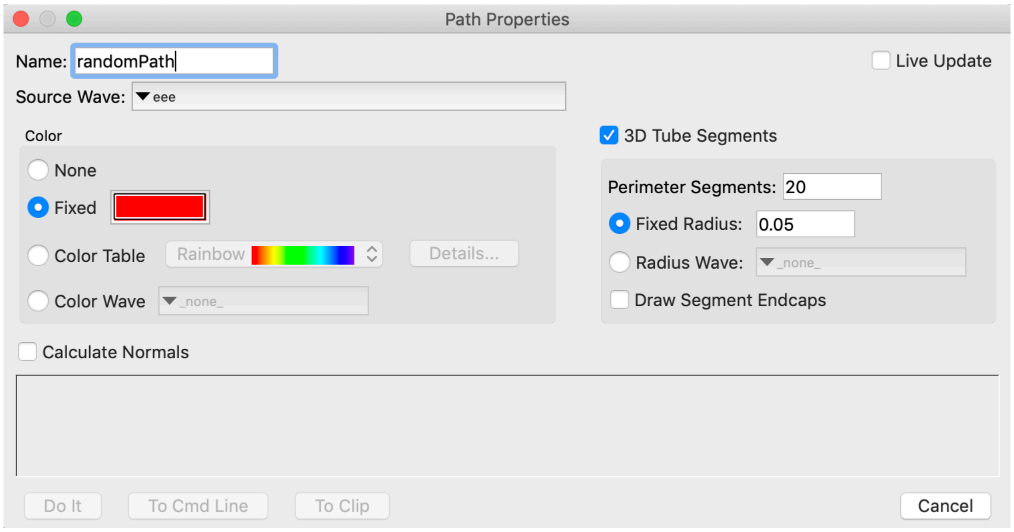
Each data point in a path plot is connected in sequential order by a straight line to each adjacent point.



To draw markers at the individual points, you need to append a scatter plot based on the same data wave.

A path plot requires a triplet wave, which is a 2D wave of 3 columns for X, Y, Z coordinates, respectively, of each vertex. A color wave for a path plot is a 2D wave in which each row specifies the color of the corresponding vertex in the data wave. The color wave has 4 columns which specify RGBA entries in the range of [0,1].

The Path Properties dialog looks like this:



Path Properties

Name: randomPath

□ Live Update

Source Wave: eee

Color

○ None

● Fixed 🟥

○ Color Table Rainbow

○ Color Wave _none_

☑ 3D Tube Segments

Perimeter Segments: 20

● Fixed Radius: 0.05

○ Radius Wave: _none_

□ Draw Segment Endcaps

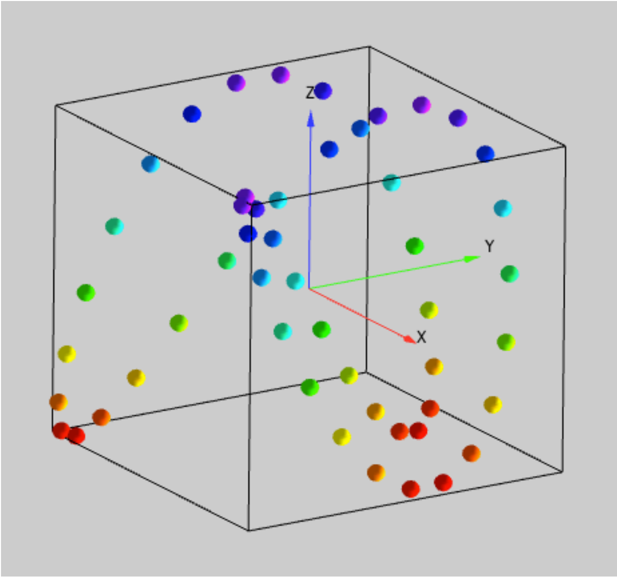
□ Calculate Normals

The full list of available options is given under **ModifyGizmo**.

You can find an example of 3D tube segments where the tube segments are used to display chemical bonds by choosing File→Example Experiments→Visualization→Molecule.

3D Scatter Plots

Each data point in a scatter plot is displayed as 3D marker.



If you want to connect the markers, you need to append a path plot object that uses the same triplet wave.

A scatter plot requires a triplet wave, which is a 2D wave of 3 columns for X, Y, Z coordinates, respectively, of each marker.

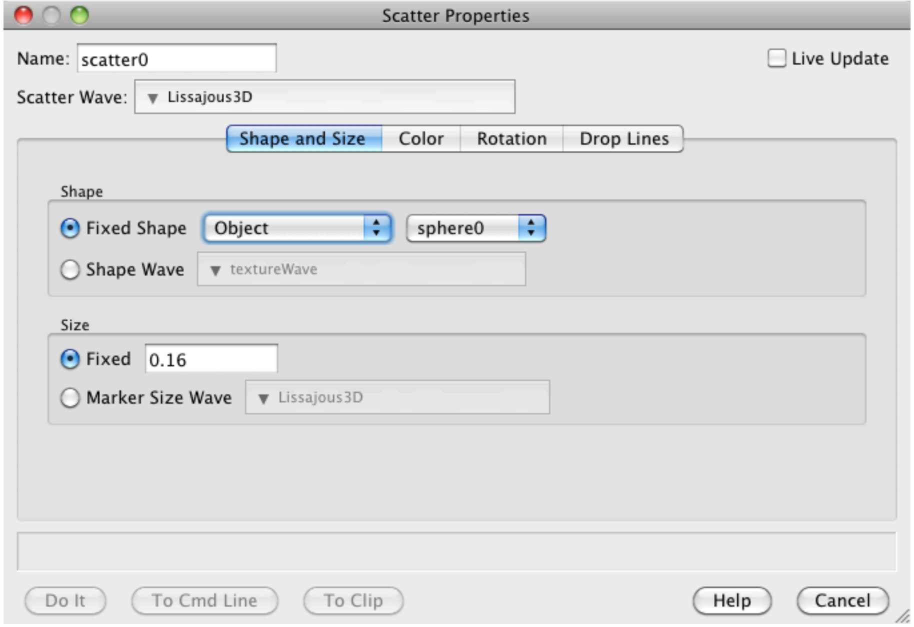
Each marker can be represented by a Gizmo object. You can select any one of the built-in drawing objects (e.g., box, sphere, cylinder or disk) or you can also select any object in the object list. This is useful when you have a very small number of points and you want to display each marker at high resolution. In that case you would add an object to the display list that has the required resolution and then choose this object as your marker for the path plot. Choose File→Example Experiments→Visualization→Molecule for an example.

You can specify the size of the scatter objects to be a constant size, or provide a wave that contains a size specification for each scatter point. The size is specified by a triplet wave where each row contains scale factors for the X, Y and Z dimensions of each marker.

You can also specify the rotation of each scatter point. The rotation is specified by a 2D wave containing four columns. The first column specifies the rotation angle in degrees and the remaining three columns specify the normalized rotation axis vector, just as in the rotate operation.

A scatter plot can be colored using a constant color, a color taken from one of the built-in color tables, or a user-specified color wave. A color wave for a scatter plot is a 2D wave in which each row specifies the color of the corresponding element in the data wave. The color wave has 4 columns which specify RGBA entries in the range of [0,1].

The Scatter Properties dialog allows you to set all properties of the object. If you use the dialog when the object is already in the display list then checking the Live Update box lets you to see all changes as you make them.



Scatter Properties

Name: scatter0

□ Live Update

Scatter Wave: Lissajous3D

Shape

● Fixed Shape: Object, sphere0

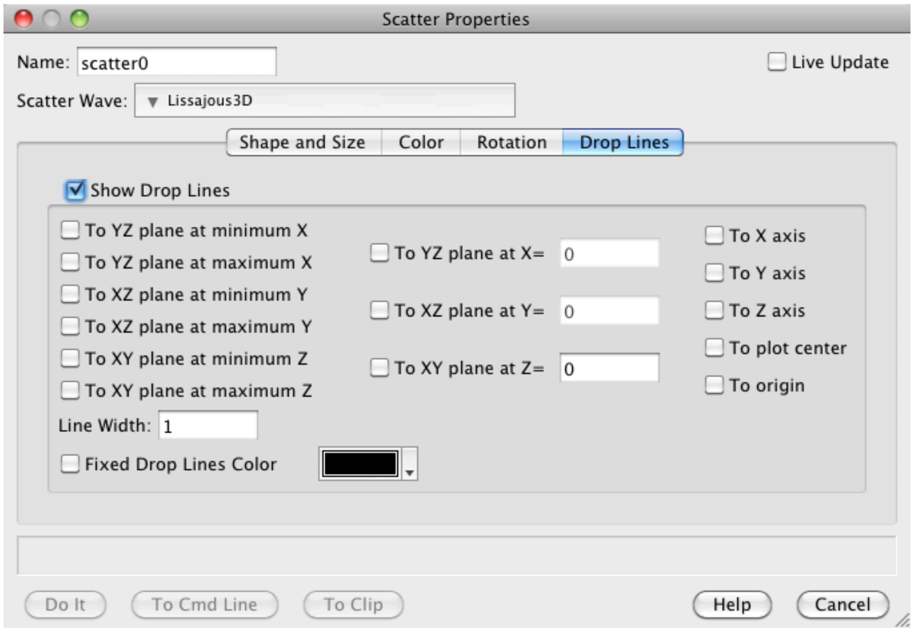
○ Shape Wave: textureWave

Size

● Fixed: 0.16

○ Marker Size Wave: Lissajous3D

One useful feature of scatter plots is the ability to draw a "drop line". Drop lines start at the center of each marker and extend to one of 14 possible points, lines or planes. You can choose any combination of drop lines from the Drop Lines tab of the Scatter Properties dialog.



Scatter Properties

Name: scatter0

□ Live Update

Scatter Wave: Lissajous3D

☑ Show Drop Lines

□ To YZ plane at minimum X
□ To YZ plane at maximum X
□ To XZ plane at minimum Y
□ To XZ plane at maximum Y
□ To XY plane at minimum Z
□ To XY plane at maximum Z

□ To YZ plane at X= 0
□ To XZ plane at Y= 0
□ To XY plane at Z= 0

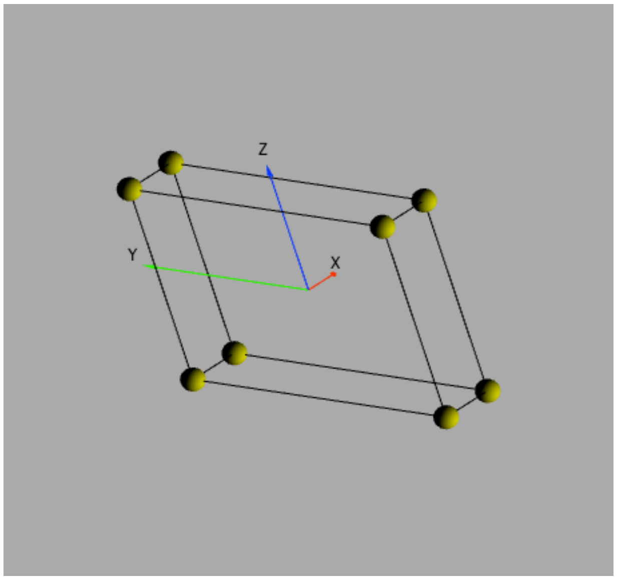
□ To X axis
□ To Y axis
□ To Z axis
□ To plot center
□ To origin

Line Width: 1

□ Fixed Drop Lines Color ■

The full list of available options is given under **ModifyGizmo**.

You can display crystal structures in Gizmo using scatter plots. Here is an example:



Crystals are easy to display as scatter objects that are centered at coordinates provided in a triplet wave. For example, you can use a sphere as the object drawn at each center. You can specify additional waves to control the color and size of each sphere.

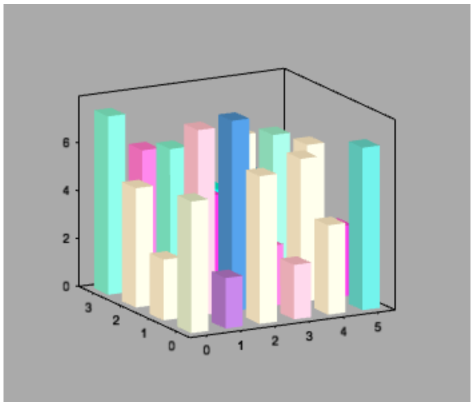
Igor provides two built-in transformations that convert triplet waves from crystallography coordinates to rectangular coordinates and vice versa. See the Crystal Demo experiment and the **WaveTransform** operation with the keywords crystalToRect and rectToCrystal.

3D Bar Plots

A 3D bar plot consists of a number of straight (parallel to all three axes) rectangular bars. The bars may be arranged on a regular grid or may be completely scattered within the volume.

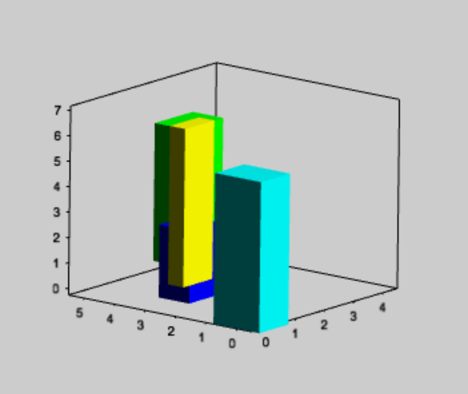
There are two modes for 3D bar plots: basic and refined.

In a basic 3D bar plot, the bars all start at zero and extend in the positive Z direction. Here is an example:



The basic 3D bar plot represents a 2D matrix of positive values. Each value is displayed as a zero-based bar according to its row/column in the matrix. All bars have the same width.

In a refined bar plot, bars can be drawn at arbitrary positions and all bar dimensions are under your control. Here is an example:



The refined 3D Bar plot displays requires a 6-column input wave where each row represents a single bar and the columns contain the following information:

Column 0:	The X center of the bar
Column 1:	The Y center of the bar
Column 2:	The lower Z value
Column 3:	The upper Z value
Column 4:	The width of the bar in the X direction
Column 5:	The width of the bar in the Y direction

Using the refined mode it is possible to create stacked and overlapping 3D bars plots as well as bars of varying sizes.

To find out more open the 3D Bar Plot Demo experiment.

Gizmo Image Plots

A Gizmo Image is a form of a quad object that is internally textured by image data.

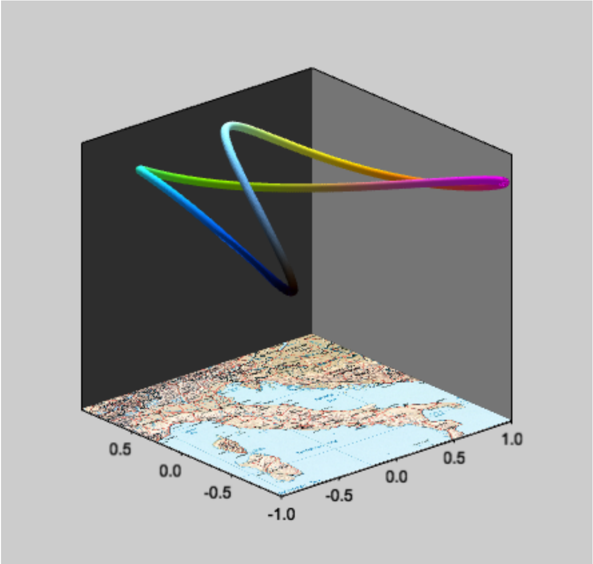
The image source wave can be in one of three formats:

- A 2D matrix of Z values
- A 3D RGB wave of type unsigned byte with 3 layers
- A 3D RGBA wave of type unsigned byte with 4 layers

The latter two formats are created by the **ImageLoad** operation.

By default, the image is displayed at the bottom of the display volume but using rotation, translation and scaling options you can place the image anywhere within it. If you need to register the image relative to the data you can modify the axis range of the Gizmo display or change the scaling of the image. The scaling is uniform about the center of the image.

Igor includes a flight path example that combines a dense scatter plot with a Gizmo image object. In this case the image consists of a map:



To find out more open the Flight Path Demo experiment.

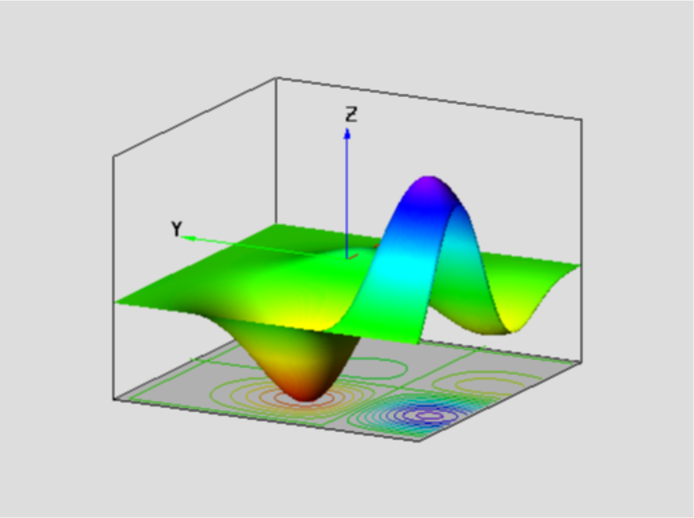
Surface Plots

A surface plot consists of a sheet connecting a grid of data values. You can specify surface colors from built-in color tables or custom color waves; see **Color Waves** on page II-430 for details. In addition to the surface, data values can also be displayed as points or as grid lines which can have their own color specification.

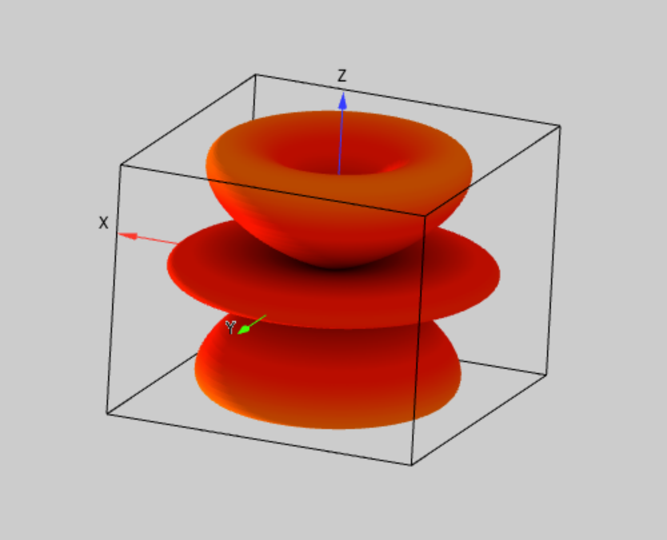
Typically data consist of an MxN 2D matrix of Z values which comprise the surface; see Surface Object Data Formats for details. Also supported are parametric surfaces which are 3D waves where each successive layer contains of the X, Y, and Z values in order; see **Parametric Surface Data Formats** on page II-452 for details.

The full list of available options is documented under **ModifyGizmo**.

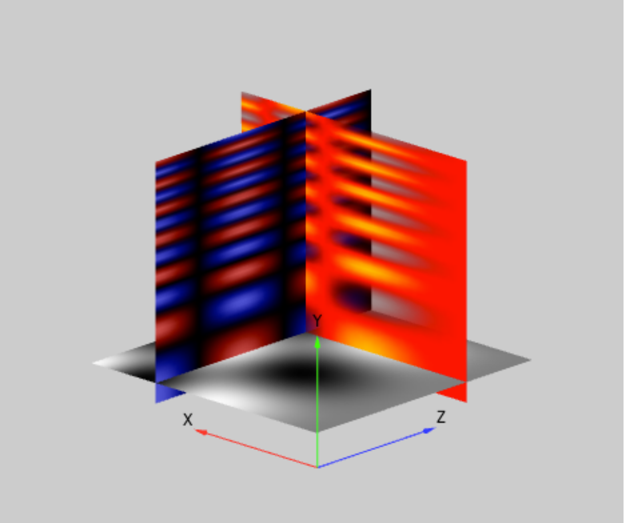
This example shows a surface plot with a contour map at the bottom:



This example shows a parametric surface plot of a spherical harmonic function:



This example shows orthogonal slices of volumetric data:



To display the data by sampling on non-orthogonal slices you can create an arbitrary parametric surface and color it using data sampled at the vertices of the parametric surface. Here is an example.

Suppose you have the following simple 3D data set:

```
Make/O/N=(100,100,100) ddd=z
```

Now create a parametric surface that describes a sphere in this range of data. Here is the MakeSphere function from the GizmoSphere demo experiment:

```
Function MakeSphere(pointsx,pointsy)
   Variable pointsx,pointsy

   Variable i,j,rad
   Make/O/N=(pointsx,pointsy,3) sphereData
   Variable anglePhi,angleTheta
   Variable dPhi,dTheta

   dPhi=2*pi/(pointsx-1)
   dTheta=pi/(pointsy-1)
   Variable xx,yy,zz
   Variable sig

   for(j=0;j<pointsy;j+=1)
      angleTheta=j*dTheta
      zz=sin(angleTheta)
      if(angleTheta>pi/2)
         sig=-1
      else
         sig=1
      endif
      for(i=0;i<pointsx;i+=1)
         anglePhi=i*dPhi
         xx=zz*cos(anglePhi)
         yy=zz*sin(anglePhi)
         sphereData[i][j][0]=xx
         sphereData[i][j][1]=yy
         sphereData[i][j][2]=sig*sqrt(1-xx*xx-yy*yy)
      endfor
   endfor
End
```

You can execute the function like this:

```
MakeSphere(100,100)
```

Then shift the result to the limits of the sample data using

```
sphereData*=48          // Slightly inside the boundary
sphereData+=50
```

Next create a scale wave that will contain the samples of the data at the vertices of the parametric surface:

```
Make/N=(100,100)/O scaleWave
scaleWave=Interp3D(ddd,sphereData[p][q][0],sphereData[p][q][1],sphereData[p][q][2])
```

To create a color wave we first open a Gizmo window and then have Gizmo create the color wave:

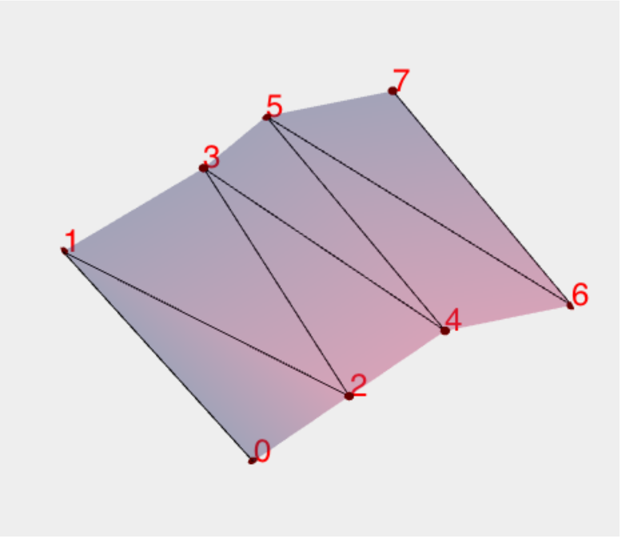
```
NewGizmo
AppendToGizmo surface=root:sphereData,name=surface0
ModifyGizmo modifyObject=surface0,objectType=surface,property={ srcMode,4}
ModifyGizmo makeParametricColorWave={sphereData,scaleWave,Rainbow,0}
```

Apply the color wave to the surface:

```
ModifyGizmo modifyObject=surface0,objectType=surface,
                         property={ surfaceColorType,3}
ModifyGizmo modifyObject=surface0,objectType=surface,
                         property={ surfaceColorWave,root:sphereData_C}
```

Ribbon Plots

In a ribbon plot, data points are connected by a surface that defines the ribbon object. A ribbon is constructed from a list of triangles with alternating vertices as shown here:



To display the individual points or connections, you need to append a scatter plot or a path plot.

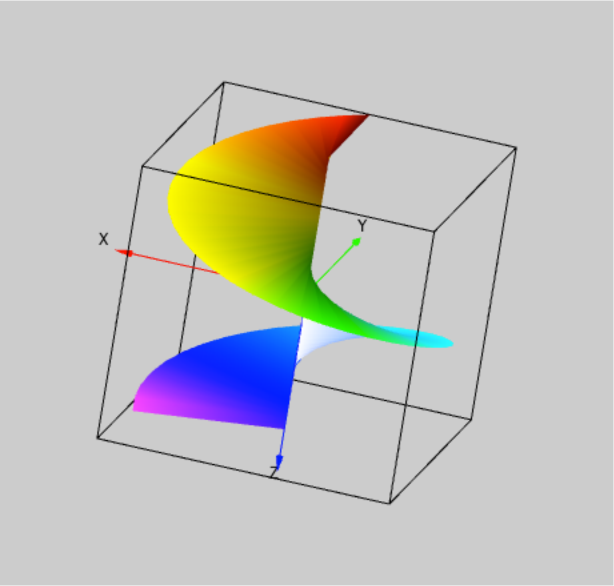
Data for a ribbon plot consist of a Nx3 matrix of values. Each row contains the X, Y, and Z values for the spatial coordinates of a point on the edge of the ribbon. The coordinates for each alternating edge of the ribbon follow in sequential order. The order of vertices for a ribbon is shown in the illustration above.

A ribbon must have at least four vertices and the total number of vertices must be even.

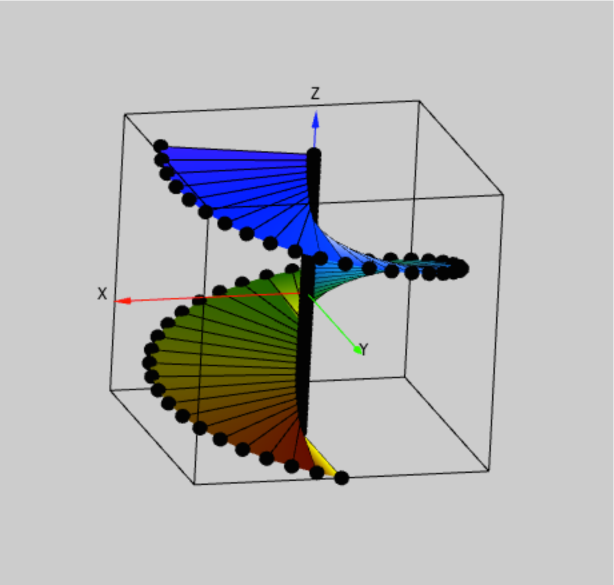
A ribbon plot can be colored using a constant color, a color taken from one of the built-in color tables, or a user-specified color wave. A color wave for a ribbon plot is a 2D wave in which each row specifies the color of the corresponding element in the data wave. The color wave has 4 columns which specify RGBA entries in the range of [0,1].

The full list of available options is given under **ModifyGizmo**.

This is an example of a simple ribbon plot:



This is an example of a ribbon plot overlayed with path and scatter plots of the same data in order to show the positioning of data points along the ribbon edges:



Voxelgram Plots

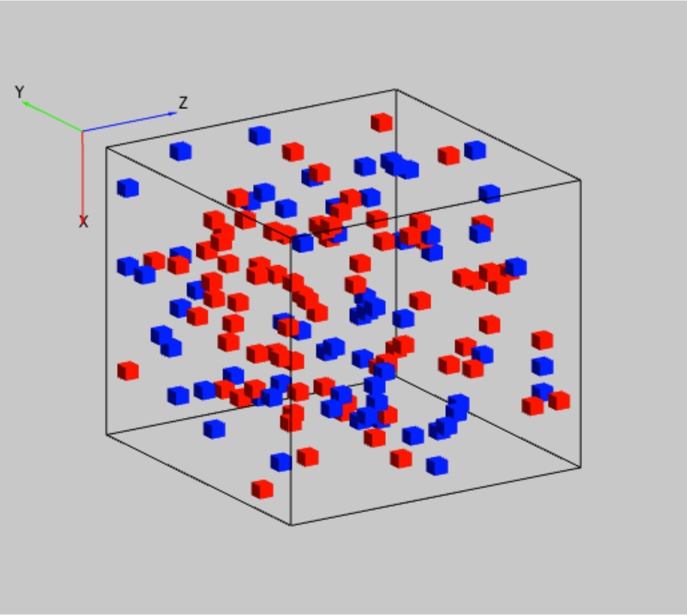
"Voxel" is short for "volume element".

A voxelgram is a representation of a 3D wave that uses color to indicate the wave elements containing certain values. You specify 1 to 5 values and, for each value, an associated RGBA color. If a given wave element's value matches one of the specified values within a specified tolerance, Gizmo displays that element using the associated RGBA color. If a wave element matches none of the specified levels then it is not displayed at all.

Voxels can be represented by cubes or points.

The full list of available options is given under **ModifyGizmo**.

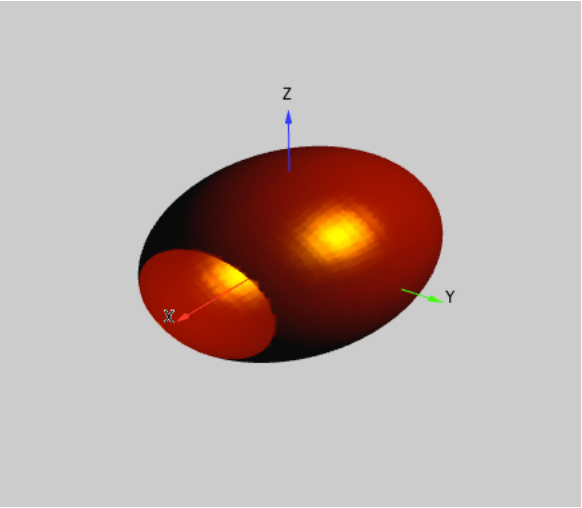
This example shows a voxelgram that uses two specified values:



Isosurface Plots

An isosurface is a 3D analog of 2D contour line. It is a surface drawn at the specified scalar value called the "isovalue". The surface may be represented by any combination of colors, grids, or points.

Here is an example of a translucent isosurface:



Isosurface generation is computationally intensive because it subdivides the volume into tetrahedra. If the isovalue is found in any tetrahedron, the intersection (one or two triangles) is calculated. The isosurface itself is composed of the collections of these triangles.

Isosurfaces are constructed from 3D volumetric data waves.

The full list of available options is given under **ModifyGizmo**.

Printing Gizmo Windows

You can print the image in the Gizmo display window by choosing File→Print when the Gizmo display window is the front window.

The window is printed at a multiple of screen resolution. The multiple is controlled by the Default Output Resolution Factor in the Gizmo section of the Miscellaneous Settings dialog which you can access via the Misc menu. The factory default value for this multiple is 2.

You can override the default printing resolution by executing:

```
ModifyGizmo outputResFactor = n
```

where n is a positive integer, typically 1, 2, 4 or 8. This applies to the active window only. It affects subsequent printing and overrides the Default Output Resolution Factor setting. This setting is not stored in the recreation macro for the Gizmo window and therefore does not persist. The maximum value of n that will work depends on the amount of video memory (VRAM) that you have in your graphics hardware.

You can also improve the output by anti-aliasing objects. To do this, add to the display list a blend function with GL_SRC_ALPHA and GL_ONE_MINUS_SRC_ALPHA and add an enable operation with GL_LINE_SMOOTH. You can also enable the GL_POINT_SMOOTH to smooth points in a scatter object.

If you are unable to print an image from Gizmo you may have run out of VRAM. This may produce a blank or distorted graphic. Some of the things that you should try are:

- Close any other Gizmo window that you might have open.
- Reduce the size of the Gizmo display window.
- Reduce the resolution as set by the Default Output Resolution Factor setting or via ModifyGizmo outputResFactor.
- If you are working on a system with more than one monitor move the display window to the one driven by a graphics card with the most VRAM.
- Run the experiment on hardware that has more VRAM.

Exporting Gizmo Windows

You can export a Gizmo plot using one of these techniques:

- Choose File→Save Graphics to export to a PNG, JPEG or TIFF file. This generates a **SavePICT** command.
- Choose Edit→Export Graphics to export to the clipboard as PNG, JPEG or TIFF.
- Choose Edit→Copy. This exports to the clipboard using the settings last set in the Export Graphics dialog.
- Right-click and choose Copy to Clipboard from the contextual pop-up menu. This is the same as choosing Edit→Copy.

The **ExportGizmo** operation is also available for backward compatibility only. It is obsolete and you should use SavePICT instead.

The Export Graphics dialog and the SavePICT operation give you control of the output resolution as a multiple of screen resolution. Exporting at high resolution requires sufficient video memory (VRAM). Most hardware supports 2x (two times screen resolution). You may be able to increase resolution further depending on the available VRAM.

If you are unable to export an image from Gizmo you may have run out of VRAM. This may produce a blank or distorted graphic. Some of the things that you should try are:

- Close any other Gizmo window that you might have open.
- Reduce the size of the Gizmo display window.
- Reduce the resolution as set in the Export Graphics or Save Graphics dialogs.
- If you are working on a system with more than one monitor move the display window to the one

driven by a graphics card with the most VRAM.

- Run the experiment on hardware that has more VRAM.

Advanced Gizmo Techniques

There are many advanced Gizmo options that are, by default, hidden. To display them you need to check the Display Advanced Options Menus checkbox in the Gizmo section of the Miscellaneous Settings dialog which you can access via the Misc menu. Most Gizmo users will not need these options.

Group Objects

A Gizmo group object is an encapsulation of existing Gizmo objects and operations that are treated as a single object in the top-level Gizmo.

To work with group objects you must enable advanced Gizmo menus in Miscellaneous Settings options. If you want to try the example in this section, make sure that the Display Advanced Options in Menus checkbox is checked in the Gizmos section of the Miscellaneous Settings dialog (Misc menu).

The following example illustrates how you can use a group object to create a custom marker for a scatter plot.

Start a new Igor experiment and execute this command:

```
NewGizmo/JUNK=3
```

This creates a simple scatter plot with default red sphere markers.

Create a group object using the + icon below the object list in the info window.

Double-click the group0 object in the object list. This opens a new info window for the group with display, object and attribute lists. Add a blue sphere object with a radius of 0.15 and a red cylinder object with top and bottom radii of 0.05 and a height of 0.3 to the group. Drag the sphere and cylinder objects to the group's display list.

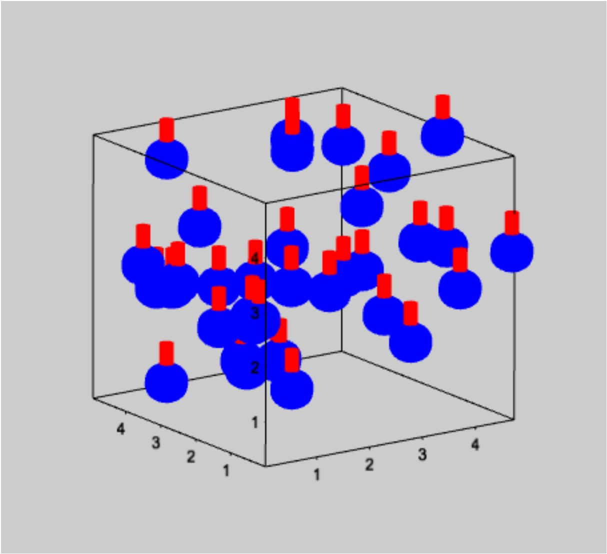
In the Gizmo0 Info window, drag the group0 object from the object list to the bottom of the display list. The sphere+cylinder group appears in the Gizmo plot.

Select the group0 object in the Gizmo0 Info window display list and delete it by clicking the - icon. The sphere+cylinder group disappears from the Gizmo plot.

Close the group0 info window.

In the Gizmo0 Info window, double-click the randomScatter object to open the Scatter Properties dialog. In the Shape and Size tab of the dialog, select Object from the Fixed Shape pop-up menu and group0 from the pop-up menu just to the right of it. Click Do It.

The resulting graph should look something like this:



A similar example of scatter objects can be found in the Scatter Arrows demo experiment where the marker object is a line that has a cylindrical body and cone head.

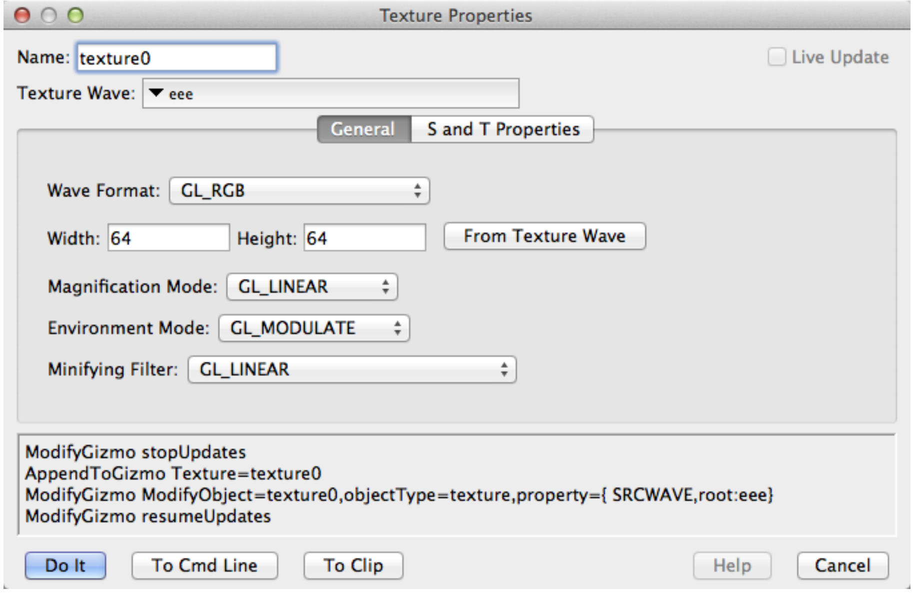
Texture Objects

Textures are used extensively in OpenGL so most graphics hardware are optimized to use them. An in-depth discussion of texture is beyond the scope of this document so we will focus on the use of textures in Gizmo. To work with texture objects you must enable the advanced Gizmo menus in Miscellaneous Settings.

A Gizmo texture object represents an OpenGL object that allows you to apply image information to surfaces of arbitrary shape. Textures in Gizmo are 1D or 2D. They can be applied to quad objects, quadric objects (spheres, cylinders and disks) and parametric surfaces. If you intend to use a texture on a simple quad you should use an image plot instead; see **Gizmo Image Plots** on page II-458.

You usually create a texture from an image that was loaded using **ImageLoad** and is in the form of an unsigned byte 3D RGB wave where the color for each pixel is stored in sequential layers. Before you create a Gizmo texture you need to convert the standard image format into a 1D wave where the color entries for each pixel are stored sequentially as RGB or RGBA. This conversion can be accomplished by **ImageTransform** with the keyword imageToTexture.

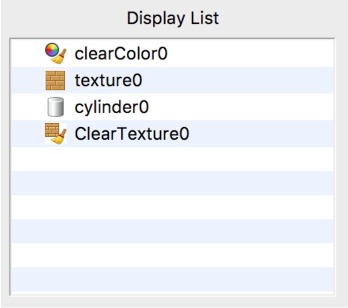
The Texture Properties dialog determines how the texture wave is converted into a texture object and how the texture is applied in a drawing.



```
ModifyGizmo stopUpdates
AppendToGizmo Texture=texture0
ModifyGizmo ModifyObject=texture0,objectType=texture,property={ SRCWAVE,root:eee}
ModifyGizmo resumeUpdates
```

Before invoking the dialog it is useful to know the dimensions of the texture wave, its packing format and the original image size. ImageTransform stores this information in the wave note of the texture wave. The dialog loads the texture information from the wave note if you click the From Texture Wave button.

Once you create a texture object you can apply it to other objects. For example, to apply texture0 to cylinder0 your display list should contain the following sequence:



clearColor0
texture0
cylinder0
ClearTexture0

where cylinder0 is preceeded by texture0 and followed by clearTexture0. You must check the Uniform Texture Coordinates checkbox in the Cylinder Properties dialog. The texture object is placed just before the cylinder and the clear texture operation follows so that subsequent drawings are free of textures.

The Gizmo Earth demo experiment illustrates the use of textures. In this case a high-resolution rectangular texture is added to a parametric surface representing the sphere. The parametric surface consists of 61x61 vertices.

Matrix4x4 Objects

Matrix4x4 objects are used with Gizmo operations that affect the transformation matrix. The encapsulate a 2D wave with 4 rows and 4 columns.

Gizmo Subwindows

You can embed a Gizmo subwindow in a graph, panel or layout window. Here is a simple example:

```
NewPanel/N=MyPanel
NewGizmo/HOST=MyPanel/JUNK=2
```

You can direct any ModifyGizmo commands using full subwindow specification. For example, to change the colormap of the demo surface you can execute:

```
ModifyGizmo/N=MyPanel#GZ0 modifyObject=sampleSurface, objectType=surface,
      property={surfaceCTab,Blue}
```

For more information see **Subwindow Syntax** on page III-92.

Gizmo Troubleshooting

This section provides tips for resolving problems you may encounter while using Gizmo.

Troubleshooting Gizmo Lighting Problems

1. Make sure that the light object appears in the display list above the objects that you expect to be illuminated.
2. Check that your light specifies non-black colors for the ambient, diffuse and specular components.
3. Start with directional light which has fewer options than positional light with less to go wrong. Open the Light Properties dialog, check the Live Update checkbox, change the direction controls and check for changes in the scene. If you do not see changes in the scene then the direction of the light is more than likely not the problem.
4. Verify that you have checked the Compute Normals checkbox for each object that you expect to be illuminated.
5. Check that the normal orientation is what you expect. In the case of quadric objects, toggle the normals orientation between inside and outside. For other objects, consider the definitions of front and back surfaces. If you have doubt, add a front face operation to the display list right before an object. The operation allows you to define the direction representing front and back surfaces.

6. If you are drawing an object that looks fine at normal scale but appears to have the wrong lighting effects when drawn after a scale operation you should compute the object at the final size and avoid scaling. For example, a scatter plot draws markers by default with an isotropic scaling factor of 1 (no scaling). If you create a sphere object with radius 1 for use in a scatter plot, you would need to reduce its size somewhere. You can do so by setting the scale in the Scatter Properties dialog or by reducing the radius of the sphere.

Troubleshooting Gizmo Transparency Problems

1. Check the display list to make sure that there is a GL_BLEND enable operation above the transparent objects and that a blend function exists and has the correct parameters. See **Transparency and Translucency** on page II-432 for details.
2. Check the drawing order of objects. Transparent objects must be after opaque objects in the display list.
3. If you have transparent surfaces that have some overlapping facets, you have no choice but to decompose the surface into triangles and order them according to depth. You can find an example of this in the Depth Sorting demo experiment.

Troubleshooting Gizmo Clipping Problems

Try these tips if an object or part of an object is clippled or not visible.

1. Set the axis range to auto for all axes using Gizmo Menu→Axis Range.
2. Make sure that the display list does not contain any clipping planes.
3. Make sure that the object in question does not follow in the display list after translate, rotate or scale operations.
4. Check the zoom level and the main transformation. If you do not have any transformation operations on the display list insert a default ortho operation (+/-2 for all axes) at the top of the display list.

Gizmo Compatibility

Before Igor7 Gizmo was implemented based on the original OpenGL 1.0 specification. Changes in OpenGL, especially since OpenGL 3.0, required that we modify various features of Gizmo. We have made an effort to allow Gizmo windows in old Igor experiments to open without error and to appear as close as possible to their original form. Notable exceptions include the orientation of string objects, axis labels and tick mark labels; see **Changes to Gizmo Text** on page II-471.

We have also marked in the Gizmo reference documentation some features as "obsolete" or "deprecated". "Obsolete" means that the feature is no longer supported. "Deprecated" means that it may be removed from a future version of Igor and you should use an alternative feature if possible.

Gizmo Commands in User-Defined Functions

In Igor7 and later Gizmo commands can be used in user-defined functions. Supporting this required some changes to the command syntax. The main syntax change is the addition of the objectType keyword in ModifyGizmo. Prior to Igor7 you could use:

```
ModifyGizmo modifyObject=quad0, property={calcNormals,1}
```

The Igor7 syntax requires adding the objectType=<type> keyword as in:

```
ModifyGizmo modifyObject=quad0, objectType=Quad, property={calcNormals,1}
```

So that Igor can interpret Gizmo recreation macros created by Igor6, the objectType keyword is required only in user-defined functions, not from the command line or in macros.

For backward compatibility, the Gizmo XOP as of Igor Pro 6.30 accepts the objectType keyword.

Gizmo Recreation Macro Changes

Prior to Igor7 every Gizmo recreation macro contained the line:

```
ModifyGizmo startRecMacro
```

In Igor7 and later the syntax includes a version number e.g.,

```
ModifyGizmo startRecMacro=700
```

For backward compatibility, the Gizmo XOP as of Igor Pro 6.30 accepts the new startRecMacro syntax.

In Igor7 and later Gizmo uses counter-clockwise as the front face default except when executing Gizmo recreation macros from previous versions. This is unlikely to affect most you but if it does, you can override this default by specifying an explicit frontFace operation in the Gizmo display list.

Use of GL Constants in Gizmo Commands

Previous versions of Gizmo supported the use of OpenGL constants in commands. For example, you could write:

```
AppendToGizmo attribute blendFunc={GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA},
      name=blend0
```

In Igor7 and later the GL constants, such as GL_SRC_ALPHA, are not recognized and you must use the numeric equivalent instead. For example:

```
AppendToGizmo attribute blendFunc={770,771}, name=blend0
```

You can execute GetGizmo to determine what numeric value represents a given constant. For example:

```
GetGizmo constant="GL_SRC_ALPHA"
```

This prints "770" to the history. You can copy the printed number and past it into your procedure.

Exporting Gizmo Graphics

In Igor7 and later you can export Gizmo graphics using File→Save Graphics which generates a **SavePICT** command. The **ExportGizmo** operation is only partially supported for some degree of backward compatibility. It can export to the clipboard or to an Igor wave and it can print but it can no longer export to a file. Use SavePICT instead.

Changes to Gizmo Text

Prior to Igor7 Gizmo displayed axis labels, tick mark labels, and string objects by composing a 3D filled polygon for each character. The polygon representation allowed Gizmo to handle each string as a true 3D object which could be drawn at any position in the display volume independent of the orientation of the axes. This approach had three disadvantages:

- Labels were not always legible and could disappear completely in some orientations
- The conversion into polygons made it impractical to use anti-aliasing to smooth the characters
- Font sizes were inconsistent across platforms

In Igor7 we moved Gizmo to a new text rendering technology that generates 2D smooth text. While this technology addresses the three issues mentioned, it does not produce text objects that match those produced by previous versions. Consequently you will see differences when loading pre-Igor7 experiments that use text, such as axis labels, with offsets and rotations which are no longer supported.

In Igor7 and later you can use formatted text to construct complex axis labels. Double-click an axis item in the Display or object list to display the Axis Properties dialog and click the Axis Labels tab.

In Igor7 and later you can add standard Igor annotations, including textboxes and colorscales, to a Gizmo window using Gizmo→Add Annotation. These annotations appear in an overlay in front of the 3D graphics and behave like annotations in a graph window.

Miscellaneous Gizmo Changes

In Igor7 and later arguments to the shininess attribute have changed to front and back values.

In Igor7 and later a Gizmo object optionally has an internal color attribute. When you create an object you have the option to specify a color or to leave it unspecified. If you specify a color, Gizmo creates a default color material for the object. The default color material has the GL_FRONT_AND_BACK and GL_AMBIENT_AND_DIFFUSE settings. If you don't specify a color then Gizmo does not create a default color material and you must create a color material yourself. This color material affects all objects that appear later in the display list if they have no default color material. This change was necessary in order to support creation of shiny surfaces.

Gizmo Hook Functions

This section is for advanced programmers only.

A hook function is a user-defined function called by Igor when certain events occur. It allows a programmer to react to events and possibly modify Igor's behavior. A window hook function is a hook function that is called for events in a particular window.

Igor's support for this feature is described under **Window Hook Functions** on page IV-293 and, as of Igor7, applies to Gizmo as well as other types of windows.

Because Gizmo was previously implemented as an XOP, it has its own hook function mechanism separate from the Igor mechanism. This section describes Gizmo's specific hook function support.

You can use either Igor hook functions or Gizmo hook functions or both for a Gizmo window. However using both may lead to confusion. If you install both an Igor hook function and a Gizmo hook function on a given Gizmo window, the Igor hook function is called first.

As in Igor itself, Gizmo originally had just one window hook function, installed by the ModifyGizmo hookFunction keyword. Later a named hook function, installed by ModifyGizmo namedHook, was added. Unnamed hooks are obsolete. We recommend that you use named hooks.

Gizmo Named Hook Functions

A named Gizmo window hook function takes one parameter - a WMGizmoHookStruct structure. This built-in structure provides your function with information about the status of various window events.

You install a named Gizmo hook function using **ModifyGizmo** with the namedHook or namedHookStr keywords. The hookEvents keyword is not relevant for named hook functions.

The hook function should usually return 0. In the case of mouse wheel hook events returning a non-zero value prevents Gizmo from rotating in response to the wheel.

The named window hook function has this format:

```
Function MyGizmoHook(s)
   STRUCT WMGizmoHookStruct &s

   strswitch(s.eventName)
      case "mouseDown":
         break
      case "mouseMoved":
         break
      case "rotation":
         break
      case "killed":
         break
      case "scaling":
         break
   endswitch

   return 0
End
```

See **WMGizmoHookStruct** for details on the structure.

As of this time the following event names are defined: mouseDown, mouseMoved, rotation, killed and scaling.

For an example using a named Gizmo window hook, open the Gizmo Window Hook demo experiment and look at the GizmoRotationNamedHook function in the GizmoRotation.ipf procedure file. This is a packed procedure file. It is in an independent module so you need to enable **Independent Modules** to see it.

Gizmo Unnamed Hook Functions

Unnamed hooks are obsolete though still supported for backward compatibility. Use named hooks instead. The following documentation is for historical reference only.

Each Gizmo window can have one and only one unnamed Gizmo window hook function. You designate a function as the unnamed window hook function using the **ModifyGizmo** operation with the hookFunction keyword.

The unnamed hook function is called when various window events take place. The reason for the hook function call is stored as an event code in the hook function's infoStr parameter. Certain events must be enabled using the **ModifyGizmo** operation with the hookEvents keyword.

The unnamed hook function has the following syntax:

```
Function functionName(infoStr)
   String infoStr

   String event = StringByKey("EVENT",infoStr)
   ...
   return 0          // Return value is ignored
End
```

infoStr is a string which containing a semicolon-separated list of keyword:value pairs. For documentation see “Gizmo Unnamed Hook Functions” in the “3D Graphics” help file.

Chapter

II-18

Page Layouts

Overview

A page layout, or layout for short, is a type of window that you can use to compose pages containing:

- Graphs
- Tables
- 3D Gizmo plots
- Annotations (textboxes and legends)
- Pictures
- Drawing elements (lines, arrows, rectangles, polygons, etc.)

You create a layout by choosing New Layout from the Windows menu.

When the active window is a page layout window, the Layout menu appears in the menu bar. It contains items that apply to page layout windows only.

You can use a page layout to make complex graphics for publication and slide shows for presentation. A layout can have multiple pages. You can have as many layouts as memory allows.

Each page layout page has a number of layers. One layer, the layout layer, is for graphs, tables, 3D Gizmo plots, annotations and pictures. The other layers are for drawing elements. Drawing is discussed in detail in Chapter III-3, **Drawing**. This chapter is primarily devoted to the layout layer.

Here are the notable features of page layouts:

- You can combine graphs, tables, 3D Gizmo plots, pictures, annotations and drawing elements.
- Graphs, tables, Gizmo plots and legends in layouts are updated automatically.
- Layouts can contain multiple pages.
- You can export all or part of a layout page to another program as a graphics file.
- You can use a layout to create a full-screen slide show.

There are two ways to add a graph, table or Gizmo plot to a page layout:

- By creating a layout *object* in the layout layer. A layout object represents the contents of a standalone graph window, table window, or 3D Gizmo plot window. Layout objects are described under **Page Layout Objects** on page II-485.
- By creating an graph, table or Gizmo *subwindow* in a drawing layer. Subwindows are described under **Page Layout Subwindows** on page II-497.

The subwindow is a power-user feature. It is described in detail in Chapter III-4, **Embedding and Subwindows**. Graph, table and Gizmo objects are less powerful but simpler to use and more intuitive. We recommend using objects until you have had time to read and understand Chapter III-4.

In this chapter, the term “object” refers to a graph, table, 3D Gizmo plot, annotation or picture object, not to a subwindow.

The following documentation discusses page layouts from the top down, covering these objects:

- Windows
- Pages
- Layers
- Layout objects (in the layout layer)
- Drawing elements (in the drawing layers)
- Subwindows (in the drawing layers)

Page Layout Windows

To create a page layout window, choose Windows→New Layout.

Page Layout Names and Titles

Every page layout window that you create has a name. This is a short Igor-object name that you or Igor can use to reference the layout from a command or procedure. When you create a new layout, Igor assigns it a name of the form Layout0, Layout1 and so on. You will most often use a layout's name when you kill and recreate the layout, see **Killing and Recreating a Layout** on page II-477.

A layout also has a title. The title is the text that appears at the top of the layout window. Its purpose is to identify the layout visually. It is not used to identify the layout from a command or procedure. The title can consist of any text, up to 255 bytes.

You can change the name and title of a layout using the Window Control dialog. This dialog is a collection of assorted window-related things. Choose Window Control from the Control submenu of the Windows menu.

Hiding and Showing a Layout

You can hide a layout window by pressing the Shift key while clicking the close button.

You can show a layout window by choosing its name from the Windows→Layouts submenu.

Killing and Recreating a Layout

Igor provides a way for you to kill a layout and then later to recreate it. This temporarily gets rid of a layout that you expect to be of use later.

You kill a layout by clicking the layout window's close button or by using the Close item in the Windows menu. When you kill a layout, Igor offers to create a **window recreation macro**. Igor stores the window recreation macro in the procedure window of the current experiment. You can invoke the window recreation macro later to recreate the layout. The name of the window recreation macro is the same as the name of the layout.

For further details, see **Closing a Window** on page II-46 and **Saving a Window as a Recreation Macro** on page II-47.

Page Layout Zooming

You can zoom the page using the Zoom submenu in the Layout menu or the Zoom submenu in the Layout menu or the Zoom pop-up menu in the lower-left corner of the layout window.

By zooming out you see the entire page at once. You can zoom in to place drawing elements with higher precision.

Igor stores the position of layout objects with a precision of one point (nominally 1/72th of an inch, about 0.35 mm).

Page Layout Background Color

You can choose a background color for a page layout. This is useful for creating slides.

The background color applies to all pages of the page layout.

You can specify thebackground color by:

- Using the Background Color submenu in the Layout menu.
- Using the Background Color submenu in the Misc pop-up menu.
- Using the NewLayout command line operation.
- Using the ModifyLayout command line operation.

The background color is white by default. If you wish, after selecting a background color, you can capture your preferred background color by choosing Capture Layout Prefs from the Layout menu.

Page Layout Pages

Each page can have any number of pages. You use the page sorter to add and delete pages.

The Page Sorter

The page sorter occupies the left side of the layout window. It provides an overview of all pages in the layout by displaying a thumbnail view of each page.

Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the page sorter area displays a contextual menu from which you can add a page, insert a new page between existing pages, or delete existing pages. You can also add or delete pages using the controls at the bottom of the page sorter.

The editing area of the layout displays only one page at a time. The displayed page is called the "active page". You set the active page by clicking one of the thumbnails in the sorter. Igor identifies the active page by drawing a red outline around its thumbnail.

Although only one page can be active, you can select multiple pages. You would do this to delete or reorder multiple pages at a time. Igor identifies selected pages by drawing a darker outline around their thumbnails.

Starting from a single selected page, you can select a range of pages by clicking on a non-selected thumbnail while pressing the shift key. Alternatively, you can select or deselect individual pages by pressing the command key (*Macintosh*) or control key (*Windows*) while clicking the corresponding thumbnail.

To change the order of pages, select one or more thumbnails and drag them to the desired place in the sorter.

You can also manipulate the pages in a layout programatically using the **LayoutPageAction** operation.

You can resize the page sorter by dragging the divider between it and the page editing area to the left or right. Dragging the divider all the way to the left hides the page sorter entirely.

Page Layout Page Sizes

Each layout stores a global page size and page margins. You can also set the size and margins for specific individual pages. The global page size and margins apply to any page in the layout for which you have not set an explicit size. You can set both the global and the per-page dimensions using the Layout Page Size dialog via the Layout menu, or using the **LayoutPageAction** operation.

When it is created, a layout is given a default page size and page margins based on preferences. You can change these default dimensions by setting the global dimensions for a page layout and choosing Layout→Capture Layout Prefs.

Compatibility with Previous Versions of Igor

Page layouts were originally conceived, decades ago, primarily for use in printing hard copy. Because of this, in Igor Pro 6 and earlier versions, the layout page size was controlled using the system Page Setup dialog, which is part of the printing system.

Over time, the use of hard copy has diminished, replaced by on-screen formats such as HTML and PDF. It is more common now to create graphics for display in a web page or for inclusion in an electronic document to be sent to a journal editor. For these purposes, you usually need to control the size of the graphics independent of any paper size.

Consequently, in Igor Pro 7 and later, each layout has its own size setting that is independent of the page setup. The page setup affects printing of the layout only, not the size of the page on the screen or when exported. Because of this change, a layout created in Igor7 or later may appear with a different size when opened in Igor6.

You may be accustomed to controlling the size and orientation of a page layout page using File→Page Setup. In Igor7 and later, you need to use Layout→Page Size instead.

An ancillary benefit of this change is that it eliminates operating-system dependencies, making behavior across platforms more consistent.

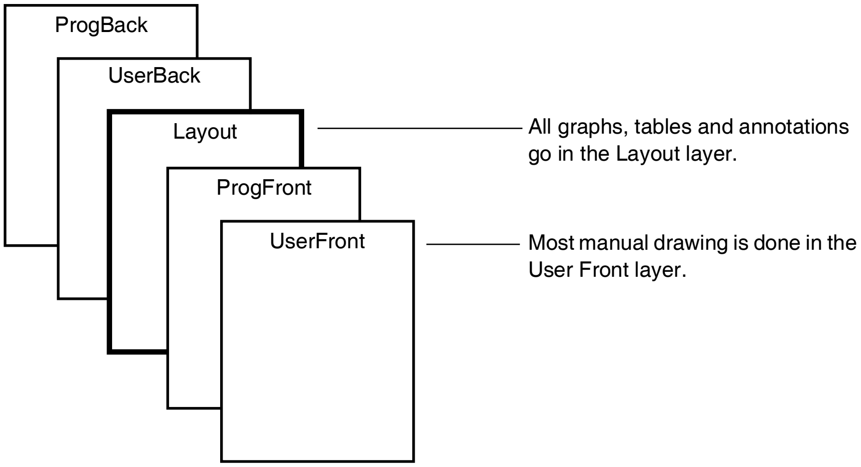
Printing Page Layouts

When you print a page layout, Igor aligns the top/left corner of the layout page, as set in the Layout Page Size dialog, with the top/left corner of the printer's printable area, as set in the Page Setup dialog. The printer's printable area is controlled by the printer margins as set in the Page Setup dialog.

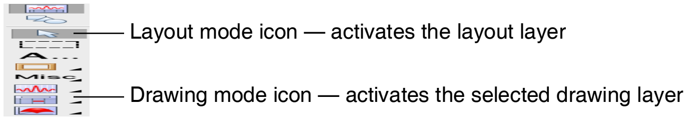
You can preview the printed output using the Print Preview dialog from the File menu.

Page Layout Layers

A page in a layout has six layers. There is one layer for layout objects, four layers for drawing elements, and one layer (not shown in this graphic) for user-interface elements.


ProgBack
UserBack
Layout
ProgFront
UserFront
All graphs, tables and annotations go in the Layout layer.
Most manual drawing is done in the User Front layer.

The two icons in the top-left corner of the layout window control whether you are in layout mode or drawing mode.

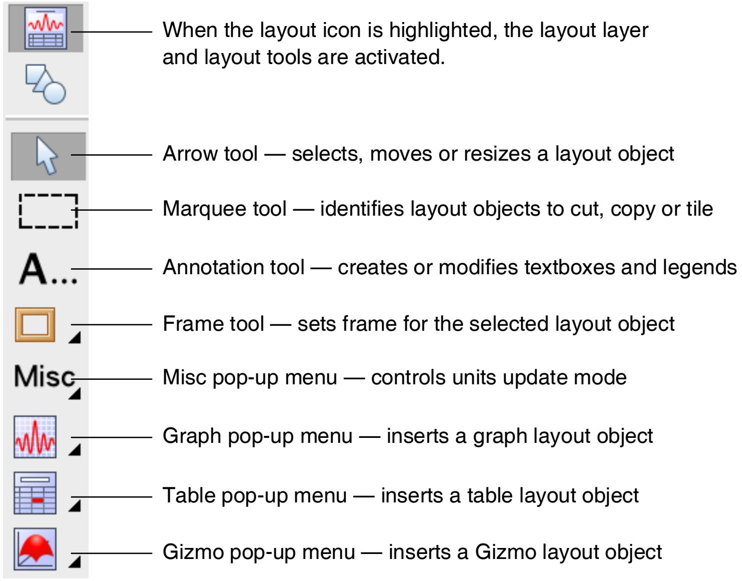

Layout mode icon — activates the layout layer
Drawing mode icon — activates the selected drawing layer

The layout layer is most useful for presenting multiple graphs and for annotations that refer to multiple graphs. The drawing layers are useful for adding simple graphic elements such as arrows between graphs.

The top layer (not shown in the graphic above) is the Overlay layer. It is provided for programmers who wish to add user-interface drawing elements without disturbing graphic elements. It is not included when printing or exporting graphics. This layer was added in Igor Pro 7.00.

Activating the Layout Layer

When you click the layout mode icon, the layout layer is activated. You can use the layout tools to add objects to or modify objects in the layout layer only.



When the layout icon is highlighted, the layout layer and layout tools are activated.

Arrow tool — selects, moves or resizes a layout object

Marquee tool — identifies layout objects to cut, copy or tile

Annotation tool — creates or modifies textboxes and legends

Frame tool — sets frame for the selected layout object

Misc pop-up menu — controls units update mode

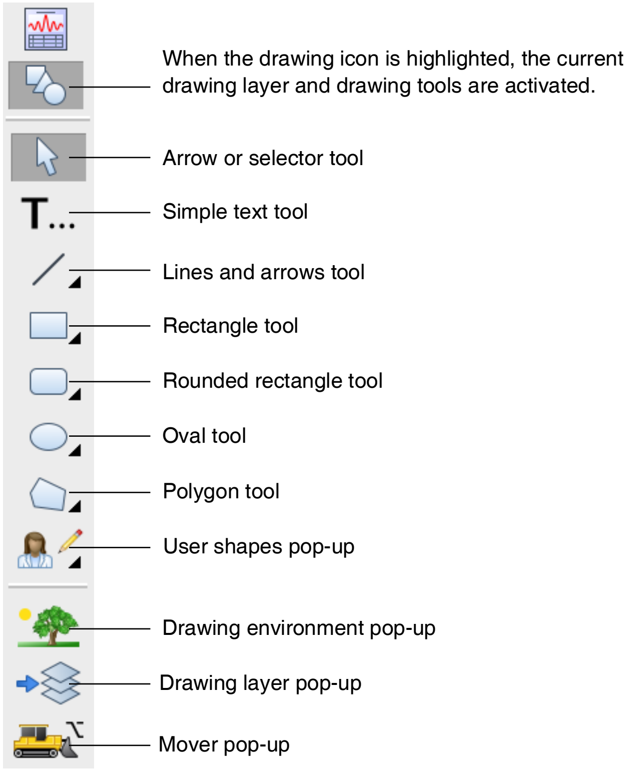
Graph pop-up menu — inserts a graph layout object

Table pop-up menu — inserts a table layout object

Gizmo pop-up menu — inserts a Gizmo layout object

Activating the Current Drawing Layer

When you click the drawing mode icon, the current drawing layer is activated. You can use the drawing tools to add elements to or modify elements in the current drawing layer only.



When the drawing icon is highlighted, the current drawing layer and drawing tools are activated.

Arrow or selector tool

Simple text tool

Lines and arrows tool

Rectangle tool

Rounded rectangle tool

Oval tool

Polygon tool

User shapes pop-up

Drawing environment pop-up

Drawing layer pop-up

Mover pop-up

Changing the Current Drawing Layer

Initially, the UserFront drawing layer will be the current drawing layer. To select a different drawing layer, click the layer tool and select the layer from the pop-up menu.

You may never need to use the drawing layer pop-up menu. Most users will need to use just the layout layer and the UserFront drawing layer. The ProgFront and ProgBack layers are intended to be used from Igor procedures only.

If you click an element that is not in the active layer, Igor ignores the click.

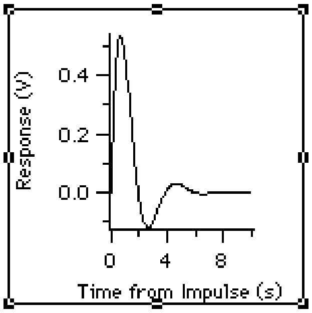
The Layout Layer Tool Palette

When you click the top icon in the tool palette, Igor displays the tools for the layout layer.

Arrow Tool

When you click the arrow tool, it becomes highlighted. The arrow tool is used to select, move or resize a graph, table, annotation or picture object. To select an object, click it. Adjustment handles appear on the selected object.

When you position the cursor over an object, the info panel shows the name of the object under the cursor. If you click and drag, the object will follow the cursor as you drag it. While you drag, the info panel shows the left and top coordinates of the object as well as its width and height. If you press Shift while dragging an object, the direction of movement is constrained either horizontally, vertically, or diagonally, depending on the motion direction. You must press Shift after clicking the object - otherwise the Shift-click will deselect it.



If you press Shift while dragging an object, the direction of movement is constrained either horizontally or vertically, depending on the first motion direction.

You can set the object's width and height by dragging the selected object's handles. While you drag, the info panel shows the width and height of the object.

When you adjust the size of a table and then release the mouse, the table is auto-sized to an appropriate integral number of rows and columns.

You can quickly force an annotation or picture back to its unadjusted size (100% by 100%) by pressing Option (*Macintosh*) or Alt (*Windows*) and double-clicking the object.

Double-clicking an object while the arrow tool is selected brings up the Modify Objects dialog, described in **Modifying Layout Objects** on page II-487. Use this dialog to set the object's properties.

To select multiple objects, click to select the first object and Shift-click to select other objects. Alternatively, you can click in the page and drag diagonally. If you Shift-click an object that is already selected, it becomes deselected.

When two or more objects are selected, you can align them using the Align submenu in the Layout menu. You can also make the widths, heights, or widths and heights of the selected objects the same using items in the Layout menu. In all of these actions, the first object that you select is used as the basis for aligning or resizing other objects. Click in a blank area of the page to deselect all objects. Then click the object whose position or size you want to replicate. Now Shift-click to select additional objects. Finally, choose the desired action from the Layout menu.

While an object is selected, you can control its front-to-back ordering in the layout layer by choosing Bring to Front, Move Forward, Send to Back or Move Backward from the Layout menu. This changes the order of objects *within the layout layer only*. It has no effect on the drawing layers.

If you select a graph, table or 3D Gizmo object, you can then double-click the name of the object in the info panel at the bottom of the layout window. This activates the associated graph or table window.

With multiple objects selected, you can perform the following actions:

- Delete the objects, using the Delete key or the Edit→Clear menu item.
- Copy the objects, using Edit→Copy.
- Cut the objects, using Edit→Cut.
- Drag the objects to a new location.

- Nudge the objects with the arrow keys.
- Change the frame on the objects using the frame tool.

You can not do the following actions on multiple objects:

- Change the order of the objects in the object list (move to front, move to back).
- Adjust the size of the objects using the mouse.

Marquee Tool

When you click the marquee tool, it becomes highlighted and the cursor changes to a crosshair. You can use the marquee tool to identify multiple objects for cutting, copying, clearing or arranging. You can also use it to indicate a part of the layout for export to another application and to control pasting of objects from the clipboard.

To use the marquee tool, click the mouse and drag it diagonally to indicate the region of interest. Igor displays a dashed outline around the region. This outline is called a marquee. A marquee has handles and edges that allow you to refine its size and position.

To refine the size of the marquee, move the cursor over one of the handles. The cursor changes to a double arrow which shows you the direction in which the handle adjusts the edge of the marquee. To adjust the edge, simply drag it to a new position.

To refine the position of the marquee, move the cursor over one of the edges away from the handles. The cursor changes to a hand. To move the marquee, drag it.

To make it possible to export any section of a layout, an object is considered selected if it intersects the marquee. This is in contrast to selection with the arrow tool, which requires that you completely enclose the object.

This table shows how to use the marquee.

To Accomplish This	Do This
Cut, copy or clear multiple objects	Drag a marquee around them and then use the Edit menu to cut, copy or clear.
Paste objects into a particular area	Drag a marquee where you want to paste and then use the Edit menu to paste.
Tile or stack objects	Drag a marquee to indicate the area into which you want to tile or stack and then choose Arrange Objects from the Layout menu.
Export a section of the layout as a picture	Drag a marquee to indicate the section that you want to export and then choose Export Graphics from the Edit menu (to use the clipboard) or choose Save Graphics from the File menu (to save in a disk file).

When you click inside the marquee Igor presents a pop-up menu, called the Layout Marquee menu, from which you can choose Cut, Copy, Paste, or Clear. This cuts, copies, pastes or clears the all objects that intersect the marquee. These marquee items do the same thing as the corresponding items in the Edit menu.

The Marquee menu also contains items that allow you to insert a subwindow.

It is possible to add your own menu items to the Layout Marquee menu. See **Marquee Menus** on page IV-137 for details.

See **Copying Objects from the Layout Layer** on page II-493 and **Pasting Objects into the Layout Layer** on page II-493 for more details on copying and pasting. See **Arranging Layout Objects** on page II-487 for details on tiling and stacking.

When the marquee tool is selected, any selected object is deselected. Double-clicking while the marquee tool is selected has no effect.

Annotation Tool

When you click the annotation tool, it becomes highlighted and the cursor changes to an I-beam. The annotation tool creates new annotations or modifies existing annotations. Annotations include textboxes, legends, and colorscales.

Clicking an existing annotation invokes the Modify Annotation dialog. Clicking anywhere else on the page invokes the Add Annotation dialog which you use to create a new annotation. See **Page Layout Annotations** on page II-494 for details.

Frame Pop-Up Menu

When an object is selected, you can change its frame by choosing an item from the Frame pop-up menu. Each object can have no frame or a single, double, triple or shadow frame.

When you change the frame of a graph, table or picture object, its outer dimensions (width and height) do not change. Since the different frames have different widths, the inner dimensions of the object *do* change. In the case of graphs this is usually the desired behavior. For tables, changing the frame shows a non-integral number of rows and columns. You can restore the table to an integral number of rows and columns by pressing Option (*Macintosh*) or Alt (*Windows*) and double-clicking the table. For pictures, changing the frame slightly resizes the picture to fit into the new frame. To restore the picture to 100% sizing, press Option (*Macintosh*) or Alt (*Windows*) and double-click the picture.

When you change the frame of an annotation object, Igor *does* change the outer dimensions of the object to compensate for the change in width of the frame.

Misc Pop-Up Menu

The Misc pop-up menu adjusts some miscellaneous settings related to the layout.

You can choose Points, Inches, or Centimeters. This sets the units used in the info panel.

You can enable or disable the DelayUpdate item. If DelayUpdate is on, when a graph or table which corresponds to an object in the layout changes, the layout is not updated until you activate it (make it the front window). If you disable DelayUpdate then changes to graphs or tables are reflected immediately in the layout. This also affects drawing commands. If you want to see the effect of drawing commands immediately, turn the DelayUpdate setting off.

DelayUpdate does not affect embedded graph and table subwindows.

DelayUpdate is a global setting that affects all existing and future layouts. When you change it in one layout, you change it for all layouts in all experiments.

You can use the Background Color submenu to change the layout's background color. See **Page Layout Background Color** on page II-477 for details.

Graph Pop-Up Menu

The Graph pop-up menu provides a handy way to append a graph object to the layout layer. It contains a list of all the graph windows that are currently open. Choosing the name of a graph appends the graph object to the layout layer. The initial size of the graph object in the layout is taken from the size of the graph window.

Table Pop-Up Menu

The Table pop-up menu provides a handy way to append a table object to the layout layer. It contains a list of all the table windows that are currently open. Choosing the name of a table appends the table object to the layout layer.

Gizmo Pop-Up Menu

The Gizmo pop-up menu provides a handy way to append a 3D Gizmo plot object to the layout layer. It contains a list of all the Gizmo windows that are currently open. Choosing the name of a Gizmo window appends the Gizmo object to the layout layer.

The Layout Layer Contextual Menu

When the layout layer is active, Control-clicking (*Macintosh*) or right-clicking (*Windows*) displays the Layout Layer contextual menu. The contents of the menu depend on whether you click directly on an object or on a part of the page where there is no object.

Layout Contextual Menu for a Single Object

If you Control-click (*Macintosh*) or right-click (*Windows*) directly on an object, the layout contextual menu includes these items:

Activate Object's Window

Activates the graph, table or 3D Gizmo plot window associated with the object.

Recreate Object's Window

Recreates the graph, table or 3D Gizmo plot window associated with the object by running the window recreation macro that was created when the window was killed.

Kill Object's Window

Kills the graph, table or 3D Gizmo plot window associated with the object. Before it is killed, Igor displays a dialog that you can use to create or update its window recreation macro.

If you press the Option key on Macintosh while selecting this item, the window is killed with no dialog and without creating or updating the window recreation macro. Any changes you made to the window will be lost so use this feature carefully. This feature does not work on Windows because the Alt key interferes with menu behavior.

Show Object's Window

Shows the corresponding window if it is hidden.

Hide Object's Window

Hides the corresponding if it is visible.

Scale Object

Changes the size of the layout object in terms of percent of its current size or percent of its normal size. Although this can work on any type of object, it is most useful for scaling pictures relative to their normal size.

For a picture or annotation object, "normal" size is the inherent size of the picture or annotation before any shrinking or expanding. For a graph, table or Gizmo object, "normal" size means the size of the corresponding window.

If a graph's size is hardwired via the Modify Graph dialog, the corresponding layout object can not be scaled.

Tip: You can quickly return a picture or annotation to its normal size by double-clicking it while pressing Option (*Macintosh*) or Alt (*Windows*).

Convert Object to Embedded

This item converts a graph, table or Gizmo object to an embedded subwindow. In doing so, the standalone window which the object represented is killed, leaving just the embedded subwindow.

Layout Contextual Menu for a Selected Object

If you Control-click or right-click on a part of the page where there is no object while objects are selected, the Layout contextual menu includes these items:

Recreate Selected Objects' Windows

Runs the recreation macro for each selected graph, table or Gizmo object for which the corresponding window was killed.

Kill Selected Objects' Windows

Kills the window corresponding to each selected graph, table, and Gizmo object. Before each window is killed, Igor displays a dialog that you can use to create or update its window recreation macro.

If you press the Option key on Macintosh while selecting this item, each window is killed with no dialog and without creating or updating the window recreation macro. Any changes you made to the window will be lost so use this feature carefully. This feature does not work on Windows because the Alt key interferes with menu behavior.

Show Selected Objects' Window

Shows the corresponding windows if they are hidden.

Hide Selected Objects' Window

Hides the corresponding windows if they are visible.

Scale Selected Objects

Changes the size of each selected layout object in terms of percent of its current size or percent of its normal size.

Page Layout Objects

The layout layer of a page layout page can contain five kinds of objects: graph windows, table windows, 3D Gizmo plot windows, annotations, and pictures. The term "layout object" references these objects in the layout layer only, not drawing elements or subwindows in other layers.

This table shows how you can add each of these objects to the layout layer.

Object Type	To Add Object to the Layout Layer
Graph	Use the Graph pop-up menu in the layout window. Use the Append to Layout dialog. Use the **AppendLayoutObject** operation.
Table	Use the Table pop-up menu in the layout window. Use the Append to Layout dialog. Use the **AppendLayoutObject** operation.
3D Gizmo Plots	Use the Gizmo pop-up menu in the layout window. Use the Append to Layout dialog. Use the **AppendLayoutObject** operation.

Object Type	To Add Object to the Layout Layer
Annotations	Click the text (“A”) tool and then click in the page area. Use the Add Annotation dialog. Use the **TextBox** or **Legend** operations.
Pictures	Paste from the clipboard. Use the Pictures dialog (Misc menu). Use the **AppendLayoutObject** operation if the picture already exists in the current experiment’s picture gallery.

Layout Object Names

Each object in the layout layer has a name so that you can manipulate it from the command line or from an Igor procedure as well as with the mouse. When you position the cursor over an object, its name, position and dimensions are shown in the info panel at the bottom of the layout window.

For a graph, table, or Gizmo object, the object name is the same as the name of the corresponding window. For an annotation, the object name is determined by the Textbox or Legend operation that created the annotation. When you paste a picture from the clipboard into a page layout, Igor automatically gives it a name like PICT_0 and adds it to the current experiment’s picture gallery which you can see by choosing Misc→Pictures.

Layout Object Properties

This table shows the properties of each object in the layout layer.

Object Property	Comment
Left coordinate	Measured from the left edge of the paper. Set using mouse or Modify Objects dialog.
Top coordinate	Measured from the top edge of the paper. Set using mouse or Modify Objects dialog.
Width	Set using mouse or Modify Objects dialog.
Height	Set using mouse or Modify Objects dialog.
Frame	None, single, double, triple, or shadow. Set using Frame pop-up menu or Modify Objects dialog.
Transparency	Set using Modify Objects dialog.

All of the properties can also be set using the ModifyLayout operation from the command line or from an Igor procedure.

Appending a Graph, Table, or Gizmo Plot to the Layout Layer

You can append a graph, table, or 3D Gizmo plot to a layout by choosing the Append to Layout item from the Layout menu or by using the pop-up menus in the layout’s tool palette.

Removing Objects from the Layout Layer

You can remove objects from a layout by choosing the Remove from Layout item from the Layout menu.

You can also remove objects by selecting them and choosing Edit→Clear or Edit→Cut.

Removing a picture from a layout does not remove it from the picture gallery. To do that, use the Pictures dialog.

Modifying Layout Objects

You can modify the properties of layout objects using the Modify Objects dialog. To invoke it, choose Modify Objects from the Layout menu or double-click an object with the layout layer arrow tool.

The effect of each property is described under **Layout Object Properties** on page II-486.

Once you have modified an object you can select another object from the Object list and modify it.

Automatic Updating of Layout Objects

Graph, table and Gizmo objects are dynamic. When the corresponding window changes, Igor automatically updates the layout object. Also, if you change the symbol for a wave in a graph and if that symbol is used in a layout legend, Igor automatically updates the legend.

Normally, Igor waits until the layout window is activated before doing an automatic update. You can force Igor to do the update immediately by deselecting the DelayUpdate item in the Misc pop-up menu in the layout's tool palette.

Dummy Objects

If you append a graph, table, or 3D Gizmo plot to the layout layer, this creates a layout object corresponding to the window. If you then kill the original window, the layout object remains and is said to be a "dummy object". A dummy object can be moved, resized or changed just as any other object.

If you later recreate the window or create a new window with the same name as the original, the object is reassociated with the window and ceases to be a dummy object.

Arranging Layout Objects

This section applies to layout objects in the layout layer only, not to drawing elements or subwindows.

Front-To-Back Relationship of Objects

New objects added to the layout layer are added in front of existing objects. You can move objects in front of or in back of other objects using the Layout menu after selecting a single object with the arrow tool.

These menu commands affect the layout layer only. To put drawing elements in front of the layout layer, use the User Front drawing layer. To put drawing elements behind the layout layer, User Back drawing layer.

Tiling Layout Objects

You can tile or stack objects in a layout by choosing the Arrange Objects item from the Layout menu. This displays the Arrange Objects dialog. The rest of this section refers to that dialog.

Click the Tile radio button to do tiling instead of stacking.

When you tile objects, Igor needs to know the following:

- Which objects you want to tile
- The area of the page into which you want to tile the objects
- The number of rows and columns of tiles that you want
- The spacing between tiles (grout)

You can specify all of this information via the Arrange Objects dialog.

In most cases, it is worthwhile for you pre-arrange the objects to be tiled in roughly the desired sizes and positions and select those objects before summoning the dialog. This facilitates using features of the dialog to precisely arrange the objects.

Before we get into details, we will look at a simple example that demonstrates tiling concepts.

Layout Tiling Guided Tour

In this tour, we will create a page layout displaying some graphs and arrange them using tiling.

We assume that we are creating the layout to produce a graphic of a specific size for use on a web page or in a paper, not for printing hard copy.

We start by making some graphs to tiled and then create a page layout containing the graphs.

1. **In a new experiment, execute the following commands:**

```
Make/O jack=sin(x/8), fred=cos(x/8), bob=jack*fred
Display/W=(35,50,300,175) jack
Display/W=(35,200,300,325) fred
Display/W=(35,350,300,475) bob
Layout/W=(350,50,850,650) Graph0, Graph1, Graph2
ModifyLayout mag=1, units=0
```

2. **If necessary, adjust the layout magnification and window size so you can see the entire page.**

   Next we will set the page size and margins appropriate for using in a web page or paper.

3. **Choose Layout→Page Size, enter the following settings, and click Do It:**

   Units: Points

   Width: 432

   Height: 360

   Margins: 0 for all

   Margins are useful when printing but usually serve no purpose when creating graphics for a paper or web page.

4. **Choose Layout→Append to Layout and append Graph0, Graph1, and Graph2.**

5. **Use Edit→Select All to select all of the graph objects in the layout.**

6. **Choose Layout→Arrange Objects.**

   Click the Tile radio button.

   Set the other settings as follows:

   Grout: 8

   Rows: Auto

   Columns: Auto

   Click Do It.

   The graphs are tiled in a 2 row by 2 column arrangment leaving the bottom/right tile empty.

   Suppose we want to leave the bottom/left tile empty rather than the bottom right.

7. **Drag the bottom/left graph and position it roughly in the bottom/right position.**

   Use Edit→Select All to select all of the graph objects.

   Choose Layout→Arrange Objects.

   Check the Preserve Arrangement checkbox.

   Click Do It.

   The Preserve Arrangement feature tiled the last graph in the bottom/right position as you had roughly positioned it.

   Next we will assume that you want to use just a portion of the page rather than the whole thing, perhaps to leave room for textboxes or other objects.

8. **Click the Misc icon in the tool palette and choose Points.**

   Click a blank area of the page to deselect all objects.

Drag the top/left graph down until its top edge is at 72 points (as displayed in the T section of the layout status panel)

Shift-click to select the top/right graph also and choose Layout→Align→Top Edges.

Shift-click to select the bottom/right graph also.

Choose Layout→Arrange Objects.

Check the Use Bounding Box checkbox.

Click Do It.

Now Igor has tiled the objects into the bounding box surrounding the selected graph objects. The bounding box is the smallest rectangle that encompasses all of the objects to be tiled.

We have left the top 72 points of the layout open for use by textboxes or other objects.

Next we will see how to do the same thing but using the marquee instead of the bounding box and leaving 36 points instead of 72 points of free space at the top.

The marquee tool is a dashed rectangle. It appears below the arrow tool and above the Annotation tool in the page layout tool palette.

To drag out a marquee, click in the page and drag diagonally. The marquee coordinates are displayed in the layout status panel at the bottom of the layout window.

The marquee has draggable handles on the corners and in the middle of each side that you can use for fine tuning its position.

9. **Click the Marquee tool and drag out a marquee that roughly fills the entire page.**

Position the mouse over the top handle and drag it so that the top is positioned at 36 points as shown in the T section of the layout status panel.

Drag the left, bottom, and right marquee handles to the corresponding page edges.

Choose Layout→Arrange Objects.

Uncheck the Use Bounding Box checkbox to enable the Use Marquee checkbox.

Click Do It.

Now Igor has tiled the objects into the area specified by the marquee.

This concludes the layout tiling guided tour.

Selecting Tiling Objects

There are several ways to specify which objects to arrange. The easiest is to select the objects in the page layout before summoning the Arrange Objects dialog. The selected objects will then be selected in the Objects to Arrange list.

Alternatively, you can select objects in the list directly. Objects are tiled in the order in which you select them.

Finally you can include objects to be tiled using the various dialog checkboxes such as All Graphs Too and All Tables Too.

If you select no objects in the list and select none of the checkboxes, then all of the objects in the layout will be arranged.

Specifying the Tiling Area

You can specify the tiling area by clicking the marquee tool and dragging out a marquee. Then when you enter the dialog, the Use Marquee checkbox will be checked and enabled.

Alternatively, you can pre-position the objects so that their bounding box defines the tiling area. The bounding box is the smallest rectangle that completely encloses all of the objects to be tiled. To use this technique, roughly pre-position the objects. Next precisely adjust the positions of the left-most, top-most, right-most, bottom-most edges. Select the objects to be tiled, summon the dialog, and check the Use Bounding Box and Preserve Arrangement checkboxes. See **Layout Tiling Guided Tour** on page II-488 for an example.

If you uncheck both Use Marquee and Use Bounding Box, the tiling area is the entire page.

Setting the Number of Rows and Columns

You can set the number of rows and columns of tiles or you can leave them both on auto. If auto, Igor figures out a nice arrangement based on the number of objects to be tiled and the available space. Setting rows or columns to zero is the same as setting it to auto.

If you set both the rows and columns to a number between 1 and 100, Igor tiles the objects in a grid determined by your row/column specification. If you set either rows *or* columns to a number between 1 and 100 but leave the other setting on auto, Igor figures out what the other setting should be to properly tile the objects. In all cases, Igor tiles starting from the top-left cell in a grid defined by the rows and columns, moving horizontally first and then vertically.

If the grid that you specify has fewer tiles than the number of objects to be tiled, once all of the available tiles have been filled, Igor starts tiling from the top-left corner again.

Setting the Space Between Tiles

To set the space between tiles, enter a value in points in the Grout edit box.

Preserving Your Rough Arrangement

If you check the Preserve Arrangement checkbox, Igor tries to keep the tiled objects in the same approximate positions as your rough pre-positioning. See **Layout Tiling Guided Tour** on page II-488 for an example.

If your approximate positioning is not close enough and if you have left the number of rows and columns as Auto, Preserve Arrangement may get the wrong row/column arrangement. In this case, enter specific values for the number of rows and columns and try again.

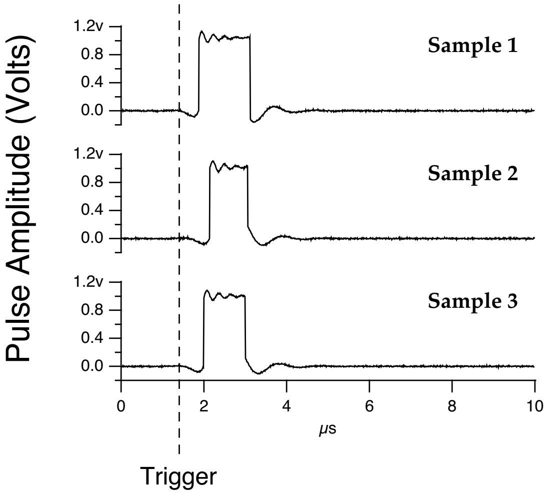
Other Tiling Issues

Regardless of the parameters you specify, Igor clips coordinates so that a tiled object is never completely off the page. Also, objects are never set smaller than a minimum size or larger than the page.

If Preserve Arrangement is unchecked, objects are tiled from left to right, top to bottom. If you select objects in the Objects to Arrange list, they are tiled in the order in which you selected them. If you select no objects in the list, the order in which objects are tiled is determined by the front to back ordering of the objects in the layout.

Aligning Layout Objects

It is a common practice to stack a group of graphs vertically in a column. Sometimes, only one X axis is used for a number of vertically stacked graph. Here is an example.



This section gives step-by-step instructions for creating a layout like the one above. It is also possible to do this using a single graph (see **Creating Stacked Plots** on page II-324 for details) or using subwindows (see Chapter III-4, **Embedding and Subwindows**).

To align the axes of multiple graph objects in a layout, it is critical to set the graph margins. This is explained in detail as follows.

The basic steps are:

1. Prepare the graphs.
2. Append the graph objects to the layout.
3. Align the left edges of the graph objects.
4. Set the width and height of the graph objects.
5. Set the vertical positions of the graph objects.
6. Set the graph plot areas and margins to uniform values.

It is possible to do steps 3, 4, and 5 at once by using the Arrange Objects dialog. However, in this section, we will do them one-at-a-time.

Prepare the Graphs

It is helpful to set the size of the graph windows approximately to the size you intend to use in the layout so that what you see in the graph window will resemble what you get in the layout. You can do this manually or you can use the MoveWindow operation. For example, here is a command that sets the target window to 5 inches wide by 2 inches tall, one inch from the top-left corner of the screen.

```
MoveWindow/I 1, 1, 1 + 5, 1 + 2
```

In the example shown above, we wanted to hide the X axes of all but the bottom graph. We used the Axis tab of the Modify Graph dialog to set the axis thickness to zero and the Label Options tab to turn the axis labels off.

Append the Graphs to the Layout

Click in the layout window or create a new layout using the New Layout item in the Windows menu. If necessary, activate the layout tools by clicking the layout icon in the top-left corner of the layout. Use the Graph pop-up menu or the Append to Layout item in the Layout menu to add the graph objects to the layout. Drag each graph object to the general area of the layout page where you want it.

Align Left Edges of Layout Objects

Drag one of the graphs to set its left position to the desired location. Then Shift-click the other graphs to select them. Now choose Align→Left Edges from the Layout menu.

Set Width and Height of Layout Objects

Set the width and height of one of the graph objects by selecting it and dragging the resulting handles or by double-clicking it and entering values in the Modify Objects dialog.

Click in a blank part of the page to deselect all objects. Now click the object whose dimensions you just set. Now Shift-click to select the other graph objects. With all of the graph objects selected, choose Make Same Width And Height from the Layout menu.

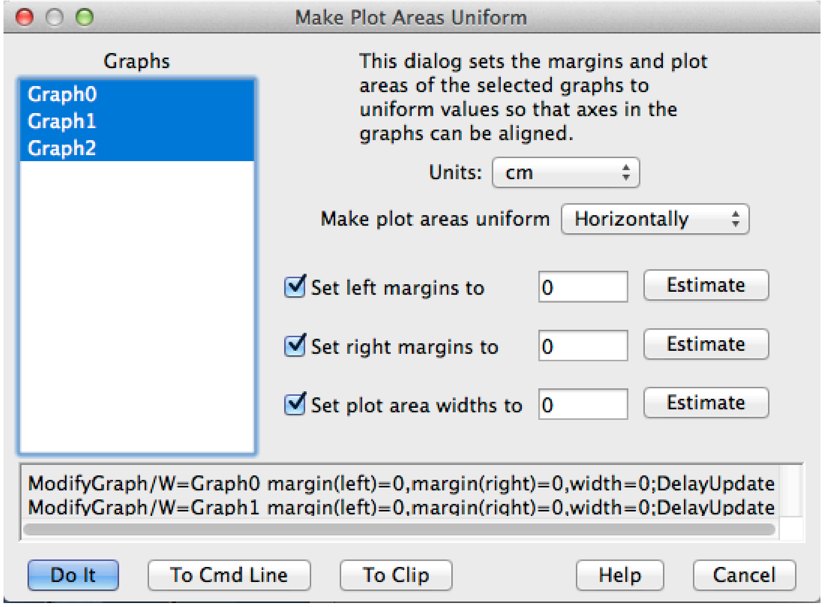
Set Vertical Positions of Layout Objects

Drag the graph objects to their approximate desired positions on the page. You can drag an object vertically without affecting its horizontal position by pressing Shift while dragging. You must press Shift after clicking the object - otherwise the Shift-click will deselect it. Once you have set the approximate position, fine tune the vertical positions using the arrow keys to nudge the selected object.

Set Graph Plot Areas and Margins

At this point, your axes would be aligned except for one subtle thing. The width of text (e.g., tick mark labels) in the left margin of each graph can be different for each graph. For example, if one graph has left axis tick mark labels in the range of 0.0 to 1.0 and another graph has labels in the range 10,000 to 20,000, Igor would leave more room in the left margin of the second graph. The solution to this problem is to set the graph margins, as well as the width of the plot areas, of each graph to the same specific value.

To do this, select all of the graph objects and then choose Make Plot Areas Uniform from the Layout menu. This invokes the following dialog:



Make Plot Areas Uniform

Graphs

Graph0
Graph1
Graph2

This dialog sets the margins and plot areas of the selected graphs to uniform values so that axes in the graphs can be aligned.

Units: cm

Make plot areas uniform: Horizontally

☑ Set left margins to 0

☑ Set right margins to 0

☑ Set plot area widths to 0

```
ModifyGraph/W=Graph0 margin(left)=0,margin(right)=0,width=0;DelayUpdate
ModifyGraph/W=Graph1 margin(left)=0,margin(right)=0,width=0;DelayUpdate
```

Because we are stacking graphs vertically, we want their horizontal margins and plot areas to be the same, which is why we have selected Horizontally from the pop-up menu. The three checkboxes are selected because we want to set both the left and right margins as well as the plot area width.

Now click each of the three Estimate buttons. When you click the Estimate button next to the Set Left Margins To checkbox, Igor sets the corresponding edit box to the largest left margin of all of the graphs selected in the list. Igor does a similar thing for the other two Estimate buttons. As a result, after clicking the three buttons, you should have reasonable values. Click Do It.

Now examine the stacked graph objects. It is possible that you may want to go back into the Make Plot Areas Uniform dialog to manually tweak one or more of the settings.

After doing these steps, the horizontal plot areas in the stacked graphs will be perfectly aligned. This does not, however, guarantee that the left axes will line up. The reason for this is the graphs' axis standoff settings. The axis standoff setting, if enabled, moves the left axis to the left of the plot area to prevent the displayed traces from colliding with the axis. If the graphs have different sized markers, for example, it will offset the left axis of each graph by a different amount. Thus, although the plot areas are perfectly-aligned horizontally, the left axes are not aligned. The solution for this is to use the Modify Axis dialog (Graph menu) to turn axis standoff off for each graph.

Copying and Pasting Layout Objects

This section discusses copying objects from the layout layer of a page layout and pasting them into the same or another page layout. Most users will not need to do this.

Copying Objects from the Layout Layer

You can copy objects to the clipboard by selecting them with the arrow tool or enclosing them with the marquee tool and then choosing Copy from the Edit menu. You can also choose Copy from the pop-up menu that appears when you click inside the marquee.

When you copy an object to the clipboard, it is copied in two formats:

- As an Igor object in a format used internally by Igor
- As a picture that can be understood by other applications

Although you can do a copy for the purposes of exporting to another application, this is not the best way. See **Exporting Page Layouts** on page II-498 for a discussion of exporting graphics to another application. This section deals with copying objects for the purposes of pasting them in the same or another layout. Since it is easy to append graphs and tables to a layout using the pop-up menus in the tool palette, the main utility of this is for copying annotations or pictures from one layout to another.

Copying as an Igor Object Only

There are times when a straightforward copy operation is not desirable. Imagine that you have some graph objects in a layout and you want to put the same objects in another layout. You could copy the graph objects and paste them into the other layout. However, if the graphs are very complex, it could take a lot of time and memory to copy them to the clipboard as a picture. If your purpose is not to export to another application, there is really no need to copy as a picture. If you press Option (*Macintosh*) or Alt (*Windows*) while choosing Copy, then Igor will do the copy only as Igor objects, not as a picture. You can now paste the copied graphs in the other layout.

Pasting Objects into the Layout Layer

This section discusses pasting Igor objects that you have copied from the same or a different page layout. For pasting a new picture that you have generated with another application, see **Inserting a Picture in the Layout Layer** on page II-496.

To paste layout objects that you have copied to the clipboard from the same Igor experiment, just choose Paste from the Edit menu.

When you copy a graph, table, Gizmo or picture layout object from a layout to the clipboard, it is copied as a picture and as an Igor object in an internal Igor format. The Igor format includes the name by which Igor knows the layout object. If you later paste into a layout, Igor will use this name to determine what object should be added to the layout. It normally does not paste the picture representation of the object. In other words, the Igor format of the object that is copied to the clipboard refers to a graph, table, Gizmo or picture by its name.

In rare cases, you may actually want to paste as a picture, not as an Igor object. You might plan to change the graph but want a representation of it as it is now in the layout. To do this, press Option (*Macintosh*) or Alt (*Windows*) while choosing Edit→Paste. This creates a new named picture in the current experiment.

Pasting into a Different Experiment

The reference in the clipboard to Igor objects by name doesn't work across Igor experiments. The second experiment may have a different object with the same name or it may have no object with the name stored in the clipboard. The best you can do when pasting from one experiment to another is to paste a *picture* of the object from the first experiment.

You can force Igor to paste the picture representation instead of the Igor object representation as described above, by pressing Option (*Macintosh*) or Alt (*Windows*) while choosing Edit→Paste.

Pasting Color Scale Annotations

For technical reasons, Igor is not able to faithfully paste a color scale annotation that uses a color index wave or that uses the lookup keyword of the ColorScale operation. If you paste such a color scale, Igor will change it to a color table color scale annotation with no lookup.

Page Layout Annotations

The term “annotation” includes textboxes, legends, tags, and color scales. You can create annotations in graphs and in page layouts. Annotations are discussed in detail in Chapter III-2, **Annotations**. This section discusses aspects of annotations that are unique to page layouts.

Annotations in page layouts exist as layout objects in the layout layer, along with graphs, tables, 3D Gizmo plots, and pictures.

In a graph, an annotation can be a textbox, legend, tag, or color scale. A legend shows the plot symbols for the waves in the graph. A tag is connected to a particular point of a particular wave. In a layout, tags are not applicable. You can create textboxes, legends, and color scales.

Annotations are distinct from the simple text elements that you can create in the drawing layers of graphs, layouts and control panels.

Creating a New Annotation

To create a new annotation, choose Add Annotation from the Layout menu or select the annotation tool and click anywhere on the page, except on an existing annotation. These actions invoke the Add Annotations dialog.

The many options in this dialog are explained in Chapter III-2, **Annotations**.

Modifying an Existing Annotation

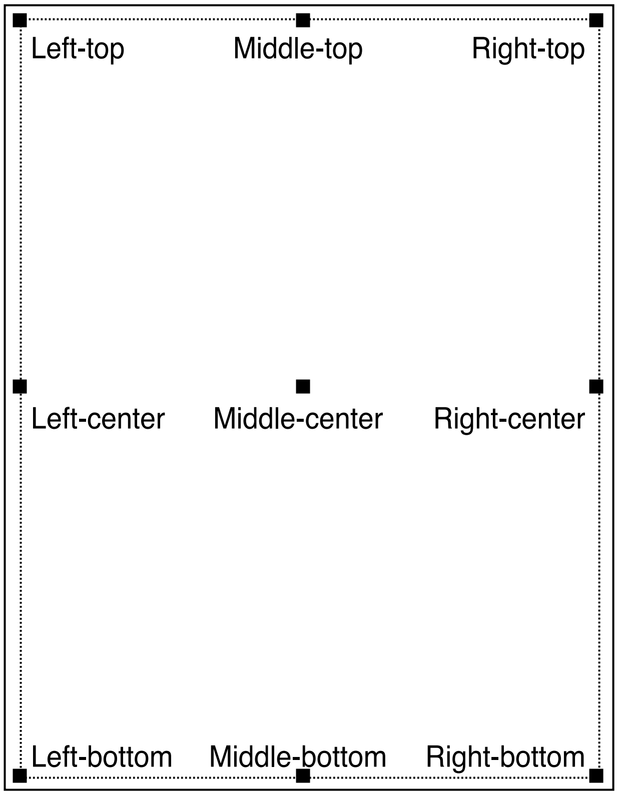
If an annotation is selected when you pull down the Layout menu, you will see a Modify Annotation item instead of the Add Annotation item. Use this to modify the text or style of the selected annotation. You can also invoke the Modify Annotation dialog by clicking the annotation while the annotation tool is selected. Double-clicking an annotation while the arrow tool is selected brings up the Modify Object dialog, not the Modify Annotation dialog.

Positioning an Annotation

An annotation is positioned relative to an anchor point on the edge of the printable part of the page. The distance from the anchor point to the textbox is determined by the X and Y offsets expressed in percent of the width and height of the page inside the margins. The X and Y offsets are automatically set for you when you drag a textbox around the page. You can also set them using the Position tab of the Modify Annotation dialog but this is usually not as easy as just dragging.

Positioning Annotations Programmatically

This diagram shows the anchor points:



Using the top-left anchor, a (0, 0) XY offset would put a tag in the top-left corner of the page:

```
Textbox/A=LT/X=0/Y=0 "Test 1"
```

An XY offset of (50, 50) would put a tag in the middle of the page.

```
Textbox/A=LT/X=50/Y=50 "Test 2"
```

Using the middle-center anchor, a (0, 0) XY offset would put a tag in the middle of the page:

```
Textbox/A=MC/X=0/Y=0 "Test 3"
```

An XY offset of (-50, 50) would put a tag in the top-left corner of the page.

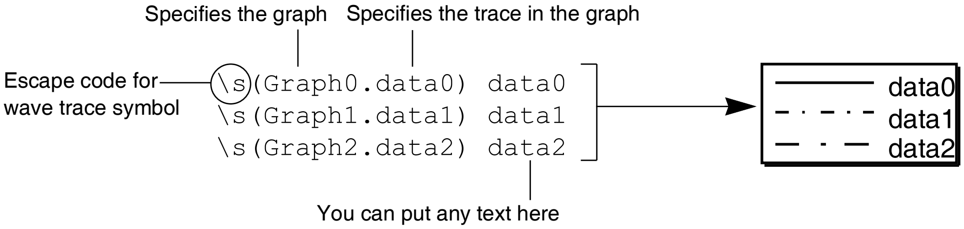
```
Textbox/A=MC/X=-50/Y=50 "Test 4"
```

For most purposes, the left-top anchor is the easiest to use and is sufficient.

The anchor sets not only the reference point on the page but also the reference point on the annotation. For example, if the anchor is right-top then the XY offset sets the position of the right-top corner of the annotation, relative to the right-top corner of the page. For this reason, if you want several textboxes to be right-aligned, you would want to use a right-top, right-center or right-bottom anchor.

Legends in the Layout Layer

When you invoke the Add Annotations dialog and choose Legend, Igor automatically sets the annotation's text to produce a legend containing a symbol for each wave in each graph object in the layout. This diagram illustrates the legend text generated by the dialog:



```
\s(Graph0.data0) data0
\s(Graph1.data1) data1
\s(Graph2.data2) data2
```

Specifies the graph
Specifies the trace in the graph
Escape code for wave trace symbol
You can put any text here
data0
data1
data2

Igor generates the lines of the legend text starting with the bottom graph object in the layout and working toward the top. You can edit the text to remove symbols that you don’t want or to change what appears after the symbol.

If you change the symbol for a trace referenced in the legend, Igor automatically updates the layout legend. If you append or remove waves to the graphs represented in the layout, Igor updates the layout legend. Updating happens when you activate the layout unless you have turned the layout’s DelayUpdate setting off, in which case it happens immediately.

You can freeze a legend by converting it to a textbox. This stops Igor from automatically updating it when waves are added to or removed from graphs. To do this, select the annotation tool and click in the legend. In the resulting Modify Annotation dialog, change the pop-up menu in the top-left corner from Legend to Textbox. You can also do this using the following command:

```
Textbox/C/N=text0    // convert legend named text0 into a textbox
```

Instead of specifying the name of the trace for a legend symbol, you can specify the trace number. For example, "\s(Graph0.#0)" displays the legend for trace number 0 of Graph0.

Default Font

By default, annotations use the default font chosen in the Default Font dialog via the Misc menu. You can override the default font using the Font pop-up menu in the Add Annotation dialog.

Page Layout Pictures

You can insert a picture that you have created in another application, for example a drawing program or equation editor, into the layout layer or into a drawing layer. If the picture has some relation to other drawing elements, you should use the drawing layer. If it has some relation to other layout objects, you should use the layout layer. The use of drawing layers is discussed under **Pasting a Picture Into a Drawing Layer** on page III-73. This section discusses pictures in the layout layer.

Inserting a Picture in the Layout Layer

All pictures displayed in the layout layer reside in the picture gallery which you can see by choosing Misc→Pictures. If you paste a picture from the clipboard into the layout layer, Igor automatically adds it to the picture gallery. If the picture is in a file on disk, you must first load it into the picture gallery. Then you can place it in the layout layer.

See **Pictures** on page III-509 for information on supported picture formats.

Resetting a Picture’s Size

If you expand or shrink a picture in the layout layer, you can reset it to its default size by pressing Option (*Macintosh*) or Alt (*Windows*) and double-clicking it with the arrow tool.

Page Layout Subwindows

The subwindow is a power-user feature. It is described in detail in Chapter III-4, **Embedding and Subwindows** and can not be effectively used without a careful reading of that chapter. In this section we discuss subwindows as they apply to page layouts.

You can create three kinds of subwindows in a page layout window: graphs, tables, and 3D Gizmo plots. To add a subwindow to a layout:

1. Activate the layout layer by clicking the layout icon.
2. Select the marquee tool (dashed-line rectangle).
3. Drag out a marquee.
4. Click inside the marquee and choose one of the following:

- New Graph Here
- New Category Plot Here
- New Contour Plot Here
- New Image Plot Here
- New Table Here
- New 3D Plot Here

You can also create a subwindow by right-clicking (*Windows*) or Control-clicking (*Macintosh*) while in drawing mode and choosing an item from the New submenu.

You can convert a layout object to an embedded subwindow by right-clicking (*Windows*) or Control-clicking (*Macintosh*) while in layout mode and choosing Convert To Embedded. Note that a graph containing a control panel or controls cannot be converted into an embedded graph, even though a graph object with controls or control panels can be added to a layout. Such a graph object does not display the controls or control panels, however.

You can convert a subwindow to a standalone window and layout object object by right-clicking (*Windows*) or Control-clicking (*Macintosh*) while in layout mode and choosing Convert To Window And Object. In a graph window, you must click in the graph background, away from any traces or axes.

Page Layout Drawing

Like graphs and control panels, you can add drawing elements to page layouts. Drawing in general is discussed under Chapter III-3, **Drawing**. This section discusses drawing issues specific to page layouts.

You enter drawing mode by clicking the second icon from the top in the tool palette, as explained under **Activating the Current Drawing Layer** on page II-480. Once in drawing mode, you can select a specific drawing layer using the drawing layer tool, as explained under **Changing the Current Drawing Layer** on page II-480.

Page Layout Slide Shows

A layout slide show is a full-screen display of the pages in a layout, suitable for a presentation to a group. You start a slide shows by clicking the button at the bottom/right corner of the layout page sorter, or using the **LayoutSlideShow** operation.

Using the Slide Show Settings dialog, which you can invoke via the Layout menu, you can control various aspects of the slide show, including:

- How pages are scaled to fit the screen
- How multiple monitors are used
- The time to wait before showing the next slide

Exporting Page Layouts

You can export a layout to another application through the clipboard or by creating a file. To export via the clipboard, use the Export Graphics item in the Edit menu. To export via a file, use the Save Graphics item in the File menu.

If you want to export a section of the page, use the marquee tool to specify the section first. To do this, the layout icon in the top-left corner of the layout window must be selected.

If you don't use the marquee, Igor exports the entire page, or the part of the page that has layout objects or drawing elements in it. The Crop to Page Contents, in the Export Graphics and Save Graphics dialogs, controls this.

The process of exporting graphics from a layout is very similar to exporting graphics from a graph. You can find the details under Chapter III-5, **Exporting Graphics (Macintosh)** and Chapter III-6, **Exporting Graphics (Windows)**. Those chapters describe the various export methods and how to select the method that will give you the best results.

Page Layout Preferences

Page layout preferences allow you to control what happens when you create a new layout or add new objects to the layout layer of an existing layout. To set preferences, create a layout and set it up to your taste. We call this your *prototype* layout. Then choose Capture Layout Prefs from the Layout menu.

Preferences are normally in effect only for *manual* operations, not for programmed operations in Igor procedures. This is discussed in more detail in Chapter III-18, **Preferences**.

When you initially install Igor, all preferences are set to the factory defaults. The dialog indicates which preferences you have changed.

The "Window Position and Size" preference affects the creation of new layouts only.

The Object Properties preference affects the creation of new objects in the layout layer. To capture this, add an object to the layout layer and use the Modify Objects dialog to set its properties. Then select the object and choose Capture Layout Prefs. Select the Object Properties checkbox and click Capture Prefs.

The page size and margins preference affects what happens when you create a new layout, not when you recreate a layout using a recreation macro.

Page Layout Style Macros

The purpose of a layout style macro is to allow you to create a number of layouts with the same stylistic properties. Using the Window Control dialog, you can instruct Igor to automatically generate a style macro from a prototype layout. You can then apply the macro to other layouts.

Igor can generate style macros for graphs, tables and page layouts. However, their usefulness is mainly for graphs. See **Graph Style Macros** on page II-350. The principles explained there apply to layout style macros also.

Page Layout Shortcuts

Action	Shortcut (*Macintosh*)	Shortcut (*Windows*)
Change the layout magnification	Click the magnification readout in the lower-left corner of the layout window.	Click the magnification readout in the lower-left corner of the layout window.
Modify layout object properties	Select the arrow tool in Layout mode and double-click the object.	Select the arrow tool in Layout mode and double-click the object.
Edit an existing annotation	Select the annotation tool in the Layout mode and click in the annotation.	Select the annotation tool in the Layout mode and click in the annotation.
Bring up a graph or table window	Select corresponding object in the layout layer and then double-click the name of the object in the info panel.	Select corresponding object in the layout layer and then double-click the name of the object in the info panel.
Auto-size a picture or annotation object to 100%	Select arrow tool in Layout mode, press Option and double-click the picture or annotation object.	Select arrow tool in Layout mode, press Alt and double-click the picture or annotation object.
Auto-size a table object to an integral number of rows and columns	Select arrow tool in Layout mode, press Option and double-click the table object.	Select arrow tool in Layout mode, press Alt and double-click the table object.
Constrain the resizing direction or dragging an object	Press Shift while resizing or dragging the object. You must press Shift after clicking the object - otherwise the Shift-click will deselect it.	Press Shift while resizing or dragging the object. You must press Shift after clicking the object - otherwise the Shift-click will deselect it.
Copy, cut, or clear multiple layout objects	Use the arrow tool or marquee tool to select the objects, then choose copy, cut or clear from the Edit menu.	Use the arrow tool or marquee tool to select the objects, then choose copy, cut or clear from the Edit menu.
Export a subset of the layout via the clipboard	Using the marquee tool, select a page area, then choose Export Graphics from the Edit menu.	Using the marquee tool, select a page area, then choose Export Graphics from the Edit menu.
Export a subset of the layout via the a graphics file	Using the marquee tool, select a page area and then choose Save Graphics from the File menu.	Using the marquee tool, select a page area and then choose Save Graphics from the File menu.
Drawing tool shortcuts	See Chapter III-3, **Drawing**.	See Chapter III-3, **Drawing**.

Volume III User's Guide: Part 2

Table of Contents

III-1 Notebooks III-1
III-2 Annotations III-33
III-3 Drawing III-61
III-4 Embedding and Subwindows III-79
III-5 Exporting Graphics (Macintosh) III-95
III-6 Exporting Graphics (Windows) III-101
III-7 Analysis III-107
III-8 Curve Fitting III-177
III-9 Signal Processing III-269
III-10 Analysis of Functions III-319
III-11 Image Processing III-351
III-12 Statistics III-383
III-13 Procedure Windows III-395
III-14 Controls and Control Panels III-413
III-15 Platform-Related Issues III-447
III-16 Text Encodings III-459
III-17 Miscellany III-495
III-18 Preferences III-515

Chapter III-1

Notebooks

Overview

A notebook is a window in which you can store text and graphics, very much like a word processor document. Typical uses for a notebook are:

- Keeping a log of your work.
- Generating a report.
- Examining or editing a text file created by Igor or another program.
- Documenting an Igor experiment.

A notebook can also be used as a worksheet in which you execute Igor commands and store text output from them.

Plain and Formatted Notebooks

There are two types of notebooks:

- Plain notebooks.
- Formatted notebooks.

Formatted notebooks can store text and graphics and are useful for reports. Plain notebooks can store text only. They are good for examining data files and other text files where line-wrapping and fancy formatting is not appropriate.

This table lists the properties of each type of notebook.

Property	Plain	Formatted
Can contain graphics	No	Yes
Allows multiple paragraph formats (margins, tabs, alignment, line spacing)	No	Yes
Allows multiple text formats (fonts, text styles, text sizes, text colors)	No	Yes
Does line wrapping	No	Yes
Has rulers	No	Yes
Has headers and footers	Yes	Yes
File name extension	.txt	.ifn
Can be opened by most other programs	Yes	No
Can be exported to word processors via Rich Text file	Yes	Yes

Plain text files can be opened by many programs, including virtually all word processors, spreadsheets and databases. The Igor formatted notebook file format is a proprietary WaveMetrics format that other applications can not open. However, you can save a formatted notebook as a Rich Text file, which is a file format that many word processors can open.

Igor does not store settings (font, size, style, etc.) for plain text files. When you open a file as a plain text notebook, these settings are determined by preferences. You can capture preferences by choosing Notebook→Capture Notebook Prefs.

Notebook Text Encodings

Igor uses UTF-8, a form of Unicode, internally. Prior to Igor7, Igor used non-Unicode text encodings such as MacRoman, Windows-1252 and Shift JIS.

All new notebook files should use UTF-8 text encoding. When you create a new notebook using Windows→New→Notebook, Igor automatically uses UTF-8. Also, the NewNotebook operation defaults to UTF-8.

Igor must convert from the old text encodings to Unicode when opening old files. It is not always possible to get this conversion right. You may get incorrect characters or receive errors when opening files containing non-ASCII text.

For a discussion of these issues, see **Text Encodings** on page III-459, **Plain Text File Text Encodings** on page III-466, and **Formatted Text Notebook File Text Encodings** on page III-472.

Creating a New Notebook File

To create a new notebook, choose Windows→New→Notebook. This displays the New Notebook dialog.

The New Notebook dialog creates a new notebook *window*. The notebook *file* is not created until you save the notebook window or save the experiment.

Normally you should store a notebook as part of the Igor experiment in which you use it. This happens automatically when you save the current experiment unless you do an explicit Save Notebook As before saving the experiment. Save Notebook As stores a notebook separate from the experiment. This is appropriate if you plan to use the notebook in multiple experiments.

Note: There is a risk in sharing notebook files among experiments. If you copy the experiment to another computer and forget to also copy the shared files, the experiment will not work on the other computer. See **References to Files and Folders** on page II-24 for more explanation.

If you do create a shared notebook file then you are responsible for copying the shared file when you copy an experiment that relies on it.

Opening an Existing File as a Notebook

You can create a notebook window by opening an existing file. This might be a notebook that you created in another Igor experiment or a plain text file created in another program. To do this, choose File→Open File→Notebook.

Opening a File for Momentary Use

You might want to open a text file momentarily to examine or edit it. For example, you might read a Read Me file or edit a data file before importing data. In this case, you would open the file as a notebook, do your reading or editing and then kill the notebook. Thus the file would not remain connected to the current experiment.

Sharing a Notebook File Among Experiments

On the other hand, you might want to share a notebook among multiple experiments. For example, you might have one notebook in which you keep a running log of all of your observations. In this case, you could save the experiment with the notebook open. Igor would then save a reference to the shared notebook file in the experiment file. When you later open the experiment, Igor would reopen the notebook file.

As noted above, there is a risk in sharing notebook files among experiments. You might want to “adopt” the opened notebook. See **References to Files and Folders** on page II-24 for more explanation.

Saving All Standalone Notebook Files

When a notebook window is active, you can save all modified standalone notebook files at once by choosing File→Save All Standalone Notebook Files. This saves only standalone notebook files. It does not save packed notebook files, or notebook windows that were just created and never saved to disk; these are saved when you save the experiment.

Autosaving Standalone Notebook Files

Igor can automatically save modified standalone notebook files. See **Autosave** on page II-36 for details.

Notebooks as Worksheets

Normally you enter commands in Igor's command line and press Return or Enter to execute them. You can also enter and execute commands in a notebook window. Some people may find using a notebook as a worksheet more convenient than using Igor's command line.

You can also execute commands from procedure windows and from help windows. The former is sometimes handy during debugging of Igor procedures. The latter provides a quick way for you to execute commands while doing a guided tour or to try example commands that are commonly presented in help files. The techniques described in the next paragraphs for executing commands from a notebook also apply to procedure and help windows.

To execute a command from a notebook, enter the command in a notebook, or select existing text, and press Control-Enter (*Macintosh*) or Ctrl-Enter (*Windows*). You can also select text already in the notebook and press Control-Enter. You can also right-click selected text and choose Execute Selection from the resulting pop-up menu.

When you press Control-Enter, Igor transfers text from the notebook to the command line and starts execution. Igor stores both the command and any text output that it generates in the notebook and also in the history area of the command window. If you don't want to keep the command output in the notebook, just undo it.

Command output is never sent to a procedure window or help window.

Command output is not sent to the notebook if it is open for read only, or if you clicked the write-protect icon, or if you disabled it by unchecking the Copy Command Output to the Notebook checkbox in the Command Window section of the Miscellaneous Settings dialog.

If you don't want to store the commands or the output in the history area, you can disable this using the Command Window section of the Miscellaneous Settings dialog. However, if command output to the notebook is disabled, the command and output are sent to the history area even if you have disabled it.

Showing, Hiding and Killing Notebook Windows

Notebook files can be opened (added to the current experiment), hidden, and killed (removed from the experiment).

When you click the close button of a notebook window, Igor presents the Close Notebook Window dialog to find out what you want to do. You can save and then kill the notebook, kill it without saving, or hide it.

If you just want to hide the window, you can press Shift while clicking the close button. This skips the dialog and just hides the window.

Killing a notebook window closes the window and removes it from the current experiment but does *not* delete the notebook file with which the window was associated. If you want to delete the file, do this on the desktop.

The Close item of the Windows menu and the keyboard shortcut, Command-W (*Macintosh*) or Ctrl+W (*Windows*), behave the same as the close button, as indicated in these tables.

Macintosh:

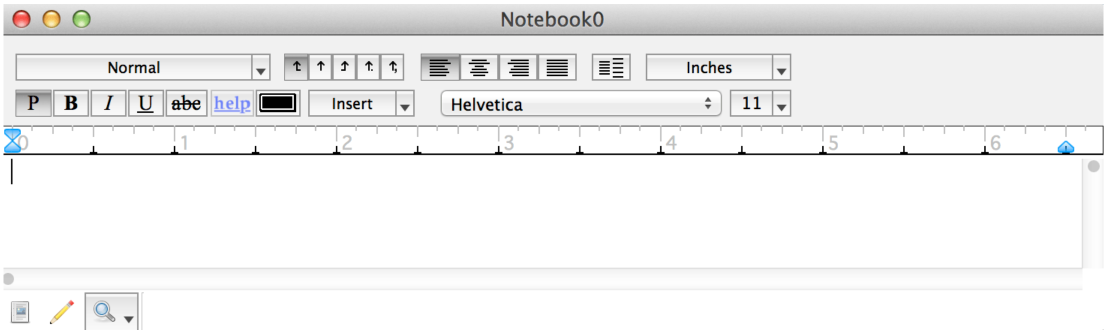
Action	Modifier Key	Result
Click close button, choose Close or press Command-W	None	Displays dialog
Click close button, choose Close or press Command-W	Shift	Hides window

Windows:

Action	Modifier Key	Result
Click close button, choose Close or press Ctrl+W	None	Displays dialog
Click close button, choose Close or press Ctrl+W	Shift	Hides window

Parts of a Notebook

This illustration shows the parts of a formatted notebook window. A plain notebook window has the same parts except for the ruler.



Ruler Bar

Rulers defined paragraph formatting, including font, text size, text style, indents, tabs, and spacing.

You can use the ruler bar to apply a ruler to the selected paragraphs, to modify a ruler, or to apply changes to the selected text without changing the current ruler.

In a formatted text notebook, you can show and hide the ruler bar using the Notebook menu. You can not show the ruler bar in a plain text notebook.

You can create as many rulers as you want in a formatted text notebook. A plain text notebook has just one ruler.

Document Info Icon

The icon in the bottom/left corner of the notebook is the document info icon. When you click it, Igor displays information about the notebook and its associated file, if any.

Write-Protect Icon

Notebooks (as well as procedure windows) have a write-enable/write-protect icon which appears in the lower-left corner of the window and resembles a pencil. If you click this icon, Igor Pro will draw a line through the pencil, indicating that the notebook is write-protected. The main purpose of this is to prevent accidental manual alteration of shared procedure files, but you can also use it to prevent accidental manual alteration of notebooks.

Note that write-protect is not the same as read-only. Write-protect prevents manual modifications while read-only prevents all modifications. See **Notebook Read/Write Properties** on page III-10 for details.

Magnifier Icon

You can magnify procedure text to make it more readable. See **Text Magnification** on page II-53 for details.

Notebook Properties

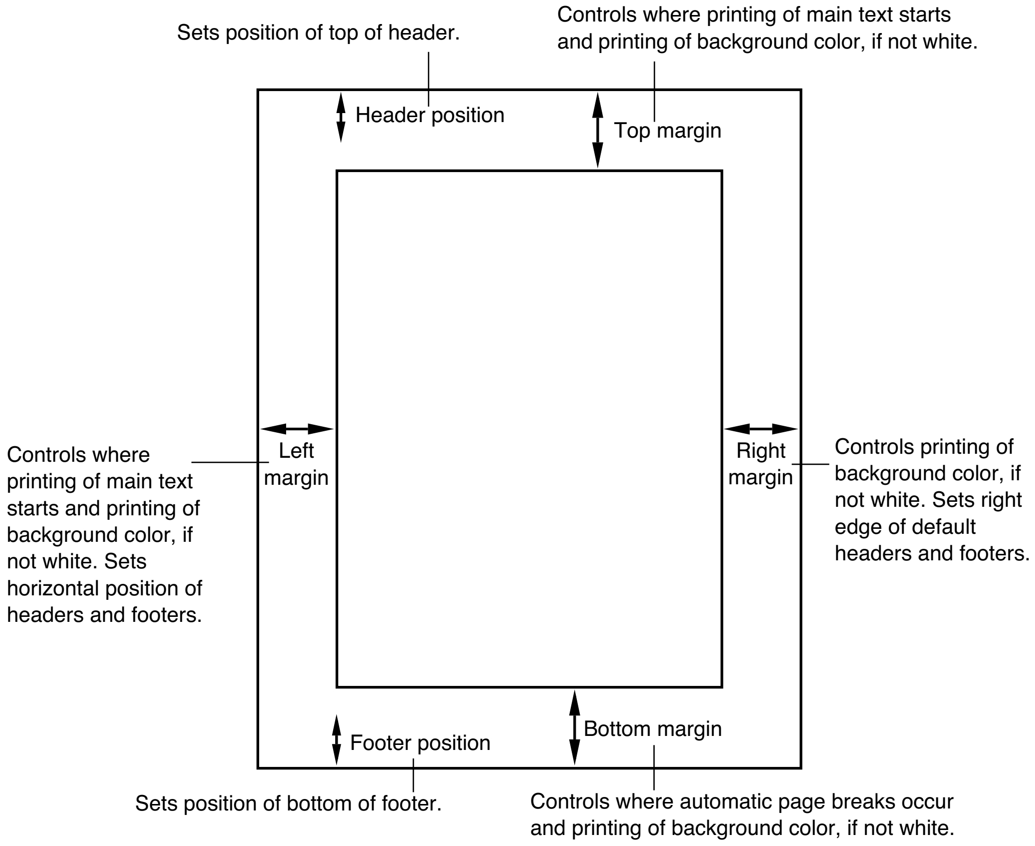
Everything in a notebook that you can control falls into one of four categories, as shown in this table.

Category	Settings
Document properties	Page margins, background color, default tab stops, headers and footers.
Paragraph properties	Paragraph margins and indents, tab stops, line alignment, line spacing, default text format.
Character properties	Font, text size, text style, text color, vertical offset.
Read/write properties	Read-only, write-protect and changeableByCommandOnly.

Notebook Document Properties

To set document properties, choose Notebook→Document Settings. These are properties that apply to the notebook as a whole, such as page margins, headers, footers, background color, and default tab stops.

The next illustration shows the effects of the page margins and header and footer position settings. In addition, these settings affect the Rich Text format file which you can use to export a notebook to a word processor.


Sets position of top of header.
Controls where printing of main text starts and printing of background color, if not white.
Header position
Top margin
Controls where printing of main text starts and printing of background color, if not white. Sets horizontal position of headers and footers.
Left margin
Right margin
Controls printing of background color, if not white. Sets right edge of default headers and footers.
Footer position
Bottom margin
Sets position of bottom of footer.
Controls where automatic page breaks occur and printing of background color, if not white.

The notebook page margins override margins set via the Page Setup dialog and the PrintSettings operation margins keyword.

Notebook Default Tabs

The default tab width setting controls the location of all tab stops in plain text notebooks and of all tab stops after the last explicit tab stop in formatted notebooks. You set this using the Document Settings dialog in the Notebook menu or using the Notebook operation defaultTab or defaultTab2 keywords.

Three modes are available: points, spaces, and mixed. In points mode, you specify the default tab width for all paragraphs in units of points. In the Document Settings dialog, you can enter this setting in inches, centimeters, or points, but it is always stored as points. Prior to Igor Pro 9.00, this was the only mode available.

In spaces mode, you specify the default tab width for all paragraphs in units of spaces.

In mixed mode, you specify the default tab width for paragraphs controlled by proportional fonts in units of points and for paragraphs controlled by monospace fonts in units of spaces.

Specifying default tabs in spaces is mostly of use for indentation and alignment of comments in notebooks that show Igor code such as Igor help files.

For formatted notebooks, the space character unit used in spaces mode and in mixed mode for monospace fonts is the width of a space character in the ruler font for a given paragraph. For plain text notebooks, it is the width of a space character in the notebook font.

In a formatted notebook, the ruler displays explicit tab stops and default tab stops. You can adjust default tab stops by dragging one of them. This sets either the default tab width in points or in spaces, depending on the units used for default tabs by the current paragraph. Default tabs in points are relative to the position of zero on the ruler. Default tabs in spaces are relative to the ruler's left margin.

When you create a new notebook window, Igor applies your preferred default tab width settings. There are separate preferences for plain text notebooks and formatted text notebooks. You can set preferences for a given type of notebook by first using the Document Settings dialog to apply the desired settings to a window of that type and then choosing Notebook→Capture Notebook Prefs as explained under **Notebook Preferences** on page III-30.

Igor stores document settings separately for each formatted text notebook file. In Igor Pro 9.00 and later, it stores the default tab width mode, width in points, and width in spaces settings. Older versions of Igor support only default tabs specified in points and ignore the other settings.

Igor does not store document settings for plain text notebook files, so your preferred default tab width settings for plain text notebooks apply to all plain text notebook files subsequently opened.

Notebook Paragraph Properties

A set of paragraph properties is called a “ruler”. In some word processors, this is called a “style”. The purpose of rulers is to make it easy to keep the formatting of a notebook consistent. This is described in more detail under **Working with Rulers** on page III-11.

Formatted Notebook Paragraph Properties

The paragraph properties for a formatted notebook are all under your control and can be different for each paragraph. A new formatted notebook has one ruler, called the Normal ruler. You can control the properties of the Normal ruler and you can define additional rulers.

The ruler font, ruler text size, ruler text style and ruler text color can be set using the pop-up menu on the left side of the ruler. They set the *default* text format for paragraphs governed by the ruler. You can use the Notebook menu to override these default properties. The Notebook menu permits you to hide or show the ruler in a formatted notebook.

The paragraph properties for formatted notebooks are:

Paragraph Property	Description
First-line indent	Horizontal position of the first line of the paragraph.
Left margin	Horizontal position of the paragraph after the first line.
Right margin	Horizontal position of the right side of the paragraph.

Paragraph Property	Description
Line alignment	Left, center, right or full.
Space before	Extra vertical space to put before the paragraph.
Min line space	Minimum height of each line in the paragraph.
Space after	Extra vertical space to put after the paragraph.
Tab stops	Left, center, right, decimal-aligned or comma-aligned tab stops.
Ruler font	The default font to use for the paragraph.
Ruler text size	Default text size to use for the paragraph.
Ruler text style	Default text style to use for the paragraph.
Ruler text color	Default text color to use for the paragraph.

Plain Notebook Paragraph Properties

For each plain notebook, there is one set of paragraph properties that govern all paragraphs. Many of the items are fixed — you can't adjust them.

Paragraph Property	Comment
First-line indent	Fixed at zero.
Left margin	Fixed at zero.
Right margin	Fixed at infinity.
Line alignment	Fixed as left-aligned.
Space before	Fixed at zero.
Min line space	Fixed at zero.
Space after	Fixed at zero.
Tab stops	None.
Font	Set using Notebook menu.
Text size	Set using Notebook menu.
Text style	Set using Notebook menu.
Text color	Set using Notebook menu.

There is only one font, text size, text style and text color for the entire document which you can set using the Notebook menu.

Although you can not set paragraph tab stops in a plain notebook, you *can* set and use the notebook's default tab stops, which affect the entire notebook.

Notebook Character Properties

The character properties are font, text size, text style, text color and vertical offset. The vertical offset is used mainly to implement superscript and subscript. A specific collection of character properties is called a "text format". You can set the text format using the Notebook menu. The Set Text Format dialog allows you to inspect and set all of the character properties.

Plain Notebook Text Formats

A plain notebook has one text format which applies to all of the text in the notebook. You can set it, using the Notebook menu, except for the vertical offset which is always zero.

Igor does not store settings (font, size, style, etc.) for a plain text files. When you open a file as a plain text notebook, these settings are determined by preferences. You can capture preferences by choosing Notebook→Capture Notebook Prefs.

Formatted Notebook Text Formats

By default, the text format for the text in a paragraph is determined by the ruler font, ruler text size, ruler text style, and ruler color of the ruler governing the paragraph. You can change these properties using the ruler pop-up menu in the ruler bar.

You can override the default text format for the selected text by applying specific formatting using the Notebook menu or the font, size, style, and color icons in the ruler bar.

You can set the text format of the selected text to the ruler default by choosing Notebook→Set Text Format to Ruler Default or by clicking the Default button in the ruler bar.

You should use the ruler to set the basic text format and use overrides for highlighting or other effects. For example, you might override the ruler text format to underline a short stretch of text or to switch to italic.

If you make a selection from the font, size, style, or color submenus of the ruler pop-up menu in the ruler bar, you redefine the current ruler, and all text governed by the ruler is updated.

If you make a selection from the font, size, style, or color submenus of the Notebook menu in the main menu bar, you override the text format for the selected text, and only the selected text is updated. This is also what happens if you use the font, size, style, and color icons in the ruler bar.

Text Sizes

The Text Size submenu in the Notebook menu contains an Other item. This leads to the Set Text Size dialog in which you can specify which sizes should appear in the Text Size submenu.

The text sizes in your Text Size menu are stored in the Igor preferences file so that the menu will include your preferred sizes each time you run Igor.

Vertical Offset

The vertical offset property is available only in formatted notebooks and is used mainly to implement superscript and subscript, as described in the next section.

Vertical offset is also useful for aligning a picture with text within a paragraph. For example, you might want to align the bottom of the picture with the baseline of the text.

The easiest way to do this is to use Control-Up Arrow and Control-Down Arrow key combinations (*Macintosh*) or Alt+Up Arrow and Alt+Down Arrow key combinations (*Windows*), which tweak the vertical offset by one point at a time.

You can set the vertical offset by choosing Notebook→Set Text Format which displays the Set Text Format dialog.

Superscript and Subscript

The last four items in the Text Size submenu of the Notebook menu have to do with superscript and subscript. Igor implements superscript and subscript by setting the text size and the vertical offset of the selected text to achieve the desired effect. They are not character properties but rather are effects accomplished using character properties.

The following table illustrates the use and effects of each of these items.

Action	Effect on Character Properties	Result
Type “XYZ”.		XYZ
Highlight “Y” and then choose Superscript.	Reduces text size and sets vertical offset for “Y”.	$X^{Y}Z$
Highlight “Z” and then choose Superscript.	Sets text size and vertical offset for “Z” to make it superscript relative to “Y”.	$X^{Y^{Z}}$
Highlight “Z” and then choose In Line.	Sets text size and vertical offset for “Z” to be same as for “Y”.	X^{YZ}
Highlight “YZ” and then choose Normal.	Sets text size for “YZ” same as “X” and sets vertical offset to zero.	XYZ

Help Links in Formatted Notebooks

You can format text as a help link by selecting the text and clicking the Help button in the ruler. Usually you do this while editing a help file.

Clicking text in a notebook that is formatted as a help link goes to the specified help topic. This allows you to easily link to help topics from a documentation notebook. The help link text must reference a topic or subtopic in a help file; the target can not be in a notebook.

Since simply clicking a help link in a notebook activates the link, to edit a help link you need to do one of the following:

- Use the arrow keys instead of simply clicking in the help link
- Press the command key (*Macintosh*) or Ctrl key (*Windows*) while clicking in the help link
- Select at least one character instead of simply clicking in the help link

Notebook Read/Write Properties

There are three properties that control whether a notebook can be modified.

Read-only

The read-only property is set if you open the file for read-only by executing **OpenNotebook**/R. It is also set if you open a file for which you do not have read/write permission.

When the read-only property is set, a lock icon appears in the bottom/left corner of the notebook window and you can not modify the notebook manually or via commands.

The read-only property can not be changed after the notebook is opened.

Use read-only if you want no modifications to be made to the notebook.

Write-protect

You can set the write-protect property to on or off by clicking the pencil icon in the bottom/left corner of the notebook window or using the **Notebook** operation with the writeProtect keyword.

The write-protect property is intended to give the user a way to prevent inadvertent manual modifications to the notebook. The user can turn the property on or off at will.

The write-protect property does not affect commands such as **Notebook** and **NotebookAction**. Even if write-protect is on, they can still modify the notebook.

Use write-protect if you want to avoid inadvertent manual modifications to the notebook but want the user to be able to take full control.

Changeable By Command Only

You can control the changeableByCommandOnly property using **NewNotebook**/OPTS=8 or using the **Notebook** operation with the changeableByCommandOnly keyword.

This property is intended to allow programmers to control whether the user can manually modify the notebook or not. Its main purpose is to allow a programmer to create a notebook subwindow in a control panel for displaying status messages and other information that is not intended to be modified by the user. There is no way to manually change this property - it can be changed by command only.

When the changeableByCommandOnly property is on, a lock icon appears in the bottom/left corner of the notebook window.

Use changeableByCommandOnly if you want no manual modifications to be made to the notebook but want it to be modifiable via commands.

The changeableByCommandOnly property is intended for programmatic use only and is not saved to disk.

For further information on notebook subwindows, see **Notebooks as Subwindows in Control Panels** on page III-91.

Working with Rulers

A ruler is a set of paragraph properties that you can apply to paragraphs in a formatted notebook. Using rulers, you can make sure that paragraphs that you *want* to have the same formatting *do* have the same formatting. Also, you can redefine the format of a ruler and all paragraphs governed by that ruler will be automatically updated.

In a simple notebook, you might use just the one built-in ruler, called Normal. In a fancier notebook, where you are concerned with presentation, you might use several rulers.

The pop-up menu on the left side of the ruler shows which ruler governs the first currently selected paragraph. You can use this pop-up menu to:

- Apply an existing ruler to the selected paragraphs
- Create a new ruler
- Redefine an existing ruler
- Find where a ruler is used
- Rename a ruler
- Remove a ruler from the document

Defining a New Ruler

To create a new ruler, choose Define New Ruler from the Ruler pop-up menu. This displays the Define New Ruler dialog.

Enter a name for the ruler. Ruler names must follow rules for standard (not liberal) Igor names. They may be up to 31 bytes in length, must start with a letter and may contain letters, numbers and the underscore character.

Use the icons in the dialog’s ruler bar to set the font, text size, text style, and color for the new ruler.

Click OK to create the new ruler.

In a sophisticated word processor, a ruler can be based on another ruler so that changing the first ruler automatically changes the second. Igor rulers do not have this capability.

Redefining a Ruler

When you redefine a ruler, all paragraphs governed by the ruler are automatically updated. There are three principal ways to redefine a ruler:

- Use the Redefine Ruler dialog.
- Use the Ruler Font, Ruler Text Size, Ruler Text Style or Ruler Text Color pop-up menu items.
- Use the Redefine Ruler from Selection item in the Ruler pop-up menu.
- Press the Command key (Macintosh) or Ctrl key (Windows) while using a margin, indent, tab, alignment, or spacing control in the ruler bar.

To invoke the Redefine Ruler dialog, choose Redefine Ruler from the Ruler pop-up menu.

Another handy way to redefine an existing ruler (e.g. Normal) is to adjust it, creating a derived ruler (e.g. Normal+). Then choose Redefine Ruler from Selection from the Ruler pop-up menu. This redefines the explicitly named ruler (Normal) to match the current ruler (Normal+).

Creating a Derived Ruler

You can adjust a ruler using its icons. When you do this, you create a *derived* ruler. A derived ruler is usually a minor variation of an explicitly created ruler.

If you redefine the Normal ruler, the Normal+ ruler is *not* automatically redefined. This is a limitation in Igor’s implementation of rulers compared to a word-processor program.

Finding Where a Ruler Is Used

You can find the next or previous paragraph governed by a particular ruler. To do this press Option (*Macintosh*) or Alt (*Windows*) while selecting the name of the ruler from the Ruler pop-up menu. To search backwards, press Shift-Option (*Macintosh*) or Shift+Alt (*Windows*) while selecting the ruler. If there is no next or previous use of the ruler, Igor will emit a beep.

Removing a Ruler

Rulers that you no longer need clutter up the Ruler pop-up menu. You can remove them from the document by choosing Remove Ruler from the Ruler pop-up menu.

You might want to know if a particular ruler is used in the document. The only way to do this is to search for the ruler. See **Finding Where a Ruler Is Used** on page III-12.

Transferring Rulers Between Notebooks

The only way to transfer a ruler from one notebook to another is by copying text from the first notebook and pasting it in the second. Rulers needed for the text are also copied and pasted. If a ruler that exists in the source notebook also exists in the destination, the destination ruler takes precedence.

If you expect to create a lot of notebooks that share the same rulers then you should create a template document with the common rulers. See **Notebook Template Files** on page III-30 for details.

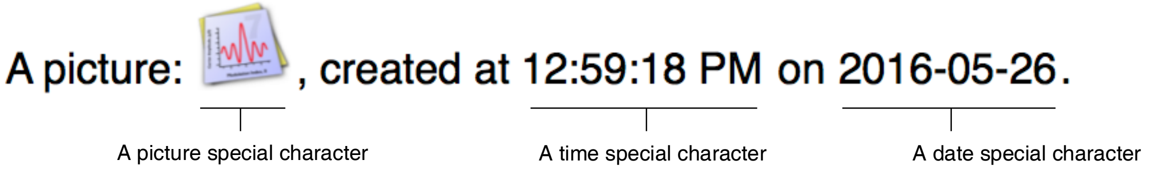
Special Characters

Aside from regular text characters, there are some special things that you can put into a paragraph in a formatted notebook. This table lists of all of the types of special characters and where they can be used.

Special Character Type	Where It Can Be Used
Picture	Main body text, headers and footers.
Igor-object picture (from graph, table, layout)	Main body text, headers and footers.
The date	Main body text, headers and footers.
The time	Main body text, headers and footers.
The name of the experiment file	Main body text, headers and footers.
Notebook window title	Headers and footers only.
Current page number	Headers and footers only.
Total number of pages	Headers and footers only.
Actions	Main body text only.

The main way in which a special character differs from a normal character is that it is not simply a text character. Another significant difference is that some special characters are dynamic, meaning that Igor can update them automatically. Other special characters, while not dynamic, are linked to Igor graphs, tables or page layouts (see **Using Igor-Object Pictures** on page III-18).

This example shows three kinds of special characters:



A picture: , created at 12:59:18 PM on 2016-05-26.

A picture special character

A time special character

A date special character

The time and date look like normal text but they are not. If you click any part of them, the entire time or date is selected. They act like a single character.

An action is a special character which, when clicked, runs Igor commands. See **Notebook Action Special Characters** on page III-14 for details.

Except for pictures, which are pasted, special characters are inserted using the Special submenu in the Notebook menu or using the Insert pop-up menu in the ruler area of a formatted notebook.

Inserting Pictures

You can insert pictures, including Igor-object pictures, by merely doing a paste. You can also insert pictures using Edit→Insert File or using the Notebook insertPicture operation.

When you insert a picture, the contents of the picture file are copied into the notebook. No link to the picture file is created.

If you use a platform-independent picture format, such as PNG (recommended), JPEG, TIFF, PDF, or SVG, then the picture is displayed correctly on all platforms. If you use a platform-specific picture format, such as Enhanced Metafile on Windows, the picture is displayed as a gray box if viewed on the other platform. On Windows, PDF is displayed correctly in Igor Pro 9.00 or later; older versions displayed PDF as a gray box.

Saving Pictures

You can save a picture in a formatted text notebook as a standalone picture file. Select one picture and one picture only. Then choose File→Save Graphics. You can also save a picture using the Notebook savePicture operation.

Special Character Names

Each special character has a name. For most types, the name is automatically assigned by Igor when the special character is created. However for action special characters you specify the name through the Special→New Action dialog. When you click a special character, you will see the name in the notebook status area. Special character names must be unique within a particular notebook.

The special character name is used only for specialized applications and usually you can ignore it. You can use the name with the Notebook findSpecialCharacter operation to select special characters. You can get a list of special character names from the **SpecialCharacterList** function (see page V-895) and get information using the **SpecialCharacterInfo** function (see page V-893).

When you copy a graph, table, layout, or Gizmo plot and paste it into a notebook, an Igor-object picture is created (see **Using Igor-Object Pictures** on page III-18). The Igor-object picture, like any notebook picture, is a special character and thus has a special character name, which, whenever possible, is the same as the source window name. However, this may not always possible such as when, for example, you paste Graph0 twice into a notebook, the first special character will be named *Graph0* and the second *Graph0_1*.

The Special Submenu

Using the Special submenu of the Notebook menu you can:

- Frame or scale pictures
- Insert special characters
- Control updating of special characters
- Convert a picture to cross-platform PNG format
- Specify an action character that executes commands

Scaling Pictures

You can scale a picture by choosing Notebook→Special→Scale or by using the Notebook command line operation. There is currently no way to scale a picture using the mouse.

Updating Special Characters

The window title, page number and total number of pages are dynamic characters—Igor automatically updates them when you print a notebook. These are useful for headers and footers. All other kinds of special characters are not dynamic but Igor makes it easy for you to update them if you need to, using the Update Selection Now or Update All Now items in the Special menu.

To prevent inadvertent updating, Igor disables these items until you enable updating, using the Enable Updating item in the Special menu. This enables updating for the active notebook.

If you are using a notebook as a form for generating reports, you will probably want to enable updating. However, if you are using it as a log of what you have done, you will want to leave updating in the disabled state.

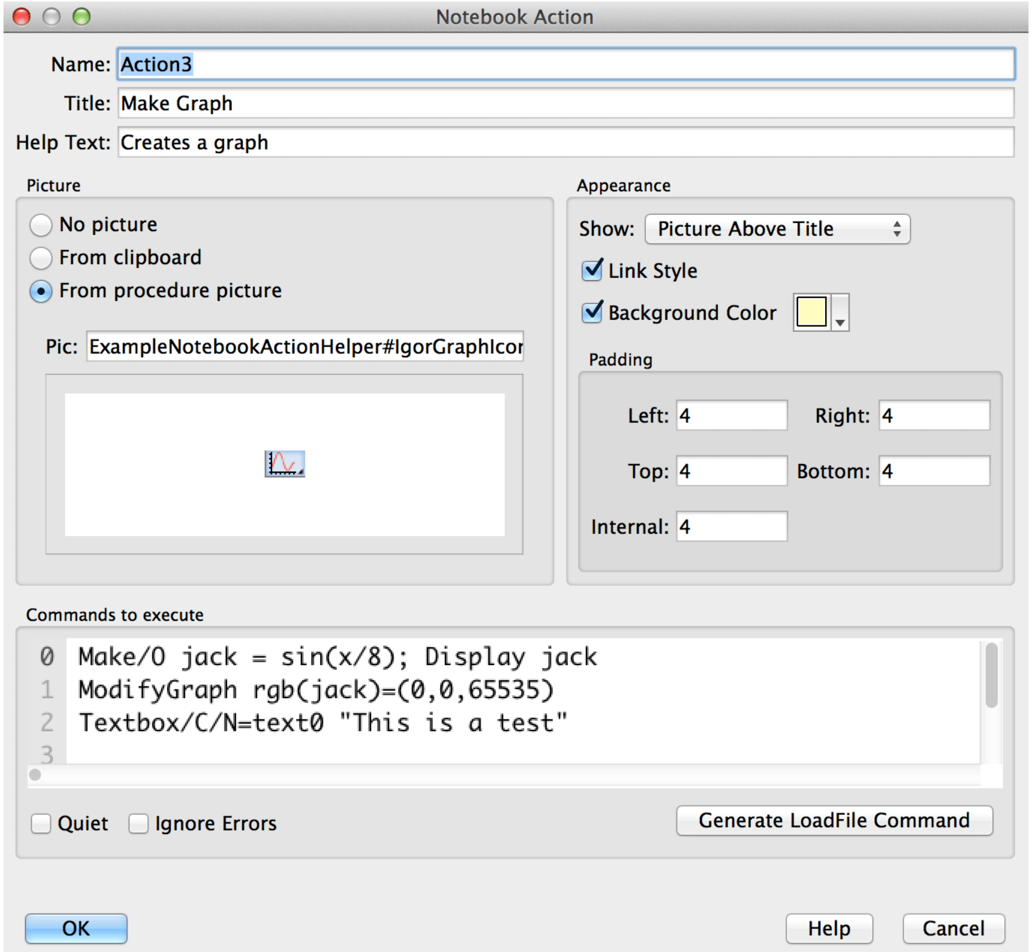
Notebook Action Special Characters

An action is a special character that runs commands when clicked. Use actions to create interactive notebooks, which can be used for demonstrations or tutorials. Help files are formatted notebook files so actions can also be used in help files.

You create actions in a formatted text notebook. You can invoke actions from formatted text notebooks or from help files.

For a demonstration of notebook actions, see the Notebook Actions Demo experiment.

To create an action use the **NotebookAction** operation (see page V-711) or choose Notebook→Special→New Action to invoke the Notebook Action dialog:



Name: Action3

Title: Make Graph

Help Text: Creates a graph

○ No picture
○ From clipboard
● From procedure picture

Pic: ExampleNotebookActionHelper#IgorGraphIcor

Show: Picture Above Title

☑ Link Style

☑ Background Color

Left: 4 Right: 4

Top: 4 Bottom: 4

Internal: 4

```
Make/O jack = sin(x/8); Display jack
ModifyGraph rgb(jack)=(0,0,65535)
Textbox/C/N=text0 "This is a test"
```

☐ Quiet ☐ Ignore Errors

Each action has a name that is unique within the notebook.

The title is the text that appears in the notebook. The text formatting of the notebook governs the default text formatting of the title.

If the Link Style checkbox is selected, the title is displayed like an HTML link — blue and underlined. This style overrides the color and underline formatting applied to the action through the Notebook menu.

The help text is a tip that appears when the cursor is over an action, if tips are enabled in the Help section of the Miscellaneous Settings dialog.

An action can have an associated picture that is displayed instead of or in addition to the title. There are two ways to specify a picture. You can paste one into the dialog using the Paste button or you can reference a Proc Picture stored in a procedure file. The latter source may be useful for advanced programmers (see **Proc Pictures** on page IV-56 for details).

For most purposes it is better to use a picture rather than a Proc Picture. One exception is if you have to use the same picture many times in the notebook, in which case you can save disk space and memory by using a Proc Picture.

If you designate a Proc Picture using a module name (e.g., `MyProcPictures#MyPicture`), then the Proc Picture must be declared static.

If you specify both a Proc Picture and a regular picture, the regular picture is displayed. If you specify no regular picture and your Proc Picture name is incorrect or the procedure file that supplies the Proc Picture is not open or not compiled, `"???"` is displayed in place of the picture.

In order for a picture to display correctly on both Macintosh and Windows, it must be in a cross-platform format such as PNG. You can convert a picture to PNG by clicking the Convert To PNG button. This affects the regular picture only.

Pictures and Proc Picture in actions are drawn transparently. The background color shows through white parts of the picture unless the picture explicitly erases the background.

The action can display one of six things as determined by the Show popup menu:

- The title
- The picture
- The picture below the title
- The picture above the title
- The picture to the left of the title
- The picture to the right of the title

If there is no picture and you choose one of the picture modes, just the title is displayed.

You can add padding to any external side of the action content (title or picture). The Internal Padding value sets the space between the picture and the title when both are displayed. All padding values are in points.

If you enable the background color, the rectangle enclosing the action content is painted with the specified color.

You can enter any number of commands to be executed in the Commands area. When you click the action, Igor sends each line in the Commands area to the Operation Queue, as if you called the Execute/P operation, and the commands are executed.

In addition to regular commands, you can enter special operation queue commands like INSERTINCLUDE, COMPILEPROCEDURES, and LOADFILE. These are explained under **Operation Queue** on page IV-278.

For sophisticated applications, the commands you enter can call functions that you define in a companion “helper procedure file” (see **Notebook Action Helper Procedure Files** on page III-17).

If the Quiet checkbox is selected, commands are not sent to the history area after execution.

If the Ignore Errors checkbox is selected then command execution errors are not reported via error dialogs.

The Generate LoadFile Command button displays an Open File dialog and then generates an Execute/P command to load the file into Igor. This is useful for generating a command to load a demo experiment, for example. This button inserts the newly-generated command at the selection point in the command area so, if you want the command to replace any pre-existing commands, delete any text in the command area before clicking the button. If the selected file is inside the Igor Pro Folder or any subdirectory, the generated path will be relative to the Igor Pro Folder. Otherwise it will be absolute.

Creating a Hyperlink Action

You can use a notebook action to create a hyperlink that displays a web page by calling the **BrowseURL** operation from the action's command.

Modifying Action Special Characters

You can modify an existing action by Control-clicking (*Macintosh*) or right-clicking (*Windows*) on it and choosing Modify Action from the pop-up menu, or by selecting the action special character, and nothing else, and then choosing Notebook→Special→Modify Action.

If you have opened a notebook as a help file and want to modify an action, you must close the help file (press Option or Alt and click the close button) and reopen it as a notebook (choose File→Open

File→Notebook). After editing the action, save the changes, close the notebook, and reopen it as a help file (choose File→Open File→Help File).

Modifying the Action Frame

If the notebook action has a picture, you can frame the action by choosing a frame style from the Notebook→Special→Frame submenu.

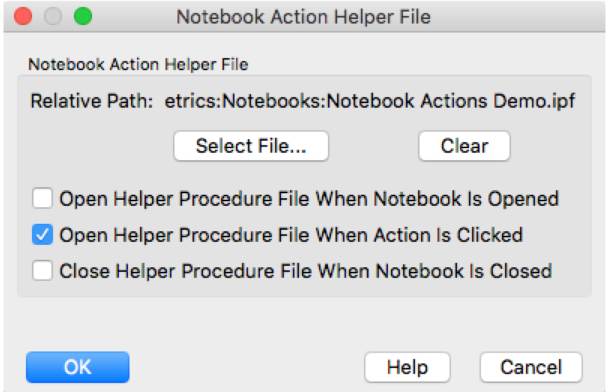
Modifying the Action Picture Scaling

If the notebook action has a picture, you can scale the picture by choosing an item from the Notebook→Special→Scale submenu.

Notebook Action Helper Procedure Files

In some instances you may want an action to call procedures in an Igor procedure file. The notebook action helper procedure file feature provides a convenient way to associate a notebook or help file with a procedure file.

Each formatted notebook (and consequently each help file) can designate only one procedure file as an action helper procedure file. Before choosing the helper file you must save the notebook as a standalone file on disk. Then choose Notebook→Special→Action Helper.



Notebook Action Helper File

Notebook Action Helper File

Relative Path: etrics:Notebooks:Notebook Actions Demo.ipf

□ Open Helper Procedure File When Notebook Is Opened

☑ Open Helper Procedure File When Action Is Clicked

□ Close Helper Procedure File When Notebook Is Closed

Click the File button to choose the helper procedure file for the notebook.

For most cases we recommend that you name your action helper procedure file with the same name as the notebook but with the .ipf extension. This indicates that the files are closely associated.

The helper file will usually be located in the same directory as the notebook file. Less frequently, it will be in a subdirectory or in a parent directory. It must be located on the same volume as the notebook file because Igor finds the helper using a relative path, starting from the notebook directory. If the notebook file is moved, the helper procedure file must be moved with it so that Igor will be able to find the helper using the relative path.

If Open Helper Procedure File When Notebook Is Opened is selected, the helper procedure file is opened along with the notebook. This checkbox can usually be left deselected. However, if you use Proc Pictures stored in the helper file, you should select it so that the pictures can be correctly rendered when the notebook is opened.

If Open Helper Procedure File When Action Is Clicked is selected, then, when you click an action, the procedure file loads, compiles, and executes automatically. This should normally be selected.

In both of these situations, the procedure file loads as a “global” procedure file, which means that it is not part of the current experiment and is not closed when creating a new experiment.

If Close Helper procedure File When Notebook Is Closed is selected and you kill a notebook or help file that has opened a helper file, the helper file is also killed. This should normally be selected.

To avoid unanticipated name conflicts between procedures in your helper file and elsewhere, it is a good idea to declare the procedures static (see **Static Functions** on page IV-105). In order to call such private routines you also need to assign a module name to the procedure file and use the module name when invoking the routines (see **Regular Modules** on page IV-236). For an example see the Notebook Actions Demo experiment.

Using Igor-Object Pictures

You create a picture from an Igor graph, table, page layout or Gizmo plot by choosing Edit→Export Graphics to copy a picture to the clipboard. For graphs, layouts, and Gizmo plots, you can also choose Edit→Copy. When you do this, Igor puts on the clipboard information about the window from which the picture was generated. When you paste into a notebook, Igor stores the window information with the picture. We call this kind of picture an “Igor-object” picture.

The Igor-object information contains the name of the window from which the picture was generated, the date/time at which it was generated, the size of the picture and the export mode used to create the picture. Igor uses this information to automatically update the picture when you request it.

Because of backward compatibility issues, this feature works only if the name of the window is 31 bytes or less.

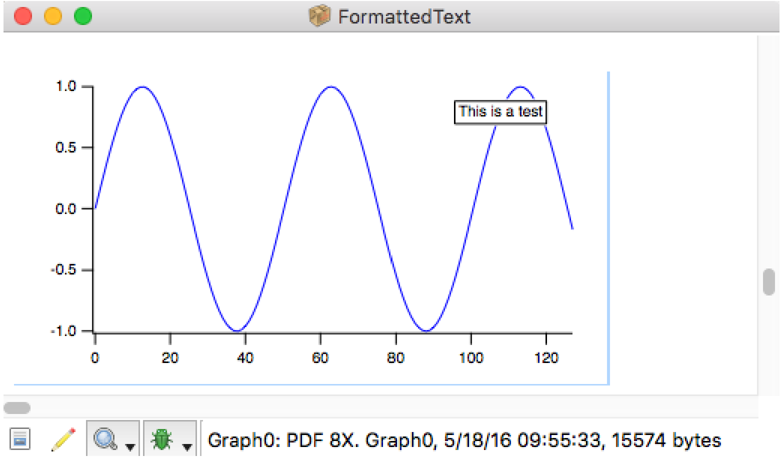
Igor can not link Igor-object pictures to a window in a different Igor experiment.

For good picture quality that works across platforms, we recommend that you use a high-resolution PNG format.

Updating Igor-Object Pictures

Before updating Igor object pictures, you must enable updating using the Notebook→Special→Enable Updating menu item. This is a per-notebook setting.

When you click an Igor-object picture, Igor displays the name of the object from which the picture was generated and the time at which it was generated in the notebook’s status area.



Graph0: PDF 8X. Graph0, 5/18/16 09:55:33, 15574 bytes

The first Graph0 shown in the status area is the name of the picture special character and the second Graph0 is the name of the source graph for the picture. There is no requirement that these be the same but they usually will be.

If you change the Igor graph, table, layout or Gizmo plot, you can update the associated picture by selecting it and choosing Update Selection Now from the Notebook→Special menu or by right-clicking and choosing Update Selection from the contextual menu. You can update all Igor-object pictures as well as any other special characters in the notebook by clicking anywhere so that nothing is selected and then choosing Update All Now from the Notebook→Special menu.

An Igor object picture can be updated even if it was created on another platform using a platform-dependent format. For example, you can create an EMF Igor object picture on Windows and paste it into a notebook. If you open the notebook on Macintosh, the EMF will display as a gray box because EMF is a Windows-specific format. However, if you right-click the EMF picture and choose Update Selection, Igor will regenerate it using a Macintosh format.

An Igor-object picture never updates unless you do so. Thus you can keep pictures of a given object taken over time to record the history of that object.

The Size of the Picture

The size of the picture is determined when you initially paste it into the notebook. If you update the picture, it will stay the same size, even if you have changed the size of the graph window from which the picture is derived. Normally, this is the desired behavior. If you want to change the size of the picture in the notebook, you need to repaste a new picture over the old one.

Activating The Igor-Object Window

You can activate the window associated with an Igor-object picture by double-clicking the Igor object picture in the notebook. If the window exists it is activated. If it does not exist but the associated window recreation macro does exist, Igor runs the window recreation macro.

Breaking the Link Between the Object and the Picture

Lets say you create a picture from a graph and paste it into a notebook. Now you kill the graph. When you click the picture, Igor displays a question mark after the name of the graph in the notebook's status area to indicate that it can't find the object from which the picture was generated. Igor can not update this picture. If you recreate the graph or create a new graph with the same name, this reestablishes the link between the graph and the picture.

If you change the name of a graph, this breaks the link between the graph and the picture. To reestablish it, you need to create a new picture from the graph and paste it into the layout.

Compatibility Issues

A Windows format picture, when updated on Macintosh, is converted to a Macintosh format, and vice versa.

Igor does not recognized Igor-object pictures created by Igor versions before 6.10.

If you save a notebook containing a Gizmo picture and open it in Igor6 version 6.37 or before, you will get errors in Igor6. If you open it in Igor6 version 6.38 or later, it will display correctly.

Cross-Platform Pictures

If you want to create a notebook that contains pictures that display correctly on both Macintosh and Windows, you can use the PDF or PNG (Portable Network Graphics) format. If some pictures are already in JPEG or TIFF format, these too will display correctly on either platform.

PDF pictures are displayed correctly on Windows in Igor Pro 9.00 or later. In earlier versions on Windows, PDF pictures are displayed as gray boxes.

You can convert other types of pictures to PNG using the Convert to PNG item in the Special submenu of the Notebook menu.

Page Breaks

When you print a notebook, Igor automatically goes to the next page when it runs out of room on the current page. This is an automatic page break.

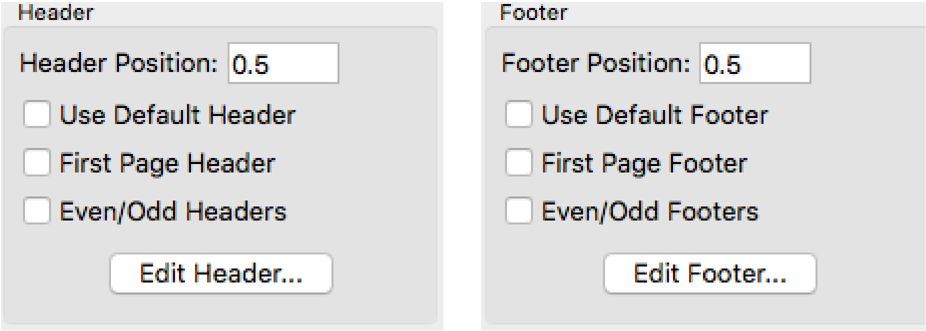
In a formatted notebook, you can insert a *manual* page break using the Insert Page Break item in the Edit menu. Igor displays a manual page break as a dashed line across the notebook window. You can't insert a manual page break into a plain notebook.

There is no way to see where Igor will put automatic page breaks other than by printing the document or using the Print Preview feature.

Headers and Footers

Both formatted and plain notebooks can have headers or footers.

You create headers and footers using the Document Settings dialog.



Header

Header Position: 0.5

□ Use Default Header

□ First Page Header

□ Even/Odd Headers

Footer

Footer Position: 0.5

□ Use Default Footer

□ First Page Footer

□ Even/Odd Footers

The header position is specified as a distance from the top of the paper. The footer position is specified as a distance from the bottom of the paper. The units are your preferred units as specified in the Miscellaneous section of the Miscellaneous Settings dialog.

For most uses, the default header or default footer will be sufficient. The default header consists of the date, window title and page number. The other options are intended for use in fancy reports.

Clicking Edit Header or Edit Footer displays a subdialog in which you can enter text as well as special characters.

The Page Number and Number of Pages special characters are displayed on the screen as # characters but are printed using the actual page number and number of pages in the document. The Date, Time and Window Title special characters are automatically updated when the document is printed.

Header and footer settings are saved in formatted notebook files but not in plain text notebook files. Consequently, settings for plain text notebook files revert to preferred settings, as set by the Capture Notebook Prefs dialog, when you open a file as a plain text notebook.

Printing Notebooks

To print an entire notebook, click so that no text is selected and then choose File→Print Notebook Window.

To print part of a notebook, select the section that you want to print and then choose File→Print Notebook Selection.

Import and Export Via Rich Text Format Files

The Rich Text Format (RTF) is a file format created by Microsoft Corporation for exchanging formatted text documents between programs. Microsoft also calls it the Interchange format. Many word processors can import and export RTF.

You can save an Igor plain or formatted notebook as an RTF file and you can open an RTF file as an Igor formatted notebook. You may find it useful to collect text and pictures in a notebook and to later transfer it to your word processor for final editing.

An RTF file is a plain text file that contains RTF codes. An RTF file starts with "\rtf". Other codes define the text, pictures, document formats, paragraph formats, and text formats and other aspects of the file.

When Igor writes an RTF file from a notebook, it must generate a complex sequence of codes. When it reads an RTF file, it must interpret a complex sequence of codes. The RTF format is very complicated, has evolved and allows some flexibility. As a result, each program writes and interprets RTF codes somewhat differently. Because of this and because of the different feature sets of different programs, RTF translation is sometimes imperfect and requires that you do manual touchup.

Saving an RTF File

To create an RTF file, choose Save Notebook As from the File menu. Select Rich Text Format from the Save File dialog's file type pop-up menu, and complete the save.

The original notebook file, if any, is not affected by saving as RTF, and the notebook retains its connection to the original file.

Opening an RTF File

When Igor opens a plain text file as a notebook, it looks for the "\rtf" code that identifies the file as an RTF file. If it sees this code, it displays a dialog asking if you want to convert the rich text codes into an Igor formatted notebook.

If you answer Yes, Igor creates a new, formatted notebook. It then interprets the RTF codes and sets the properties and contents of the new notebook accordingly. When the conversion is finished, you sometimes need to fix up some parts of the document that were imperfectly translated.

If you answer No, Igor opens the RTF file as a plain text file. Use this to inspect the RTF codes and, if you are so inclined, to tinker with them.

Exporting a Notebook as HTML

Igor can export a notebook in HTML format. HTML is the format used for Web pages. For a demo of this feature, choose File→Example Experiments→Feature Demos→Web Page Demo.

This feature is intended for two kinds of uses. First, you can export a simple Igor notebook in a form suitable for immediate publishing on the Web. This might be useful, for example, to automatically update a Web page or to programmatically generate a series of Web pages.

Second, you can export an elaborate Igor notebook as HTML, use an HTML editor to improve its formatting or tweak it by hand, and then publish it on the Web. It is unlikely that you could use Igor alone to create an elaborately formatted Web page because there is a considerable mismatch between the feature set of HTML and the feature set of Igor notebooks. For example, the main technique for creating columns in a notebook is the use of tabs. But tabs mean nothing in HTML, which uses tables for this purpose.

Because of this mismatch between notebooks and HTML, and so your Web page works with a wide variety of Web browsers, we recommend that you keep the formatting of notebooks which you intend to write as HTML files as simple as possible. For example, tabs and indentation are not preserved when Igor exports HTML files, and you can't rely on Web browsers to display specific fonts and font sizes. If you restrict yourself to plain text and pictures, you will achieve a high degree of browser compatibility.

There are two ways to export an Igor notebook as an HTML file:

- Choose File→Save Notebook As
- Using the `SaveNotebook/S=5` operation

The **SaveNotebook** operation (see page V-823) includes a /H flag which gives you some control over the features of the HTML file:

- The file's character encoding
- Whether or not paragraph formatting (e.g., alignment) is exported
- Whether or not character formatting (e.g., fonts, font sizes) is exported
- The format used for graphics

When you choose File→Save Notebook As, Igor uses the following default parameters:

- Character encoding: UTF-8 (see **HTML Text Encoding** on page III-23)
- Paragraph formatting is not exported
- Character formatting is not exported
- Pictures are exported in the PNG (Portable Network Graphics) format

By default, paragraph and character formatting is not exported because this formatting is often not supported by some Web browsers, is at cross-purposes with Web browser behavior (e.g., paragraph space-before and space-after), or is customarily left in the hands of the Web viewer (e.g., fonts and font sizes).

For creating simple Web pages that work with a majority of Web browsers, this is all you need to know about Igor's HTML export feature. To use advanced formatting, to use non-Roman characters, to use different graphics formats, and to cope with diverse Web browser behavior, you need to know more. Unfortunately, this can get quite technical.

HTML Standards

Igor's HTML export routine writes HTML files that conform to the HTML 4.01 specification, which is available from:

```
http://www.w3.org/TR/1999/PR-html40-19990824
```

It writes style information that conforms to the CSS1 (Cascading Style Sheet - Level 1) specification, which is available from:

```
http://www.w3.org/TR/1999/REC-CSS1-19990111
```

HTML Horizontal Paragraph Formatting

Tabs mean nothing in HTML. A tab behaves like a single space character. Consequently, you can not rely on tabs for notebooks that are intended to be written as HTML files. HTML has good support for tables, which make tabs unnecessary. However, Igor notebooks don't support tables. Consequently, there is no simple way to create an HTML file from an Igor notebook that relies on tabs for horizontal formatting.

HTML files are often optimized for viewing on screen in windows of varying widths. When you make the window wider or narrower, the browser automatically expands and contracts the width of the text. Consequently, the roles played by the left margin and right margin in notebooks are unnecessary in HTML files. When Igor writes an HTML file, it ignores the left and right paragraph margin properties.

HTML Vertical Paragraph Formatting

The behavior of HTML browsers with regard to the vertical spacing of paragraphs makes it difficult to control vertical formatting. For historical reasons, browsers typically add a blank line after each paragraph (<P>) element and they ignore empty paragraph elements. Although it is possible to partially override this behavior, this only leads to more problems.

In an Igor notebook, you would usually use the space-before and space-after paragraph properties in place of blank lines to get paragraph spacing that is less than one line. However, because of the aforementioned browser behavior, the space-before and space-after would add to the space that the browser already adds and you would get more than one line's space when you wanted less. Consequently, Igor ignores the space-before and space-after properties when writing HTML files.

The minimum line height property is written as the CSS1 line-height property, which does not serve exactly the same purpose. This will work correctly so long as the minimum line height that you specify is greater than or equal to the natural line height of the text.

HTML Character Formatting

In an Igor notebook, you might use different fonts, font sizes, and font styles to enhance your presentation. An HTML file is likely to be viewed on a wide range of computer systems and it is likely that your enhancements would be incorrectly rendered or would be a hindrance to the reader. Consequently, it is customary to leave these things to the person viewing the Web page.

If you use the **SaveNotebook** operation (see page V-823) and enable exporting font styles, only the bold, underline and italic styles are supported.

In notebooks, the vertical offset character property is used to create subscripts and superscripts. When writing HTML, Igor uses the CSS vertical-align property to represent the notebook's vertical offset. The HTML property and the Igor notebook property are not a good match. Also, some browsers do not support the vertical-align property. Consequently, subscripts and superscripts in notebooks may not be properly rendered in HTML. In this case, the only workaround is to use a picture instead of using the notebook subscript and superscript.

HTML Pictures

If the notebook contains one or more pictures, Igor writes PNG or JPEG picture files to a “media” folder. For example, if the notebook contains two pictures and you save it as “Test.htm”, Igor writes the file Test.htm and creates a folder named TestMedia. It stores in the TestMedia folder two picture files: Picture0.png (or .jpg) and Picture1.png (or .jpg). The names of the picture files are always of the form Picture<N> where N is a sequential number starting from 0. If the folder already exists when Igor starts to store pictures in it, Igor deletes all files in the folder whose names start with “Picture”, since these files are most likely left over from a previous attempt to create the HTML file.

When you choose Save Notebook As from the File menu, Igor always uses the PNG format for pictures. If you want to use the JPEG format, you must execute a **SaveNotebook** operation (see page V-823) from the command line, using the /S=5 flag to specify HTML and the /H flag to specify the graphics format.

PNG is a lossless format that is excellent for storing web graphics and is supported by virtually all recent web browsers. JPEG is a lossy format commonly used for web graphics. We recommend that you use PNG.

HTML does not support Igor's double, triple, or shadow picture frames. Consequently, when writing HTML, all types of notebook frames are rendered as HTML thin frames.

HTML Text Encoding

By default, Igor uses UTF-8 text encoding when you save a notebook as HTML. For historical reasons, the SaveNotebook operation /H flag allows you to specify other text encodings. However, there is no reason to use anything other than UTF-8.

Embedding HTML Code

If you are knowledgeable about HTML, you may want to access the power of HTML without completely giving up the convenience of having Igor generate HTML code for you. You can do this by embedding HTML code in your notebook, which you achieve by simply using a ruler named HTMLCode.

Normally, Igor translates the contents of the notebook into HTML code. However, when Igor encounters a paragraph whose ruler is named HTMLCode, it writes the contents of the paragraph directly into the HTML file. Here is a simple example:

Living things are generally classified into 5 kingdoms:

```
<OL>
<LI>Monera
<LI>Protista
<LI>Fungi
```

```
<LI>Plantae
<LI>Animalia
</OL>
```

In this example, the gray lines are governed by the HTMLCode ruler. Igor writes the text in these line directly to the HTML file. This example produces a numbered list, called an “ordered list”, which is announced using the HTML “OL” tag.

By convention, we make the ruler font color for the HTMLCode ruler gray. This allows us to distinguish at a glance the HTML code from the normal notebook text. The use of the color gray is merely a convention. It is the fact that the ruler is named HTMLCode that makes Igor write the contents of these paragraphs directly to the HTML file.

Here is an example that shows how to create a simple table:

```
<TABLE border="1" summary="Example of creating a table in HTML.">
<CAPTION><EM>A Simple Table</EM></CAPTION>
<TR><TH><TH>Col 1<TH>Col 2<TH>Col 3
<TR><TH>Row 1<TD>10<TD>20<TD>30
<TR><TH>Row 2<TD>40<TD>50<TD>60
</TABLE>
```

Here is an example that includes a link:

```
<P>Visit the <A HREF="http://www.wavemetrics.com/">WaveMetrics</A> web site</P>
```

Finding Notebook Text

You can find text in a notebook using the *find bar*. Choose Edit→Find or press Command-F (*Macintosh*) or Ctrl+F (*Windows*) to display it. The find bar allows you to set search parameters and search forward and backward.

On Macintosh, you can search for other occurrences of a string with minimal use of the mouse as follows:

1. Select the first occurrence of the string you want to find.
2. Press Command-E (Edit→Use Selection for Find).
3. Press Command-G (Edit→Find Same) to find subsequent occurrences.
4. Press Shift-Command-G (Edit→Find Same) to find previous occurrences.

On Windows you can search for other occurrences of a string with minimal use of the mouse as follows:

1. Select the first occurrence of the string you want to find.
2. Press Command-H (Edit→Find Selection) to find the next occurrence.
3. Press Command-G (Edit→Find Same) to find subsequent occurrences.
4. Press Shift-Command-G (Edit→Find Same) to find the previous occurrences.

You can also perform a Find on multiple help, procedure and notebook windows at one time. See **Finding Text in Multiple Windows** on page II-53.

Replacing Notebook Text

You can replace text in a notebook using the *find bar*. Choose Edit→Replace or press Command-R (*Macintosh*) or Ctrl+R (*Windows*) to display the bar in replace mode.

Another method for finding and replacing text consists of these steps:

1. Copy the replacement text to the clipboard.
2. Do a find to find the first occurrence of the text to be replaced.
3. Press Command-V (*Macintosh*) or Ctrl+V (*Windows*) to paste.

4. Press Command-G (*Macintosh*) or Ctrl+G (*Windows*) to find the next occurrence.

Notebook Names, Titles and File Names

This table explains the distinction between a notebook's name, its title and the name of the file in which it is saved.

Item	What It Is For	How It Is Set
Notebook name	Used to identify a notebook from an Igor command.	Igor automatically gives new notebooks names of the form Notebook0. You can change it using the Window Control dialog or using the DoWindow/C operation.
Notebook title	For visually identifying the window. The title appears in the title bar at the top of the window and in the Other Windows submenu of the Windows menu.	Initially, Igor sets the title to the concatenation of the notebook name and the file name. You can change it using the Window Control dialog or using the DoWindow/T operation.
File name	This is the name of the file in which the notebook is stored.	You enter this in the New Notebook dialog. Change it on the desktop.

Igor automatically opens notebooks that are part of an Igor experiment when you open the experiment. If you change a notebook's file name outside of the experiment, Igor will be unable to automatically open it and will ask for your help when you open the experiment.

A notebook file stored inside a packed experiment file does not exist separately from the experiment file, so there is no way or reason to change the notebook's file name.

Notebook Info Dialog

You can get general information on a notebook by selecting the Info item in the Notebook menu or by clicking the icon in the bottom/left corner of the notebook. This displays the File Information dialog.

The File Information dialog shows you whether the notebook has been saved and if so whether it is stored in a packed experiment file, in an unpacked experiment folder or in a stand-alone file.

Programming Notebooks

Advanced users may want to write Igor procedures to automatically log results or generate reports using a notebook. The operations that you would use are briefly described here.

Operation	What It Does
NewNotebook	Creates a new notebook window.
OpenNotebook	Opens an existing file as a notebook.
SaveNotebook	Saves an existing notebook to disk as a stand-alone file or packed into the experiment file.
PrintNotebook	Prints all of a notebook or just the selected text.
Notebook	Provides control of the contents and all of the properties of a notebook except for headers and footers. Also sets the selection and to search for text or graphics.
NotebookAction	Creates or modifies notebook action special characters.

Operation	What It Does
GetSelection	Retrieves the selected text.
SpecialCharacterList	Returns a list of the names of special characters in the notebook.
SpecialCharacterInfo	Returns information about a specific special character.
KillWindow	Kills a notebook.

There is currently no way to set headers and footers from Igor procedures. A workaround is to create a stationery (*Macintosh*) or template (*Windows*) notebook file with the headers and footers that you want and to open this instead of creating a new notebook.

In addition, the **SpecialCharacterList** function (see page V-895) and **SpecialCharacterInfo** function (see page V-893) may be of use.

The Notebook Demo #1 experiment, in the Examples:Feature Demos folder, provides a simple illustration of generating a report notebook using Igor procedures.

See **Notebooks as Subwindows in Control Panels** on page III-91 for information on using a notebook as a user-interface element.

Some example procedures follow.

Logging Text

This example shows how to add an entry to a log. Since the notebook is being used as a log, new material is always added at the end.

```
// Function AppendToLog(nb, str, stampDateTime)
// Appends the string to the named notebook.
// If stampDateTime is nonzero, appends date/time before the string.
Function AppendToLog(nb, str, stampDateTime)
   String nb              // name of the notebook to log to
   String str             // the string to log
   Variable stampDateTime // nonzero if we want to include stamp

   Variable now
   String stamp

   Notebook $nb selection={endOfFile, endOfFile}
   if (stampDateTime)
      now = datetime
      stamp = Secs2Date(now,0) + ", " + Secs2Time(now,0) + "\r"
      Notebook $nb text=stamp
   endif
   Notebook $nb text= str+"\r"
End
```

You can test this function with the following commands:

```
NewNotebook/F=1/N=Log1 as "A Test"
AppendToLog("Log1", "Test #1\r", 1)
AppendToLog("Log1", "Test #2\r", 1)
```

The **sprintf** operation (see page V-902) is useful for generating the string to be logged.

Inserting Graphics

There are two kinds of graphics that you can insert into a notebook under control of a procedure:

- A picture generated from a graph, table, layout or Gizmo plot (an “Igor-object” picture).
- A copy of a named picture stored in the current experiment’s picture gallery.

The command

```
Notebook Notebook0 picture={Graph0(0,0,360,144), -1, 0}
```

creates a new picture of the named graph and inserts it into the notebook. The numeric parameters allow you to control the size of the picture, the type of picture and whether the picture is black and white or color. This creates an anonymous (unnamed) picture. It has no name and does not appear in the Pictures dialog. However, it is an Igor-object picture with embedded information that allows Igor to recognize that it was generated from Graph0.

The command

```
Notebook Notebook0 picture={PICT_0, 1, 0}
```

makes a *copy* of the named picture, PICT_0, stored in the experiment's picture gallery, and inserts the copy into the notebook as an anonymous picture. The inserted anonymous picture is no longer associated with the named picture from which it sprang.

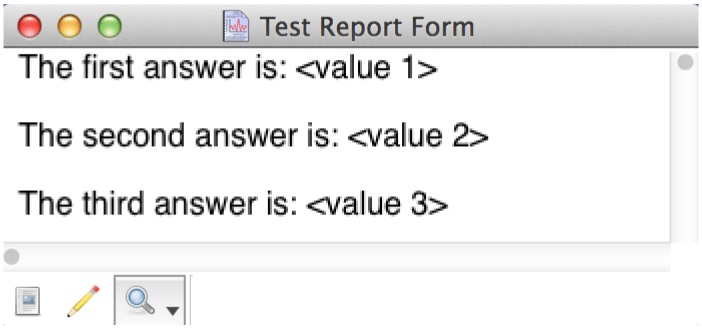
See **Pictures** on page III-509 for more information on pictures.

Updating a Report Form

In this example, we assume that we have a notebook that contains a form with specific values to be filled in. These could be the results of a curve fit, for example. This procedure opens the notebook, fills in the values, prints the notebook and then kills it.

```
// DoReport(value1, value2, value3)
// Opens a notebook file with the name "Test Report Form",
// searches for and replaces "<value 1>", "<value 2>" and "<value3>".
// Then prints the notebook and kills it.
// "<value 1>", "<value 2>" and "<value 3>" must appear in the form
// notebook, in that order.
// This procedure assumes that the file is in the Igor folder.
Function DoReport(value1, value2, value3)
   String value1, value2, value3

   OpenNotebook/P=IgorUserFiles/N=trf "Test Report Form.ifn"
   Notebook trf, findText={"<value 1>", 1}, text=value1
   Notebook trf, findText={"<value 2>", 1}, text=value2
   Notebook trf, findText={"<value 3>", 1}, text=value3

   PrintNotebook/S=0 trf
   KillWindow trf
End
```

To try this function, enter it in the Procedure window. Then create a formatted notebook that contains "<value 1>", "<value 2>" and "<value 3>" and save it in the Igor User Files folder using the file name "Test Report Form.ifn". The notebook should look like this:



The first answer is: <value 1>

The second answer is: <value 2>

The third answer is: <value 3>

Now kill the notebook and execute the following command:

```
DoReport("123", "456", "789")
```

This will print the form using the specified values.

Updating Igor-Object Pictures

The following command updates all pictures in the notebook made from Igor graphs, tables, layouts or Gizmo plots from the current experiment.

```
Notebook Notebook0 specialUpdate=0
```

More precisely, it will update all dynamic special characters, including date and time characters as well as Igor-object pictures.

This next fragment shows how to update just one particular Igor-object picture.

```
String nb = "Notebook0"
Notebook $nb selection={startOfFile, startOfFile}
Notebook $nb findPicture={"Graph0", 1}
if (V_Flag)
   Notebook $nb specialUpdate=1
else
   Beep              // can't find Graph0
endif
```

Igor will normally refuse to update special characters unless updating is enabled, via the Enable Updating dialog (Notebook menu). You can override this and force Igor to do the update by using 3 instead of 1 for the specialUpdate parameter.

Retrieving Notebook Text

Since you can retrieve text from a notebook, it is possible to use a notebook as an input mechanism for a procedure. To illustrate this, here is a procedure that tags each point of a wave in the top graph with a string read from the specified notebook. The do-loop in this example shows how to pick out each paragraph from the start to the end of the notebook.

```
#pragma rtGlobals=1  // Make V_Flag and S_Selection be local variables.

// TagPointsFromNotebook(nb, wave)
// nb is the name of an open notebook.
// wave is the name of a wave in the top graph.
// TagPointsFromNotebook reads each line of the notebook and uses it
// to tag the corresponding point of the wave.
Function TagPointsFromNotebook(nb, wave)
   String nb         // name of notebook
   String wave       // name of the wave to tag

   String name       // name of current tag
   String text       // text for current tag
   Variable p

   p = 0
   do
      // move to current paragraph
      Notebook $nb selection={(p, 0), (p, 0)}
      if (V_Flag)          // no more lines in file?
         break
      endif

      // select all characters in paragraph up to trailing CR
      Notebook $nb selection={startOfParagraph, endOfChars}

      GetSelection notebook, $nb, 2    // Get the selected text
      text = S_Selection      // S_Selection is set by GetSelection
      if (strlen(text) > 0)   // skip if this line is empty
         name = "tag" + num2istr(p)
         Tag/C/N=$name/F=0/L=0/X=0/Y=8 $wave, pnt2x($wave, p), text
      endif
      p += 1
   while (p < numpnts($wave)) // break if we hit the end of the wave
End
```

More Notebook Programming Examples

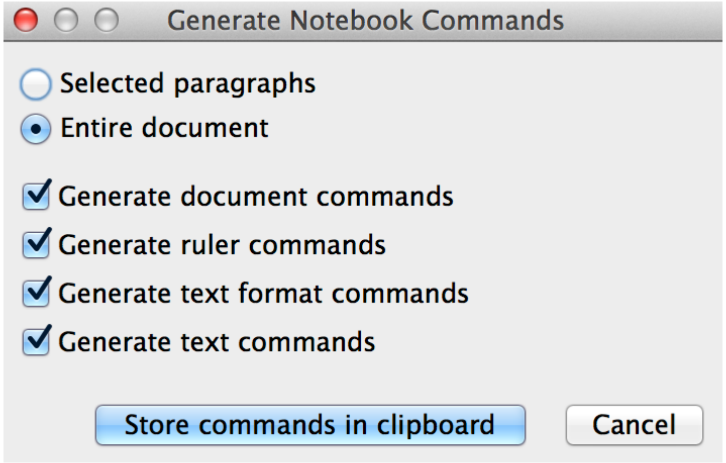
For examples using notebook action special characters, see the Notebook Actions Demo example experiment.

These example experiments illustrate notebook programming:

```
Igor Pro Folder:Examples:Feature Demos:Notebook Actions Demo.pxp
Igor Pro Folder:Examples:Testing:Notebook Operations Test.pxp
Igor Pro Folder:Examples:Testing:Notebook Picture Tests.pxp
Igor Pro Folder:Examples:Feature Demos 2:Notebook in Panel.pxp
```

Generate Notebook Commands Dialog

The Generate Notebook Commands dialog automatically generates the commands required to reproduce a notebook or a section of a notebook. This is intended to make programming a notebook easier. To use it, start by manually creating the notebook that you want to later create automatically from an Igor procedure. Then choose Generate Commands from the Notebook menu to display the corresponding dialog:


Generate Notebook Commands
○ Selected paragraphs
● Entire document
☑ Generate document commands
☑ Generate ruler commands
☑ Generate text format commands
☑ Generate text commands
Store commands in clipboard
Cancel

After clicking Store Commands in Clipboard, open the procedure window and paste the commands into a procedure.

For a very simple formatted notebook, the commands generated look like this:

```
String nb = "Notebook2"
NewNotebook/N=$nb/F=1/V=1/W=(5,40,563,359)
Notebook $nb defaultTab=36,pageMargins={54,54,54,54}
Notebook $nb showRuler=0,rulerUnits=1,updating={1,60}
Notebook $nb newRuler=Normal,justification=0,margins={0,0,504}
Notebook $nb spacing={0,0,0},tabs={}
Notebook $nb rulerDefaults={"Helvetica",10,0,(0,0,0)}
Notebook $nb ruler=Normal,text="This is a test."
```

To make it easier for you to modify the commands, Igor uses the string variable nb instead of repeating the literal name of the notebook in each command.

If the notebook contains an Igor-object picture, you will see a command that looks like

```
Notebook $nb picture={Graph0(0,0,360,144), 0, 1}
```

However, if the notebook contains a picture that is not associated with an Igor object, you will see a command that looks like

```
Notebook $nb picture={putGraphicNameHere, 1, 0}
```

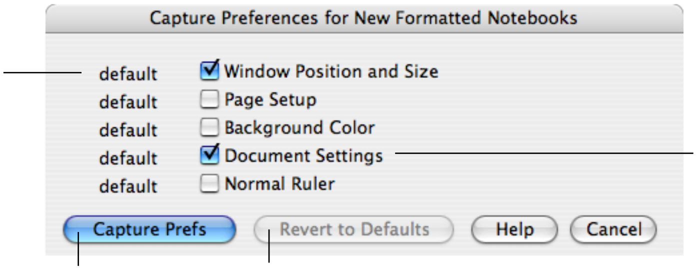
You will need to replace “putGraphicNameHere” with the name of a picture. Use the Pictures dialog, via the Misc menu, to see what named pictures are stored in the current experiment or to add a named picture. See **Pictures** on page III-509 for more information.

There is a shortcut that generates commands without going through the dialog. Select some text in the notebook, press Option (*Macintosh*) or Alt (*Windows*) and choose Copy from the Edit menu. This generates commands for the selected text and text formats. Press the Shift-Option (*Macintosh*) or Shift+Alt (*Windows*) to also generate document and ruler commands.

Notebook Preferences

The notebook preferences affect the creation of *new* notebooks. There is one set of preferences for plain notebooks and another set of preferences for formatted notebooks.

To set notebook preferences, set the attributes of any notebook of the desired type (plain or formatted) and then use the Capture Notebook Prefs item in the Notebook menu.



Capture Preferences for New Formatted Notebooks

default ☑ Window Position and Size
default □ Page Setup
default □ Background Color
default ☑ Document Settings
default □ Normal Ruler

“default” indicates that this preference was never changed from the factory default or was changed and then reverted.

This includes whether or not the ruler is initially showing as well as all settings in the Document Settings dialog.

Click to capture preferences for the selected

Click to revert to factory defaults for the selected items.

To determine what the preference settings are you must create a new notebook and examine its settings.

Notebook windows each have their own Page Setup values. New notebook windows will have their own copy of the captured Page Setup values.

Preferences are stored in the Igor Preferences file. See Chapter III-18, **Preferences**, for further information on preferences.

Notebook Template Files

A template notebook provides a way to customize the initial contents of a new notebook. When you open a template notebook, Igor opens it normally but leaves it untitled and disassociates it from the template notebook file. This leaves you with a new notebook based on your prototype. When you save the untitled notebook, Igor creates a new notebook file.

Template notebooks have ".ift" as the file name extension instead of ".ifn".

To make a template notebook, start by creating a prototype formatted text notebook with whatever contents you would like in a new notebook.

Choose File→Save Notebook As, choose IGOR Formatted Notebook Template from the file type pop-up menu, and save the template notebook.

You can convert an existing formatted text notebook file into a template file by changing the extension from ".ifn" to ".ift".

The Macintosh Finder’s file info window has a Stationery Pad checkbox. Checking it turns a file into a stationery pad. When you double-click a stationery pad file, Mac OS X creates a copy of the file and opens the copy. For most uses, the template technique is more convenient.

Notebook Shortcuts

To view text window keyboard navigation shortcuts, see **Text Window Navigation** on page II-51.

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To revisit a previously-viewed location	Press Cmd-Option-Left Arrow (Go Back) or Cmd-Option-Right Arrow (Go Forward).	Press Alt+Left Arrow (Go Back) or Alt+Right Arrow (Go Forward).
To get a contextual menu of commonly-used actions	Press Control and click in the body of the notebook window.	Right-click the body of the notebook window.
To execute commands in a notebook window	Select the commands or click in the line containing the commands and press Control-Return or Control-Enter.	Select the commands or click in the line containing the commands and press Ctrl+Enter.
To display the Find dialog	Press Command-F.	Press Ctrl+F.
To find the same text again	Press Command-G.	Press Ctrl+G.
To find again but in the reverse direction	Press Command-Shift-G.	Press Ctrl+Shift+G.
To find selected text	Press Command-E and Command-G.	Press Ctrl+H.
To find the selected text but in the reverse direction	Press Command-E and Command-Shift-G.	Press Ctrl+Shift+H.
To select a word	Double-click.	Double-click.
To select an entire line	Triple-click.	Triple-click.
To change a named ruler without using the Redefine Ruler dialog	Press Command while adjusting the icons in the ruler.	Press Ctrl while adjusting the icons in the ruler.
To find the next occurrence of a ruler	Press Option while selecting a ruler from the pop-up menu.	Press Alt while selecting a ruler from the pop-up menu.
To find the previous occurrence of a ruler	Press Shift-Option while selecting a ruler from the pop-up menu.	Press Shift+Alt while selecting a ruler from the pop-up menu.
To get miscellaneous information on notebook	Click the document icon in the bottom-left corner of the window.	Click the document icon in the bottom-left corner of the window.
To generate Notebook commands that will recreate the selected text	Press Option and choose Copy from the Edit menu. This puts the commands in the Clipboard for text and text formats.	Press Alt and choose Copy from the Edit menu. This puts the commands in the Clipboard for text and text formats.
	Press Option-Shift while copying to also generate document and ruler commands.	Press Alt+Shift while copying to also generate document and ruler commands.
To nudge a picture by one point up or down	Select the picture and press Control-Up Arrow or Control-Down Arrow.	Select the picture and press Alt+Up Arrow or Alt+Down Arrow.
To save all modified standalone procedure files	When a procedure window is active, choose File→Save All Standalone Procedure Files.	When a procedure window is active, choose File→Save All Standalone Procedure Files.

Chapter

III-2

Annotations

Overview

Annotations are custom objects that add information to a graph, page layout or Gizmo plot.

Most annotations contain text that you might use to describe the contents of a graph, point out a feature of a wave, identify the axis that applies to a wave, or create a legend. An annotation can also contain color scales showing the data range associated with colors in contour and image plots.

There are four types of annotation: textboxes, legends, color scales, and tags.

A textbox contains one or more lines of text which optionally may be surrounded by a frame, rotated, colorized and aligned.

A legend is similar to a textbox except that it contains trace symbols for one or more waves in a graph. Legends are automatically updated when waves are added to or removed from the graph, or when a wave's appearance is modified.

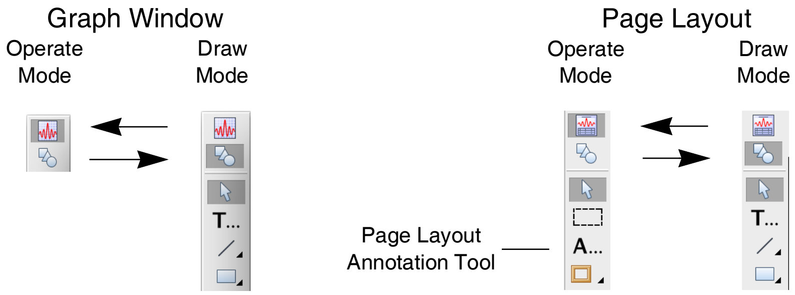
A tag is also similar to a textbox except that it is attached to a point of a trace or image and can contain dynamically updated text describing that point. Tags can be added to graphs, but not to page layouts or Gizmo plots. In contour plots, Igor automatically generates tags to label the contour lines.

A color scale contains a color bar with an axis that spans the range of colors associated with the data. Color scales are automatically updated when the associated data changes. A color scale can also be completely disassociated from any data by directly specifying a named color table and an explicit numeric range for the axis.

Annotations Quick Start

To Do This	Do This
To add an annotation to a graph	Choose Add Annotation from the Graph menu.
To add an annotation to a page layout	Choose Add Annotation from the Layout menu or click with the annotation ("A") tool.
To add an annotation to a Gizmo plot	Choose Add Annotation from the Gizmo menu.
To modify an annotation in a graph or Gizmo plot	Double-click the annotation. This invokes the Modify Annotation dialog.
To modify an annotation in a page layout	Single-click the annotation with the annotation tool. This invokes the Modify Annotation dialog.
To change the annotation type	Use the Annotation pop-up menu in the Modify Annotation dialog, or use the proper Tag, TextBox, ColorScale, or Legend operation.
To move an existing annotation	Click in the annotation and drag it to the new position. If the annotation is frozen, this won't work — double-click it and make it moveable in the Annotation Position tab. To change a tag's attachment point, press Option (*Macintosh*) or Alt (*Windows*) and drag the tag text to the new attachment point on the wave. This works whether or not the tag is frozen.
To duplicate an existing annotation	Double-click the annotation, then click the Duplicate Textbox button in upper-right corner of the Modify Annotation dialog. If the annotation is a tag, the button is titled Duplicate Tag, etc.
To delete an annotation	Double-click the annotation, then click the Delete button in the Modify Annotation dialog. A tag can be deleted by dragging its attachment point off the graph.

When manipulating annotations with the mouse, be sure that the graph or page layout are in the “operate” mode; *not* the “drawing” mode. The tool palette indicates which mode the window is in:



The Annotation Dialog

You can use the annotation dialog to create new annotations or modify existing annotations. The annotation dialog seems complex but comprises only a few major functions:

- A pop-up menu to choose the annotation type.
- A Name setting.
- Tabs that group related Annotations settings.
- A Preview box to show what the annotation will look like or to display commands.
- The normal Igor dialog buttons.

Modifying Annotations

If an annotation is already in a graph or Gizmo plot, you can modify it by double-clicking it while in the “operate” mode. This invokesModify Annotation dialog.

In a page layout, to modify an annotation, you must first select the Annotation (“**A**”) tool in layout mode. Then single-click the annotation to invoke the Modify Annotation dialog.

Annotation Text Content

You enter text into the Annotation text entry area in the Text tab.

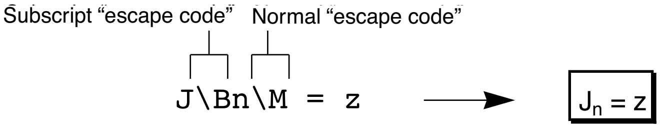
The annotation text may contain both plain text and “escape code” text which produces special effects such as superscript, font, font size and style, alignment, text color and so on. The text can contain multiple lines. At any point when entering plain text, you can choose a special effect from a pop-up menu within the Insert group, and Igor will insert the corresponding escape code. Igor wizards can type them in directly.

As you type annotation text, the Preview box shows what the resulting annotation will look like. You can not enter text in the Preview box.

Annotation Text Escape Codes

An escape code consists of a backslash character followed by one or more characters. It represents the special effect you selected. The effects of the escape code persist until overridden by a following escape code. The escape codes are cryptic but you can see their effects in the Preview box.

In the adjacent example, the subscript escape code “\B” begins a subscript and is not displayed in the annotation; the “n” that follows is plain text displayed as a subscript. The normal escape code “\M” overrides the subscript mode so that the plain text “= z” that follows has the original size and Y position (vertical offset) used for the “J”.



The section **Annotation Escape Codes** on page III-53 provides detailed documentation for the escape codes.

Font Escape Codes

Choosing an item from the Font pop-up menu inserts a code that changes the font for subsequent characters in the annotation. The checked font is the font currently in effect at the current insertion point in the annotation text entry area.

If you don't choose a font, Igor uses the default font or the graph font for annotations in graphs. You can set the default font using the Default Font item in the Misc menu, and the graph font using the Modify Graph item in the Graph menu. The Font pop-up menu also has a "Recall font" item. This item is used in elaborate annotations and is described under **Text Info Variable Escape Codes** on page III-55.

Font Size Escape Codes

Choosing an item from the Font Size pop-up menu inserts a code that changes the font size for subsequent characters in the annotation. The checked font size is the size currently in effect at the current insertion point in the annotation text entry area.

To insert a size not shown, choose any shown size, and edit the escape code to contain the desired font size. Annotation font sizes may be 03 to 99 points; two digits are required after the "`\Z`" escape code.

If you specify no font size escape code for annotations in graphs, Igor chooses a font size appropriate to the size of the graph unless you've specified a graph font size in the Modify Graph dialog. The default font size for annotations in page layouts is 10 points. The Font Size pop-up menu contains a "Recall size" item. This item is used in elaborate annotations and is described under **Text Info Variable Escape Codes** on page III-55.

Relative Font Size Escape Codes

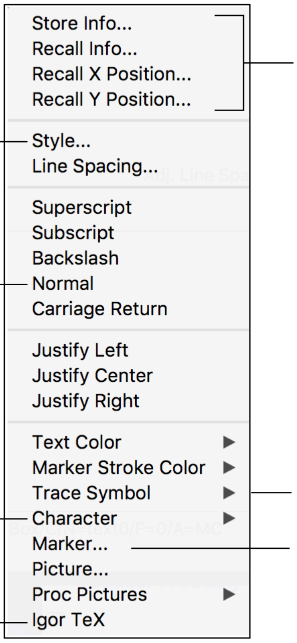
Choosing an item from the Rel. Font Size pop-up menu inserts a code that changes the relative font size for subsequent characters in the annotation. Use values larger than 100 to increase the font size, and values smaller than 100 to decrease the font size.

To insert a size not shown, choose any shown relative size, and edit the escape code to contain the desired relative font size. Annotation relative font sizes may be 001 to 999 (1% to 999%). Three digits are required after the "\Zr" escape code.

Don't use, say, 50% followed by 200% and expect to get exactly the original font size back; rounding inaccuracies will prevent success (because font sizes are handled as only integers). For example, if you start with 15 point text and use `\Zr050` (50%) the result is 7 point text. 200% of 7 points is only 14 point text. Instead, use the Normal "\M" escape code, or an absolute font size or a recalled font size, to return to a known font size.

Special Escape Codes

Choosing an item from the Special pop-up menu inserts an escape code that makes subsequent characters superscript, subscript or normal, affects the style, position or color of subsequent text, or inserts the symbol with which a wave is plotted in a graph.



Text Info Variables.

Bold, Italic, etc.

Return to starting font size and no offset from baseline.

Symbol that a trace is plotted with in a graph.

Insert symbols from a character map.

Choose a marker symbol independent of any traces.

Insert a TeX formula template.

The first four items, Store Info, Recall Info, Recall X Position, and Recall Y Position are used to make elaborate annotations and are described under **Text Info Variable Escape Codes** on page III-55.

The Style item invokes a subdialog that you use to change the style (bold, italic, etc.) for the annotation at the current insertion point in the annotation text entry area. This subdialog has a Recall Style checkbox that is used in elaborate annotations with text info variables.

The Superscript and Subscript items insert an escape code that makes subsequent characters superscript or subscript. Use the Normal item to return the text to the original text size and Y position.

The Backslash item inserts a code to insert a backslash that prints, rather than one which introduces an escape code. Igor does this by inserting two backslashes, which is an escape code that represents a backslash.

The Normal item inserts a code to return to the original font size and baseline. More precisely, Normal sets the font size and baseline to the values stored in text info variable 0 (see **Text Info Variable Escape Codes** on page III-55). The font and style are not affected.

The Justify items insert codes to align the current and following lines.

The Color item inserts a code to color the following text. The initial text color and the annotation background color are set in the Frame Tab.

The Wave Symbol item inserts a code that prints the symbol (line, marker, etc.) used to display the wave trace in the graph. This code is inserted automatically in a legend. You can use this menu item to manually insert a symbol into a tag, textbox, or color scale. For graph annotations, the submenu lists all the trace name instances in the top graph. For layout annotations, all the trace name instances in all graphs in the layout are listed.

The Character item presents a table from which you can select text and special characters to add to the annotation.

The Marker item inserts a code to draw a marker symbol. These symbols are independent of any traces in the graph.

Dynamic Escape Codes for Tags

The Dynamic pop-up menu inserts escape codes that apply only to tags. These codes insert information about the wave or point in the wave to which the tag is attached. This information automatically updates whenever the wave or the attachment point changes.

Dynamic Item	**Effect**
Wave name	Displays the name of the wave to which the tag is attached.
Trace name and instance	Same as wave name but appends an instance number (e.g., #1) if there is more than one trace in the graph associated with a given wave name.
Attach point number	Displays the number of the tag attachment point.
Attach point X value	Displays the X value of the tag attachment point.
Attach point Y value	Displays the Y value of the tag attachment point.
Attach point Z value	Displays the Z value of the tag attachment point. Available only for contour traces, waterfall plots, or image plots.
Attach X offset value	Displays the trace's X offset.
Attach Y offset value	Displays the trace's Y offset.

See also **TagVal and TagWaveRef Functions** on page III-38. These functions provide the same information as the Dynamic pop-up menu items but with greater flexibility.

Other Dynamic Escape Codes

You can enter the dynamic text escape sequence which inserts dynamically evaluated text into any kind of annotation using the escape code sequence:

\{*dynamicText*}

where *dynamicText* may contain numeric and string expressions. This technique is explained under **Dynamic Text Escape Codes** on page III-56.

TagVal and TagWaveRef Functions

If the annotation is a tag, you can use the functions **TagVal** (page V-1022) and **TagWaveRef** (page V-1023) to display information about the data point to which the tag is attached. For example, the following displays the Y value of the tag's data point:

```
\{"%g", TagVal(2)}
```

This is identical in effect to the "\0Y" escape code which you can insert by choosing the "Attach point Y value" item from the Dynamic pop-up menu. The benefit of using the TagVal function is that you can use a formatting technique other than %g. For example:

```
\{"%5.2f",TagVal(2)}
```

TagVal is capable of returning all of the information that you can access via the Dynamic menu escape codes. Use it when you want to control the numeric format of the text.

The TagWaveRef function returns a reference to the wave to which the tag is attached. You can use this reference just as you would use the name of the wave itself. For example, given a graph displaying a wave named wave0, the following tag text displays the average value of the wave:

```
\{"%g",mean(wave0)}
```

This is fine, but if you move the tag to another wave it will still show the average value of wave0. Using TagWaveRef, you can make this show the average value of whichever wave is tagged:

```
\{"%g",mean(TagWaveRef())}
```

The TagVal and TagWaveRef functions work only while Igor is in the process of evaluating the annotation text, so you should use them only in annotation dynamic text or in a function called from annotation dynamic text.

Annotation Tabs

The Text Tab's Annotation text area actually has two functions which are controlled by the pop-up menu at its top-left corner. If you choose Set Tabs from this pop-up menu, Igor shows the tab stops for the annotation.

By default, an annotation has 10 tab stops spaced 1/2 inch apart. You can change the tab stops by dragging them along the ruler. You can remove a tab stop by dragging it down off the ruler. You can add a tab by dragging it from the tab storage area at the left onto the ruler.

Igor supports a maximum of 10 tab stops per annotation and they are always left-aligned tabs. There is only one set of tab stops per annotation and they affect the entire annotation.

General Annotation Properties

Most annotation properties are common to all kinds of annotations.

Annotation Name

You can assign a name to the annotation with the Name item. In the Modify Annotation dialog, this is the Rename item. The name is used to identify the annotation in a Tag, TextBox, ColorScale, or Legend operation. Annotation names must be unique in a given window. See **Programming with Annotations** on page III-52 for more information.

Annotation Frame

In the Frame Tab, the Frame and Border pop-up menus allow you to frame the annotation with a box or shadow box, to underline the textbox, or to have no frame at all. The line size of the frames and the shadow are set by the Thickness and Shadow values.

By default, framed annotations also have a 1-point "halo" that surrounds them to separate them from their surroundings. The halo takes on the color of the annotation's background color. You can change the width of this halo to a value between 0 and 10 points by setting the desired thickness in the Halo box in the Frame tab. A fractional value such as 0.5 is permitted.

Specifying a negative value for Halo allows the halo thickness to be overridden by the global variable V_TBBufZone in the root data folder. If the variable doesn't exist, the absolute value of the entered value is used. The default halo value is -1. You can override the default halo by setting the V_TBBufZone global in a IgorStartOrNewHook hook function. See the example in **User-Defined Hook Functions** on page IV-280.

Annotation Color

The Frame tab contains most of the annotation's color settings.

Use the Foreground Color pop-up menu to set the initial text color. You can change the color of the text from the initial foreground color by inserting a color escape code using the Special pop-up menu in the Text tab.

Use the Background pop-up menu to set the background mode and color:

Background Color Mode	**Effect**
Opaque	The annotation background covers objects behind. You choose the background color from a pop-up menu.
Transparent	Objects behind the annotation show through.

Graph color	The background is opaque and is the same color as the graph background color. This is not available for annotations added to page layout windows.
Window color	The background is opaque and is the same color as the window background color.
Opaque	The annotation background covers objects behind. You choose the background color from a pop-up menu.
Transparent	Objects behind the annotation show through.

Annotation Positioning

You can rotate the annotation into the four principal orientations with the in the Position tab's Rotation pop-up menu. You can also enter an arbitrary rotation angle in integral degrees. Tags attached to contour traces and color scales have specialized rotation settings; see **Modifying Contour Labels** on page II-378 and **ColorScale Size and Orientation** on page III-48.

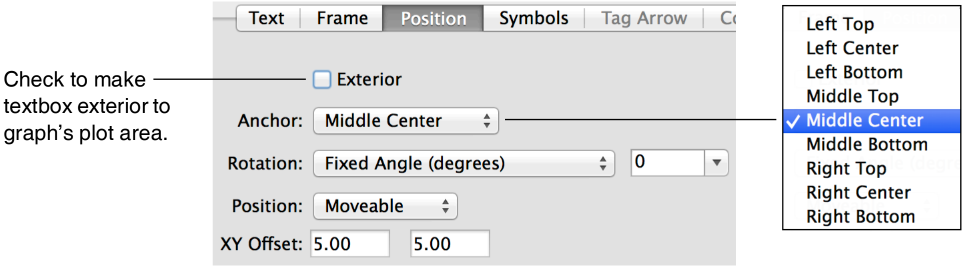
You can position an annotation anywhere in a window by dragging it and in many cases this is all you need to know. However, if you attend to a few extra details you can make the annotation go to the correct position even if you resize the window or print the window at a different size.

This is particularly important when a graph is placed into a page layout window, where the size of the placed graph usually differs from the size of the graph window.

Annotations are positioned using X and Y offsets from “anchor points”. The meaning of these offsets and anchors depends on the type of annotation and whether the window is a graph, layout or Gizmo plot. Tags, for instance, are positioned with offsets expressed as a percentage of the horizontal and vertical sizes of the graph. See **Tag Positioning** on page III-46.

Textbox, Legend, and Color Scale Positioning in a Graph

A textbox, legend, and color scale are positioned identically, so this description will use “textbox” to refer to all of them. A textbox in a graph can be “interior” or “exterior” to the graph’s plot area. You choose this positioning option with the Exterior checkbox:



☐ Exterior

Anchor: Middle Center

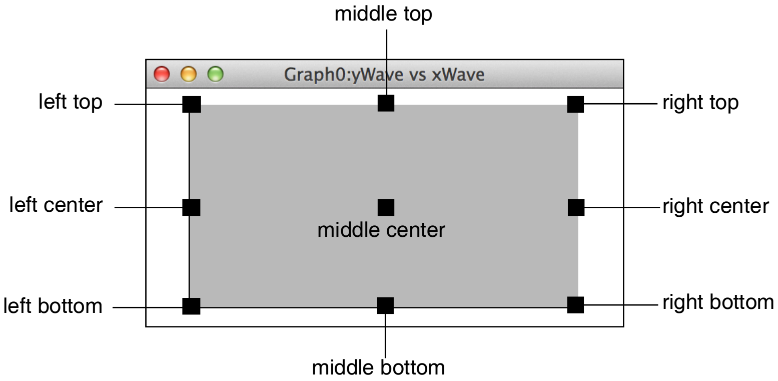
Rotation: Fixed Angle (degrees) 0

Position: Moveable

XY Offset: 5.00 5.00

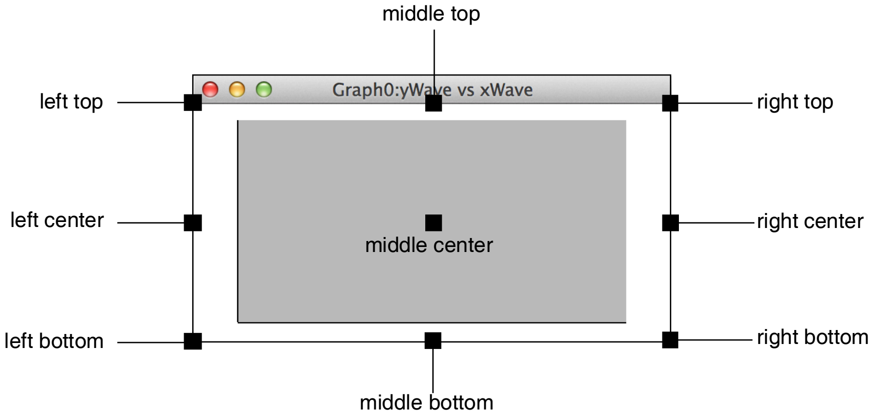
The Anchor pop-up menu specifies the precise location of the reference point on the plot area or graph window edges. It also specifies the location *on the textbox* which Igor considers to be the “position” of the textbox.

An interior textbox is positioned relative to a reference point on the edge of a graph’s plot area. (The plot area is the central rectangle in a graph window where traces are plotted. The standard left, right, bottom, and top axes surround this rectangle.)



Anchor Points for Interior Textboxes

An exterior textbox is positioned relative to a reference point on the edge of the window and the textbox is normally outside the plot area.



Anchor Points for Exterior Textboxes

The purpose of the exterior textbox is to allow you to place a textbox away from the plot area of the graph. For example, you may want it to be above the top axis of a graph or to the right of the right axis. Igor tries to keep exterior textboxes away from the graph by pushing the graph away from the textbox.

The direction in which it pushes the graph is determined by the textbox's anchor. If, for example, the textbox is anchored to the top then Igor pushes the graph down, away from the textbox. If the anchor is middle-center, Igor does not attempt to push the graph away from the textbox. So, an exterior textbox anchored to the middle-center behaves like an interior textbox.

If you specify a margin, using the Modify Graph dialog, this overrides the effect of the exterior textbox, and the exterior textbox will not push the graph.

The XY Offset in the Position Tab gives the horizontal and vertical offset from the anchor to the textbox as a percentage of the horizontal and vertical sizes of the graph's plot area for interior textboxes or the window sizes for exterior textboxes.

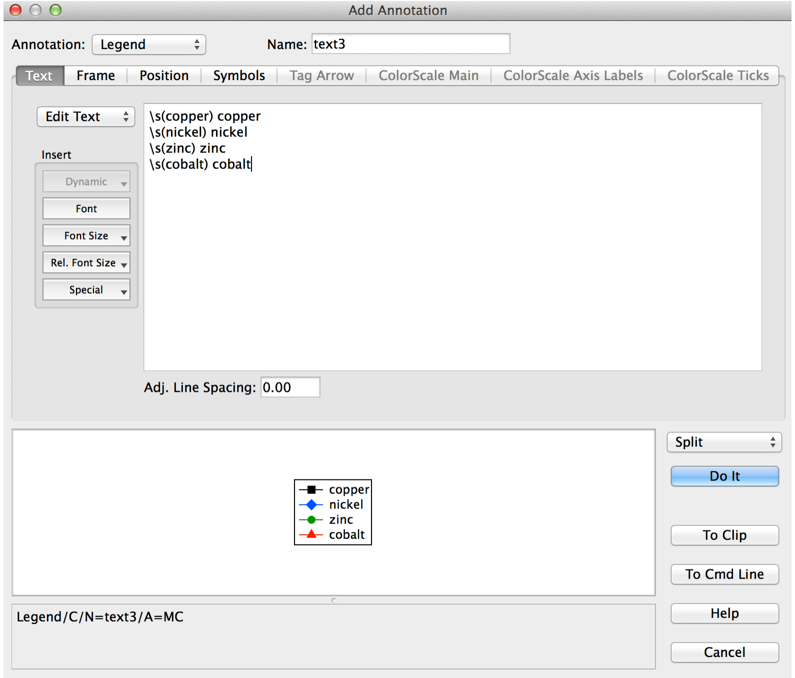
The Position pop-up menu allows you to set the position to moveable or frozen. "Frozen" means that the position of the textbox so that it moved with the mouse. This is useful if you are using the textbox to label an axis tick mark and don't want to accidentally move it.

Textbox and Legend Positioning in a Page Layout

Annotations in a page layout window are positioned relative to an anchor point on the edge of the printable part of the page. The distance from the anchor point to the textbox is determined by the X and Y offsets which are in percent of the printable page. Annotations in a page layout can not be “frozen” as they can in a graph.

Legends

A legend is very similar to a textbox. It shows the symbol for some or all of the traces in a graph or page layout. To make a legend, choose Add Annotation from the Graph or Layout menu.



Add Annotation

Annotation: Legend

Name: text3

```
\s(copper) copper
\s(nickel) nickel
\s(zinc) zinc
\s(cobalt) cobalt
```

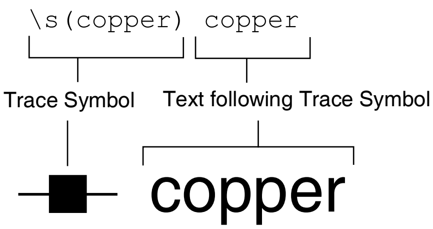
Adj. Line Spacing: 0.00

```
Legend/C/N=text3/A=MC
```

The pop-up menu at the top left of the dialog sets the type of the annotation: TextBox, Tag, Legend or ColorScale. If you choose Legend *when there is no text in the text entry area*, Igor automatically generates the text needed for a “standard legend”. To keep the standard legend, just click Do It. However, you can also modify the legend text as you can for any type of annotation.

Legend Text

The legend text consists of an escape sequence to specify the trace whose symbol you want in the legend plus plain text. In this example dialog above, `\s(copper)` is the escape sequence that inserts the trace symbol (a line and a filled square marker) for the trace whose name is copper. This escape sequence is followed by a space and the name of the wave. The part after the escape sequences is plain text that you can edit as needed.



Instead of specifying the name of the trace for a legend symbol, you can specify the trace number. For example, "`\s(#0)`" displays the legend for trace number 0.

There are only two differences between a legend and a textbox. First, text for a legend is automatically generated when you choose Legend from the pop-up menu while there is no text in the text entry area. Second, if you append or remove a wave from the graph or rename a wave, the legend is automatically updated by adding or removing trace symbols. Neither of these two actions occur for a textbox, tag or color scale.

See **Trace Names** on page II-282 for details on trace names.

Symbol Conditions at a Point

You can create a legend symbol that shows the conditions at a specific point of a trace by appending the point number in brackets to the trace name. For example \s(copper[3]). This feature is useful when a trace uses f(z) mode or when a single point on a trace has been customized.

Freezing the Legend Text

Occasionally you may not want the legend to update automatically when you append waves to the graph. You can freeze the legend text by converting the annotation to a textbox. To create a non-updating legend, invoke the Add Annotation dialog. Choose Legend from the pop-up menu to get Igor to create the legend text, then choose TextBox from the pop-up menu. Now you have converted the legend to a textbox so it will not be automatically updated.

Marker Size

A trace symbol includes a marker if the trace is drawn with one.

Normally the size of the marker drawn in the annotation is based on the font size in effect when the marker is drawn. When you set the font size before the trace symbol escape code, both the marker and following text are adjusted in proportion. You can also change the font size after the trace symbol, which sets the size of the following text without affecting the marker size.

The second method for setting the size of a marker is to choose “Same as on Graph” from the Marker Size pop-up menu in the Symbols Tab. Then the marker size matches the size of the corresponding marker in the graph, regardless of the size of the annotation’s text font.

Trace Symbol Centering

Some trace symbols are vertically centered relative to either the text that precedes or the text that follows the trace symbol escape code, and other symbols are drawn with their bottom at the baseline.

Among the trace styles whose symbols are centered are lines between points, dots, markers, lines and markers, and cityscape. Among the trace styles whose symbols are drawn from the baseline are lines from zero, histogram bars, fill to zero, and sticks and markers.

Trace Symbol Width

The trace symbol width is the width in which all trace symbols in a given legend are drawn. This width is controlled by the font size of the text preceding the trace symbol, or it is set explicitly to a given number of points using the Symbol Width value in the Symbols Tab.

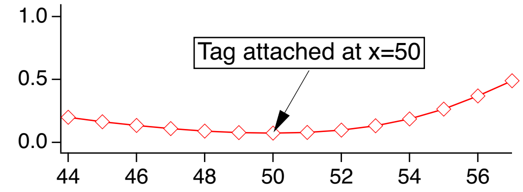
You can widen or narrow the overall symbol size by entering a nonzero width value for Symbol Width. If you use large markers with small text, you may find it necessary to reduce the trace symbol width using this setting. For some line styles that have long dash/gap patterns, you will want to enter an explicit value large enough to show the pattern.

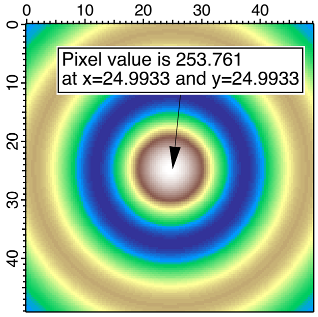
Symbol With Color as f(z)

If you create a graph that uses color as f(z) you may want to create a legend. See **Color as f(z) Legend Example** on page II-301 for a discussion of how to do this.

Tags

A tag is like a textbox but with several added capabilities. A tag is attached to a particular point on a particular trace, image, or waterfall plot in a graph:





Tags can not be added to page layouts or Gizmo plots. However, a graph containing a tag can be added to a page layout.

Igor automatically generates tags to label contour plots.

Tag Text

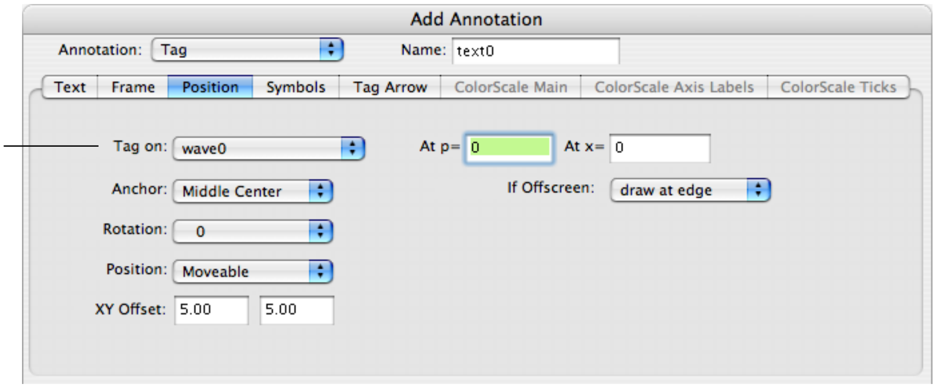
Text in a tag can contain anything a textbox or legend can handle, and more.

The Dynamic pop-up menu of the Text Tab inserts escape codes that apply only to tags. These codes insert information about the wave the tag is attached to, or about the point in the wave to which the tag is attached. This information is “dynamically” updated whenever the wave or the attachment point changes. See **Dynamic Escape Codes for Tags** on page III-38.

The TagVal and TagWaveRef functions are also useful when creating a tag with dynamic text. See **TagVal and TagWaveRef Functions** on page III-38.

Tag Wave and Attachment Point

You specify which wave the tag is attached to in the Position Tab, by choosing a wave from the “Tag on” pop-up menu.



Add Annotation

Annotation: Tag
Name: text0

Tag on: wave0
At p= 0
At x= 0
Anchor: Middle Center
If Offscreen: draw at edge
Rotation: 0
Position: Moveable
XY Offset: 5.00 5.00

Choose which trace or image the tag is attached to

You specify which point the tag is attached to by entering the point number in the “At p=” entry or an X value in the “At x=” entry. The X value is in terms of the X scaling of the wave to which you are attaching the tag. This is not necessarily the same as the X axis position of the point if the wave is displayed in XY mode. It is the X value of the Y wave point to which the tag is attached. If this distinction mystifies you, see **Waveform Model of Data** on page II-62.

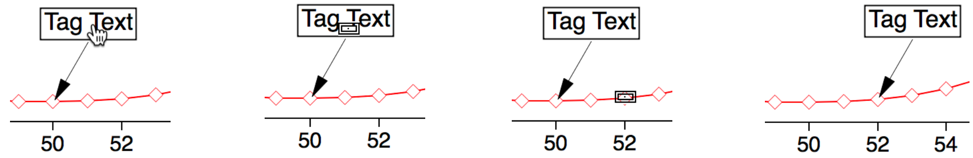
The attachment point of a tag in a (2D) image or waterfall plot is treated a bit differently than for 1D waves. In images it is the sequential point number linearly indexed into the matrix array. The dialog converts this point number, entered as the “At p=” setting, into the X and Y values, and vice versa.

Since it is the point number that determines the actual attachment point, because of rounding, the tag is not necessarily attached exactly at the entered “At x=” and “At y=” values.

As described in the next section, you can position the tag manually by dragging.

Changing a Tag’s Attachment Point

Once a tag is on a graph you can attach it to a different point by pressing Option (*Macintosh*) or Alt (*Windows*), clicking in the tag, and dragging the special tag cursor ▭ to the new attachment point on the trace. You must drag the tag cursor to the point on the trace to which you want to attach the tag, not to the position on the screen where you want the tag text to appear. The dot in the center of the tag cursor shows where the tag will be attached.

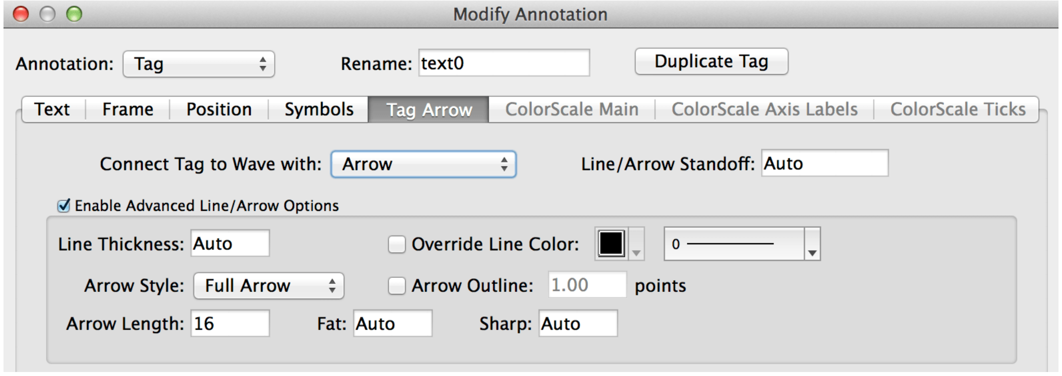


1. Move cursor over tag text
2. Press Option or Alt
3. Drag to new attach point
4. Tag attaches to new point

If you drag the tag cursor off the graph, the tag is deleted from the graph.

Tag Arrows

You can indicate a tag’s attachment point with an arrow or line drawn from the tag’s anchor point using the “Connect Tag to Wave with” pop-up menu in the Tag Arrows tab.



Modify Annotation

Annotation: Tag   Rename: text0   Duplicate Tag

Text | Frame | Position | Symbols | Tag Arrow | ColorScale Main | ColorScale Axis Labels | ColorScale Ticks

Connect Tag to Wave with: Arrow   Line/Arrow Standoff: Auto

☑ Enable Advanced Line/Arrow Options

Line Thickness: Auto   ☐ Override Line Color: ■   0 ————

Arrow Style: Full Arrow   ☐ Arrow Outline: 1.00 points

Arrow Length: 16   Fat: Auto   Sharp: Auto

You can adjust how close the arrow or line comes to the data point by setting the Line/Arrow Standoff distance in units of points.

The Advanced Line/Arrow options give you added control of the line and arrow characteristics.

A Line Thickness value of 0 corresponds to the default line thickness of 0.5 points. Otherwise, enter a value up to 10.0 points. To make the line disappear, select No Line from the “Connect Tag to Wave with” popup menu.

The line color is normally set by the annotation frame color (in the Frame tab). You can override this by checking the Override Line Color checkbox and choosing a color from the popup menu.

Change the attachment line’s style from the default solid line using the line style popup menu.

If “Connect Tag to Wave with” popup is set to Arrow, you can control the appearance of the arrowhead using the remaining controls. Options include full or half arrowhead, filled or outlined arrowhead, arrow length, fatness, and sharpness.

The Arrow Length setting is in units of points with 0 or Auto giving the default length.

The Sharp setting is a small value between -1.0 and 1.0. 0 or Auto gives the default sharpness.

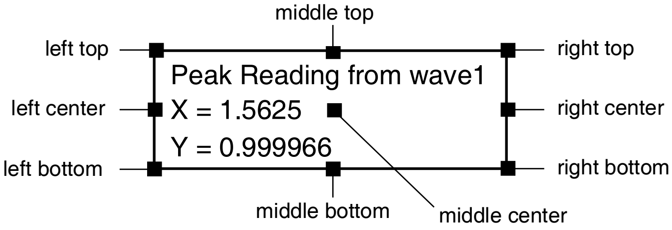
The Fat option specifies the width-to-length ratio. 0 or Auto gives the default ratio of 0.5. Larger numbers result in fatter arrows. If the number is small (say, 0.1), the arrow may seem to disappear unless the arrow length is made longer. Printed arrows can appear narrower than screen-displayed arrows.

Tag Line and Arrow Standoff

You can specify how close to bring the line or arrow to that trace with the Line/Arrow Standoff setting. You can enter an explicit distance in points. If you enter Auto or 0, Igor varies the distance according to the output device resolution and graph size. Use a value of 1 to bring the line as near to the trace as possible. When the wave is graphed with markers, you might prefer to set the standoff to a value larger than the marker size so that the line or arrow does not intersect the marker.

Tag Anchor Point

A tag has an anchor point that is on the tag itself. If there is an arrow or line, it is drawn from the anchor point on the tag to the attachment point on the trace. The anchor setting also determines the precise spot on the tag which represents the position of the tag.



The line is always drawn behind the tag so that if the anchor point is middle center the line doesn't interfere with the text.

Tag Positioning

The position of a tag is determined by the position of the point to which it is attached and by the XY Offset settings in the Position tab. The XY Offset gives the horizontal and vertical distance from the attachment point to the tag's anchor in percentage of the horizontal and vertical sizes of the graph's plot area.

Once a tag is on a graph you can change its XY offset and therefore its position by merely dragging it. You can prevent the tag from being dragged by choosing "frozen" in the Position pop-up menu in the Position Tab. Igor freezes tags when it creates them for contour labels.

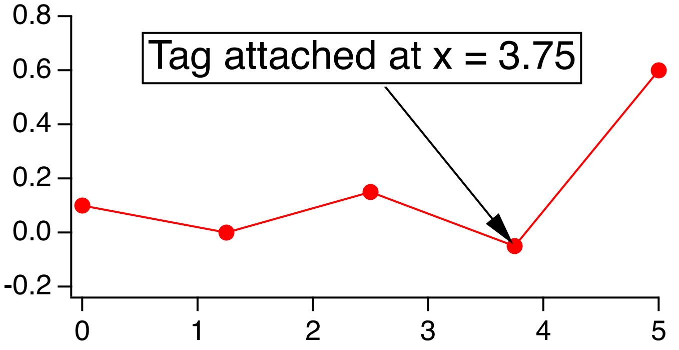
The interior/exterior setting used with textboxes does not apply to tags.

Tags Attached to Offscreen Points

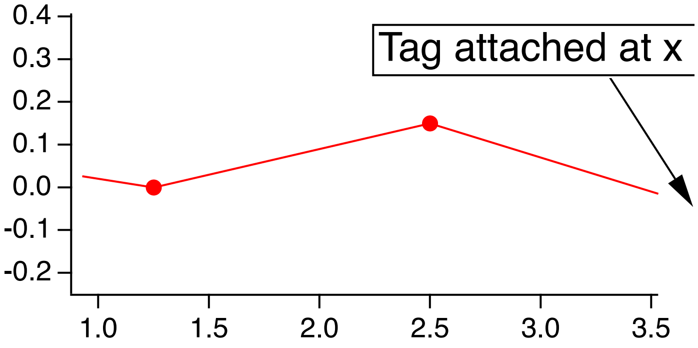
When only a portion of a wave is shown in a graph, it is possible that the attachment point of a tag isn't shown in the graph; it is "off screen" or "out-of-range".

This usually occurs because the graph has been manually expanded or the axes are not autoscaled. Igor draws the attachment line toward the offscreen attachment point.

In this example graph, the attachment point at x=3.75 falls within the range of displayed X axis values:



If we zoom the graph's X range to exclude the x=3.75 attachment point, the tag attachment point is offscreen but the tag is still drawn:



You can suppress the drawing of a tag whose attachment point is offscreen by choosing “hide the tag” from the If Offscreen pop-up menu in the Position Tab.

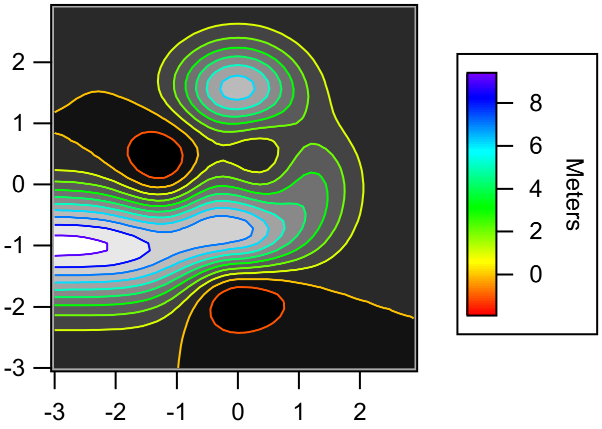
If you want to see or modify a tag that is hidden, autoscale the graph so that it is no longer hidden.

Contour Labels Are Tags

Igor uses specialized tags to create the numerical labels for contour plots. The specialization adds a "tangent" feature to automatically orient the tag along the path of the contour lines. See **Contour Labels** on page II-377 for details.

Color Scales

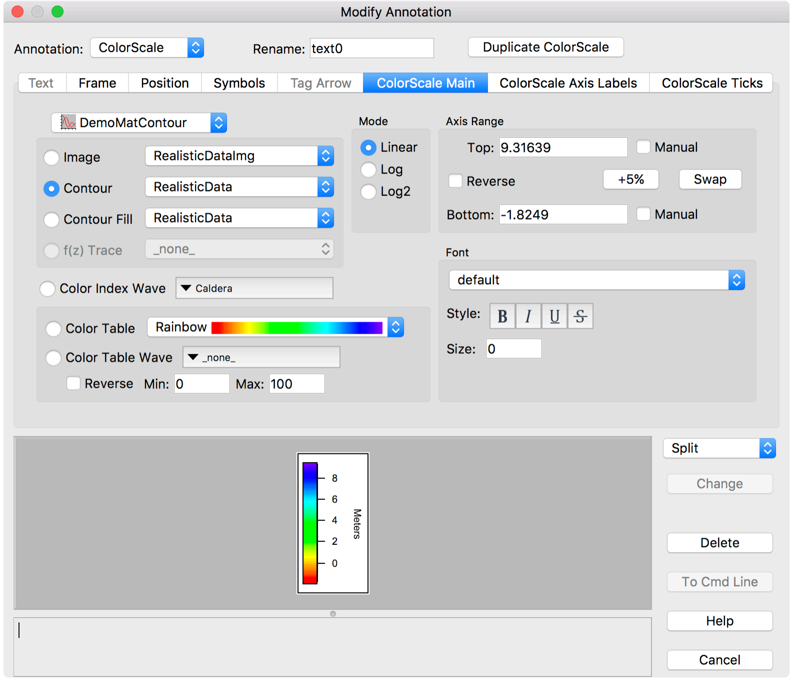
A color scale annotation summarizes the range of data using a color bar and one or more axes.



You create and modify color scales using the **ColorScale** operation which you can invoke via the Add Annotation and Modify Annotation dialogs.

ColorScale Main Tab

A color scale is associated with an f(z) trace, image plot, contour plot in a graph, the first surface object in a Gizmo plot, or with any color index wave or color table. This association can be changed in the ColorScale Main tab.



Modify Annotation

Annotation: ColorScale   Rename: text0

DemoMatContour

○ Image RealisticDataImg

● Contour RealisticData

○ Contour Fill RealisticData

○ f(z) Trace _none_

○ Color Index Wave Caldera

○ Color Table Rainbow

○ Color Table Wave _none_

□ Reverse Min: 0 Max: 100

Mode: ● Linear ○ Log ○ Log2

Axis Range

Top: 9.31639 □ Manual

□ Reverse

Bottom: -1.8249 □ Manual

Font: default

Style: B I U S

Size: 0

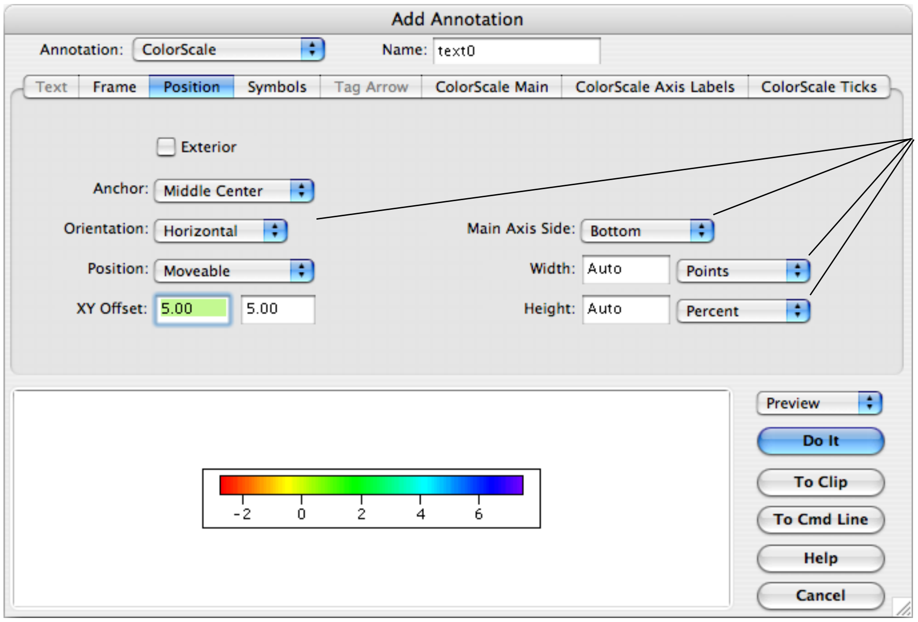
Split

You can infer from the Graph pop-up menu that color scales can be associated with image and contour plots and f(z) traces in a graph other than the graph (or layout) in which the color scale is displayed.

By choosing a graph or Gizmo plot from the pop-up menu, you can create a color scale based on an image, contour plot or f(z) trace in another graph or based on a surface plot in another gizmo window.

ColorScale Size and Orientation

The size and orientation of the color scale is set in the dialog’s Position tab:



Add Annotation

Annotation: ColorScale   Name: text0

□ Exterior

Anchor: Middle Center

Orientation: Horizontal

Position: Moveable

XY Offset: 5.00   5.00

Main Axis Side: Bottom

Width: Auto   Points

Height: Auto   Percent

These settings apply only to ColorScales.

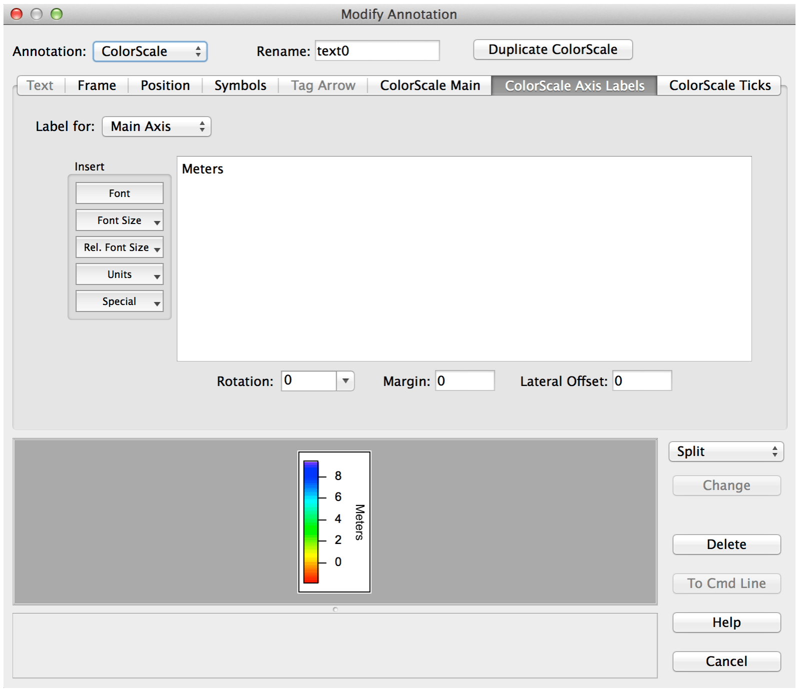
The size of a color scale is indirectly controlled by the size and orientation of the “color bar” inside the annotation, and by the various axis and ticks parameters. The annotation adjusts itself to accommodate the color bar, tick labels, and axis labels.

When set to Auto or 0, the Width and Height settings cause the color scale to auto-size with the window along the color scale’s axis dimension. Horizontal color scales auto-size horizontally but not vertically, and vice versa. The long dimension of the color bar is automatically maintained at 75% of the graph plot area or Gizmo plot dimension. The short dimension is set to 15 points.

You can enter a custom setting for either scale dimension in units of percent or points. Choosing Percent from the menu causes Igor to resize the corresponding dimension in response to graph size changes. Choosing Points fixes the dimension so that it never changes.

ColorScale Axis Labels Tab

You set the axis label for the main axis, and for the second axis if any, in the ColorScale Axis Labels tab:



Annotation: ColorScale

Rename: text0

Label for: Main Axis

Meters

Rotation: 0 Margin: 0 Lateral Offset: 0

The axis label text is limited to one line. This text is the same as is used for text boxes, legends, and tags in the Text tab, but it is truncated to one line when the Annotation pop-up menu is changed to ColorScale.

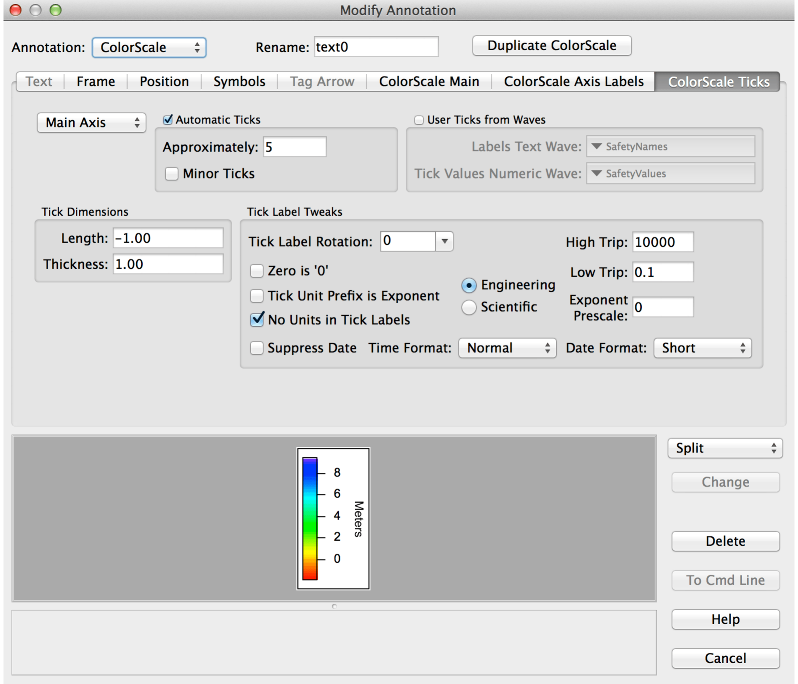
The Units pop-up menu inserts escape codes related to the data units of the item the color scale is associated with. In the case of an image, contour or surface plot, the codes relate to the data units of the image, contour or surface matrix, or of an XYZ contour's Z data wave.

Rotation, Margin, and Lateral Offset adjust the axis label's orientation and position relative to the color axis.

The second axis label is enabled only if the Color Scale Ticks tab has created a second axis through user-supplied tick value and label waves.

ColorScale Ticks Tab

The color scale’s axis ticks settings are similar to those for a graph axis:



Modify Annotation

Annotation: ColorScale   Rename: text0   Duplicate ColorScale

Text | Frame | Position | Symbols | Tag Arrow | ColorScale Main | ColorScale Axis Labels | ColorScale Ticks

Main Axis

☑ Automatic Ticks
Approximately: 5
□ Minor Ticks

□ User Ticks from Waves
Labels Text Wave: SafetyNames
Tick Values Numeric Wave: SafetyValues

Tick Dimensions
Length: -1.00
Thickness: 1.00

Tick Label Tweaks
Tick Label Rotation: 0
□ Zero is '0'
□ Tick Unit Prefix is Exponent
☑ No Units in Tick Labels
● Engineering
○ Scientific
High Trip: 10000
Low Trip: 0.1
Exponent Prescale: 0
□ Suppress Date   Time Format: Normal   Date Format: Short

The main axis tick marks can be automatically computed by checking Automatic Ticks, or you can control tick marks manually by checking User Ticks from Waves. In the latter case, you provide two waves: one numeric wave which specifies the tick mark positions, and one text wave which specifies the corresponding tick mark labels.

You can also specify user-defined tick values and labels to create a second axis. This might be useful, for example, to display a temperature range in Fahrenheit and in Celsius.You do this by choosing Axis 2 from the pop-up menu that normally shows Main Axis. The second axis is drawn on the opposite side of the color bar from the main axis.

The Tick Dimensions settings apply to both axes. A length of -1 means Auto. Otherwise the dimensions are in points. To hide tick marks, set Thickness to 0.

Elaborate Annotations

It is possible to create elaborate annotations with subscripts, superscripts, and math symbols in Igor. Doing so requires entering escape codes. The Add Annotation dialog provides menus for inserting many of the escape codes that you might need.

This is feasible for relatively simple math expressions such as you might use for axis labels or to label graphs. For complex equations, you should use an Igor TeX formula or use a real equation editor and paste a picture representing the equation into an Igor graph or page layout.

Text Info Variables

The **text info variable** is a mechanism that uses escape codes that have a higher degree of “intelligence” than simple changes of font, font size, or style. Using text info variables, you can create quite elaborate annotations if you have the patience to do it. Since you need to know about them only to do fancy things, if you are satisfied with simple annotations, skip the rest of these Text Info Variable sections.

A text info variable saves information about a particular “spot” (text insertion point) in an annotation. Specifically, it saves the font, font size, style (bold, italic, etc.), and horizontal and vertical positions of the spot.

Each annotation has 10 text info variables, numbered 0 through 9. You can embed an escape sequence in an annotation's text to store information about the insertion point in a particular variable. Later, you can embed an escape sequence to recall part or all of that information. In the Label Axis and Add Annotation dialogs, there are items in the Font, Font Size and Special pop-up menus to do this.

See **Text Info Variable Escape Codes** on page III-55 for list of escape sequences.

Text Info Variable Example

To get a feel for this, let's look at a simple example. We want to create a textbox that shows the formula for the chemical compound ethanol: CH_3CH_2OH

To create a textbox showing this formula in 24 point type, we need to enter this, which consists of regular text plus escape codes (shown in red), in the Text tab of the Add Annotations dialog:

```
\Z24\[0CH\B3\MCH\B2\MOH
```

You can enter the escape codes by simply typing them or by making selections from the pop-up menus in the Insert section of the dialog. In this example, the font size escape code, **\Z24**, was generated using the Font Size pop-up menu and the rest of the escape codes were generated using the Special pop-up menu.

Here is what the escape codes mean:

\Z24	Set font size to 24 points.
\[0	Capture the current state as text info variable 0. (Text info variable 0 stores the "normal" state).
\B	Subscript.
\M	Return to normal state (as stored in text info variable 0).
\B	Subscript.
\M	Return to normal state (as stored in text info variable 0).

One way to enter this is to enter the regular text first and then add the escape codes. Here is what the annotation preview would show at each step of this process:

CH3CH2OH	CH3CH2OH
\Z24CH3CH2OH	CH3CH2OH (but in 24 point type)
\Z24**\[0**CH3CH2OH	CH3CH2OH (no visible change)
\Z24\[0CH**\B**3CH2OH	CH$_{3CH2OH}$
\Z24\[0CH\B3**\M**CH2OH	CH$_3$CH2OH
\Z24\[0CH\B3\MCH**\B**2OH	CH$_3$CH$_{2OH}$
\Z24\[0CH\B3\MCH\B2**\M**OH	CH$_3$CH$_2$OH

Programming with Annotations

You can create, modify and delete annotations with the **TextBox**, **Tag**, **Legend**, and **ColorScale** operations. The **AnnotationInfo** function returns information about one existing annotation. The **AnnotationList** returns a list of the names of existing annotations.

Changing Annotation Names

Each annotation has a name which is unique within the window it is in. You supply this name to the TextBox, Tag, Legend, ColorScale, and AnnotationInfo routines to identify the annotation you want to change.

You can rename an annotation by using the /C/N=*oldName*/R=*newName* syntax with the operations. For example:

```
TextBox/C/N=oldTextBoxName/R=newTextBoxName
```

Changing Annotation Types

To change the type of an annotation, apply the corresponding operation to the named annotation. For example, to change a tag or legend into a textbox, use:

```
TextBox/C/N=annotationName
```

Changing Annotation Text

To change the text of an existing annotation, identify the annotation using /N=*annotationName*, and supply the new text. For example, to supply new text for the textbox named text0, use:

```
TextBox/C/N=text0 "This is the new text"
```

To append text to an annotation, use the AppendText operation:

```
AppendText/N=text0 "and this text appears on a new line"
```

You can append text without creating a new line using the /NOCR flag.

Generating Text Programmatically

You can write an Igor procedure to create or update an annotation using text generated from the results of an analysis or calculation. For example, here is a function that creates or updates a textbox in the top graph or layout window. The textbox is named FitResults.

```
Function CreateOrUpdateFitResults(slope, intercept)
   Variable slope, intercept

   String fitText
   sprintf fitText, "Fit results: Slope=%g, Intercept=%g", slope, intercept
   TextBox/C/N=FitResults fitText
End
```

You would call this function, possibly from another function, after executing a CurveFit command that performed a fit to a line, passing coefficients returned by the CurveFit operation.

Deleting Annotations

To programmatically delete an annotation, use:

```
TextBox/K/N=text0
```

Annotation Escape Codes

Annotation escape codes provide formatting control and other features in annotations, including textboxes, tags, legends, and color scales. They can also be used in axis labels, control titles, **SetVariable** values using the styledText keyword, **ListBox** control contents, and with the **DrawUserShape** operation.

Using these escape codes you can control the font, size, style and color of text, create superscripts and subscripts, create dynamically-updated text, insert legend symbols, and apply other effects.

General Escape Codes

These escape codes are can be used in any text that supports annotation escape codes:

\B — Subscript.

\F'*fontName*' — Use the specified font (e.g., \F'Helvetica').

\f*dd* — *dd* is a bitwise parameter with each bit controlling one aspect of the font style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

\JR — Right align text.

\JC — Center align text.

\JL — Left align text.

\K(*r*,*g*,*b*) — Use specified color for text. *r*, *g* and *b* are integers from 0 to 65535.

You can optionally include a fourth alpha parameter which specifies opacity in the range 0 to 65535.

\K also sets the marker fill color for markers added by \W. For setting the marker stroke color, use \k.

\KB(*r*,*g*,*b*) — Use specified color for text background. *r*, *g* and *b* are integers from 0 to 65535.

You can optionally include a fourth alpha parameter which specifies opacity in the range 0 to 65535.

Use \KB0; to turn background color off.

\KB was added in Igor Pro 9.00.

\k(*r*,*g*,*b*) — Use specified color for marker stroke (line color). *r*, *g* and *b* are integers from 0 to 65535. Use before \W*tdd* to change marker stroke color from the default of black (0,0,0).

You can optionally include a fourth alpha parameter which specifies opacity in the range 0 to 65535.

For setting the marker fill color for markers added by \W, use \K.

\L*dtss* — Draws a line from the x position specified in text info variable *d* to the current x position. Uses current text color. Thickness is encoded by digit *t* with values of 4,5,6 and 7 giving 0.25, 0.5, 1.0 and 1.5 pt. Line style is specified by 2 digit number *ss*.

\M — Use normal (main) script (reverts to main line and size).

\$PICT$name=*pictName*$/PICT$

Inserts specified picture. *pictName* can be a ProcPict or the name of a picture as listed in the Misc→Pictures dialog. This is useful for inserting math equations created by another program.

\$WMTEX$ *formula* $/WMTEX$

Inserts a math equation using a subset of LaTeX. See Igor TeX for details.

\S — Superscript.

\sa+*dd* — Adds extra space above line. *dd* is two digits in units of half points (1/144 inch). Can go anywhere in a line.

\sa-*dd* Reduces space above line. *dd* is two digits in units of half points (1/144 inch). Can go anywhere in a line.

\sb+*dd* Adds extra space below line. *dd* is two digits in units of half points (1/144 inch). Can go anywhere in a line.

\sb-*dd* Reduces space below line. *dd* is two digits in units of half points (1/144 inch). Can go anywhere in a line.

\W*tdd* Draws a marker symbol *dd* using current font size and color.

\W*tddd* Draws a marker symbol *ddd* using current font size and color.

The marker outline thickness is specified by the one-digit number *t* with 1, 4, 5, 6, 7 and 8 giving 0.0, 0.25, 0.5, 1.0, 1.25 and 1.5 points. A *t* value of 1, which sets the outline thickness to zero, is useful only for filled markers as it makes unfilled markers disappear.

The marker symbol number is specified by a two-digit number *dd* or a three-digit number *ddd*. New code should use the three-digit marker.

The three-digit syntax was added in Igor Pro 6.30 to support custom markers. This addition causes a two-digit *dd* escape sequence that is directly followed by a digit to be incorrectly interpreted. For example, the sequence:

```
\W718500 m
```

was intended to display marker 18 followed by "500 m". It is now interpreted as a three-digit marker number (185) followed by "00 m". To fix this, change the two-digit marker number to a three-digit marker number by adding a leading zero, like this:

```
\W7018500 m
```

Use \k to set the marker stroke color. Use \K to set the marker fill color.

\x+*dd* Moves the current X position right by 2**dd* percent of the current font max width.

\x-*dd* Moves the current X position left by 2**dd* percent of the current font max width.

\y+*dd* Moves the current Y position up by 2**dd* percent of the current font height.

\y-*dd* Moves the current Y position down by 2**dd* percent of the current font height.

\Z*nn* Use font size *nn*. *nn* must be exactly two digits.

\Zr*nnn* *nnn* is 3 digit percentage by which to change the current font size. *nnn* must be exactly three digits.

Tag Escape Codes

\ON Inserts the name of the wave to which the tag is attached.

\On Inserts the name of the trace and its instance number if greater than 0.

\OP Inserts the point number to which the tag is attached.

\OX Inserts the X value of the point to which the tag is attached.

\OY Inserts the Y value of the point to which the tag is attached.

\OZ Inserts the Z value of the point to which the tag is attached for contour level traces. Inserts NaN for other traces.

Text Info Variable Escape Codes

A text info variable is an internal Igor structure created by Igor. Using the escape codes described in this section, you can manipulate text info variables to create sophisticated annotations. These escape codes are can be used in any text that supports annotation escape codes.

Each annotation is created with 10 text info variables, numbered 0 through 9. Each text info variable is initialized to the default font, font size and style for the graph or page layout hosting the annotation. The X and Y positions are initially undefined.

For background information and an example, see **Text Info Variables** on page III-51.

Here are the supported text info variable escape codes. <digit> means one of 0, 1, 2, ... 9:

\[<digit>	Saves the font, size, style and current X and Y positions in the text info variable.
\]<digit>	Restores all but the X and Y positions from the text info variable.
\X<digit>	Restores the X position from the text info variable.
\Y<digit>	Restores the Y position from the text info variable.
\F]<digit>	Restores the font from the text info variable.
\Z]<digit>	Restores the font size from the text info variable.
\f]<digit>	Restores the style from the text info variable.

Text info variable 0 has a special property. It defines the "main" font size which is restored using the \M escape sequence. \M also sets the Y offset for the baseline to zero. No other text info variable 0 settings (font, style, or offsets) are set by \M.

Dynamic Text Escape Codes

You can enter the dynamic text escape sequence which inserts dynamically evaluated text into any kind of annotation using the escape code sequence:

Dynamic text allows you to create automatically updated annotations. The syntax for inserting dynamic text is:

```
\{dynamicText}
```

dynamicText may contain numeric and string expressions which reference global variables and waves and which invoke user-defined functions. Igor automatically reevaluates *dynamicText* when a global variable or a wave referenced directly in *dynamicText* or indirectly through a function call changes. It also reevaluates *dynamicText* whenever the annotation is updated.

Note: The use of dynamic text creates dependencies that can be fragile and confusing. In most cases, it is better to generate static text programmatically, as described in **Generating Text Programmatically** on page III-53.

Note: You can not reference local variables in *dynamicText* because local variables exist only while a procedure is executing and Igor must reevaluate it at other times.

The numeric and string expressions are evaluated in the context of the root data folder. Use the full data folder path of any non-root objects in the expressions.

dynamicText can take two forms, an easy form for a single numeric expression and a more complex form that provides precise control over the formatting of the result.

The easy form is:

```
\{ numeric-expression }
```

This evaluates the expression and prints with generic ("%g") formatting. For example:

```
TextBox "Two times PI is \\{2*PI}"
```

creates a textbox with this text:

```
Two times PI is 6.28319
```

The use of two backslashes in the TextBox literal string parameter is explained under **Backslashes in Annotation Escape Sequences** on page III-58.

The full form is:

```
\{ formatStr, list-of-numeric-or-string-expressions }
```

formatStr and *list-of-numeric-or-string-expressions* are treated as for the printf operation. For instance, this example has a format string, a numeric expression and a string expression:

```
TextBox "\\{\"Two times PI is %1.2f, and today is %s\", 2*PI, date()}"
```

It produces this result:

```
Two times PI is 6.28, and today is Thu, April 9, 2015
```

You can not use any other annotation escape codes in the format string or numeric or string expressions. They don't work within the \{ ... } context.

Also, the format string and string expressions do not support multiline text. If you need to use multiline text, use the technique described in **Generating Text Programmatically** on page III-53.

As an aid in typing the expressions, Igor considers carriage returns between the braces to be equivalent to spaces. In the Add Annotation dialog, rather than typing:

```
\{"Two times PI is %1.2f, and today is %s",2*PI,date()}
```

you can type:

```
\{
   "Two times PI is %1.2f, and today is %s",
   2*PI,
   date()
}
```

Legend Symbol Escape Codes

You can insert a legend symbol in an annotation.

The syntax for inserting a symbol in a graph is:

\s(*traceName*)

The syntax for inserting a symbol in a page layout is:

\s(*graphName*.*traceName*)

\s is usually used in a legend, in which symbols are created and removed automatically, but can also be used in tags, textboxes, and axis labels where the symbol is updated, but not automatically added or removed. See **Legends** on page III-42.

Axis Label Escape Codes

These escape codes are supported in axis labels and in axis tags:

\c	Inserts the name of the wave that controls the axis. This is the first wave graphed against that axis.
\E	Inserts power of 10 scaling with leading "x". This can be ambiguous and we recommend that you use either \U or \u.
\e	Like \E but inverts the sign of the exponent. This can be ambiguous and we recommend that you use either \U or \u.
\U	Inserts units with automatic prefixes

\u	Inserts power of 10 scaling but without the leading "x" as used by \E. No action if axis is not scaled. Use in front of custom or compound unit strings. Example label: "Field Strength (\\u Volts/Meter)" will produce something like "Field Strength (106 Volts/Meter)".
\u#1	This is a variant of \u that inserts the inverse of \u (e.g., 10-6 instead of 106).
\u#2	Prevents automatic insertion of any units or scaling. Normally, if you set a wave's units and scaling, using the Change Wave Scaling dialog or SetScale operation, and if you do not explicitly specify an axis label, Igor will automatically generate an axis label from the units and scaling. \u#2 provides a way to suppress this behavior when it gets in the way.

Note that escape codes are case sensitive; \u and \U insert different substrings.

Backslashes in Annotation Escape Sequences

An annotation escape code is introduced by a backslash character. Because backslash is itself an escape character in Igor strings, when entering an escape code in a literal string, you need to enter two backslashes to produce one.

For example:

```
TextBox/C/N=text0 "\\Z14Bigger"
```

The Igor command parser converts "\\" to a single backslash. The TextBox operation sees the single backslash and interprets \Z to mean that you want to change the font size.

The next example shows a case where you do not want a double-backslash:

```
TextBox/C/N=text0 "First line\rSecond line"
```

The Igor command parser converts "\r" to a carriage return character. The TextBox operation sees the carriage return character and interprets it to mean that you want to start a new line. There are no annotation escape codes in this example, just a regular escape code, so you must not use a double backslash.

The Add Annotation dialog generates a command to create an annotation. It knows the rules for backslashes. Consequently, if you use the Add Annotation dialog to create an annotation, you can see the correct use of backslashes by observing the command that it generates.

Igor TeX

As of Igor Pro 8, you can use a subset of LaTeX syntax to easily insert math formulas that would otherwise require tedious escape codes or the insertion of a picture generated by another program. LaTeX expressions are delimited by "\\$WMTEX$" and "\\$/WMTEX$" as illustrated here:

```
Display/W=(35,45,339,154)
TextBox/C/N=TeXTest/A=MC "\Z18" + "\\$WMTEX$ \frac{3x}{2} \\$/WMTEX$"
```

which creates

$$\frac{3x}{2}$$

As this example shows, you need double backslashes in the opening and closing escape sequences ("\\$WMTEX$" and "\\$/WMTEX$"). Inside those escape sequences is TeX markup text in which double backslashes are not used ("\frac{3x}{2}").

In the Add Annotation and Modify Annotation dialogs, you can insert the opening and closing escape sequences by choosing the Igor TeX item in the Special pop-up menu. When using the dialogs, do not enter double backslashes as Igor does this for you as necessary.

The example above shows white space after the opening and before the closing escape sequence. Such white space is optional.

You can use local string variables to construct the annotation text. For example:

```
static StrConstant kTeXOpen = "\\$WMTEX$"
static StrConstant kTeXClose = "\\$/WMTEX$"
static StrConstant kTeXFontSize = "\Z18"

Function TeXTest()
   DoWindow/F TexTestGraph
   if (V_Flag == 0)
      Display /N=TexTestGraph /W=(35,45,339,154)
   endif

   String TeXFormula = "\frac{3x}{2}"
   String annotationText = kTeXFontSize + kTeXOpen + TeXFormula + kTeXClose
   TextBox/C/N=TeXTest/A=MC/F=0 annotationText
End
```

For more examples open the Igor TeX Demo experiment by choosing File→Graphing Techniques→Igor TeX Demo.

By default, formulas use an inline style. You can switch to a larger style that is commonly used for formulas on their own line by adding "\displaystyle" at the start of the TeX formula.

Igor Supports a Subset of TeX

Igor does not contain a full LaTeX interpreter. Rather it uses code patterned after Knuth's TeX.web with a subset that supports the most common math syntax that an Igor user is likely to use. Formulas are drawn directly using Igor's normal text and line drawing code — Igor does not first create a picture or .dvi file.

Igor's subset supports LaTeX's \frac but does not support standard TeX's \over and does not support macros. If you discover syntax that Igor does not support but really should, please let us know.

You can use \rm to force letters to be upright (rather than italic.) This is useful in chemical formulas such as ethanol: \rm CH_3CH_2OH

Fonts Used with Igor TeX

For purposes of determining what font is used, each component of a TeX formula is classified as one of these:

- A Greek letter specified by a TeX code like \alpha, \beta, and \gamma
- A math symbol specified by a TeX code like \neg, \prod, \sum, and \int
- Other text (letters, numbers, function names, and anything else other than Greek letters and math symbols as defined above)

By default, Igor uses these fonts for components classified as Greek letters and math symbols:

Macintosh	Hiragino Sans with backups Cambria Math and Symbol
Windows	Symbol with backup Cambria Math

All other text is rendered using the font in effect in the annotation before the TeX formula.

The default Greek and math symbol fonts were chosen based on appearance and support for special characters such as square bracket extensions (used when building a tall square bracket).

You can override these defaults within a given annotation by storing a font name in text info variable 8 for Greek characters and text info variable 9 for math symbols. See **Text Info Variables** on page III-51. You can experiment with different fonts using the Igor TeX Demo experiment.

Although you can use a Greek character such as α (alpha) directly, in order to use the specific font chosen for Greek character display, you should still use the \alpha syntax of TeX.

The fonts used in a formula affect not only the visual appearance of characters but also the font embedding situation when exporting to the graphics formats Igor PDF and EPSF. See **Font Embedding** on page III-99 (*Macintosh*) and **Font Embedding** on page III-105 (*Windows*). When exporting Igor PDF and EPSF with font embedding for standard fonts turned off, you should examine the output carefuly and use full font embedding if problems are found.

Igor TeX Demo

To learn about LaTeX syntax and capabilites, open the Igor TeX Demo experiment by choosing File→Graphing Techniques→Igor TeX Demo.

Igor TeX References

These web sites are useful for generating TeX formulas:

https://latex.codecogs.com/eqneditor/editor.php

http://www.sciweavers.org/free-online-latex-equation-editor

https://oeis.org/wiki/List_of_LaTeX_mathematical_symbols

Chapter

III-3

Drawing

Overview

Igor's drawing tools are useful in page layout and graph windows for highlighting material with boxes, circles and arrows and can also be used to create simple diagrams. These drawing tools are object-oriented and optimized for the creation of publication quality graphics.

You can graphically edit a wave's data values when it is displayed in a graph; data points can be deleted, added or modified. You can also create new waves by simply drawing them.

In control panel windows, you can use the drawing tools to create a fancy background for controls.

Like all other aspects of Igor, drawing tools are fully programmable. This allows programmers to create packages of code that add new graph types to Igor's repertoire. Although only programmers can create such packages, everyone can make use of them.

The drawing tools are available only in page layout, graph and control panel windows.

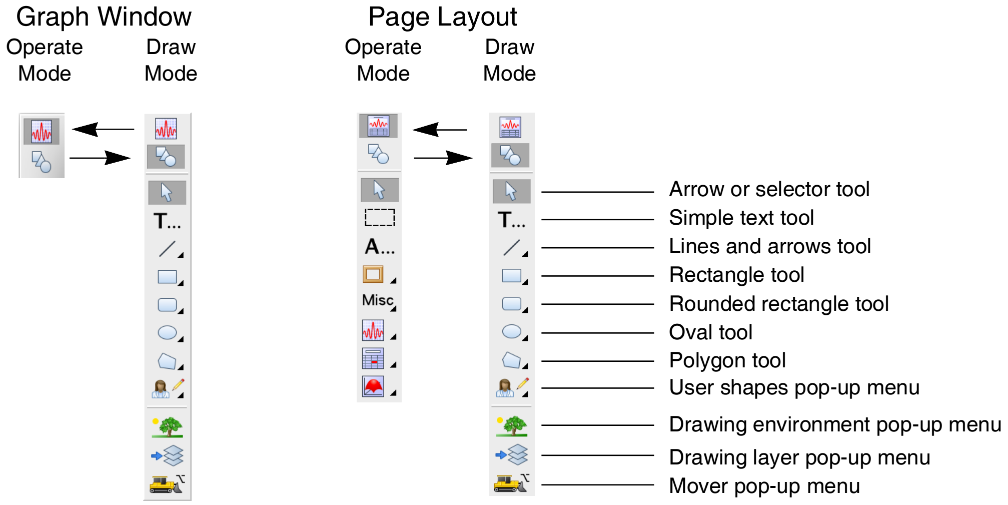
The Tool Palette

To use the drawing tools you first need to invoke the tool palette. In graphs and control panels this is done by the Show Tools menu command in the Graph or Panel menu. This command adds a tool palette along the left edge of the window. The tool palette is always available in page layout windows.

With the exception of Undo, Cut, Copy, Clear, Paste, Select All and Duplicate in the Edit menu, all drawing commands are located in the tool palette.

Once displayed, the tool palette works in two modes:

- **Operate mode:** Click the top (operate) button to enter Operate mode. In this mode, the drawing tool palette is not available, and you interact with the window as normal.
- **Drawing mode:** Click the second (drawing) button to enter Drawing mode. The entire tool palette will then be displayed, and you can add text, arrows, boxes, and other shapes to the window.



The three bottom icons in the tool palette — the Drawing Environment icon (tree and grass), Drawing Layer icon, and the Mover icon (bulldozer) — are pop-up menus. If you press Option (*Macintosh*) or Alt (*Windows*) while clicking the Mover icon, you get a pop-up menu that allows you to retrieve offscreen drawing elements, if any.

The icons above, with the triangle symbol, when clicked, select a mode of operation for creating various shapes using the mouse. If you click them and hold, they present pop-up menus that allow you to invoke dialogs for creating the shapes.

Arrow Tool

Use the arrow tool to select, move and resize one or more drawing objects or user-defined controls.

Selecting, Moving, and Resizing Objects

Click a drawing object once to select it. When selected, small squares called **handles** appear, defining the object's size. You can drag these handles to resize single or multiple objects. By pressing Shift while clicking a handle, an object resize can be constrained to the horizontal, vertical, or diagonal directions depending on how close the cursor is to these directions. By pressing Option (*Macintosh*) or Alt (*Windows*) instead, the resize is constrained to entirely horizontal or entirely vertical, depending on the position of the cursor.

If you click an object and it does not become selected then the object may be in a different drawing layer or it may not in fact be a drawing object at all.

You can select multiple drawing objects by shift clicking additional objects. You can remove an object from a selection by shift clicking a selected object.

You can also select multiple drawing objects by dragging a selection rectangle around a set of drawing objects. Unless you first press Option (*Macintosh*) or Alt (*Windows*), only objects that are completely enclosed by the selection rectangle are included in the selection. With Option (*Macintosh*) or Alt (*Windows*) pressed, objects that are merely touched by the selection rectangle are included. You can append additional objects to a selection by pressing Shift before dragging out a selection rectangle.

You can select all objects in the current layer with the Select All item in the Edit menu.

Rotating Objects

You can rotate draw objects by clicking just beyond a selection's visible resizing handles.

When the mouse is over an invisible rotation handle, the cursor changes shape, indicating that you can rotate the object. Click and drag the invisible rotation handle to rotate it.

Duplicating Objects

You can duplicate selected drawing objects using the Duplicate command found in the Edit menu. If, right after duplicating an object, you reposition the duplicated object and duplicate again then the new object will be offset from the last by the amount you just defined.

Deleting Objects

You can delete selected drawing objects by pressing Delete.

Modifying Objects

Double-clicking a drawing object displays a dialog for that object. You can then change any of the settings available in the dialog. Click the Do it button to apply the changes.

Simple Text Tool

The text tool is usually used to create a single line of simple text with the same font, size, style and rotation. You can also create formatted text using multiple lines and all the escape codes supported by annotations - see **Annotations** on page III-33.

The text tool is the only way to add text to control panel windows.

To create a line of text, click the simple text icon. The cursor will change shape to a text entry vertical line. Then move the cursor to the location in the window where you want the text origin, and click there to invoke the Create Text dialog.

Casual users can ignore the Anchor pop-up menu. It is used mainly by programmers to ensure that text is aligned properly about the numeric origin (X0,Y0). Any rotation is applied first and then the rotated text is aligned relative to the origin as specified by the anchor setting.

To edit or change a text object, double-click the object with the Arrow tool.

Lines and Arrows Tool

You can use the Lines tool to draw lines by clicking at the desired starting point and then dragging to the desired ending point. Press Shift while drawing to constrain the line to be vertical or horizontal.

If you click and hold on the Lines icon, you get a pop-up menu that takes you to a dialog where you can specify the line numerically. You see a similar dialog if you use the arrow tool to double-click a line. It allow you to change the properties of the line.

The dash pattern pop-up palette is the same as the one used for graph traces. You can adjust the dash patterns by use of the Dashed Lines command in the Misc menu.

The arrow fatness parameter is the desired ratio of the width of the arrow head to its length.

The line thickness and arrow length parameters are specified in terms of points and may be fractional.

Line start and end coordinates depend on the chosen coordinate system. See **Drawing Coordinate Systems** on page III-66 for detailed discussion. Programmers should note that the ends of a line are *centered* on the start and end coordinates. This is more obvious when the line is very thick.

Rectangle, Rounded Rectangle, Oval

The Rectangle, Rounded Rectangle, and Oval tools create objects that are defined by an enclosing rectangle. Click and hold on the appropriate icon to invoke a dialog where you can specify the object numerically. You see a similar dialog if you use the arrow tool to double-click the object.

Press Shift while initially dragging out the object to create a square or circle. If you press Shift while resizing the object with the arrow tool, you constrain the object in the horizontal, vertical, or diagonal directions depending on how close the cursor is to one of these directions. Thus, when you Shift-drag along a diagonal the sides are constrained to equal length, but if you Shift-drag along a horizontal or vertical direction, the object is resized along only one of these directions. If instead you hold down Option (*Macintosh*) or Alt (*Windows*), dragging along a diagonal resizes the object proportionally.

The Erase fill mode functions by filling the area with the current background color of the window. The fill background color is used only when a fill pattern other than Solid or Erase is chosen.

An object is always drawn inside the mathematical rectangle defined by its coordinates no matter how thick the lines. This differs from straight lines which are centered on their coordinates.

To adjust the corners of a rounded rectangle, double-click the object and edit the RRect Radius setting in the resulting dialog. Units are in points.

Arcs and Circles

To draw an arc or a circle by center and radius, click and hold on the Oval icon and choose Draw Arc from the resulting pop-up menu. Click and drag to define the center and the radius or start angle. Click again without moving the mouse to create a circle or move and click to define the stop angle for an arc. A variety of click and click-drag methods are supported and experimentation is encouraged.

To edit an arc, click and hold on the Oval icon in the draw tool panel and then choose Edit Arc from the resulting pop-up menu. If necessary, click on an arc to select it. You can then drag the origin, radius, start angle, and stop angle.

To change the appearance of an arc, double click to get to the Modify Arc dialog. Unlike Ovals, Arcs honor the current dash pattern and arrowhead setting in the same way as polygons and Beziers. The center of an arc or circle can be in any coordinate system (see **Drawing Coordinate Systems** on page III-66) but the radius is always in points.

Polygon Tool

The polygon tool creates or edits drawing objects called polygons and, in graphs, it can create or edit waves. For details, see **Drawing Polygons and Bezier Curves** on page III-69.

The User Shapes Tool

This tool creates drawing objects defined by procedure code written by you or another Igor user. For details see the **DrawUserShape** operation and for examples choose File→Example Experiments→Programming→User Draw Shapes.

Drawing Environment Pop-Up Menu

TheDrawing Environment icon allows you to change properties such as line thickness, color, fill pattern, and other visual attributes.

You can change the attributes of existing objects, or you can change the default attributes of objects you are yet to create.

To change the attributes of existing objects, first select them. Then use the Drawing Environment pop-up menu to modify the attributes.

To change the default attributes of objects yet to be created, make sure no objects are selected. Then use the Drawing Environment pop-up menu to change attributes. From that point on, all new objects will have the new attributes, until you change them again.

The items in the menu do not affect all types of objects. The Fill Mode and Fill Color commands affect only enclosed shapes. The Line Dash and Line Arrow commands do not affect rectangles and ovals.

You can invoke the Modify Draw Environment dialog to change multiple attributes by choosing All from the Drawing Environment pop-up menu or by double-clicking an object.

Double-clicking multiple selected objects or groups of object with the selector tool also invokes the Modify Draw Environment dialog. In this case, the properties shown are those of the *first* selected object but if you change a property then all selected objects are affected.

Double-clicking a single drawing object with the selector tool invokes a specific dialog for objects of that type.

Drawing Layer Pop-up Menu

The Drawing Layer pop-up menu selects the active drawing layer. You can create and edit drawing objects in the active drawing layer only. See **Drawing Layers** on page III-68 for details.

Mover Pop-Up Menu

The Mover pop-up menu performs various actions:

- Changing the front-to-back relationship of drawing objects in a given layer
- Aligning drawing objects or controls to each other
- Distributing the space between drawing objects or controls
- Grouping and ungrouping drawing objects
- Retrieving drawing objects or controls that are off screen

Use the Bring to Front, Send to Back, Forward and Backward commands to adjust the drawing order within the current drawing layer.

The Align command adjusts the positions of all the selected drawing objects relative to the first selected object. This works on controls as well as drawing objects.

The Distribute command evens up the horizontal or vertical spacing between selected objects. The original order is maintained. This operation is especially handy when working with buttons or other controls in a user-defined panel. This works on controls as well as drawing objects.

The Retrieve command is used to bring offscreen objects back into the viewable area. You can retrieve an offscreen object by selecting it from the Retrieve submenu of the Mover pop-up menu. Alternatively, if you press Option (*Macintosh*) or Alt (*Windows*) and select an object from the resulting pop-up menu, Igor selects the object and displays the Properties dialog. Use this to set the numeric coordinates for an object to bring it back onscreen. Or you can cancel out of the dialog and then press Delete to remove the object. Retrieve works on controls as well as drawing objects.

The Grid submenu provides options for controlling the grid. See **Drawing Grid** on page III-66 for details.

Drawing Grid

You can display a grid and force objects to snap to the grid, it is visible or not. You do this using the Mover pop-up menu Grid submenu.

The default grid is in inches with 8 subdivisions. The grid origin is the top-left corner of the window or subwindow. Use the **ToolsGrid** to set grid properties. You can independently specify the X and Y grids and set the origin, major grid spacing, and number of subdivisions.

When grid snap is on, you can turn it off temporarily by engaging Caps Lock.

When dragging an object, the corner nearest to where you clicked to start dragging the object is the corner that will be snapped to the grid. You can also snap existing objects to the grid by selecting the Align to Grid from the Mover popup menu.

Set Grid from Selection

If a single object is selected, Set Grid from Selection will set the grid origin at the top left corner of the object. It two objects are selected, the origin will be set to the top left corner of the first object and the major grid spacing will be defined by the distance to the top left corner of the second object. If either the horizontal or vertical separation is small then a uniform (equal X and Y) grid is defined by the larger distance. Otherwise the horizontal and vertical grids are set from the corresponding distances.

Grid Style Function

The Style Function submenu allows you use to create a style function or to run one that you previously created. Style functions are created in the main procedure window with names like *MyGridStyle00*. You can edit these to provide more meaningful names.

Drawing Coordinate Systems

A unique feature of Igor’s drawing tools is the ability to choose different coordinate systems. You can choose different systems on an object-by-object basis and for X and Y independently. This capability is mainly for use in graphs to allow your drawings to adjust to changes in window size or to changes in axis scaling.

You specify the coordinate system using pop-up menus found in the drawing Modify dialogs. The available coordinate systems are:

- Absolute
- Relative
- Plot Relative
- Axis Relative
- Axis

Absolute

In absolute mode, coordinates are measured in points, or **Control Panel Units** for control panels, relative to the top-left corner of the window. Positive x is toward the right and positive y is toward the bottom. In this mode the position and size of objects are unaffected by changes in window size. This is the default and recommended mode in page layouts and control panels.

If you shrink a window, it is possible that some objects will be left behind and may find themselves outside of the window (offscreen). In addition, if you copy an object with absolute coordinates from one window and then paste it in another smaller window it will be placed where the coordinates specify, even if it is offscreen. If you think this has happened, use the Mover pop-up menu to retrieve any offscreen objects or expand the window until the stray objects are visible.

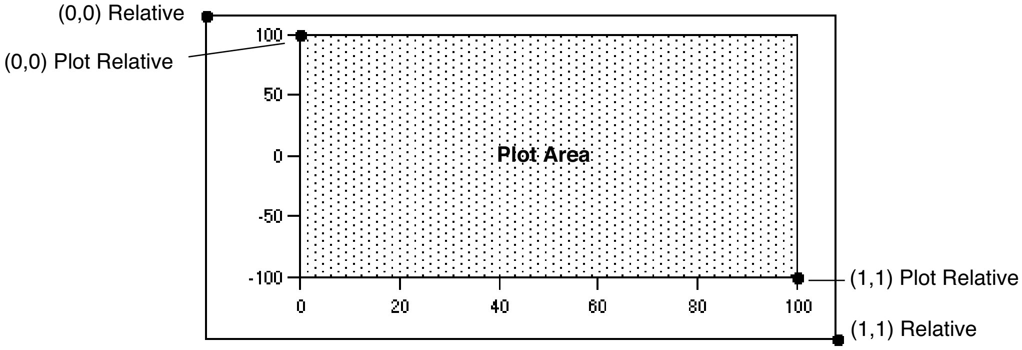
Relative

In this mode, coordinates are measured as fractions of the size of the window. Coordinate values x=0, y=0 represent the top-left corner while x=1,y=1 corresponds to the bottom-right corner. Use this mode if you want your drawing object to remain in the same relative position as you change the window size.

This mode will produce near but not exact WYSIWYG results in graphs. This is because the margins of a graph depend on many factors and only loosely on the window size. This mode gives good results for objects that don't have to be positioned precisely, such as an arrow pointing from near a trace to near an axis. It would not be suitable if you want the arrow to be positioned precisely at a particular data point or at a particular spot on an axis. For that you would use one of the next three coordinate systems.

Plot Relative (Graphs Only)

This system is just like Relative except it is based on the plot rectangle rather than the window rectangle. The plot rectangle is the rectangle enclosed by the default left and bottom axes and their corresponding mirror axes. The coordinates x=0, y=0 represent the top-left corner while x=1,y=1 corresponds to the bottom-right corner. This is the default and recommended mode for graphs.



The Plot Relative system is ideal for objects that should maintain their size and location relative to the axes. A good example is cut marks as used with split axes. In most cases, Plot Relative or Axis Relative is a better choice than the more complex **Axis-Based (Graphs Only)** system discussed below.

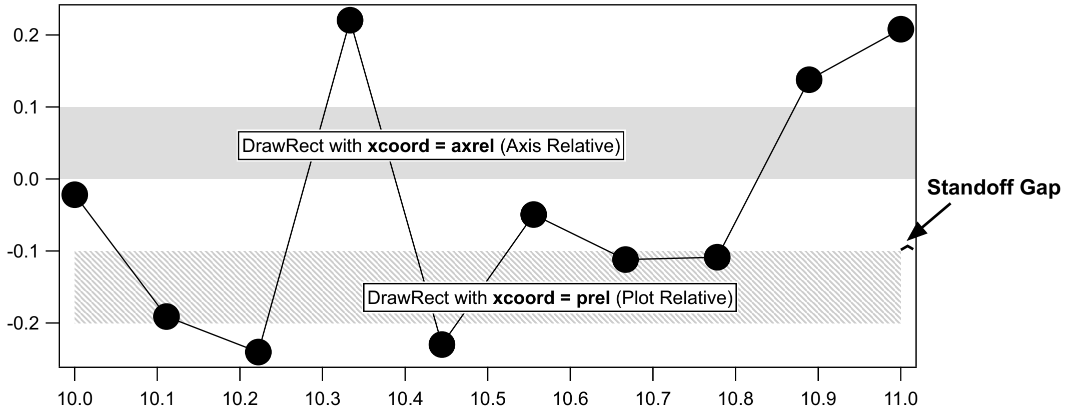
Axis Relative (Graphs Only)

This system is just like Plot Relative except it is based on the plot rectangle expanded by any axis standoff. If there is no axis standoff, the result is the same as using Plot Relative. Axis standoff is described under **Axis Tab** on page II-307.

Axis Relative coordinates require Igor Pro 9.00 or later.

The Axis Relative system is ideal for objects that should maintain their size and location relative to the axes including the standoff areas. A good example is a background highlighting color rectangle that starts or ends at the expanded rectangle's edges, avoiding a standoff gap.

The following graphic shows two rectangles, each with an X range of 0 to 1 but using axis relative mode for the top rectangle and plot relative mode for the bottom rectangle:



Both rectangles use axis-based coordinates, described next, for their Y coordinates.

Axis-Based (Graphs Only)

The pop-up menu for the X coordinate system includes a list of the horizontal axes and the pop-up menu for the Y coordinate includes a list of the vertical axes. When you choose an axis coordinate system, the position on the screen is calculated just as it is for wave data plotted against that axis, with the exception that drawing object coordinates are not limited to the plot area. This mode is ideal when you want an object to stick to a feature in a wave even if you zoom in and out.

Axes are treated as if they extend to infinity in both directions. For this reason along with the fact that axis ranges can be very dynamic, it is very easy to end up with objects that are offscreen. You can use the Mover pop-up menu Retrieve submenu to retrieve objects or, if you press Option (*Macintosh*) or Alt (*Windows*) before clicking the Mover icon, you can edit the numerical coordinates of each offscreen object. You can also end up with objects that are huge or tiny. It is best to have the graph in near final form before using axis-based drawing objects.

Axis-based coordinates are of particular interest to programmers but are also handy for a number of interactive tasks. For example you can easily create a rectangle that shades an exact area of a plot. If you use axis coordinate systems then the rectangle remains correct as the graph is resized and as the axis ranges are changed. You can also create precisely positioned drop lines and scale (calibrator) bars.

Drawing Layers

Layers allow you to control the front-to-back layering of drawing objects relative to other window components. For example, if you want to demarcate a region of interest in a graph, you can draw a shaded rectangle into a layer behind the graph traces. If you drew the same rectangle into a layer above the traces then the traces would be covered up.

Each window type supports a number of separate drawing layers. For example, in graphs, Igor provides three pairs of drawing layers. You can see the layer structure for the current window and change to a different layer by clicking the layer icon. The current layer is indicated by a check mark.

Drawing layers have names. This table shows the names of the layers and which window types support which layers:

Graphs	**Page Layouts**	**Control Panels**
ProgBack	ProgBack	ProgBack
UserBack	UserBack	UserBack
ProgAxes		

UserAxes		
ProgFront	ProgFront	
UserFront	UserFront	
Overlay	Overlay	Overlay

Drawing layers come in pairs named ProgSomething and UserSomething. User layers are provided for interactive drawing while Prog layers are provided for Igor programmers. This usage is just a recommended convention and is not enforced. The purpose of the recommendation is to give Igor procedures free access to the Prog layers. If you were to draw into a Prog layer and then ran a procedure that used that layer then your drawing could be damaged or erased.

The top layer is the Overlay layer. It is provided for programmers who wish to add user-interface drawing elements without disturbing graphic elements. It is not included when printing or exporting graphics. This layer was added in Igor Pro 7.00.

Only drawing objects in the current layer can be selected. If you find you can not select a drawing object then it must be in a different drawing layer. You will have to try the other layers until you find the right one.

To move an object between layers you have to cut, switch layers and then paste.

Drawing Polygons and Bezier Curves

The polygon tool creates or edits drawing objects called polygons and, in graphs, it can create or edit waves. For details, see **Drawing Polygons and Bezier Curves** on page III-69.

A polygon is an open or closed shape with one or more line segments, or edges. Polygons can be filled with a color and pattern and can have arrow heads attached to the start or end.

Although you may create a closed polygon by making the beginning and ending points the same, Igor does not recognize it as a closed shape. You can thus open the Polygon by moving either the beginning or ending points. This is subject to change in a future release.

Creating a New Polygon

You can create a polygon in one of two ways:

- **Segment Mode:** Each click defines a new vertex.
- **Freehand Mode:** Igor adds new vertices as you sweep out a smooth curve.

To create a polygon using segment mode, click the polygon icon once. Then click at the desired location for the beginning of the polygon. As you move the cursor, you create a line segment. A second click anchors the first line segment, and begins the second. You can keep drawing line segments until the polygon is finished.

Pressing the Shift key while dragging constrains movement to angles that are increments of 15 degrees from horizontal or vertical.

Stop drawing by double-clicking to define the last vertex or by clicking at the first vertex. You then automatically enter edit mode, the cursor changes to ⌖ , and vertices are marked with square handles. (For Bezier curves, vertices are called “anchors”.) In edit mode, you can reshape the polygon. To exit edit mode click the arrow tool.

To create a polygon using freehand mode, to edit an existing polygon, or to draw or edit a Bezier curve, click and hold on the polygon icon until the pop-up menu appears. Then choose one of these items:

- **Draw Poly:** Enters the create-segmented-polygon mode in which a click starts a new segmented polygon. This is identical to a single click on the icon.

- **Freehand Poly:** Enters create-freehand-polygon mode in which a click starts a freehand polygon. Click and drag to sweep out a smooth curve as long as you press the mouse button. When you release the mouse button, you automatically enter edit mode, where you can change the shape.
- **Edit Poly:** Enters edit-polygon mode for editing an existing polygon as described in the next section.
- **Draw Bezier:** Enters create-Bezier-polygon mode in which a click starts a Bezier polygon. Click and drag to define anchor and control points. Click on the first point to close the curve.
- **Edit Bezier:** Enters edit-Bezier-polygon mode for editing an existing Bezier polygon as described in the next section. You may need to click on a Bezier curve to select it.

Editing a Polygon

To enter edit mode, click and hold on the polygon icon, and choose Edit Poly from the pop-up menu. Then click the polygon object you want to edit.

While in edit mode you can move, add, and delete vertices, and move line segments:

- **Move a vertex:** Click and drag the vertex to move it and stretch the associated edges.
- **Create a new vertex:** Click between vertices in a line segment.
- **Delete vertices:** Press Option (*Macintosh*) or Alt (*Windows*) and click the vertex you want to delete.
- **Offset pairs of vertices:** Press Command (*Macintosh*) or Ctrl (*Windows*), click a line segment, and drag.

Segmented Polygons

It is possible to separate a polygon into segments by adding coordinate pairs that are NaN. By default, such segments are drawn as if they are separate polygons. If you fill such a polygon with a color having transparency, any overlapping areas will darken because they are painted twice.

If you use SetDrawEnv subpaths=1, the segments are sub paths within a single polygon. The segments are drawn with no line linking the subpaths, but when filled the entire polygon are treated as a single polygon, making it possible to create a polygon with internal holes. The way those holes are filled is affected by the SetDrawEnv fillRule keyword.

The subpaths keyword applies only to polygons created with **DrawPoly** and **DrawBezier**, not to those created manually. You can manually create subpaths by entering Edit Polygon mode, right-clicking a line segment of the polygon, and choosing Break Line.

A side-effect of setting SetDrawEnv subpaths=1 is a change to the way arrows are added to segmented polygons: with SetDrawEnv subpaths=0 (the default), arrows are added to each segment as if they are separate polygons. With subpaths=1, the arrows are added only to the first or last points in the entire polygon.

Editing a Bezier Curve

There are a number of operations you can perform to edit a Bezier curve:

- **Move an anchor point:** Click and drag the anchor point to move it and stretch the associated curves.
- **Move a control point:** Click and drag it. If an anchor point has two control points, this move both.
- **Move only one of two control points associated with an anchor point:** Click the control point, then press Option (*Macintosh*) or Alt (*Windows*), and drag the control point.
- **Move a control point that is directly above an anchor point**: Press Command (*Macintosh*) or Ctrl (*Windows*), click the control point (and anchor), then drag the control point.
- **Create a new anchor point:** Click the curve between anchor points.
- **Delete an anchor point:** Press Option (*Macintosh*) or Alt (*Windows*) and then click the anchor point.
- **Modify an anchor point's control points**: Control-click (Macintosh only) or right-click an anchor to display a contextual menu that sets the lengths and angles of the control points on each side of the anchor.

For example, choosing Make Sharp Corner sets the length of the control points to 0, so they lay directly on top of the anchor, making that point of the Bezier identical to a regular polygon. During editing mode, a "sharp corner" looks like a square marker with a red round marker inside of it.

- **Modify a control point or the "other" control point**: Control-click (*Macintosh only*) or right-click a control point to display a contextual menu that sets the lengths and angles of the clicked control point or the control point on the "other side" of the control point's anchor.

  For example, choosing Make Other Control Point Parallel sets the angle of the other control point so that the Bezier curve is tangent on both sides of the anchor to the angle already defined by the clicked control point.

- **Modify the control points that define the curve between two anchors**: Control-click (*Macintosh only*) or right-click the curve between anchors to display a contextual menu that sets the lengths and angles of the two control points attached to the two anchors that start and end that curve.

  For example, choosing Make 90 Degree Arc sets the angles and lengths of the two control points to draw a 90 degree arc between the two anchor points. In order for this choice to be available, the line between the two anchors must have some curve to it so that the correct orientation of the arc can be chosen.

  Choosing Break Curve to Start Another Bezier replaces the two control points between the anchors with one (NaN,NaN) coordinate pair to draw two separate Beziers curves. See **Segmented Bezier Curves** below for details. Choosing Remove All Bezier Breaks removes all (NaN,NaN) coordinate pairs from the Bezier object.

Pressing the Shift key while dragging a Bezier control handle constrains movement to angles that are increments of 15 degrees from horizontal/vertical.

Pressing the Shift key while dragging while dragging a polygon or Bezier anchor point snaps the anchor location to the nearest of lines through the original anchor location and through the locations of the two neighboring anchor points.

Pressing Shift while dragging constrains movement to horizontal or vertical directions.

Exit edit mode by clicking the arrow tool.

After exiting edit mode, you can use the environment icon to adjust the other attributes of the polygon. You can even add arrows to the start or end of the polygon. Or you can double-click a polygon to invoke the Modify Polygon dialog.

The X0 and Y0 settings determine the location of the first point.

You can change the size of the polygon by modifying the Xscale and Yscale parameters in the dialog. For example, enter 0.5 for both settings to shrink the polygon to half its normal size.

Segmented Bezier Curves

It is possible to separate a Bezier curve into segments by adding coordinate pairs that are NaN. By default, such segments are drawn as if they are separate Bezier curves. If you fill such a curve with a color having transparency, any overlapping areas will darken because they are painted twice.

If you use SetDrawEnv subpaths=1, the segments are treated as subpaths within a single Bezier curve. The segments are drawn with no line linking the subpaths, but when filled the entire curve are treated as a single Bezier curve, making it possible to create a curve with internal holes. The way those holes are filled is affected by the SetDrawEnv fillRule keyword.

The subpaths keyword applies only to polygons created with **DrawPoly** and **DrawBezier**, not to those created manually. You can manually create subpaths by entering Edit Bezier mode, right-clicking the curve, and choosing Break Curve.

A side-effect of setting SetDrawEnv subpaths=1 is a change to the way arrows are added to segmented Bezier curves. With SetDrawEnv subpaths=0 (the default), arrows are added to each segment as if they are separate Bezier curves. With subpaths=1, the arrows are added only to the first or last points in the entire Bezier curve.

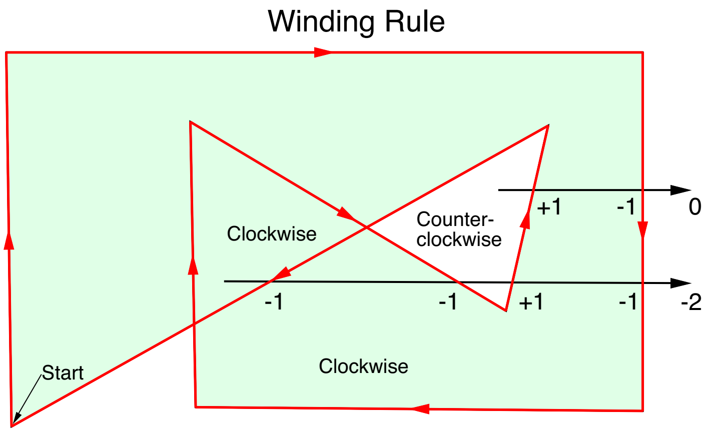
For a demonstration of segmented Bezier curves, execute this in Igor:

```
DisplayHelpTopic "Segmented Bezier Curves"
```

Polygon and Bezier Curve Fill Rules

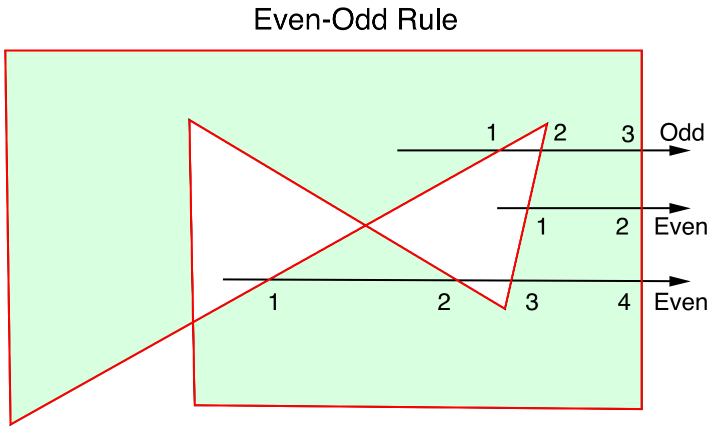
Simple polygons and Bezier curves with no intersecting edges are filled unambiguously - points within the shape are filled with color. But how do you decide what is inside and what is outside if the shape has intersecting edges?

There are two rules in common use: the Even-Odd rule and the Non-Zero Winding rule. Igor uses the Winding rule by default:



Starting at a given point, draw a line to infinity. If an edge is crossed and that edge is drawn from up to down (or clockwise with respect to the start of the line) subtract one. If the edge is drawn from down to up (or counterclockwise), add one. If the result is non-zero, it is inside and should be colored.

You can request the Even-Odd rule using `SetDrawEnv fillRule=1`. The fillRule keyword applies only to polygons and Bezier curves created with **DrawPoly** and **DrawBezier**, not to those created manually. Manually-created polygons and Bezier curves follow the winding rule by default; you can change the rule using the Modify Polygon or Modify Bezier dialogs.



Starting at a given point, draw a line to infinity. Count the number of edges crossed. If the result is an odd number, the point is inside. If it is even, it is outside.

Drawing and Editing Waves

In a graph, you can use the polygon tool to create or edit waves using the same techniques just described for drawing polygons. Click and hold on the polygon tool. Igor displays a pop-up menu containing, in addition to the usual polygon and Bezier commands, the following wave-drawing commands:

```
Draw Wave
Draw Wave Monotonic
Draw Freehand Wave
Draw Freehand Wave Monotonic
Edit Wave
Edit Wave Monotonic
```

The first four commands create and add a pair of waves with names of the form W_XPoly*nn* and W_YPoly*nn* where *nn* are digits chosen to ensure the names are unique. Draw Wave and Draw Freehand Wave work exactly like the corresponding polygon drawing described above. The monotonic variants prevent you from backtracking in the X direction. As with polygons, you enter edit mode when you finish drawing.

You can edit an existing wave, or pair of waves if displayed as an XY pair, by choosing one of the Edit Wave commands and then clicking the wave trace you wish to edit. Again, the monotonic variant prevents backtracking in the X direction. If you edit a wave that is not displayed in XY mode then you can not adjust the X coordinates since they are calculated from point numbers.

You can use the **GraphWaveDraw** and **GraphWaveEdit** operations as described in **Programmatic Drawing** on page III-73 to start wave drawing or editing.

Drawing Exporting and Importing

Copy/Paste Within Igor

You can use the Edit menu to cut, copy, clear and paste drawing objects just as you would expect.

Drawn objects retain all of their Igor properties as long as they are not modified by any other program. If, however, you export an Igor drawing to a program and then copy it back to Igor, the picture will no longer be editable by Igor, even if you made no changes to the picture.

When selected drawing objects are copied to the clipboard and then pasted, they retain their coordinates. However, this can cause the pasted objects to be placed offscreen if the object's coordinates don't fall within the displayed portion of the coordinate systems.

If you find that pasting does not yield what you expected, perhaps it is because some objects were pasted off-screen. You can use the Mover icon to examine or retrieve any of these offscreen objects.

Pasting a Picture Into a Drawing Layer

Pasting a picture from a drawing program may work differently than you expect. Igor does not attempt to take the picture apart to give you access to the component objects. Instead, Igor treats the entire picture as a single object that you can move and resize but not otherwise adjust.

You can change the scale of a pasted picture by either dragging the handles when the object is selected or by double-clicking the object and then setting the x and y scale factors in the resulting dialog.

Programmatic Drawing

All of the drawing capabilities in Igor can be used from Igor procedures. This section describes drawing programming in general terms and provides strategies for use along with example code.

The programmable nature is especially useful in creating new graph types. For example, even though Igor does not support polar plots as a native graph type we were able to create a polar plot package that produces high-quality polar graphs. (To see a demo of the package, choose File→Example Experiments→Graphing Techniques→New Polar Graph Demo.) Nonprogrammers can use the package as-is while programmers can modify the code to suit their purposes or can extract useful code snippets for their own projects.

You can get a quick start on a drawing programming project by first drawing interactively and then asking Igor to create a recreation macro for the window (click the close button and look in the Procedure window). You can then extract useful code snippets for your project. Frequently all you will have to do is replace literal coordinate values with calculated values and you are in business.

Drawing Operations

Here is a list of operations related to drawing.

DrawAction	**DrawArc**	**DrawBezier**	**DrawLine**
DrawOval	**DrawPICT**	**DrawPoly**	**DrawRect**
DrawRRect	**DrawText**	**DrawUserShape**	**GraphNormal**
GraphWaveDraw	**GraphWaveEdit**	**HideTools**	**SetDashPattern**
SetDrawEnv	**SetDrawLayer**	**ShowTools**	**ToolsGrid**

SetDrawLayer Operation

Use **SetDrawLayer** to specify which layer the following drawing commands will affect. If you use the /K flag then the current contents of the given drawing layer are deleted. See **Drawing Programming Strategies** on page III-76 for considerations in the use of SetDrawLayer and the /K flag.

SetDrawEnv Operation

This is the workhorse command of the drawing facility. It is used to specify the characteristics for a single object, to specify the default drawing environment for future objects, and to create groups of objects.

You can issue several SetDrawEnv commands in sequence; their effect is cumulative. By default, the group of SetDrawEnv commands affects only the next drawing command. Drawing commands that follow the first will use the default settings that were in effect before the SetDrawEnv commands were issued. For instance, these SetDrawEnv commands change the font and font size for only the first of the two DrawText commands:

```
SetDrawEnv fname="Courier New"
SetDrawEnv fsize=18     // 18 point Courier New, commands accumulate
DrawText 0,1,"This is in 18 point Courier New"
DrawText 0,0,"Has font and size in use before SetDrawEnv commands"
```

Use the save keyword in the SetDrawEnv specification to make the settings permanent. The usual use of the save keyword is at the end of the last SetDrawEnv command in a series. The permanent settings allow you to draw a number of objects all with the same characteristics without having to reissue SetDrawEnv commands before each object.

To create a grouping of objects, simply bracket a group of drawing commands with SetDrawEnv commands using the gstart and gstop keywords. Grouping is purely a user interface concept. Objects are drawn exactly the same regardless of grouping.

Draw<*object*> Operations

These operations, along with SetDrawEnv, operate differently depending on whether or not drawing objects are selected in the target window. If, for example, a rectangle is selected in the target window and a DrawRect command is executed then the *selected* rectangle is changed. If, on the other hand, no rectangle is selected then *a new* rectangle is created. This behavior exists to support interactive drawing and is not useful to Igor programmers, since there is no programmatic way to select a drawing object. Normally, you will be creating new objects rather than modifying existing objects.

As you can see from the format of the commands, generally all you specify in the commands themselves are the coordinates. Properties such as color and line thickness are specified by SetDrawEnv commands preceding the Draw<*object*> commands. The exception is DrawText where you specify the text to be drawn.

DrawPoly and DrawBezier Operations

The **DrawPoly** operation on page V-176 and **DrawBezier** operation on page V-174 come in the following two types:

- **Literal:** You can specify the vertices or control points with a set of literal numbers. (Polygons and bezier curves created interactively are always of the literal variety.)
- **Wave:** You can use waves to define the vertices or control points.

Because polygons and bezier curves can be of unlimited length, the /A flag allows object definitions to extend over multiple lines.

It is legal to specify a polygon with only a single point. Use this to set up a loop to append vertices to the origin vertex. Note that if you fail to add vertices and leave the polygon with just one vertex then the user will not be able to see or select the polygon.

Literal Versus Wave

As a programmer, you must choose either the literal or the wave polygon type when creating a polygon or bezier curve. This section explains the differences between the two.

The advantage of the literal method is that it does not clutter the experiment with numerous waves that may be distracting to the user. It also has the advantage that all such objects are independent of one another.

With the wave method, objects are not independent. If the user duplicates an object, or runs a window recreation macro several times, then all the objects would be linked via the wave. If the user then edits one of the objects, those edits would affect all of the associated objects; this could also be considered an advantage of the method.

A disadvantage of the literal method is that when Igor creates a recreation macro for a window containing literal method objects then all of the vertices or control points have to be specified in text. This can create huge macros that take a lot of time to create and to run. Because Igor uses the recreation macro technique when saving and restoring experiments, the use of large literal method objects can dramatically lengthen experiment save and restore time.

Wave method objects do not have this disadvantage. One nifty feature of the wave method is that you can read back the vertices or control points after the user has edited the object. Another advantage of the wave method is that you can calculate new vertices or control points at any time and the dependent objects will be automatically updated.

Screen Representation

It is important to note that the value of the first polygon vertex does not determine the location of the first vertex on the screen. The location is specified by the *xOrg*, *yOrg* parameters. Effectively the value of the first vertex is *subtracted* from all the vertices and then the value of the origin is *added* to all vertices. Thus the following commands create the exact same representation on the screen:

```
DrawPoly 120,50,1,1,{0,0,20,40,60,15}
DrawPoly 120,50,1,1,{200,300,220,340,260,315}
```

When programming, the first vertex is usually `0,0`. The *hScaling*, *vScaling* parameters are probably not of any interest to programmers; use 1 for both values.

GraphWaveDraw, GraphWaveEdit, and GraphNormal

These operations relate to graph modes that are only tangentially related to drawing.

- **GraphWaveDraw** puts the graph in a mode where the user can draw a wave using the same user interface as polygon drawing.
- **GraphWaveEdit** allows the user to edit a wave using the same user interface as polygon editing.

- **GraphNormal** puts the user into normal operation mode, and is the equivalent of clicking the top icon in the tool palette.

These operations are provided so a program can allow the user to sketch a region in a graph. The program can then read back what the user did. Unlike the other drawing modes, these wave drawing and edit modes allow user-defined buttons to be active. This is so you can provide a “done” button for the user. The button procedure should call GraphNormal to exit the drawing or edit mode.

The GraphWaveEdit command operates a little differently depending on whether or not you specify a wave with the command. If you do specify a wave then only that wave can be edited by the user. If you let the user choose a wave then he or she can switch to a new trace by clicking it.

Drawing Programming Strategies

There are three distinct ways you can structure your drawing program:

- **Append:** You can append the contents of one or more layers.
- **Replace Layer:** You can replace the contents of the layers.
- **Replace Group**: You can replace the contents of a named group.

The Replace Layer Method

Use this method when you want to maintain a fairly complex drawing completely under program control. For example you may want to extend Igor by adding a new axis type or a new display method or you may want to create a completely new kind of graph. The Polar Graphs package mentioned above utilizes the replace method.

The key to the replace method is the use of the /K flag with the SetDrawLayer command. This “kills” (deletes) the entire contents of the specified layer. After clearing out the layer you must then redraw the entire contents. To do this you will usually have to maintain some sort of data structure to hold all the information required to maintain the drawing.

For example if you are creating an artificial axis package, you will need to maintain user settings similar to those you see in Igor’s modify axis dialog. In many cases setting up a few global variables or waves in a data folder will be sufficient. As an example, see the Drawing Axes procedure file in the WaveMetrics Procedures folder.

The Replace Group Method

With named groups created with the SetDrawEnv gname keyword, you can use DrawAction to delete the group or to set the insertion point for new drawing commands. See the **DrawAction** operation on page V-172 for an example.

The Append Method

Use this method to provide convenience features that automate creation of simple drawings. You add a small drawing, such as a drop line, calibration bar, or shading rectangle, to any existing drawing objects in a given layer when the user runs a procedure or clicks a button. Such drawings are often small and modular — .

Generally, the drawing will be something the user could have done manually and may want to modify. If you need to specify a layer at all it should be a User layer. Often there will be no need to set the drawing layer at all — just use the current layer.

You may, however, need to set the layer for specific circumstances. A shading rectangle is an example of an object that should go in a specific layer, since it must be below the traces of a graph. In this case, if you use the SetDrawLayer operation, then you should set the current layer back with “SetDrawLayer UserTop”.

If you are using the append method, you should avoid using the Prog layers. This is because they are intended for use where the entire layer is to be replaced under program control.

Another consideration is whether or not you should set the default drawing environment. In general you should not but instead leave the uer in control. This varies depending on specific objectives.

Grouping

It is a good idea to use grouping for your drawing. This allows the user to move the entire drawing by merely selecting and dragging it.

Example: Drop Lines

In this example, we show a procedure that adds a line from a particular point on a wave, identified by cursor A, to the bottom axis. The procedure assumes that the graph has left and bottom axes and that cursor A is on a trace. For clarity, error checking is omitted.

```
Menu "Graph"
   "Add Drop Line", AddDropLine()
End

Function AddDropLine()
   Variable includeArrow = 2           // 1=No, 2=Yes
   Prompt includeArrow, "Include arrow head?", popup "No;Yes"
   DoPrompt "Add Drop Line", includeArrow
   includeArrow -= 1                   // 0=No, 1=Yes

   Variable xCursorPosition = hcsr(A)
   Variable yCursorPosition = vcsr(A)

   GetAxis/Q left
   Variable axisYPosition = V_min

   SetDrawEnv xcoord=bottom, ycoord=left
   if (includeArrow)
      SetDrawEnv arrow=1, arrowlen=8, arrowfat=0.5
   else
      SetDrawEnv arrow=0
   endif
   DrawLine xCursorPosition,yCursorPosition,xCursorPosition,axisYPosition
End
```

Drawing Shortcuts

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To temporarily invoke the Arrow tool while in Operate mode	Press Command-Option. You can now select an object and move or resize it. This shortcut works even if the tool palette is not showing.	Press Ctrl+Alt. You can now select an object and move or resize it. This shortcut works even if the tool palette is not showing.
To invoke the dialog to modify a control or drawing object while in Operate mode	Press Command-Option and double-click the control or drawing object. This shortcut works even if the tool palette is not showing. Also, if the palette is showing, the Drawing mode's Arrow tool becomes selected after the dialog is dismissed.	Press Ctrl+Alt and double-click the control or drawing object. This shortcut works even if the tool palette is not showing. Also, if the palette is showing, the Drawing mode's Arrow tool becomes selected after the dialog is dismissed.
To nudge a selected drawing object or control	Use the Arrow keys. Press Shift to nudge faster.	Use the Arrow keys. Press Shift to nudge faster.

Chapter

III-4

Embedding and Subwindows

Overview

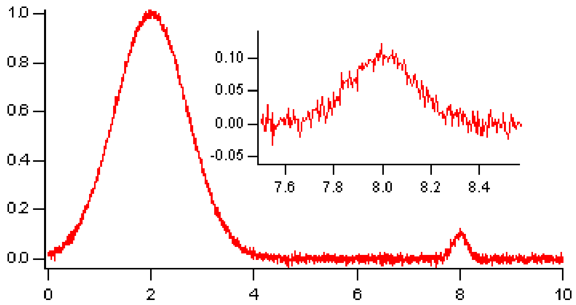
You can embed graphs, tables, Gizmo plots, and control panels into other graphs and control panels.

You can embed graphs, tables, and Gizmo plots into page layouts.

Finally, you can embed notebooks in control panels only.

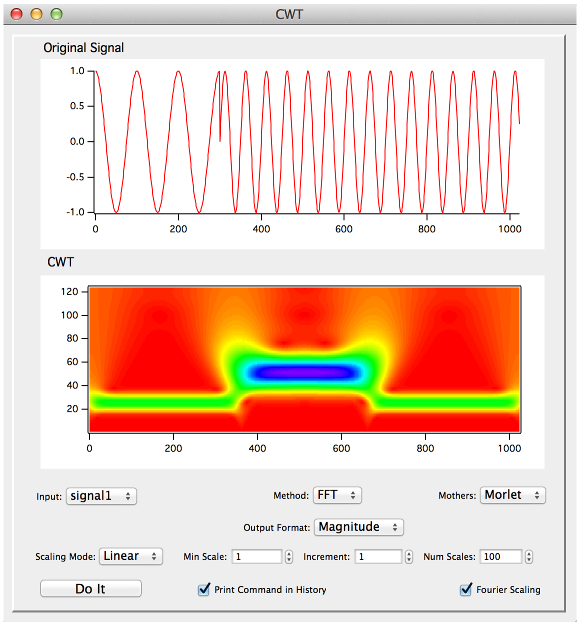
The embedded window is called a *subwindow* and the enclosing window is called the *host*. Subwindows can be nested in a hierarchy of arbitrary depth. The top host window in the hierarchy is known as the *base*.

In this example, the smaller, inset graph is a subwindow:



Although you can create graphs like this by careful positioning of free axes, it is much easier to accomplish using embedding.

In the next example, the two graphs are subwindows embedded in a host panel:



Input: signal1

Method: FFT

Mothers: Morlet

Output Format: Magnitude

Scaling Mode: Linear

Min Scale: 1

Increment: 1

Num Scales: 100

☑ Print Command in History

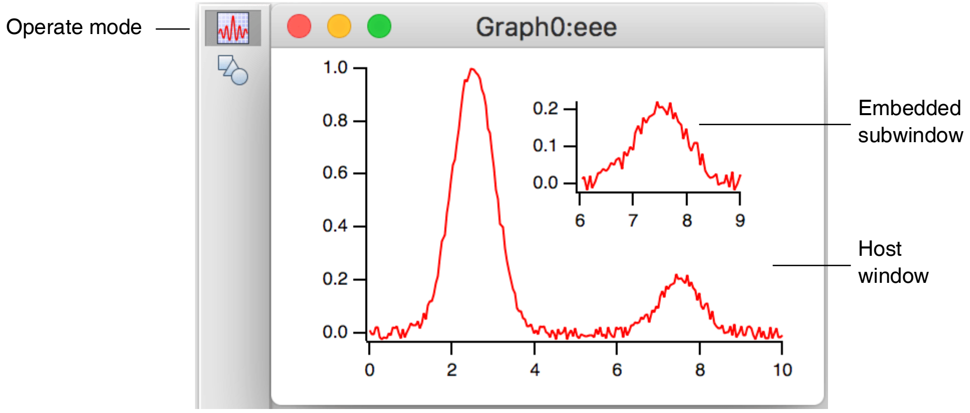
☑ Fourier Scaling

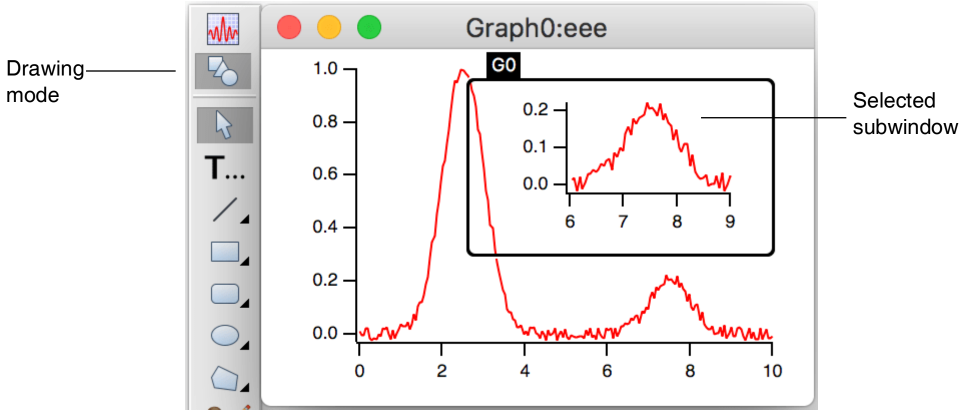
This example is derived from the CWT demo experiment which you can find in the Analysis section of your Examples folder.

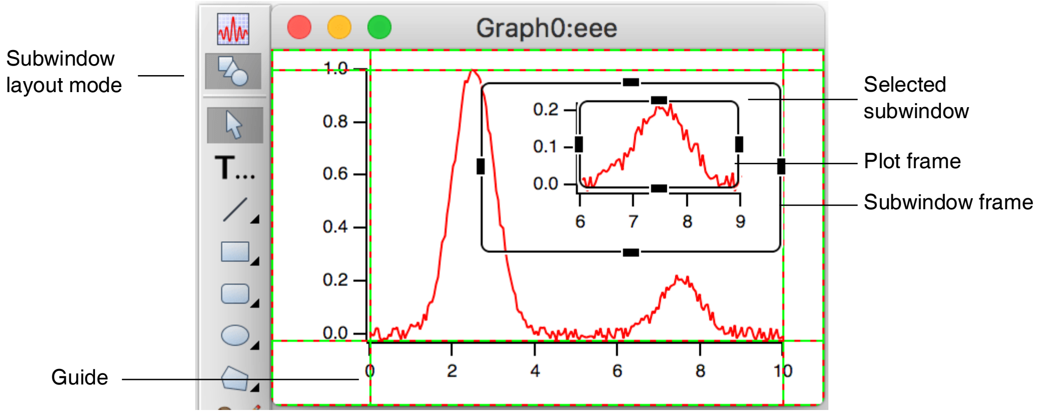
Subwindow Terminology

When a window is inserted into another window it is said to be *embedded*. In some configurations (see **Subwindow Restrictions** on page III-82), an embedded window does not support the same functionality that it has as a standalone window. It is then called a *presentation-only* object. For example, when a table is embedded in a panel, it has scroll bars and data entry features just like a standalone table. But when a table is embedded in a graph or in a page layout, it is a presentation-only object with no scroll bars or other user interface elements.

The following pictures illustrate additional subwindow terminology:







A graph, layout, or control panel window operates in one of three modes:

- Operate mode (also called normal mode)
- Drawing mode
- Subwindow layout mode

When the top icon in the tool palette is selected, the window is in operate mode. This is the mode in which the user normally uses the window or active subwindow.

When the second icon in the tool palette is selected, the window is in drawing mode. In this mode, you can use drawing tools on the window or active subwindow.

If you click the black frame around a subwindow while in edit mode, you enter subwindow layout mode. In this mode, you can position, and resize subwindows, as well as create and adjust guides.

Subwindow Restrictions

The following table summarizes the rules for allowed host and embedded subwindow configurations.

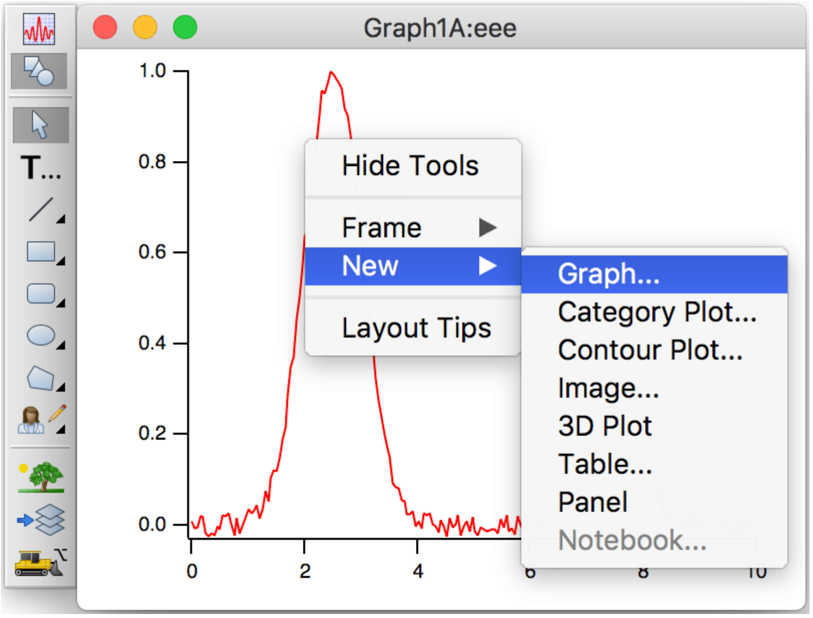
		Host			
		Graph	**Table**	**Panel**	**Layout**
Subwindow	**Graph**	Yes	No	Yes	Yes
	Table	Yes[*]	No	Yes	Yes[*]
	Panel	Yes[†]	No	Yes	No
	Layout	No	No	No	No
	Notebook	No	No	Yes	No
	Gizmo	Yes	No	Yes	No

[*] Tables embedded in graphs or layouts are presentation-only objects. They do not support editing of data.

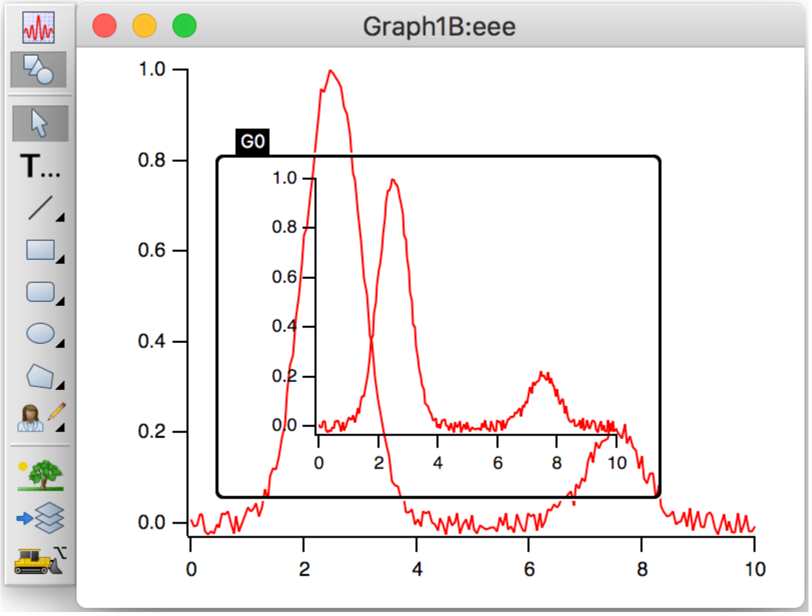
[†] Panels can be embedded in base graphs only.

Creating Subwindows

You can create subwindows either from the command line (see **Subwindow Command Concepts** on page III-92) or interactively using contextual menus. To add a subwindow interactively, show the tool palette of the base window, click the second icon to enter drawing mode, and then right-click (*Windows*) or Control-click (*Macintosh*) in the interior of the window and choose the desired type of subwindow from the New menu:



Igor presents the normal dialog for creating a new window but the result will be a subwindow:



You can position the subwindow by clicking on its heavy frame to enter subwindow layout mode (see **Subwindow Layout Mode and Guides** on page III-85). Finally, click the top icon of the tools to adjust the graphs in operate mode.

Positioning and Guides

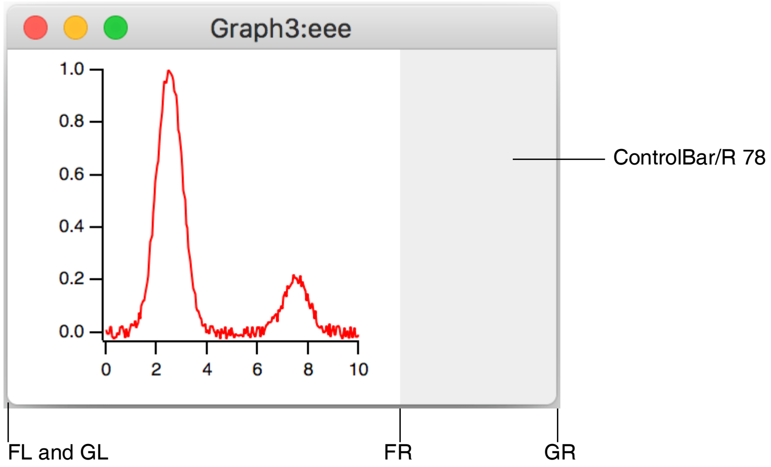
Subwindows can be positioned in their hosts in a wide variety of ways. You can specify the position of a subwindow numerically using either absolute (fixed distance) or relative modes. You can also attach key locations in a subwindow to named guides.

Guides are horizontal or vertical reference locations defined by the immediate host of a subwindow and may be either fixed (built-in) or moveable (user-defined). Built-in guides represent fixed locations of the host such as its frame or the interior plot area of graphs. Built-in guides can not be moved except by moving the object to which the guide refers.

All host windows have built-in guides named *FL*, *FT*, *FR*, and *FB* for Frame Left, Frame Top, Frame Right, and Frame Bottom. Graphs also have the corresponding *PL*, *PR*, *PT*, and *PB* for the interior plot area. In addition, base graphs have built-in guides *GL*, *GR*, *GT*, and *GB* for the graph area.

	Left	**Right**	**Top**	**Bottom**
Host Window Frame	FL	FR	FT	FB
Host Graph Rectangle	GL	GR	GT	GB
Inner Graph Plot Rectangle	PL	PR	PT	PB

The graph area is the total area of the graph window. The frame area is the total area of the graph window excluding the areas occupied by control bars.



User-defined guides can be based on built-in or other user-defined guides. They can be defined as being either a fixed distance from another guide or a relative distance between two guides.

Reference points of a subwindow that can be attached to guides include the outer left, right, top and bottom for all subwindow types and, for graphs only, the interior plot area.

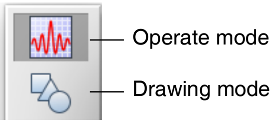
Guides are especially useful when creating stacked graphs. By attaching the plot left (PL) location on each graph to a user-defined guide, all left axes will be lined up and will move in unison when you drag the guide around. This is illustrated in **Layout Mode and Guide Tutorial** on page III-86.

Frames

You can specify a frame style for each subwindow. Frames, if any, are drawn inside the rectangle that defines the location of the subwindow and the normal content is then inset by the frame thickness. Frames can also be specified for base graph and panel windows. This is handy when you want to include a frame when you export or print a graph. You can adjust the frame for a window or subwindow by right-clicking (*Windows*) or Control-clicking (*Macintosh*) in drawing mode.

Subwindow User-Interface Concepts

Each host window has two main modes corresponding to the top two icons in the window’s tool palette. Choose Show Tools from the Graph or Panel menu to show the tool palette. Clicking the top icon selects operate mode and clicking the second icon selects drawing mode.



When using subwindows, there is a third mode: subwindow layout mode (see **Subwindow Layout Mode and Guides** on page III-85).

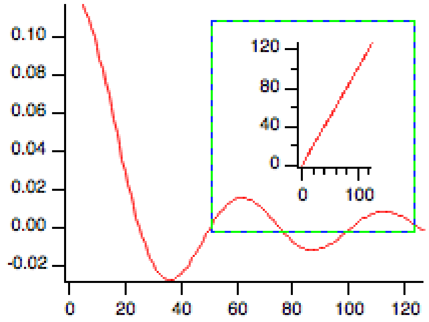
When not using subwindows, a particular window is the *target window* — the default window for command-line commands that do not explicitly specify a window. The addition of subwindows leads to the analogous concept of the *active subwindow*.

You make a subwindow the active subwindow by clicking it. In operate mode the active subwindow is indicated by a green and blue border. In drawing mode it is indicated by a heavy black border with the name of the subwindow shown in the upper left corner.

Panel subwindows are exceptions in that clicking them in operate mode does not make them the active subwindow. You must click them while in drawing mode.

As an example, execute the following:

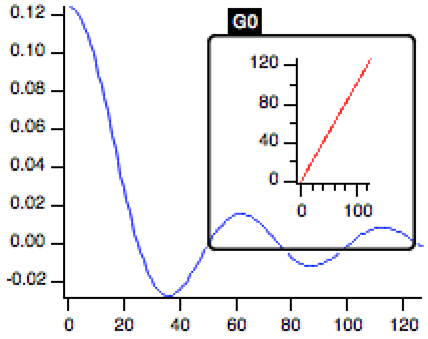
```
Make/O jack=sin(x/8)/x,sam=x
Display jack
Display/W=(0.5,0.14,0.9,0.7)/HOST=# sam
```



Notice the green and blue border around the newly created subwindow:.

This indicates that it is the active subwindow. Now double click on the curve in the host window, outside the subwindow border. Change the color of the trace `jack` to blue and notice that the subwindow is no longer active. Now move the mouse over the subwindow and notice that the cursor changes to the usual shapes corresponding to the parts of the graph that it is hovering over. Drag out a selection rectangle in the plot area of the subwindow and notice that the marquee pop-up menu is available for use on the subwindow and that the subwindow has been activated. Depending on your actions in a window, Igor activates subwindows as appropriate and generally you do not have to be aware of which subwindow is active.

Choose Show Tools from the Graph menu and notice that the tools are provided by the main window. Tools and the cursor information panel are hosted by the base window but apply to the active subwindow, if any.



Click the drawing icon in the tool palette.

Notice the subwindow, which had been indicated as active using the green and blue border, now has a heavy frame.

You are now in a mode where you can use drawing tools on the subwindow. To draw in the main window, click outside the subwindow to make the host window active.

When in operate mode, the main menu bar includes a menu (e.g., Graph) appropriate for the base window. If a subwindow is of the same type as the base window, you can target it by clicking it to make it the active subwindow and using the same main menu bar menu.

To make changes to a subwindow of a different type, you can use a context click to access a menu appropriate for the subwindow.

In drawing mode, the main menu includes a menu appropriate for the active subwindow. For example, if the base window is a graph with an table subwindow, the menu bar shows the Graph menu when in operate mode. When the table subwindow is selected in drawing mode, the main menu bar shows the Table menu.

In drawing mode, you can right-click (*Windows*) or Control-click (*Macintosh*) to get a pop-up menu from which you can choose frame styles and insert new subwindows or delete the active subwindow. Deleting a subwindow is not undoable.

The info panel (see **Info Panel and Cursors** on page II-319) in a graph targets the active subgraph. You can not simultaneously view or move cursors in two different subgraphs.

Subwindow Layout Mode and Guides

To layout one or more subwindows in a host window, enter drawing mode, click the selector (arrow) tool and click in a subwindow. A heavy frame will be drawn with the name of the subwindow in the upper left. Now click on the frame to enter subwindow layout mode. In this mode, the subwindow is drawn with a light frame with handles in the middle of each side. In addition, built-in and user guides are drawn as dashed red and green lines.

In subwindow layout mode, you can move a subwindow dragging its frame and resize it using the handles. If you drag a handle close to a guide, it will snap in place and attach itself to the guide. However, if one or more handles are attached to guides and you drag the subwindow using the frame, all attachments are deleted.

A graph subwindow is drawn using two frames. The inner frame represents the plot area of the graph. Its handles can be attached to guides to allow easy alignment of multiple graph subwindows.

You can create user-defined guides by pressing Alt (*Windows*) or Option (*Macintosh*) and then click-dragging an existing guide. By default, the new guide will be a fixed distance from its parent.

You can convert the new guide to relative mode, where the guide position is specified as a fraction of the distance between two other guides. You do this by right-clicking (*Windows*) or Control-click (*Macintosh*) on the new guide to display a contextual menu and then choosing a partner guide from the "make relative to" submenu.

You can also use the contextual menu to convert a relative guide to fixed or to delete a guide, if it is not in use.

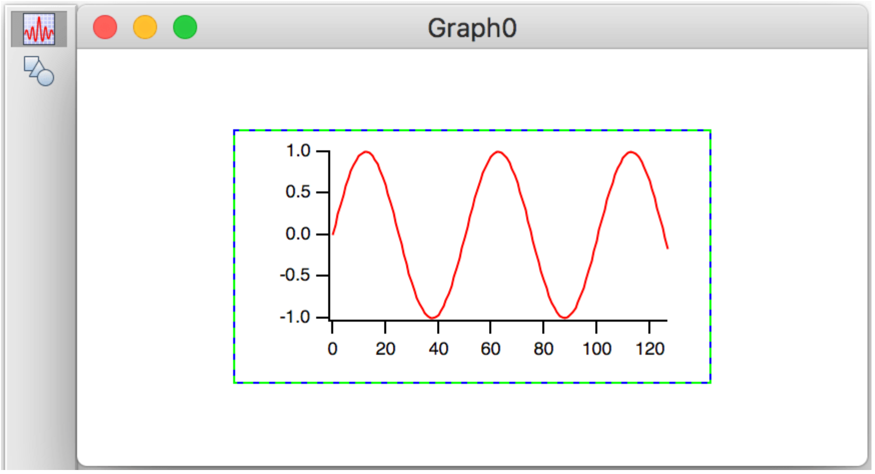
The contextual menu appears only for user-defined guides, not for built-in guides.

Layout Mode and Guide Tutorial

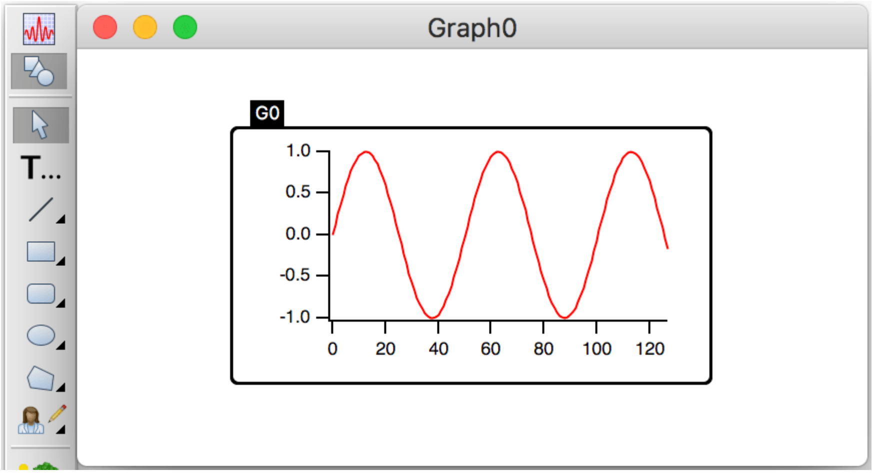
In a new experiment, execute these commands:

```
Make/O jack=sin(x/8),sam=cos(x/8)
Display
Display/HOST=# jack
ShowTools/A
```

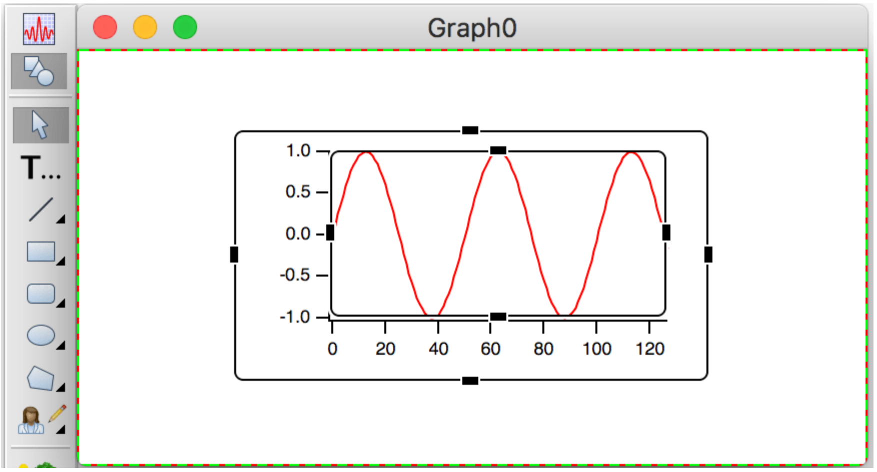
The first `Display` command created an empty graph and the second inserted a subgraph. The graph is in operate mode and it looks like this:



Click the lower icon in the tool palette to enter drawing mode and notice the subwindow is drawn with a black frame with the name of the subwindow (`G0`):

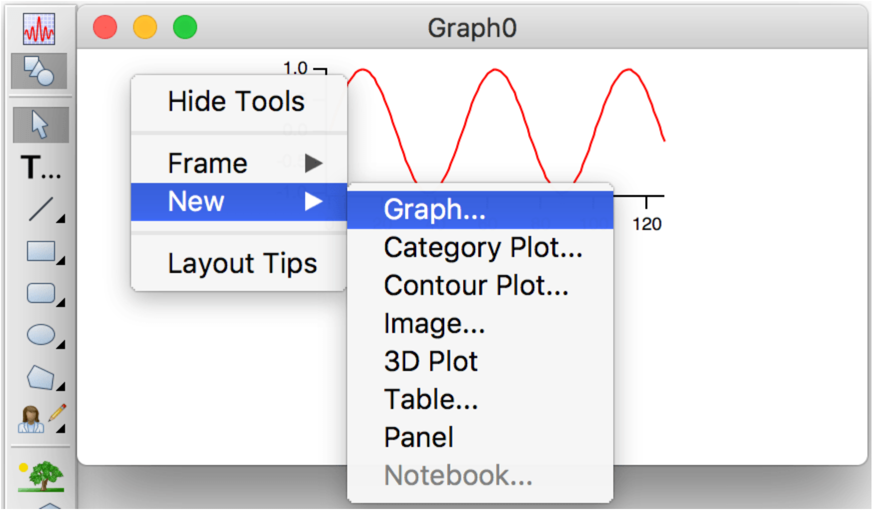


Use the arrow tool to click on the black frame around the subgraph. You are now in subwindow layout mode as indicated by the rectangles with handles on each edge of the subgraph.



Position the mouse over the outer rectangle until the cursor changes to a four-headed arrow. Drag the subwindow up as high as it will go and then drag the bottom handle up to just above the halfway point so that the subgraph is in the upper half of the window.

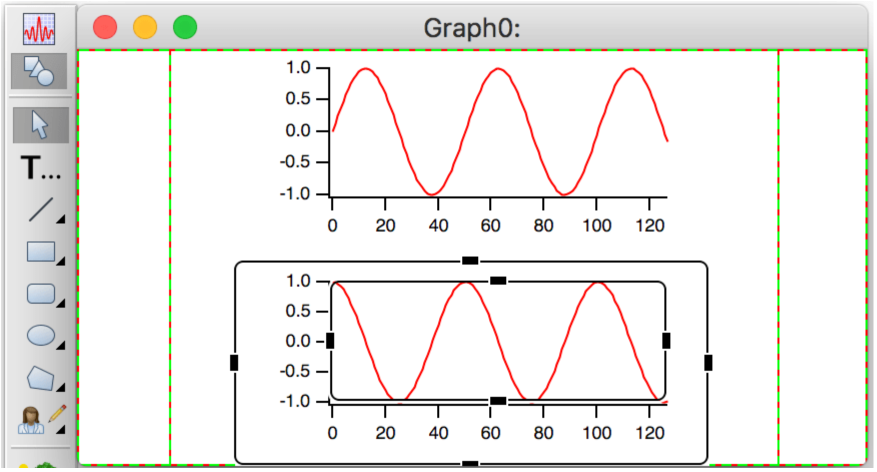
Click outside the subwindow to leave subwindow layout mode and then click again to select the main (empty) graph as the active subwindow. Right-click (*Windows*) or Control-click (*Macintosh*) below the subgraph and choose the New→Graph from the pop-up menu:



Pick sam as the Y wave in the resulting dialog and click Do It. This creates a new subwindow and makes it active. Click on the heavy frame to enter subwindow layout mode for the new subgraph and position it in the lower half of the window.

While still in subwindow layout mode for the second graph, notice the red and green dashed lines around the periphery. These are fixed guides and are properties of the base window. Press Alt (*Windows*) or Option (*Macintosh*) and move the mouse over the left hand dashed line. When you notice the cursor changing to a two headed arrow, click and drag to the right about 3 cm to create a user-defined guide.

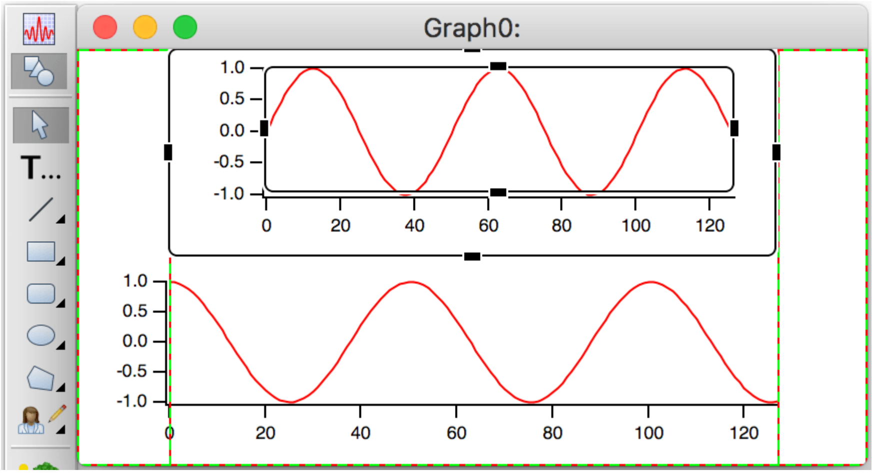
Use the same technique to create another user-defined guide based on the right edge also inset by about 3 cm:



Move the mouse over the new guides and notice the cursor changes to a two headed arrow indicating they can be moved.

While still in subwindow layout mode for the second graph, click in the black handle centered on the left axis and drag the handle over the position of the left user guide. Notice that it snaps into place when it is near the guide. Release the mouse button and use the same technique to connect the right edge of the plot area to the right user guide.

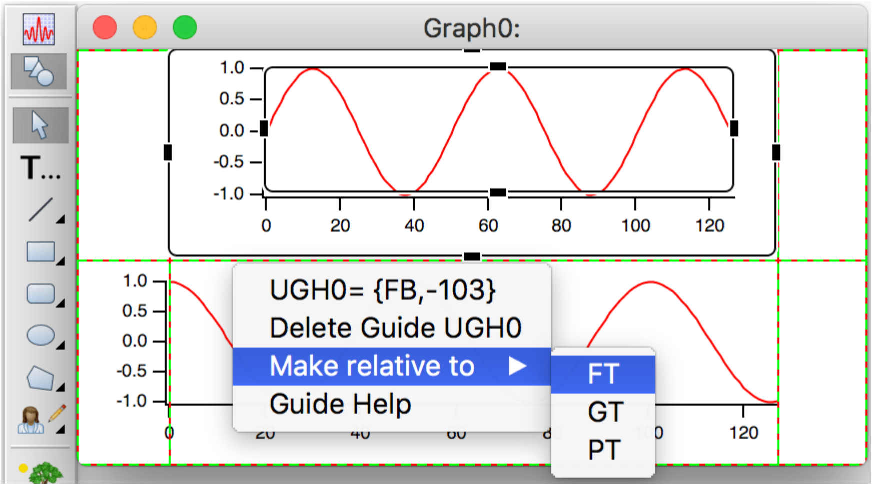
Now place the top subgraph in subwindow layout mode and connect its left and right plot area handles to the user guides:



While still in subwindow layout mode, drag the user guides around and notice that both graphs follow.

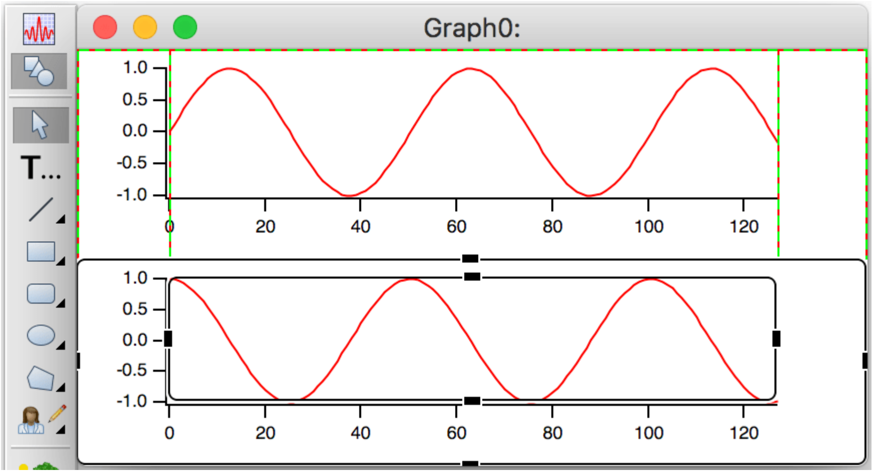
The two guides we created are a fixed distance from another guide (the frame left (FL) and frame right (FR) in this case). We will now create a relative guide.

Press Alt (*Windows*) or Option (*Macintosh*) and move the mouse over the bottom dashed line near the window frame. When you notice the cursor changing to a two headed arrow, click and drag up to about the middle of the graph to create another user-defined guide. Position the mouse over the new guide, right-click (*Windows*) or Control-click (*Macintosh*), and choose Make Relative to→FT from the pop-up menu.



Now, as you resize the window, the guide remains at the same relative distance between the bottom (FB) and the top (FT).

Use the handles to attach the bottom of the top graph to the new guide and then put the bottom graph into subwindow layout mode and attach its top to the guide:



Graph Control Bars and Subpanels

A control bar is a control panel subwindow attached to an edge of a graph.

You can drag out control bar areas from all four sides of a graph window.You do this by entering drawing mode, clicking just inside an outer edge of the window, and dragging toward the center of the window. This creates a subpanel and anchors it to appropriate guides.

Page Layouts and Subwindows

Subwindows in page layouts windows can be a bit confusing because you use two different modes to position conventional graph and table objects versus subwindows. In the normal mode for a page layout (top icon in the tool palette), you can adjust the positioning of conventional layout objects but not subwindows because they are in operate mode.

In operate mode, subwindow graphs act just like conventional graph windows and allow marquee expansion, double clicking on graph elements to bring up dialogs, dragging textboxes and axes and other normal graph behavior. But to adjust the position of subwindows, you have to enter subwindow layout (see **Subwindow Layout Mode and Guides** on page III-85).

Although a bit confusing, the use of subwindows in page layouts is very useful because the page layout is then self-contained and need not refer to other windows.

Here are a few reasons to use the conventional window or object layout method rather than subwindows:

- You can use cursors and buttons in a full graph but not in a subgraph.
- You can place the same graph in multiple layouts.
- You can have a graph window be a different, and more convenient, size than the layout object.

In a page layout, you can insert a graph subwindow or table by first using the marquee tool to specify the desired location and then using the pop-up menu available in the interior of the marquee to choose one of several subwindow types. If you need to reposition the new subwindow, you will need to enter drawing mode and use the selector (arrow) tool of the drawing tool palette, not the layout arrow tool.

You can convert conventional layout graph and table objects to subwindows via the contextual menu for the object. In operate mode (select the top icon in the layout tool palette), Control-click or right-click anywhere on the layout object and choose Convert Graph/Table To Embedded. Portions of a graph that are not allowed in layouts, such as buttons and subpanels, are lost in the conversion.

You can convert a subgraph or subtable to a conventional window and layout object via the contextual menu. In operate mode, Control-click or right-click and choose Convert To Graph/Table and Object. In a graph you must click in an area free of traces or axes, such as in the graph margin, to get the correct popup menu.

Notebooks as Subwindows in Control Panels

You can create a notebook subwindow in a control panel using the NewNotebook operation. A notebook subwindow might be used to present status information to the user or to permit the user to enter multi-line text. Here is an example:

```
NewPanel /W=(150,50,654,684)
NewNotebook /F=1 /N=nb0 /HOST=# /W=(36,36,393,306)
Notebook # text="Hello World!\r"
```

The notebook subwindow can be plain text (/F=0) or formatted text (/F=1).

By default, the notebook ruler is hidden when a notebook subwindow is created. You can change this using the Notebook operation.

The status bar is never shown in a notebook subwindow and there is no way to show it.

To make it easier to use for text input or display, when a formatted text notebook subwindow is first created, and when you resize the width of the subwindow, Igor automatically adjusts the Normal ruler's right indent so that all of the text governed by the Normal ruler fits in the subwindow. This adjustment is done for the Normal ruler only. Other rulers, including Normal+ (variations of Normal) rulers, are not adjusted.

You can programmatically insert text in the notebook using the **Notebook** operation.

If you create a window recreation macro for the control panel, by default the contents of the notebook subwindow are saved in the recreation macro. If you later run the macro to recreate the control panel, the notebook subwindow's contents are restored. This also applies to experiment recreation which automatically uses window recreation macros.

If you do not want the contents of the notebook subwindow to be preserved in the recreation macro, you must disable the autosave property, like this:

```
Notebook Panel0#nb0, autosave=0
```

When you create a window recreation macro while autosave is on, it will contain commands that look something like this:

```
Notebook kwTopWin, zdata="GaqDU%ejN7!Z)ts!+J\\.F^>EB"
Notebook kwTopWin, zdata= "jmRiCVsF?/]21,HG<k,\"@i1,&\\.F^>EB"
Notebook kwTopWin, zdataEnd=1
```

The Notebook zdata command sends to the notebook encoded binary data in an Igor-private format that represents the contents of the notebook when the recreation macro was created. In real life, there would be a number of zdata commands, one after the other, which cumulatively define the contents of the notebook. The notebook accumulates all of the zdata text. The zdataEnd command causes the notebook to decode the binary data and use it to restore the notebook's contents.

When you save an experiment containing a control panel, a window recreation macro is created for you by Igor. When you open the experiment, Igor runs the recreation macro to recreate the control panel. If autosave is off, after saving and reopening the experiment, the notebook will be empty. If autosave is on, the window recreation macro will include zdata and zdataEnd commands that restore the contents of the notebook subwindow.

The encoded binary data includes a checksum. If the Notebook zdata commands have been altered, the checksum will fail and you will receive an error when the Notebook zdataEnd command executes.

For a demonstration of notebook subwindows, choose File→Example Experiments→Feature Demos2→Notebook in Panel.

Subwindow Command Concepts

All operations that create window types that can be subwindows can take a `/HOST=`*hcSpec* flag in order to create a subwindow in a specific host. In addition, operations and functions that can modify or operate on a subwindow can affect a specific subwindow using the `/W=`*hcSpec* flag, for operations, or an *hcSpec* as a string parameter, for functions.

Subwindow Syntax

This table summarizes the command line syntax for identifying a subwindow:

Subwindow Specification	Location
baseName	Base host window
baseName#*sub1*	Absolute path from base host window
#*sub1*	Relative path from the active window or subwindow
#	Active window or subwindow
##	Host of active subwindow

The window path uses the # symbol as a separator between a window name and the name of a subwindow. If you have a panel subwindow named P0 inside a graph subwindow named G0 inside a panel named Panel0, the absolute path to the panel subwindow would be `Panel0#G0#P0`. The relative path from the main panel to the panel subwindow would be `#G0#P0`.

Subwindow Syntax for Page Layouts

For page layout windows, the standard syntax described above applies to subwindows in the currently active page only. To access a subwindow in any page, whether that page is the active page or not, use a page number in square brackets:

Subwindow Specification	Location
LayoutName	Base host window, active page
LayoutName[page]	Base host window, specified page
LayoutName#*sub1*	Absolute path on active page
LayoutName[page]#*sub1*	Absolute path on specified page

Page layout page numbers start at page 1.

This example embeds a graph in page 2 of the layout regardless of what the active page is:

```
NewLayout
LayoutPageAction appendPage         // Create page 2
LayoutPageAction page=1             // Page 1 is now the current page
Make/O wave0 = sin(x/8)
Display/HOST=Layout0[2] wave0       // Add a graph subwindow to page 2
ModifyGraph/W=Layout0[2]#G0 mode=3  // Set markers mode in graph subwindow
```

Subwindow Sizing

When `/HOST` is used in conjunction with **Display**, **NewPanel**, **NewWaterfall**, **NewImage**, **NewGizmo** and **Edit** commands to create a subwindow, the values used with the window size `/W=(`*a*`,`*b*`,`*c*`,`*d*`)` flag can have one of two different meanings. If all the values are less than 1.0, then the values are taken to be fractional relative to the host's frame. If any of the values are greater than 1.0, then they are taken to be fixed locations measured in points relative to the top left corner of the host.

Guides may override the numeric positioning set by `/W`. All operations supporting `/HOST` take the `/FG=(`*`gleft`*`,`*`gtop`*`,`*`gright`*`,`*`gbottom`*`)` flag where the parameters are the names of built-in or user-defined guides. FG stands for frame guide and this flag specifies that the outer frame of the subwindow is attached to the guides. A `*` character in place of a name indicates that the default value should be used.

The inner plot area of a graph subwindow may be attached to guides using the analogous PG flag. Thus a subgraph may need up to three specifications. For example:

```
Display /HOST=# /W=(0,10,400,200) /FG=(FL,*,FR,*) /PG=(PL,*,PR,*)
```

When the subwindow position is fully specified using guides, the /W flag is not needed but it is OK to include it anyway.

Subwindow Operations and Functions

Here are the main operations and functions that are useful in dealing with subwindows:

ChildWindowList(*hostName*)

DefineGuide [/W= *winName*] *newGuideName* = {[*guideName1*, *val*
[, *guideName2*]]} [,…]

KillWindow *winNameStr*

MoveSubwindow [/W=*winName*] *key* = (*values*)[, *key* = (*values*)]…

RenameWindow *oldName*, *newName*

SetActiveSubwindow *subWinName*

Chapter

III-5

Exporting Graphics (Macintosh)

Overview

This chapter discusses exporting graphics from Igor graphs, page layouts, tables, and Gizmo plots to another program on Macintosh. You can export graphics through the clipboard by choosing Edit→Export Graphics, or through a file, by choosing File→Save Graphics.

Igor Pro supports a number of different graphics export formats. You can usually obtain very good results by choosing the appropriate format, which depends on the nature of your graphics and the characteristics of the program to which you are exporting.

Unfortunately, experimentation is sometimes required to find the best export format for your particular circumstances. This section provides the information you need to make an informed choice.

This table shows the available graphic export formats on Macintosh:

Export Format	Export Method	Notes
Quartz PDF	Clipboard, file	Platform-independent and high quality, generated via the operating system.
LowRes PDF	Clipboard, file	Obsolete. Use Quartz PDF or Igor PDF instead.
Igor PDF	Clipboard, file	Platform-independent and high quality. Igor PDF with CMYK color does not support transparency.
EPS (Encapsulated Postscript)	File only	Platform-independent. Supports high resolution. EPS does not support transparency. Useful only when exporting to PostScript-savvy program (e.g., Adobe Illustrator, Tex).
PNG (Portable Network Graphics)	Clipboard, file	Platform-independent bitmap format. Uses lossless compression. Supports high resolution.
JPEG	Clipboard, file	Platform-independent bitmap format. Uses lossy compression. Supports high resolution. PNG is a better choice for scientific graphics.
TIFF	Clipboard, file	Platform-independent bitmap format. Supports high resolution but not compression.
SVG	Clipboard, file	Platform-independent vector graphics format. As of this writing, few Macintosh programs support SVG.

PDF Format

PDF (Portable Document Format) is Adobe's platform-independent vector graphics format that has been adopted by Apple as the standard graphics format for OS X. This is the best format as long as your destination program supports it.

The Quartz PDF format is generated by the operating system while the Igor PDF format uses the Qt application framework. For most graphics they produce similar output, but if you have an issue, it is worth trying both formats to see which is better. If you request CMYK color with the Igor PDF format, older Igor code that does not support transparency is used.

Blurry Images in PDF

When Igor exports an image plot, it exports the image as a single image object when possible. However, some PDF viewers, most notably Apple's, take it upon themselves to blur the pixels. To get around this, you can tell Igor to draw image pixels as individual rectangles using the **ModifyImage** interpolate keyword with a value of -1. You should do this only when necessary as the resulting PDF will be much larger.

Encapsulated PostScript (EPS) Format

Encapsulated PostScript was a widely-used, platform-independent vector graphics format consisting of PostScript commands in plain text form. It usually gives the best quality, but it works only when printed to a PostScript printer or exported to a PostScript-savvy program such as Adobe Illustrator. You should use only PostScript fonts (e.g., Helvetica).

Encapsulated PostScript was a widely-used platform-independent vector graphics format consisting of PostScript commands in plain text form. EPS is largely obsolete but still in use. It usually gives good quality, but it works only when printed to a PostScript printer or exported to a PostScript-savvy program such as Adobe Illustrator. You should use only PostScript fonts such as Helvetica. EPS does not support transparency.

Prior to Igor Pro 7, Igor embedded a screen preview in EPS files. This is no longer done because the preview was not cross-platform and caused problems with many programs.

EPS files normally use the RGB encoding to represent color but you can also use CMYK. See **Exporting Colors** on page III-99 for details.

Igor Pro exports EPS files using PostScript language level 2. This allows much better fill patterns when printing and also allows Adobe Illustrator to properly import Igor's fill patterns. For backwards compatibility with old printers, you can force Igor to use level 1 by specifying /PLL=1 with the SavePICT operation.

If the graph or page layout that you are exporting as EPS contains a non-EPS picture imported from another program, Igor exports the picture as an image incorporated in the output EPS file.

Igor can embed TrueType fonts as outlines. See **Font Embedding** on page III-99 and **Symbols with EPS and Igor PDF** on page III-493 for details.

SVG Format

SVG (Scalable Vector Graphics) is an XML-based platform-independent 2D vector and raster graphics format developed by the World Wide Web Consortium. It is often used for displaying graphics in web pages and is a good choice for other uses if the destination program supports it. However, as of this writing, few Macintosh programs support SVG. Safari supports it but you can not import an SVG file or paste an SVG graphic into Preview.

Platform-Independent Bitmap Formats

PNG (Portable Network Graphics) is a platform-independent bitmap format that uses lossless compression and supports high resolution. It is a superior alternative to JPEG or GIF. Although Igor can export and import PNG images via files and via the clipboard, some programs that allow you to insert PNG files do not allow you to paste PNG images from the clipboard.

JPEG is a lossy image format whose main virtue is that it is accepted by all web browsers. However all modern web browsers support PNG so there is little reason to use JPEG for scientific graphics. Although Igor can export and import JPEG via the clipboard, not all programs can paste JPEGs.

TIFF is an Adobe format often used for digital photographs. Igor's implementation of TIFF export does not use compression. TIFF files normally use the RGB scheme to specify color but you can also use CMYK. See **Exporting Colors** on page III-99 for details. There is no particular reason to use TIFF over PNG unless you are exporting to a program that does not support PNG. Igor can export and import TIFF via files and via the clipboard and most graphics programs can import TIFF.

Choosing a Graphics Format

Because of the wide variety of types of graphics, destination programs, printer capabilities, operating system behaviors and user-priorities, it is not possible to give definitive guidance on choosing an export format. But here is an approach that will work in most situations.

If the destination program accepts PDF or SVG, then they are probably your best choice because of their high-quality vector graphics and platform-independence.

Encapsulated PostScript (EPS) is also a very high quality format which works well if the destination program supports it but it does not support transparency.

If SVG, PDF and EPS are not appropriate, your next choice would be a high-resolution bitmap. The PNG format is preferred because it is platform-independent and is compressed.

Exporting Graphics Via the Clipboard

To export a graphic from the active window via the clipboard, choose Edit→Export Graphics. This displays the Export Graphics dialog.

When you click the OK button, Igor copies the graphics for the active window to the clipboard. You can then switch to another program and do a paste.

When a graph, page layout, or Gizmo plot is active and in operate mode, choosing Edit→Copy copies to the clipboard whatever format was last used in the Export Graphics dialog. For a table, Edit→Copy copies the selected numbers to the clipboard and does not copy graphics.

When a page layout has an object selected or when the marquee is active, choosing Edit→Copy copies an Igor object in a format used internally by Igor along with a PDF and does not use the format from the Export Graphics dialog

Although Igor can export a number of different formats, not all programs can recognize them on the clipboard. You may need to export via a file.

Igor can export PNG images to the clipboard and can then paste them back in. On the Macintosh, the clipboard type is `'PNGf'` but because there is no standard for PNG on the clipboard it is therefore unlikely that other programs can import them except as files.

Exporting Graphics Via a File

To export a graphic from the active window via a file, choose File→Save Graphics. This displays the Save Graphics File dialog.

The controls in the Format area of the dialog change to reflect options appropriate to each export format.

When you click the Do It button, Igor writes the graphic to a file. You can then switch to another program and import the file.

If you select _Use Dialog_ from the Path pop-up menu, Igor presents a Save File dialog in which you can specify the name and location of the saved file.

Exporting a Graphic File for Transfer to a Windows Computer

The best method for transferring Igor graphics to a Windows computer is to transfer the entire Igor experiment file, open it in Igor for Windows, and export the graphic via one of the Windows-compatible methods available in Igor for Windows.

Prior to Igor Pro 9, PDF pictures were not displayed by Igor on Windows. They are nowrendered as high-resolution bitmaps.

Exporting a Section of a Layout

To export a section of a page layout, use the marquee tool to identify the section and then choose Edit→Export Graphics or File→Save Graphics.

If you don’t use the marquee and the Crop to Page Contents checkbox is checked, Igor exports the area of the layout that is in use plus a small margin. If it is unchecked, Igor exports the entire page.

Exporting Colors

The PDF, EPS and TIFF graphics formats normally use the RGB scheme to specify color. Some publications require the use of CMYK instead of RGB, although the best results are obtained if the publisher does the RGB to CMYK conversion using the actual characteristics of the output device. For those publications that insist on CMYK, you can use the SavePICT /C=2 flag

Font Embedding

This section is largely obsolete. It applies only to EPS files and PDF files generated for CMYK colors. Even then you need this information only in unusual circumstances.

You can embed TrueType fonts in EPS files and in PDF files. This means you can print EPS or PDF files on systems lacking the equivalent PostScript fonts. This also helps for publications that require embedded fonts.

Font embedding is done automatically for the Quartz PDF format and you do not need to bother with this section unless you are using EPS or Igor PDF formats.

Font embedding is always on and the only option is to not embed standard fonts. For most purposes, embedding only non-standard fonts is the best choice.

Igor embeds TrueType fonts as synthetic PostScript Type 3 fonts derived from the TrueType font outlines. Only the actual characters used are included in the fonts.

Not all fonts and font styles on your system can be embedded. Some fonts may not allow embedding and others may not be TrueType or may give errors. Be sure to test your EPS files on a local printer or by importing into Adobe Illustrator before sending them to your publisher. You can test your PDF files with Adobe Reader. You can also use the “TrueType Outlines.pxp” example experiment to validate fonts for embedding. Choose Files→Example Experiments→Feature Demos 2→TrueType Outlines.

For EPS, Igor determines if a font is non-standard by attempting to look up the font name in a table described in **PostScript Font Names (OS X)** on page III-100 after doing any font substitution using that table. In addition, if a nonplain font style name is the same as the plain font name, then embedding is done. This means that standard PostScript fonts that do not come in italic versions (such as Symbol), will be embedded for the italic case but not for the plain case.

For PDF, non-standard fonts are those other than the basic fonts guaranteed by the PDF specification to be built-in to any PDF reader. Those fonts are Helvetica and Times in plain, bold, italic and bold-italic forms as well as Symbol and Zapf Dingbats only in plain style. If embedding is not used or if a font can not be embedded, fonts other than those just listed will be rendered as Helvetica and will not give the desired results.

PostScript Font Names (OS X)

When generating PostScript, Igor needs to generate proper PostScript font names. This presents problems under Macintosh OS X. Igor also needs to be able to substitute PostScript fonts for non-PostScript fonts. Here is a list of font names that are translated into the standards:

TrueType Name	PostScript Name
Helvetica	Helvetica
Arial	Helvetica
Helvetica-Narrow	Helvetica-Narrow
Arial Narrow	Helvetica-Narrow
Palatino	Palatino
Book Antiqua	Palatino
Bookman	Bookman
Bookman Old Style	Bookman
Avant Garde	AvantGarde
Century Gothic	AvantGarde
New Century Schlbk	NewCenturySchlbk
Century Schoolbook	NewCenturySchlbk
Courier	Courier
Courier New	Courier
Zapf Chancery	ZapfChancery
Monotype Corsiva	ZapfChancery
Zapf Dingbats	ZapfDingbats
Monotype Sorts	ZapfDingbats
Symbol	Symbol
Times	Times
Times New Roman	Times

Chapter

III-6

Exporting Graphics (Windows)

Overview

This chapter discusses exporting graphics from Igor graphs, page layouts, tables, and Gizmo plots to another program on Windows. You can export graphics through the clipboard by choosing Edit→Export Graphics, or through a file, by choosing File→Save Graphics.

Igor Pro supports a number of different graphics export formats. You can usually obtain very good results by choosing the appropriate format, which depends on the nature of your graphics, your printer and the characteristics of the program to which you are exporting.

Unfortunately, experimentation is sometimes required to find the best export format for your particular circumstances. This section provides the information you need to make an informed choice.

This table shows the available graphic export formats on Windows:

Export Format	Export Method	Notes
EMF (Enhanced Metafile)	Clipboard, file	Windows-specific vector format.
BMP (Bitmap)	Clipboard, file	Windows-specific bitmap format. Does not use compression.
Igor PDF	Clipboard, file	Platform-independent and high quality. Igor PDF with CMYK color does not support transparency.
EPS (Encapsulated Postscript)	File only	Platform-independent except for the screen preview. Supports high resolution. EPS does not support transparency. Useful only when exporting to PostScript-savvy program (e.g., Adobe Illustrator, Tex).
PNG (Portable Network Graphics)	Clipboard, file	Platform-independent bitmap format. Uses lossless compression. Supports high resolution.
JPEG	Clipboard, file	Platform-independent bitmap format. Uses lossy compression. Supports high resolution. PNG is a better choice for scientific graphics.
TIFF	Clipboard, file	Platform-independent bitmap format. Supports high resolution but not compression.
SVG	Clipboard, file	Platform-independent vector graphics format. A good choice if the destination program supports SVG. As of this writing, few Windows programs support SVG.

Metafile Formats

The metafile formats are Windows vector graphics formats that support drawing commands for the individual objects such as lines, rectangles and text that make up a picture. Drawing programs can decompose a metafile into its component parts to allow editing the individual objects. Most word processing programs treat a metafile as a black box and call the operating system to display or print it.

Enhanced Metafile (EMF) is the primary Windows-native graphics format. It comes in two flavors: the older EMF and a newer EMF+. Igor “dual EMF” by default. A dual EMF contains both a plain EMF and an EMF+; applications that don’t support EMF+ will use the plain EMF component. EMF+ is needed if transparency (colors with an alpha channel) is used. You can export using the older EMF format if the destination program does not work well with EMF+ - see **Graphics Technology** on page III-506 for details.

EMF is easy to use because nearly all Windows programs can import it and because it can be copied to the clipboard as well as written to a file. Some programs, notably some older versions of Microsoft Office, require that you choose Paste Special rather than Paste to paste an EMF from the clipboard.

Although drawing programs can decompose an EMF into its component parts to allow editing the individual objects, they often get it wrong due to the complexity of the metafile format. Some applications that decompose an EMF correctly are confused by the dual EMF. For such programs, you need to export a plain EMF by setting the graphics technology to GDI.

BMP Format

BMP is a Windows bitmap format. It is accepted by a wide variety of programs but requires a lot of memory and disk space because it is not compressed. A BMP is also known as a DIB (device-independent bitmap).

If the program to which you are exporting supports PNG then PNG is a better choice.

PDF Format

PDF (Portable Document Format) is Adobe's platform-independent vector graphics format. This is the best format as long as your destination program supports it. It is more widely supported on Macintosh than on Windows.

The Igor PDF format is generated by Igor's own code rather than by the OS. As of Igor Pro 9, Igor PDF supports transparency and does a better job of font embedding. However, if you request CMYK color, the older code is used.

When Igor exports graphics as PDF on Windows, any pictures, including PDF pictures, imported into Igor are exported as bitmap images.

Blurry Images in PDF

When Igor exports an image plot, it exports the image as a single image object when possible. However, some PDF viewers, most notably Apple's, take it upon themselves to blur the pixels. To get around this, you can tell Igor to draw image pixels as individual rectangles using the **ModifyImage** interpolate keyword with a value of -1. You should do this only when necessary as the resulting PDF will be much larger.

Encapsulated PostScript (EPS) Format

Encapsulated PostScript was a widely-used, platform-independent vector graphics format consisting of PostScript commands in plain text form. It usually gives the best quality, but it works only when printed to a PostScript printer or exported to a PostScript-savvy program such as Adobe Illustrator. You should use only PostScript fonts (e.g., Helvetica).

Encapsulated PostScript was a widely-used platform-independent vector graphics format consisting of PostScript commands in plain text form. EPS is largely obsolete but still in use. It usually gives good quality, but it works only when printed to a PostScript printer or exported to a PostScript-savvy program such as Adobe Illustrator. You should use only PostScript fonts such as Helvetica. EPS does not support transparency.

Prior to Igor Pro 7, Igor embedded a screen preview in EPS files. This is no longer done because the preview was not cross-platform and caused problems with many programs.

EPS files normally use the RGB encoding to represent color but you can also use CMYK. See **Exporting Colors** on page III-105 for details.

Igor Pro exports EPS files using PostScript language level 2. This allows much better fill patterns when printing and also allows Adobe Illustrator to properly import Igor’s fill patterns. For backwards compatibility with old printers, you can force Igor to use level 1 by specifying /PLL=1 with the SavePICT operation.

If the graph or page layout that you are exporting as EPS contains a non-EPS picture imported from another program, Igor exports the picture as an image incorporated in the output EPS file.

Igor can embed TrueType fonts as outlines. See **Font Embedding** on page III-105 and **Symbols with EPS and Igor PDF** on page III-493 for details.

SVG Format

SVG (Scalable Vector Graphics) is an XML-based platform-independent 2D vector and raster graphics format developed by the World Wide Web Consortium. It is often used for displaying graphics in web pages and is a good choice for other uses if the destination program supports it. As of this writing, Microsoft Office supports SVG but few other Windows programs support it.

Platform-Independent Bitmap Formats

PNG (Portable Network Graphics) is a platform-independent bitmap format. It uses lossless compression and supports high resolution. It is a superior alternative to JPEG or GIF. Although Igor can export and import PNG images via files and via the clipboard, some programs that allow you to insert PNG files do not allow you to paste PNG images from the clipboard.

JPEG is a lossy format whose main virtue is that it is accepted by all web browsers. However, all modern web browsers support PNG so there is little reason to use JPEG for scientific graphics. Although Igor can export and import JPEG via the clipboard, most Windows programs can not paste JPEGs, but Microsoft Office can.

TIFF is an Adobe format often used for digital photographs. Igor’s implementation of TIFF export does not use compression. TIFF files normally use the RGB scheme to specify color but you can also use CMYK. See **Exporting Colors** on page III-105 for details. There is no particular reason to use TIFF over PNG unless you are exporting to a program that does not support PNG. Igor can export and import TIFF via files and via the clipboard and most graphics programs can import TIFF.

Choosing a Graphics Format

Because of the wide variety of types of graphics, destination programs, printer capabilities, operating system behaviors and user-priorities, it is not possible to give definitive guidance on choosing an export format. But here is an approach that will work in most situations.

PNG is the recommended choice for exporting image plots and Gizmo plots which are inherently bitmaps.

For vector graphics, if the destination program accepts PDF or SVG, then they are probably your best choice because of their high-quality vector graphics and platform-independence.

Encapsulated PostScript (EPS) is also a very high quality format which works well if the destination program supports it, but it does not support transparency.

If PDF, SVG and EPS are not appropriate, your next choice for vector graphics would be a high-resolution bitmap. The PNG format is preferred because it is platform-independent and supports lossless compression.

Exporting Graphics Via the Clipboard

To export a graphic from the active window via the clipboard, choose Edit→Export Graphics. This displays the Export Graphics dialog.

When you click the OK button, Igor copies the graphics for the active window to the clipboard. You can then switch to another program and do a paste.

When a graph, page layout, or Gizmo plot is active and in operate mode, choosing Edit→Copy copies to the clipboard whatever format was last used in the Export Graphics dialog. For a table, Edit→Copy copies the selected numbers to the clipboard and does not copy graphics.

When a page layout has an object selected or when the marquee is active, choosing Edit→Copy copies an Igor object in a format used internally by Igor along with an enhanced metafile and does not use the format from the Export Graphics dialog.

Although Igor can export a number of different formats, not all programs can recognize them on the clipboard. You may need to export via a file.

Exporting Graphics Via a File

To export a graphic from the active window via a file, choose File→Save Graphics. This displays the Save Graphics File.

The controls in the Format area of the dialog change to reflect options appropriate to each export format.

When you click the Do It button, Igor writes a graphics file. You can then switch to another program and import the file.

If you select _Use Dialog_ from the Path pop-up menu, Igor presents a Save File dialog in which you can specify the name and location of the saved file.

Exporting a Section of a Layout

To export a section of a page layout, use the marquee tool to identify the section and then choose Edit→Export Graphics or File→Save Graphics.

If you don’t use the marquee and the Crop to Page Contents checkbox is checked, Igor exports the area of the layout that is in use plus a small margin. If it is unchecked, Igor exports the entire page.

Exporting Colors

The EPS and TIFF graphics formats normally use the RGB scheme to specify color. Some publications require the use of CMYK instead of RGB, although the best results are obtained if the publisher does the RGB to CMYK conversion using the actual characteristics of the output device. For those publications that insist on CMYK, you can use the SavePICT /C=2 flag.

Font Embedding

This section is largely obsolete. It applies only to EPS files and PDF files generated for CMYK colors. Even then you need this info only in unusual circumstances.

You can embed TrueType fonts in EPS files and in PDF Files. This means that you can print EPS or PDF files on systems lacking the equivalent PostScript fonts. This also helps with publications that require embedded fonts.

Font embedding is always on and the only option is to not embed standard fonts. For most purposes, embedding only non-standard fonts is the best choice.

Igor embeds TrueType fonts as synthetic PostScript Type 3 fonts derived from the TrueType font outlines. Only the actual characters used are included in the fonts.

Not all fonts and font styles on your system can be embedded. Some fonts may not allow embedding and others may not be TrueType or may give errors. Be sure to test your EPS files on a local printer or by importing into Adobe Illustrator before sending them to your publisher. You can also use the “TrueType Outlines.pxp” example experiment to validate fonts for embedding. You will find this experiment file in your Igor Pro Folder in the “Examples:Testing:” folder.

For EPS, Igor determines if a font is non-standard by attempting to look up the font name in a table described in **PostScript Font Names** on page III-106 after doing any font substitution using that table. In addition, if a nonplain font style name is the same as the plain font name, then embedding is done. This means that standard PostScript fonts that do not come in italic versions (such as Symbol), will be embedded for the italic case but not for the plain case.

For PDF, non-standard fonts are those other than the basic fonts guaranteed by the PDF specification to be built-in to any PDF reader. Those fonts are Helvetica and TImes in plain, bold, italic, and bold-italic styles as well as plain versions of Symbol and Zapf Dingbats. If embedding is not used or if a font can not be embedded, fonts other than those just listed will be rendered as Helvetica and will not give the desired results.

PostScript Font Names

When generating PostScript, Igor needs to generate proper PostScript font names. This presents problems under Windows. Igor also needs to be able to substitute PostScript fonts for non-PostScript fonts. Here is a list of font names that are translated into the standards:

TrueType Name	PostScript Name
Arial	Helvetica
Arial Narrow	Helvetica-Narrow
Book Antiqua	Palatino
Bookman Old Style	Bookman
Century Gothic	AvantGarde
Century Schoolbook	NewCenturySchlbk
Courier New	Courier
Monotype Corsiva	ZapfChancery
Monotype Sorts	ZapfDingbats
Symbol	Symbol
Times New Roman	Times

Chapter

III-7

Analysis

Overview

Igor Pro is a powerful data analysis environment. The power comes from a synergistic combination of

- An extensive set of basic built-in analysis operations
- A fast and flexible waveform arithmetic capability
- Immediate feedback from graphs and tables
- Extensibility through an interactive programming environment
- Extensibility through external code modules (XOPs and XFUNCs)

Analysis tasks in Igor range from simple experiments using no programming to extensive systems tailored for specific fields. Chapter I-2, **Guided Tour of Igor Pro**, shows examples of the former. WaveMetrics' "Peak Measurement" procedure package is an example of the latter.

This chapter presents some of the basic analysis operations and discusses the more common analyses that can be derived from the basic operations. The end of the chapter shows a number of examples of using Igor's programmability for "number crunching".

Discussion of Igor Pro's more specialized analytic capabilities is in chapters that follow.

See the WaveMetrics procedures, technical notes, and sample experiments that come with Igor Pro for more examples.

Analysis of Multidimensional Waves

Many of the analysis operations in Igor Pro operate on 1D (one-dimensional) data. However, Igor Pro includes the following capabilities for analysis of multidimensional data:

- Multidimensional waveform arithmetic
- Matrix math operations
- The MatrixOp operation
- Multidimensional Fast Fourier Transform
- 2D and 3D image processing operations
- 2D and 3D interpolation operations and functions

Some of these topics are discussed in Chapter II-6, **Multidimensional Waves** and in Chapter III-11, **Image Processing**. The present chapter focuses on analysis of 1D waves.

There are many analysis operations that are designed only for 1D data. Multidimensional waves do not appear in dialogs for these operations. If you invoke them on multidimensional waves from the command line or from an Igor procedure, Igor treats the multidimensional waves as if they were 1D. For example, the Histogram operation treats a 2D wave consisting of *n* rows and *m* columns as if it were a 1D wave with *n***m* rows. In some cases (e.g., WaveStats), the operation will be useful. In other cases, it will make no sense at all.

Waveform Versus XY Data

Igor is highly adapted for dealing with waveform data. In a waveform, data values are uniformly spaced in the X dimension. This is discussed under **Waveform Model of Data** on page II-62.

If your data is uniformly spaced, you can set the spacing using the SetScale operation. This is crucial because most of the built-in analysis operations and functions need to know this to work properly.

If your data is not uniformly spaced, you can represent it using an XY pair of waves. This is discussed under **XY Model of Data** on page II-63. Some of the analysis operations and functions in Igor can *not* handle XY pairs directly. To use these, you must either make a waveform representation of the XY pair or use Igor procedures that build on the built-in routines.

Converting XY Data to a Waveform

Sometimes the best way to analyze XY data is to make a uniformly-spaced waveform representation of it and analyze that instead. Most analysis operations are easier with waveform data. Other operations, such as the FFT, can be done *only* on waveform data. Often your XY data set is nearly uniformly-spaced so a waveform version of it is a very close approximation.

In fact, often XY data imported from other programs has an X wave that is completely unnecessary in Igor because the values in the X wave are actually a simple "series" (values that define a regular intervals, such as 2.2, 2.4, 2.6, 2.8, etc), in which case conversion to a waveform is a simple matter of assigning the correct X scaling to the Y data wave, using **SetScale** (or the Change Wave Scaling dialog):

```
SetScale/P x, xWave[0], xWave[1]-xWave[0], yWave
```

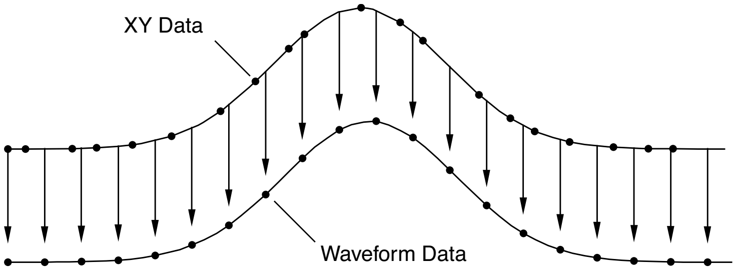
Now the X wave is superfluous and can be discarded:

```
KillWaves/Z xWave
```

The XY Pair to Waveform panel can be used to set the Y wave's X scaling when it detects that the X wave contains series data. See **Using the XY Pair to Waveform Panel** on page III-109.

If your X wave is not a series, then to create a waveform representation of XY data you need to use interpolation. To create a waveform representation of XY data you need to do interpolation. Interpolation creates a waveform from an XY pair by sampling the XY pair at uniform intervals.

The diagram below shows how the XY pair defining the upper curve is interpolated to compute the uniformly-spaced waveform defining the lower curve. Each arrow indicates an interpolated waveform value:



Igor provides three tools for doing this interpolation: The XY Pair to Waveform panel, the built-in **interp** function and the **Interpolate2** operation. To illustrate these tools we need some sample XY data. The following commands make sample data and display it in a graph:

```
Make/N=100 xData = .01*x + gnoise(.01)
Make/N=100 yData = 1.5 + 5*exp(-((xData-.5)/.1)^2)
Display yData vs xData
```

This creates a Gaussian shape. The x wave in our XY pair has some noise in it so the data is not uniformly spaced in the X dimension.

The x data goes roughly from 0 to 1.0 but, because our x data has some noise, it may not be monotonic. This means that, as we go from one point to the next, the x data usually increases but at some points may decrease. We can fix this by sorting the data.

```
Sort xData, xData, yData
```

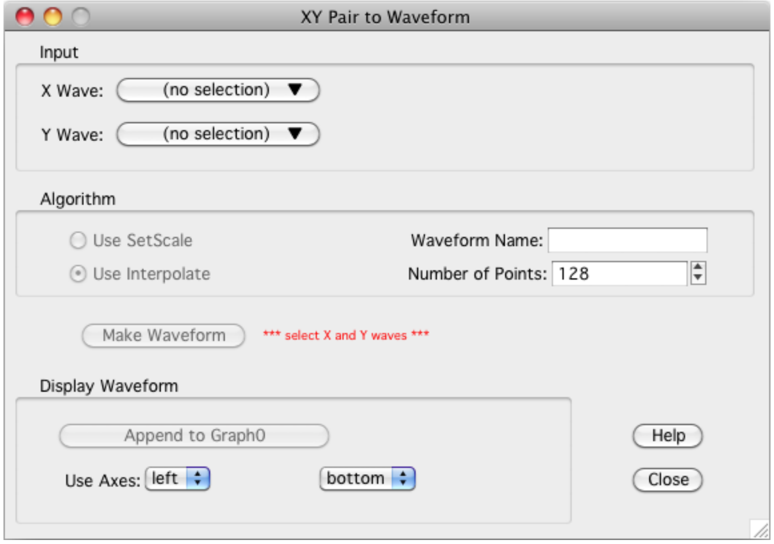
This command uses the xData wave as the sort key and sorts both xData and yData so that xData always increases as we go from one point to the next.

Using the XY Pair to Waveform Panel

The XY Pair to Waveform panel creates a waveform from XY data using the **SetScale** or Interpolate2 operations, based on an automatic analysis of the X wave's data.

The required steps are:

1. Select XY Pair to Waveform from Igor's Data→Packages submenu.
   The panel is displayed:



XY Pair to Waveform

Input

X Wave: (no selection)

Y Wave: (no selection)

Algorithm

○ Use SetScale

● Use Interpolate

Waveform Name:

Number of Points: 128

*** select X and Y waves ***

Display Waveform

Use Axes: left bottom

2. Select the X and Y waves (xData and yData) in the popup menus. When this example's xData wave is analyzed it is found to be "not regularly spaced (slope error avg= 0.52...)", which means that SetScale is not appropriate for converting yData into a waveform.
3. Use Interpolate is selected here, so you need a waveform name for the output. Enter any valid wave name.
4. Set the number of output points. Using a number roughly the same as the length of the input waves is a good first attempt. You can choose a larger number later if the fidelity to the original is insufficient. A good number depends on how uneven the X values are - use more points for more unevenness.
5. Click Make Waveform.
6. To compare the XY representation of the data with the waveform representation, append the waveform to a graph displaying the XY pair. Make that graph the top graph, then click the "Append to <Name of Graph>" button.
7. You can revise the Number of Points and click Make Waveform to overwrite the previously created waveform in-place.

Using the Interp Function

We can use the **interp** function (see page V-458) to create a waveform version of our Gaussian. The required steps are:

1. Make a new wave to contain the waveform representation.
2. Use the **SetScale** operation to define the range of X values in the waveform.
3. Use the interp function to set the data values of the waveform based on the XY data.

Here are the commands:

```
Duplicate yData, wData
SetScale/I x 0, 1, wData
wData = interp(x, xData, yData)
```

To compare the waveform representation to the XY representation, we append the waveform to the graph.

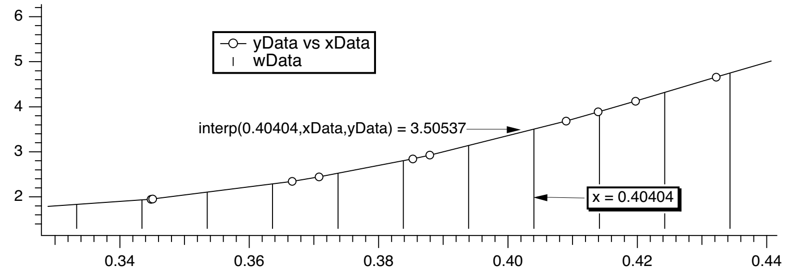
```
AppendToGraph wData
```

Let’s take a closer look at what these commands are doing.

First, we cloned yData and created a new wave, wData. Since we used Duplicate, wData will have the same number of points as yData. We could have made a waveform with a different number of points. To do this, we would use the Make operation instead of Duplicate.

The SetScale operation sets the X scaling of the wData waveform. In this example, we are setting the X values of wData to go from 0 up to and including 1.0. This means that our waveform representation will contain 100 values at uniform intervals in the X dimension from 0 to 1.0.

The last step uses a waveform assignment to set the data values of wData. This assignment evaluates the right-hand expression once for each point in wData. For each evaluation, x takes on a different value from 0 to 1.0. The interp function returns the value of the curve yData versus xData at x. For instance, x=.40404 (point number 40 of wData) falls between two points in the XY curve. The interp function linearly interpolates between those values to estimate a data value of 3.50537:



We can wrap these calculations up into an Igor procedure that can create a waveform version of any XY pair.

```
Function XYToWave1(xWave, yWave, wWaveName, numPoints)
   Wave/D xWave              // X wave in the XY pair
   Wave/D yWave              // Y wave in the XY pair
   String wWaveName          // Name to use for new waveform wave
   Variable numPoints        // Number of points for waveform

   Make/O/N=(numPoints) $wWaveName  // Make waveform.
   Wave wWave= $wWaveName
   WaveStats/Q xWave                // Find range of x coords
   SetScale/I x V_min, V_max, wWave // Set X scaling for wave
   wWave = interp(x, xWave, yWave)  // Do the interpolation
End
```

This function uses the **WaveStats** operation to find the X range of the XY pair. WaveStats creates the variables V_min and V_max (among others). See **Accessing Variables Used by Igor Operations** on page IV-123 for details.

The function makes the assumption that the input waves are already sorted. We left the sort step out because the sorting would be a side-effect and we prefer that procedures not have nonobvious side effects.

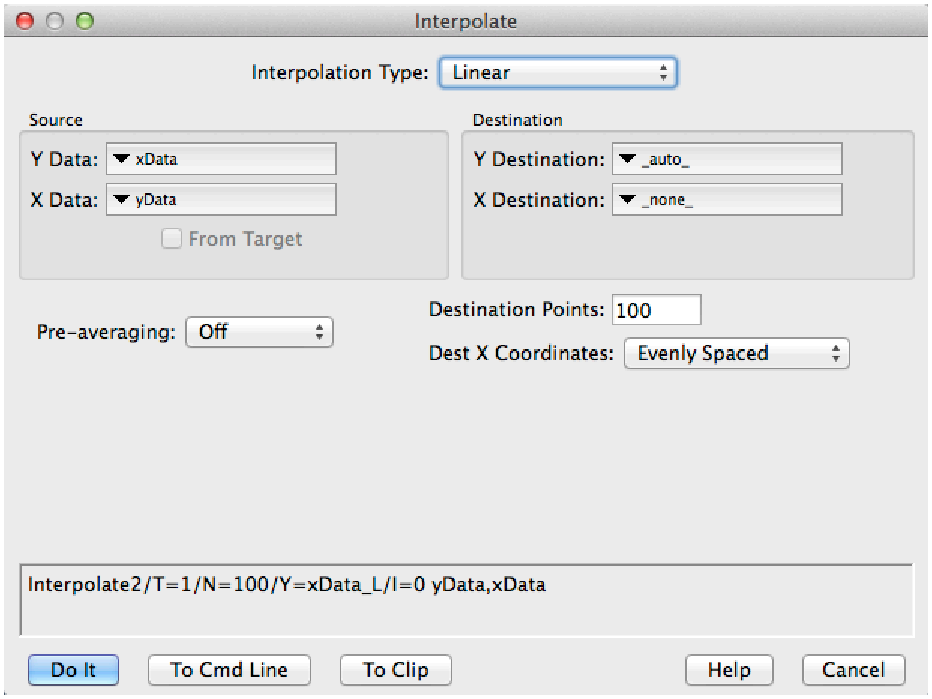
To use the WaveMetrics-supplied XYToWave1 function, include the “XY Pair To Waveform” procedure file. See **The Include Statement** on page IV-166 for instructions on including a procedure file.

If you have blanks (NaNs) in your input data, the interp function will give you blanks in your output waveform as well. The **Interpolate2** operation, discussed in the next section, interpolates across gaps in data and does not produce blanks in the output.

Using the Interpolate2 Operation

The **Interpolate2** operation provides not only linear but also cubic and smoothing spline interpolation. Furthermore, it does not require the input to be sorted and can automatically make the destination waveform and set its X scaling. It also has a dialog that makes it easy to use interactively.

To use it on our sample XY data, choose Analysis→Interpolate and set up the dialog as shown:



```
Interpolate2/T=1/N=100/Y=xData_L/I=0 yData,xData
```

Choosing "_auto_" for Y Destination auto-names the destination wave by appending "_L" to the name of the input "Y data" wave. Choosing "_none_" as the "X destination" creates a waveform from the input XY pair rather than a new XY pair.

Here is a rewrite of the XYToWave1 function that uses the Interpolate2 operation rather than the interp function.

```
Function XYToWave2(xWave, yWave, wWaveName, numPoints)
   Wave xWave                    // X wave in the XY pair
   Wave yWave                    // Y wave in the XY pair
   String wWaveName              // Name to use for new waveform wave
   Variable numPoints            // Number of points for waveform

   Interpolate2/T=1/N=(numPoints)/E=2/Y=$wWaveName xWave, yWave
End
```

Blanks in the input data are ignored.

For details on Interpolate2, see **The Interpolate2 Operation** on page III-115.

Dealing with Missing Values

A missing value is represented in Igor by the value NaN which means "Not a Number". A missing value is also called a "blank", because it appears as a blank cell in a table.

When a NaN is combined arithmetically with any value, the result is NaN. To see this, execute the command:

```
Print 3+NaN, NaN/5, sin(NaN)
```

By definition, a NaN is not equal to anything. Consequently, the condition in this statement:

```
if (myValue == NaN)
```

is always false.

The workaround is to use the **numtype** function:

```
if (NumType(myValue) == 2)          // Is it a NaN?
```

See also **NaNs, INFs and Missing Values** on page II-83 for more about how NaN values.

Some routines deal with missing values by ignoring them. The **CurveFit** operation (see page V-124) is one example. Others may produce unexpected results in the presence of missing values. Examples are the **FFT** operation and the **area** and **mean** functions.

Here are some strategies for dealing with missing values.

Replace the Missing Values With Another Value

You can replace NaNs in a wave with this statement:

```
wave0 = NumType(wave0)==2 ? 0:wave0    // Replace NaNs with zero
```

If you're not familiar with the :? operator, see **Operators** on page IV-6.

For multi-dimensional waves you can replace NaNs using **MatrixOp**. For example:

```
Make/O/N=(3,3) matNaNTest = p + 10*q
Edit matNaNTest
matNaNTest[0][0] = NaN; matNaNTest[1][1] = NaN; matNaNTest[2][2] = NaN
MatrixOp/O matNaNTest=ReplaceNaNs(matNaNTest,0)    // Replace NaNs with 0
```

Remove the Missing Values

For 1D waves you can remove NaNs using **WaveTransform** `zapNaNs`. For example:

```
Make/N=5 NaNTest = p
Edit NaNTest
NaNTest[1] = NaN; NaNTest[4] = NaN
WaveTransform zapNaNs, NaNTest
```

There is no built-in operation to remove NaNs from an XY pair if the NaN appears in either the X or Y wave. You can do this, however, using the RemoveNaNsXY procedure in the "Remove Points" WaveMetrics procedure file which you can access through Help→Windows→WM Procedures Index.

There is no operation to remove NaNs from multi-dimensional waves as this would require removing the entire row and entire column where each NaN appeared.

Work Around Gaps in Data

Many analysis routines can work on a subrange of data. In many cases you can just avoid the regions of data that contain missing values. In other cases you can extract a subset of your data, work with it and then perhaps put the modified data back into the original wave.

Here is an example of extract-modify-replace (even though Smooth properly accounts for NaNs):

```
Make/N=100 data1= sin(P/8)+gnoise(.05); data1[50]= NaN
Display data1
Duplicate/R=[0,49] data1,tmpdata1      // start work on first set
Smooth 5,tmpdata1
data1[0,49]= tmpdata1[P]               // put modified data back
Duplicate/O/R=[51,] data1,tmpdata1     // start work on 2nd set
Smooth 5,tmpdata1
data1[51,]= tmpdata1[P-51]
KillWaves tmpdata1
```

Replace Missing Data with Interpolated Values

You can replace NaN data values prior to performing operations that do not take kindly to NaNs by replacing them with smoothed or interpolated values using the **Smooth** operation (page V-878), the **Loess** operation (page V-515), or **The Interpolate2 Operation**.

Replace Missing Data Using the Interpolate2 Operation

By using the same number of points for the destination as you have source points, you can replace NaNs without modifying the other data.

If you have waveform data, simply duplicate your data and perform linear interpolation using the same number of points as your data. For example, assuming 100 data points:

```
Duplicate data1,data1a
Interpolate/T=1/N=100/Y=data1a data1
```

If you have XY data, the Interpolate2 operation has the ability to include the input x values in the output X wave. For example:

```
Duplicate data1, yData1, xData1
xData1 = x
Display yData1 vs xData1
Interpolate2/T=1/N=100/I/Y=yData1a/X=xData1a xData1,yData1
```

If, after performing an operation on your data, you wish to put the modified data back in the source wave while maintaining the original missing values you can use a wave assignment similar to this:

```
yData1 = (numtype(yData1) == 0) ? yData1 : yData1a
```

This technique can also be applied using interpolated results generated by the **Smooth** operation (page V-878) or the **Loess** operation (page V-515).

Replace Missing Data Using Median Smoothing

You can use the Smooth dialog to replace each NaN with the median of surrounding values.

Select the Median smoothing algorithm, select "NaNs" from the Replace popup, and choose "Median" for the "with:" radio button. Enter the number of surrounding points used to compute the median (an odd number is best).

You can choose to overwrite the NaNs or create a new waveform with the result. The Smooth dialog produces commands like this:

```
Duplicate/O data1,data1_smth;DelayUpdate
Smooth/M=(NaN) 5, data1_smth
```

Interpolation

Igor Pro has a number of interpolation tools that are designed for different applications. We summarize these in the table below.

Data	Operation/Function	Interpolation Method
1D waves	wave assignment, e.g., `val=wave(x)`	Linear
1D waves	**Smooth**	Running median, average, binomial, Savitsky-Golay
1D XY waves	**interp**()	Linear
1D single or XY waves	**The Interpolate2 Operation**	Linear, cubic spline, smoothing spline
1D or 2D single or XY	**Loess**	Locally-weighted regression
Triplet XYZ waves	**ImageInterpolate**	Voronoi
1D X, Y, Z waves	Data→Packages→XYZ to Matrix	Voronoi
1D X, Y, Z waves	**Loess**	Locally-weighted regression
2D waves	**ImageInterpolate**	Bilinear, splines, Kriging, Voronoi

Data	Operation/Function	Interpolation Method
2D waves	**Interp2D**()	Bilinear
2D waves (points on the surface of a sphere)	**SphericalInterpolate**	Voronoi
3D waves	**Interp3D**(), **Interp3DPath**, ImageTransform **extractSurface**	Trilinear
3D scatter data	**Interpolate3D**	Barycentric

All the interpolation methods in this table consist of two common steps. The first step involves the identification of data points that are nearest to the interpolation location and the second step is the computation of the interpolated value using the neighboring values and their relative proximity. You can find the specific details in the documentation of the individual operation or function.

The Interpolate2 Operation

The **Interpolate2** operation performs linear, cubic spline and smoothing cubic spline interpolation on 1D waveform and XY data. The cubic spline interpolation is based on a routine in "Numerical Recipes in C". The smoothing spline is based on "Smoothing by Spline Functions", Christian H. Reinsch, *Numerische Mathematic* 10, 177-183 (1967).

Prior to Igor7, Interpolate2 was implemented as part of the Interpolate XOP. It is now built-in.

The Interpolate XOP also implemented an older operation named Interpolate which used slightly different syntax. If you are using the Interpolate operation, we recommend that you convert to using Interpolate2.

The main use for linear interpolation is to convert an XY pair of waves into a single wave containing Y values at evenly spaced X values so that you can use Igor operations, like **FFT**, which require evenly spaced data.

Cubic spline interpolation is most useful for putting a pleasingly smooth curve through arbitrary XY data. The resulting curve may contain features that have nothing to do with the original data so you should be wary of using the cubic spline for analytic rather than esthetic purposes.

Both linear and cubic spline interpolation are constrained to put the output curve through all of the input points and thus work best with a small number of input points. The smoothing spline does not have this constraint and thus works well with large, noisy data sets.

The Interpolate2 operation has a feature called "pre-averaging" which can be used when you have a large number of input points. Pre-averaging was added to Interpolate2 as a way to put a cubic spline through a large, noisy data set before it supported the smoothing spline. We now recommend that you use the smoothing spline instead of pre-averaging.

The Smooth Curve Through Noise example experiment illustrates spline interpolation. Choose File→Example Experiments→Feature Demos→Smooth Curve Through Noise.

Spline Interpolation Example

Before going into a complete discussion of Interpolate2, let's look at a simple example first.

First, make some sample XY data using the following commands:

```
Make/N=10 xData, yData                          // Make source data
xData = p; yData = 3 + 4*xData + gnoise(2)      // Create sample data
Display yData vs xData                          // Make a graph
Modify mode=2, lsize=3                          // Display source data as dots
```

Now, choose Analysis→Interpolate to invoke the Interpolate dialog.

Set Interpolation Type to Cubic Spline.

Choose yData from the Y Data pop-up menu.

Choose xData from the X Data pop-up menu.

Choose _auto_ from the Y Destination pop-up menu.

Choose _none_ from the X Destination pop-up menu.

Enter 200 in the Destination Points box.

Choose Off from the Pre-averaging pop-up menu.

Choose Evenly Spaced from the Dest X Coords pop-up menu.

Click Natural in the End Points section.

Notice that the dialog has generated the following command:

```
Interpolate2/T=2/N=200/E=2/Y=yData_CS xData, yData
```

This says that Interpolate2 will use yData as the Y source wave, xData as the X source wave and that it will create yData_CS as the destination wave. _CS means "cubic spline".

Click the Do It button to do the interpolation.

Now add the yData_CS destination wave to the graph using the command:

```
AppendToGraph yData_CS; Modify rgb(yData_CS)=(0,0,65535)
```

Now let's try this with a larger number of input data points. Execute:

```
Redimension/N=500 xData, yData
xData = p/50; yData = 10*sin(xData) + gnoise(1.0)      // Create sample data
Modify lsize(yData)=1                                  // Smaller dots
```

Now choose Analysis→Interpolate to invoke the Interpolate dialog. All the settings should be as you left them from the preceding exercise. Click the Do It button.

Notice that the resulting cubic spline attempts to go through all of the input data points. This is usually not what we want.

Now, choose Analysis→Interpolate again and make the following changes:

Choose Smoothing Spline from the Interpolation Type pop-up menu. Notice the command generated now references a wave named yData_SS. This will be the output wave.

Enter 1.0 for the smoothing factor. This is usually a good value to start from.

In the Standard Deviation section, click the Constant radio button and enter 1.0 as the standard deviation value. 1.0 is correct because we know that our data has noise with a standard deviation of 1.0, as a result of the "gnoise(1.0)" term above.

Click the Do It button to do the interpolation. Append the yData_SS destination wave to the graph using the command:

```
AppendToGraph yData_SS; Modify rgb(yData_SS)=(0,0,0)
```

Notice that the smoothing spline adds a pleasing curve through the large, noisy data set. If necessary, enlarge the graph window so you can see this.

You can tweak the smoothing spline using either the smoothing factor parameter or the standard deviation parameter. If you are unsure of the standard deviation of the noise, leave the smoothing factor set to 1.0 and try different standard deviation values. It is usually not too hard to find a reasonable value.

The Interpolate Dialog

Choosing Analysis→Interpolate summons the Interpolate dialog from which you can choose the desired type of interpolation, the source wave or waves, the destination wave or waves, and the number of points in the destination waves. This dialog generates an **Interpolate2** command which you can execute, copy to the clipboard or copy to the command line.

From the Interpolation Type pop-up menu, choose Linear or Cubic Spline or Smoothing Spline. Cubic spline is good for a small input data set. Smoothing spline is good for a large, noisy input data set.

If you choose Cubic Spline, a Pre-averaging pop-up menu appear. The pre-averaging feature is largely no longer needed and is not recommended. Use the smoothing spline instead of the cubic spline with pre-averaging.

If you choose smoothing spline, a Smoothing Factor item and Standard Deviation controls appear. Usually it is best to set the smoothing factor to 1.0 and use the constant mode for setting the standard deviation. You then need to enter an estimate for the standard deviation of the noise in your Y data. Then try different values for the standard deviation until you get a satisfactory smooth spline through your data. See **Smoothing Spline Parameters** on page III-119 for further details.

From the Y Data and X Data pop-up menus, choose the source waves that define the data through which you want to interpolate. If you choose a wave from the X Data pop-up, Interpolate2 uses the XY curve defined by the contents of the Y data wave versus the contents of the X data wave as the source data. If you choose _calculated_ from the X Data pop-up, it uses the X and Y values of the Y data wave as the source data.

If you click the From Target checkbox, the source and destination pop-up menus show only waves in the target graph or table.

The X and Y data waves must have the same number of points. They do not need to have the same data type.

The X and Y source data does not need to be sorted before using Interpolate2. If necessary, Interpolate2 sorts a copy of the input data before doing the interpolation.

NaNs (missing values) and INFs (infinite values) in the source data are ignored. Any point whose X or Y value is NaN or INF is treated as if it did not exist.

Enter the number of points you want in the destination waves in the Destination Points box. 200 points is usually good.

From the Y Destination and X Destination pop-up menus, choose the waves to contain the result of the interpolation. For most cases, choose _auto_ for the Y destination wave and _none_ for the X destination wave. This gives you an output waveform. Other options, useful for less common applications, are described in the following paragraphs.

If you choose _auto_ from the Y Destination pop-up, Interpolate2 puts the Y output data into a wave whose name is derived by adding a suffix to the name of the Y data wave. The suffix is "_L" for linear interpolation, "_CS" for cubic spline interpolation and "_SS" for smoothing spline interpolation. For example, if the Y data wave is called "yData", then the default Y destination wave will be called "yData_L", "yData_CS" or "yData_SS".

If you choose _none_ for the X destination wave, Interpolate2 puts the Y output data in the Y destination wave and sets the X scaling of the Y destination wave to represent the X output data.

If you choose _auto_ from the X Destination pop-up, Interpolate2 puts the X output data into a wave whose name is derived by adding the appropriate suffix to the name of the X data wave. If the X data wave is "xData" then the X destination wave will be "xData_L", "xData_CS" or "xData_SS". If there is no X data wave then the X destination wave name is derived by adding the letter "x" to the name of the Y destination wave. For example, if the Y destination is "yData_CS" then the X destination wave will be "yData_CSx".

For both the X and the Y destination waves, if the wave already exists, Interpolate2 overwrites it. If it does not already exist, Interpolate2 creates it.

The destination waves will be double-precision unless they already exist when Interpolate2 is invoked. In this case, Interpolate2 leaves single-precision destination waves as single-precision. For any other precision, Interpolate2 changes the destination wave to double-precision.

The Dest X Coords pop-up menu gives you control over the X locations at which the interpolation is done. Usually you should choose Evenly Spaced. This generates interpolated values at even intervals over the range of X input values.

The Evenly Spaced Plus Input X Coords setting is the same as Evenly Spaced except that Interpolate2 makes sure that the output X values include all of the input X values. This is usually not necessary. This mode is not available if you choose _none_ for your X destination wave.

The Log Spaced setting makes the output evenly spaced on a log axis. This mode ignores any non-positive values in your input X data. It is not available if you choose _none_ for your X destination wave. See **Interpolating Exponential Data** on page III-118 for an alternative.

The From Dest Wave setting takes the output X values from the X coordinates of the destination wave. The Destination Points setting is ignored. You could use this, for example, to get a spline through a subset of your input data. You must create your destination waves before doing the interpolation for this mode. If your destination is a waveform, use the SetScale operation to define the X values of the waveform. Interpolate2 will calculate its output at these X values. If your destination is an XY pair, set the values of the X destination wave. Interpolate2 will create a sorted version of these values and will then calculate its output at these values. If the X destination wave was originally reverse-sorted, Interpolate2 will reverse the output.

The End Points radio buttons apply only to the cubic spline. They control the destination waves in the first and last intervals of the source wave. Natural forces the second derivative of the spline to zero at the first and last points of the destination waves. Match 1st Derivative forces the slope of the spline to match the straight lines drawn between the first and second input points, and between the last and next-to-last input points. In most cases it doesn't much matter which of these alternatives you use.

Interpolating Exponential Data

It is common to plot data that spans orders of magnitude, such as data arising from exponential processes, versus log axes. To create an interpolated data set from such data, it is often best to interpolate the log of the original data rather than the original data itself.

The Interpolate2 Log Demo experiment demonstrates how to do such interpolation. To open the demo, choose Files→Example Experiments→Feature Demos→Interpolate2 Log Demo.

Smoothing Spline Algorithm

The smoothing spline algorithm is based on "Smoothing by Spline Functions", Christian H. Reinsch, *Numerische Mathematik* 10. It minimizes

$$\int_{x_0}^{x_n} g''(x)^2 \, dx,$$

among all functions g(x) such that

$$\sum_{i=0}^{n} \left(\frac{g(x_i) - y_i}{\sigma_i} \right)^2 \leq S,$$

where $g(x_i)$ is the value of the smooth spline at a given point, y_i is the Y data at that point, σ_i is the standard deviation of that point, and S is the smoothing factor.

Smoothing Spline Parameters

The smoothing spline operation requires a standard deviation parameter and a smoothing factor parameter. The standard deviation parameter should be a good estimate of the standard deviation of the noise in your Y data. The smoothing factor should nominally be close to 1.0, assuming that you have an accurate standard deviation estimate.

Using the Standard Deviation section of the Interpolate2 dialog, you can choose one of three options for the standard deviation parameter: None, Constant, From Wave.

If you choose None, Interpolate2 uses an arbitrary standard deviation estimate of 0.05 times the amplitude of your Y data. You can then play with the smoothing factor parameter until you get a pleasing smooth spline. Start with a smoothing factor of 1.0. This method is not recommended.

If you choose Constant, you can then enter your estimate for the standard deviation of the noise and Interpolate2 uses this value as the standard deviation of each point in your Y data. If your estimate is good, then a smoothing factor around 1.0 will give you a nice smooth curve through your data. If your initial attempt is not quite right, you should leave the smoothing factor at 1.0 and try another estimate for the standard deviation. For most types of data, this is the preferred method.

If you choose From Wave then Interpolate2 expects that each point in the specified wave contains the estimated standard deviation for the corresponding point in the Y data. You should use this method if you have an appropriate wave.

Interpolate2's Pre-averaging Feature

A linear or cubic spline interpolation goes through all of the input data points. If you have a large, noisy data set, this is probably not what you want. Instead, use the smoothing spline.

Before Interpolate2 had a smoothing spline, we recommended that you use the cubic spline to interpolate through a decimated version of your input data. The pre-averaging feature was designed to make this easy.

Because Interpolate2 now supports the smoothing spline, the pre-averaging feature is no longer necessary. However, we still support it for backward compatibility.

When you turn pre-averaging on, Interpolate2 creates a temporary copy of your input data and reduces it by decimation to a smaller number of points, called nodes. Interpolate2 then usually adds nodes at the very start and very end of the data. Finally, it does an interpolation through these nodes.

Identical Or Nearly Identical X Values

This section discusses a degenerate case that is of no concern to most users.

Input data that contains two points with identical X values can cause interpolation algorithms to produce unexpected results. To avoid this, if Interpolate2 encounters two or more input data points with nearly identical X values, it averages them into one value before doing the interpolation. This behavior is separate from the pre-averaging feature. This is done for the cubic and smoothing splines except when the Dest X Coords mode is is Log Spaced or From Dest Wave. It is not done for linear interpolation.

Two points are considered nearly identical in X if the difference in X between them (dx) is less than 0.01 times the nominal dx. The nominal dx is computed as the X span of the input data divided by the number of input data points.

Destination X Coordinates from Destination Wave

This mode, which we call "X From Dest" mode for short, takes effect if you choose From Dest Wave from the Dest X Coords pop-up menu in the Interpolate2 dialog or use the Interpolate2 /I=3 flag. In this mode the number of output points is determined by the destination wave and the /N flag is ignored.

In X From Dest mode, the points at which the interpolation is done are determined by the destination wave. The destination may be a waveform, in which case the interpolation is done at its X values. Alternatively the destination may be an XY pair in which case the interpolation is done at the data values stored in the X destination wave.

Here is an example using a waveform as the destination:

```
Make /O /N=20 wave0                     // Generate source data
SetScale x 0, 2*PI, wave0
wave0 = sin(x) + gnoise(.1)
Display wave0
ModifyGraph mode=3
Make /O /N=1000 dest0                   // Generate dest waveform
SetScale x 0, 2*PI, dest0
AppendToGraph dest0
ModifyGraph rgb(dest0)=(0,0,65535)
Interpolate2 /T=2 /I=3 /Y=dest0 wave0     // Do cubic spline interpolation
```

If your destination is an XY pair, Interpolate2 creates a sorted version of these values and then calculates its output at the X destination wave's values. If the X destination wave was originally reverse-sorted, as determined by examining its first and last values, Interpolate2 reverses the output after doing the interpolation to restore the original ordering.

In X From Dest mode with an XY pair as the destination, the X wave may contain NaNs. In this case, Interpolate2 creates an internal copy of the XY pair, removes the points where the X destination is NaN, does the interpolation, and copies the results to the destination XY pair. During this last step, Interpolate2 restores the NaNs to their original locations in the X destination wave. Here is an example of an XY destination with a NaN in the X destination wave:

```
Make/O xData={1,2,3,4,5}, yData={1,2,3,4,5}           // Generate source data
Display yData vs xData
Make/O xDest={1,2,NaN,4,5}, yDest={0,0,0,0,0}         // Generate dest XY pair
ModifyGraph mode=3,marker=19
AppendToGraph yDest vs xDest
ModifyGraph rgb(yDest)=(0,0,65535)
Interpolate2 /T=1 /I=3 /Y=yDest /X=xDest xData, yData // Do interpolation
```

Differentiation and Integration

The **Differentiate** operation (see page V-158) and **Integrate** operation (see page V-446) provide a number of algorithms for operation on one-dimensional waveform and XY data. These operations can either replace the original data or create a new wave with the results. The easiest way to use these operations is via dialogs available from the Analysis menu.

For most applications, trapezoidal integration and central differences differentiation are appropriate methods. However, when operating on XY data, the different algorithms have different requirements for the number of points in the X wave. If your X wave does not show up in the dialog, try choosing a different algorithm or click the help button to see what the requirements are.

When operating on waveform data, X scaling is taken into account; you can turn this off using the /P flag. You can use the **SetScale** operation (see page V-853) to define the X scaling of your Y data wave.

Although these operations work along just one dimension, they can be targeted to operate along rows or columns of a matrix (or even higher dimensions) using the /DIM flag.

The Integrate operation replaces or creates a wave with the numerical integral. For finding the area under a curve, see **Areas and Means** on page III-120.

Areas and Means

You can compute the area and mean value of a wave in several ways using Igor.

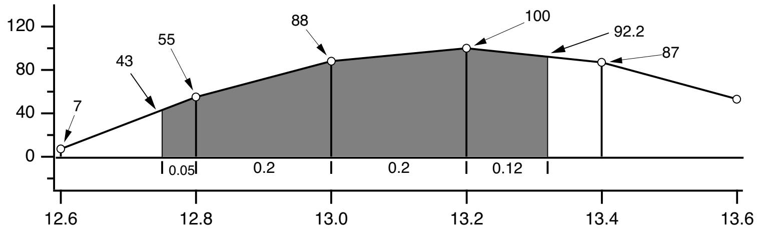
Perhaps the simplest way to compute a mean value is with the Wave Stats dialog in the Analysis menu. The dialog is pictured under **Wave Statistics** on page III-122. You select the wave, type in the X range (or use the current cursor positions), click Do It, and Igor prints several statistical results to the history area. Among them is V_avg, which is the average, or mean value. This is the same value that is returned by the **mean** function (see page V-586), which is faster because it doesn't compute any other statistics. The mean function returns NaN if any data within the specified range is NaN. The **WaveStats** operation, on the other hand, ignores such missing values.

WaveStats and the **mean** function use the same method for computing the mean: find the waveform values within the given X range, sum them together, and divide by the number of values. The X range serves only to select the values to combine. The range is rounded to the nearest point numbers.

If you consider your data to describe discrete values, such as a count of events, then you should use either WaveStats or the mean function to compute the average value. You can most easily compute the total number of events, which is an area of sorts, using the **sum** function (see page V-1007). It can also be done easily by multiplying the WaveStats outputs V_avg and V_npnts.

If your data is a sampled representation of a continuous process such as a sampled audio signal, you should use the **faverage** function to compute the mean, and the area function to compute the area. These two functions use the same linear interpolation scheme as does trapezoidal integration to estimate the waveform values between data points. The X range is *not* rounded to the nearest point. Partial X intervals are included in the calculation through this linear interpolation.

The diagram below shows the calculations each function performs for the data shown. The two values 43 and 92.2 are linear interpolation estimates.



Comparison of area, faverage and mean functions over interval (12.75,13.32)

`WaveStats/R=(12.75,13.32) wave`		
`V_avg`	= (55+88+100+87)/4 = 82.5	
`mean(wave,12.75,13.32)`	= (55+88+100+87)/4 = 82.5	
`area(wave,12.75,13.32)`	= 0.05 · (43+55) / 2	first trapezoid
	+ 0.20 · (55+88) / 2	second trapezoid
	+ 0.20 · (88+100) / 2	third trapezoid
	+ 0.12 · (100+92.2) / 2	fourth trapezoid
	= 47.082	
`faverage(wave,12.75,13.32)`	= area(wave,12.75,13.32) / (13.32-12.75)	
	= 47.082/0.57 = 82.6	

Note that only the area function is affected by the X scaling of the wave. faverage eliminates the effect of X scaling by dividing the area by the same X range that area multiplied by.

One problem with these functions is that they can not be used if the given range of data has missing values (NaNs). See **Dealing with Missing Values** on page III-112 for details.

X Ranges and the Mean, faverage, and area Functions

The X range input for the mean, faverage and area functions are optional. Thus, to include the entire wave you don’t have to specify the range:

```
Make/N=10 wave=2; Edit wave.xy    // X ranges from 0 to 9
Print area(wave)                  // entire X range, and no more
18
```

Sometimes, in programming, it is not convenient to determine whether a range is beyond the ends of a wave. Fortunately, these functions also accept X ranges that go beyond the ends of the wave.

```
Print area(wave, 0, 9)            // entire X range, and no more
18
```

You can use expressions that evaluate to a range beyond the ends of the wave:

```
Print leftx(wave),rightx(wave)
0  10
Print area(wave,leftx(wave),rightx(wave))      // entire X range, and more
18
```

or even an X range of ±×:

```
Print area(wave, -Inf, Inf)  // entire X range of the universe
18
```

Finding the Mean of Segments of a Wave

Under **Analysis Programming** on page III-170 is a function that finds the mean of segments of a wave where you specify the length of the segments. It creates a new wave to contain the means for each segment.

Area for XY Data

To compute the area of a region of data contained in an XY pair of waves, use the **areaXY** function (see page V-41). There is also an XY version of the faverage function; see **faverageXY** on page V-218.

In addition you can use the AreaXYBetweenCursors WaveMetrics procedure file which contains the AreaXYBetweenCursors and AreaXYBetweenCursorsLessBase procedures. For instructions on loading the procedure file, see **WaveMetrics Procedures Folder** on page II-32. Use the **Info Panel and Cursors** to delimit the X range over which to compute the area. AreaXYBetweenCursorsLessBase removes a simple trapezoidal baseline - the straight line between the cursors.

Wave Statistics

The **WaveStats** operation (see page V-1082) computes various descriptive statistics relating to a wave and prints them in the history area of the command window. It also stores the statistics in a series of special variables or in a wave so you can access them from a procedure.

The statistics printed and the corresponding special variables are:

Variable	**Meaning**
V_npnts	Number of points in range excluding points whose value is NaN or INF.
V_numNaNs	Number of NaNs.
V_numINFs	Number of INFs.
V_avg	Average of data values.

Variable	Meaning
V_sum	Sum of data values.
V_sdev	Standard deviation of data values, $\sigma = \sqrt{\frac{1}{V\_npnts - 1}\sum (Y_i - V\_avg)^2}$ “Variance” is V_sdev^2.
V_sem	Standard error of the mean $sem = \frac{\sigma}{\sqrt{V\_npnts}}$
V_rms	RMS (Root Mean Square) of Y values $= \sqrt{\left(\frac{1}{V\_npnts}\sum Y_i^2\right)}$
V_adev	Average deviation $= \frac{1}{V\_npnts} \sum_{i=0}^{V\_npnts-1} \lvert x_i - \bar{x} \rvert$
V_skew	Skewness $= \frac{1}{V\_npnts} \sum_{i=0}^{V\_npnts-1} \left[\frac{x_i - \bar{x}}{\sigma}\right]^3$
V_kurt	Kurtosis $= \frac{1}{V\_npnts} \sum_{i=0}^{V\_npnts-1} \left[\frac{x_i - \bar{x}}{\sigma}\right]^4 - 3$
V_minloc	X location of minimum data value.
V_min	Minimum data value.
V_maxloc	X location of maximum data value.
V_max	Maximum data value.
`V_minRowLoc`	Row containing minimum data value.
`V_maxRowLoc`	Row containing maximum data value.
`V_minColLoc`	Column containing minimum data value (2D or higher waves).
`V_maxColLoc`	Column containing maximum data value (2D or higher waves).
`V_minLayerLoc`	Layer containing minimum data value (3D or higher waves).
`V_maxLayerLoc`	Layer containing maximum data value (3D or higher waves).
`V_minChunkLoc`	Chunk containing minimum v value (4D waves only).
`V_maxChunkLoc`	Chunk containing maximum data value (4D waves only).
`V_startRow`	The unscaled index of the first row included in caculating statistics.
`V_endRow`	The unscaled index of the last row included in caculating statistics.
`V_startCol`	The unscaled index of the first column included in calculating statistics. Set only when /RMD is used.

Variable	Meaning
`V_endCol`	The unscaled index of the last column included in calculating statistics. Set only when /RMD is used.
`V_startLayer`	The unscaled index of the first layer included in calculating statistics. Set only when /RMD is used.
`V_endLayer`	The unscaled index of the last layer included in calculating statistics. Set only when /RMD is used.
`V_startChunk`	The unscaled index of the first chunk included in calculating statistics. Set only when /RMD is used.
`V_endChunk`	The unscaled index of the last chunk included in calculating statistics. Set only when /RMD is used.

To use the WaveStats operation, choose Wave Stats from the Statistics menu.

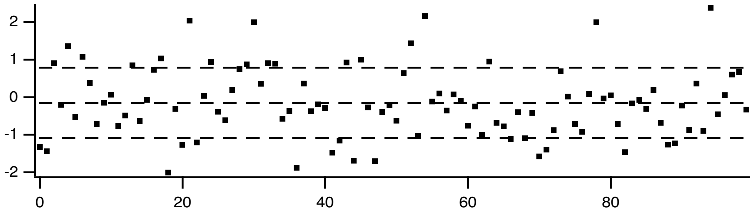
Igor ignores NaNs and INFs in computing the average, standard deviation, RMS, minimum and maximum. NaNs result from computations that have no defined mathematical meaning. They can also be used to represent missing values. INFs result from mathematical operations that have no finite value.

This procedure illustrates the use of WaveStats. It shows the average and standard deviation of a source wave, assumed to be displayed in the top graph. It draws lines to indicate the average and standard deviation.

```
Function ShowAvgStdDev(source)
   Wave source                                        // source waveform

   Variable left=leftx(source),right=rightx(source)   // source X range
   WaveStats/Q source
   SetDrawLayer/K ProgFront
   SetDrawEnv xcoord=bottom,ycoord=left,dash= 7
   DrawLine left, V_avg, right, V_avg                 // show average
   SetDrawEnv xcoord=bottom,ycoord=left,dash= 7
   DrawLine left, V_avg+V_sdev, right, V_avg+V_sdev   // show +std dev
   SetDrawEnv xcoord=bottom,ycoord=left,dash= 7
   DrawLine left, V_avg-V_sdev, right, V_avg-V_sdev   // show -std dev
   SetDrawLayer UserFront
End
```

You could try this function using the following commands.

```
Make/N=100 wave0 = gnoise(1)
Display wave0; ModifyGraph mode(wave0)=2, lsize(wave0)=3
ShowAvgStdDev(wave0)
```



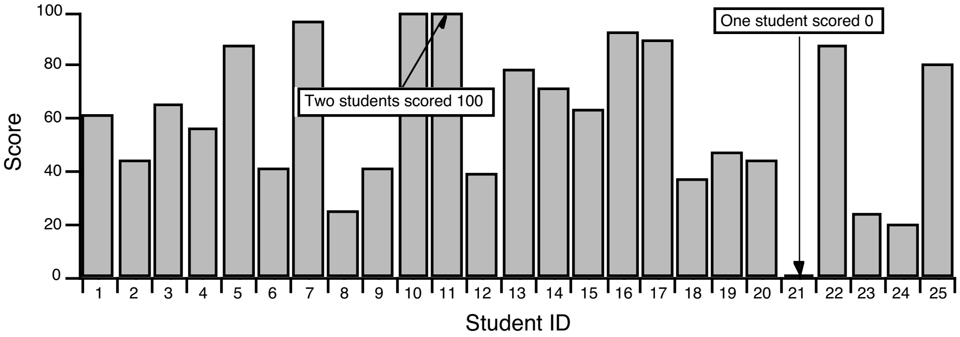
When you use WaveStats with a complex wave, you can choose to compute the same statistics as above for the real, imaginary, magnitude and phase of the wave. By default WaveStats only computes the statistics for the real part of the wave. When computing the statistics for other components, the operation stores the results in a multidimensional wave M_WaveStats.

If you are working with large amounts of data and you are concerned about computation speed you might be able to take advantage of the /M flag that limits the calculation to the first order moments.

If you are working with 2D or 3D waves and you want to compute the statistics for a domain of an arbitrary shape you should use the **ImageStats** operation (see page V-414) with an ROI wave.

Histograms

A histogram totals the number of input values that fall within each of a number of value ranges (or “bins”) usually of equal extent. For example, a histogram is useful for counting how many data values fall in each range of 0-10, 10-20, 20-30, etc. This calculation is often made to show how students performed on a test:



The usual use for a histogram in this case is to figure out how many students fall into certain numerical ranges, usually the ranges associated with grades A, B, C, and D. Suppose the teacher decides to divide the 0-100 range into 4 equal parts, one per grade. The **Histogram** operation (see page V-349) can be used to show how many students get each grade by counting how many students fall in each of the 4 ranges.

We start by creating a wave to hold the histogram output:

```
Make/N=4/D/O studentsWithGrade
```

Next we execute the Histogram command which we generated using the Histogram dialog:

```
Histogram/B={0,25,4} scores,studentsWithGrade
```

The /B flag tells Histogram to create four bins, starting from 0 with a bin width of 25. The first bin counts values from 0 up to but not including 25.

The Histogram operation analyzes the source wave (`scores`), and puts the histogram result into a destination wave (`studentsWithGrade`).

Let’s create a text wave of `grades` to plot `studentsWithGrade` versus a grade letter in a category plot:

```
Make/O/T grades = {"D", "C", "B", "A"}
Display studentsWithGrade vs grades
SetAxis/A/E=1 left
```

Everything *looks* good in the category plot. Let’s double-check that all the students made it into the bins:

```
Print sum(studentsWithGrade)
  23
```

There are two missing students. They are ones who scored 100 on the test. The four bins we defined are actually:

Bin 1:	0 - 24.99999
Bin 2:	25 - 49.99999
Bin 3:	50 - 74.99999
Bin 4:	75 - 99.99999

The problem is that the test scores actually encompass 101 values, not 100. To include the perfect scores in the last bin, we could add a small number such as 0.001 to the bin width:

Bin 1:	0 - 25.00999
Bin 2:	25.001 - 49.00199
Bin 3:	50.002 - 74.00299
Bin 4:	75.003 - 100.0399

The students who scored 25, 50 or 75 would be moved down one grade, however. Perhaps the best solution is to add another bin for perfect scores:

```
Make/O/T grades= {"D", "C", "B", "A", "A+"}
Histogram/B={0,25,5} scores,studentsWithGrade
```

For information on plotting a histogram, see Chapter II-14, **Category Plots** and **Graphing Histogram Results** on page III-127.

This example was intended to point out the care needed when choosing the histogram binning. Our example used “manual binning”.

The Histogram operation provides five ways to set binning. They correspond to the radio buttons in the Histogram dialog:

Bin Mode	What It Does
Manual bins	Sets number of points and X scaling of the destination (output) wave based on parameters that you explicitly specify.
Auto-set bins	Sets X scaling of destination wave to cover the range of values in the **source** wave. Does not change the number of points (bins) in the destination wave. Thus, you must set the number of destination wave points before computing the histogram. When using the Histogram dialog, if you select Make New Wave or Auto from the Output Wave menu, the dialog must be told how many points the new wave should have. It displays the Number of Bins box to let you specify the number.
Set bins from destination wave	Does not change the X scaling or the number of points in the destination wave. Thus, you need to set the X scaling and number of points of the destination wave before computing the histogram. When using the Histogram dialog, the Set from destination wave radio button is only available if you choose Select Existing Wave from the Output Wave menu.
Auto-set bins: 1+log2(N)	Examines the input data and sets the number of bins based on the number of input data points. Sets the bin range the same as if Auto-set bin range were selected.
Auto-set bins: 3.49*Sdev*N^-1/3	Examines the input data and sets the number of bins based on the number of input data points and the standard deviation of the data. Sets the bin range the same as if Auto-set bin range were selected.
Freedman-Diaconis method	Sets the optimal bin width to $binWidth = 2 * \mathrm{IQR} * N^{-1/3}$ where IQR is the interquartile distance (see StatsQuantiles) and the bins are evenly-distributed between the minimum and maximum values. Added in Igor Pro 7.00.

Histogram Caveats

If you use the “Set bins from destination wave” mode, you must create the destination wave, using the **Make** operation, before computing the histogram. You also must set the X scaling of the destination wave, using the **SetScale**.

The Histogram operation does not distinguish between 1D waves and multidimensional waves. If you use a multidimensional wave as the source wave, it will be analyzed as if the wave were one dimensional. This may still be useful - you will get a histogram showing counts of the data values from the source wave as they fall into bins.

If you would like to perform a histogram of 2D or 3D image data, you may want to use the **ImageHistogram** operation (see page V-379), which supports specific features that apply to images only.

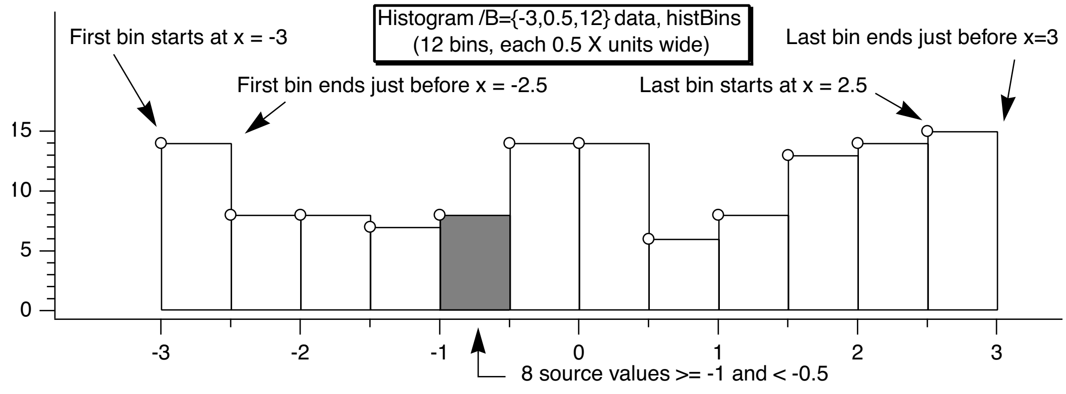
Graphing Histogram Results

Our example above displayed the histogram results as a category plot because the bins corresponded to text values. Often histogram bins are displayed on a numeric axis. In this case you need to know how Igor displays a histogram result.

For example, this histBins destination wave has 12 points (bins), the first bin starting at -3, and each bin is 0.5 wide. The X scaling is shown in the table:

Point	histBins.x	histBins.d
0	-3	14
1	-2.5	8
2	-2	8
3	-1.5	7
4	-1	8
5	-0.5	14
6	0	14
7	0.5	6
8	1	8
9	1.5	13
10	2	14
11	2.5	15

When histBins is graphed in both bars and markers modes, it looks like this:



Note that the markers are positioned at the start of the bars. You can offset the marker trace by half the bin width if you want them to appear in the center of the bin.

Alternatively, you can make a second histogram using the Bin-Centered X Values option. In the Histogram dialog, check the Bin-Centered X Values checkbox.

The Histogram Dialog

To use the Histogram operation, choose Histogram from the Analysis menu.

To use the “Manually set bins” or “Set from destination wave” bin modes, you need to decide the range of data values in the source wave that you want the histogram to cover. You can do this visually by graphing the source wave or you can use the WaveStats operation to find the exact minimum and maximum source values.

The dialog requires that you enter the starting bin value and the bin width. If you know the starting bin value and the ending bin value then you need to do some arithmetic to calculate the bin width.

A line of text at the bottom of the Destination Bins box tells you the first and last values, as well as the width and number of bins. This information can help with trial-and-error settings.

If you use the “Manually set bins” or any of the “Auto-set” modes, Igor will set the X units of the destination wave to be the same as the Y units of the source wave.

If you enable the Accumulate checkbox, Histogram does not clear the destination wave but instead adds counts to the existing values in it. Use this to accumulate results from several histograms in one destination. If you want to do this, don’t use the “Auto-set bins” option since it makes no sense to change bins in mid-stream. Instead, use the “Set from destination wave” mode. To use the Accumulate option, the destination wave must be double-precision or single-precision and real.

The “Bin-Centered X Values” and “Create Square Root(N) Wave” options are useful for curve fitting to a histogram. If you do not use Bin-Centered X Values, any X position parameter in your fit function will be shifted by half a bin width. The Square Root(N) Wave creates a wave that estimates the standard deviation of the histogram data; this is based on the fact that counting data have a Poisson distribution. The wave created by this option does not try to do anything special with bins having zero counts, so if you use the square root(N) wave to weight a curve fit, these zero-count bins will be excluded from the fit. You may need to replace the zeroes with some appropriate value.

The binning modes were added in Igor Pro. In earlier versions of Igor, the accumulate option had *two* effects:

- Did not clear the destination wave.
- Effectively used the “Set bins from destination wave” mode.

To maintain backward compatibility, the Histogram operation still behaves this way if the accumulate (“/A”) flag is used and no bin (“/B”) flag is used. This dialog always generates a bin flag. Thus, the accumulate flag just forces accumulation and has no effect on the binning.

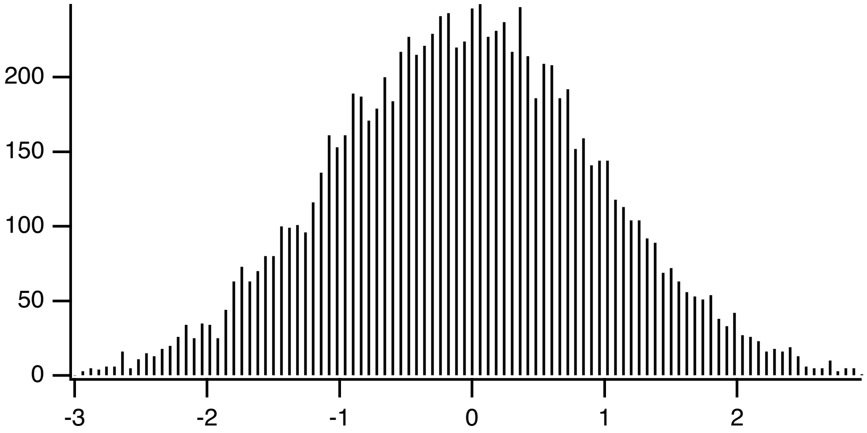
You can use the Histogram operation on multidimensional waves but they are treated as though the data belonged to a single 1D wave. If you are working with 2D or 3D waves you may prefer to use the **ImageHistogram** operation (see page V-379), which computes the histogram of one layer at a time.

Histogram Example

The following commands illustrate a simple test of the histogram operation.

```
SetRandomSeed 0.2                                      // For reproducible randomness
Make/O/N=10000 noise = gnoise(1)                              // Make raw data
Make hist                                                     // Make destination wave
Histogram/B={-3, (3 - -3)/100, 100} noise, hist           // Perform histogram
Display hist; Modify mode(hist)=1
```

These commands produce the following graph:



The raw data has 10,000 samples drawn from a Gaussian distribution having standard deviation of 1.0, and mean of zero. The histogram is made with 100 bins with width of 6/100 = 0.06. The amplitude of the histogram is expected to be:

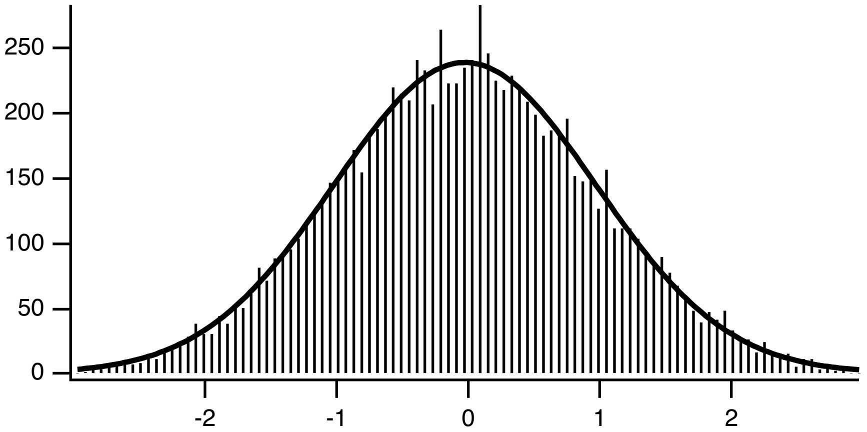
```
A = (N*dx)/sqrt(2*pi)/sigma = (10000*0.06)/sqrt(2*pi)/1 = 239.365
```

The X scaling of the wave hist uses the Histogram default of the left side of the bins.

Curve Fitting to a Histogram

Because the values in the source wave are Gaussian deviates generated by the gnoise function, the histogram should have the familiar Gaussian bell-shape. You can estimate the characteristics of the population the samples were taken from by fitting a Gaussian curve to the data. First try fitting a Gaussian curve to the example histogram:

```
CurveFit gauss hist /D     // Curve fit to histogram
```



The solution from the curve fit is:

```
y0    =  -0.85917 ± 2.74
A     =  238.23 ± 3.07
x0    =  -0.047537 ± 0.0113
width =  1.4337 ± 0.0272
```

Because of the definition of Igor's built-in gauss fit function, we expect the width coefficient to be sqrt(2)*sigma or 1.4142. The coefficients from this fit are within one standard deviation of the expected values, except for peak position x0.

The peak position x0 is shifted approximately half a bin below zero. Since the gnoise() function produces random numbers with mean of zero, we would expect x0 to be close to zero. The shifted value of x0 is a result of Igor's way of storing the X values for histogram bins. Setting the X value to the left edge is good for displaying a bar chart, but bad for curve fitting.

By default, a histogram wave has the X scaling set such that an X value gives the value at the left end of a bin. Usually if you are going to fit a curve to a histogram, you want centered X values. You can change the X scaling to centered values using SetScale. But it is easier to simply use the option to have the Histogram operation produce output with bin-centered X values by adding the /C flag to the Histogram command:

```
Histogram/C/B={-3, (3 - -3)/100, 100} noise, hist  // Perform histogram
CurveFit gauss hist /D                             // Curve fit to histogram
```

The result of the curve fit is the same as before except that x0 is closer to what we expect:

```
y0    =  -0.85917 ± 2.74
A     =  238.23 ± 3.07
x0    =  -0.017537 ± 0.0113
width =  1.4337 ± 0.0272
```

But this shifts the trace showing the histogram by half a bin on the graph. If the trace is displayed using markers or dots, this may be what is desired, but if you have used bars, the display is incorrect.

Another possibility is to make an X wave to go with the histogram data. This X wave would contain X values shifted by half a bin. Use this X wave as input to the curve fit, but don't use it on the graph:

```
Histogram/B={-3, (3 - -3)/100, 100} noise, hist    // Histogram without /C
Duplicate hist, hist_x
hist_x = x + deltax(hist)/2
CurveFit gauss  hist /X=hist_x/D
```

Use this method to graph the original histogram wave without modifying the X scaling, so a graph using bars is correct. It also gives a curve fit that uses the center X values, giving the correct x0. You could also use the Histogram operation twice, once with the /C flag to get bin-centered X values, and once without to get the shifted X scaling appropriate for bars. Both methods have the drawback of creating an extra wave that you must track.

This fit is not statistically correct. Because the histogram represents counts, the values in a histogram should have uncertainties described by a Poisson distribution. The standard deviation of a Poisson distribution is equal to the square root of the mean, which implies that the estimated uncertainty of a histogram bin depends on the magnitude of the value. This in turn implies that the errors are not constant and a curve fit will give a biased solution and possibly poor estimates of the uncertainties of the fit coefficients.

This problem can be solved approximately using a weighting wave. The appropriate weighting wave is generated by the Histogram operation if you add the /N flag or turn on the Create Square Root(N) Wave checkbox in the Histogram dialog.

The next example makes a new data set using gnoise to make gaussian-distributed values, makes a histogram with bin-centered X values and the appropriate weighting wave, and then does two curve fits, one without weighting and one with.

We expect coefficient values from the curve fit to be:

```
y0       =  0
A        =  (1024*deltax(gdata_Hist))/sqrt(2*pi)/1 = 139.863
x0       =  0
width    =  1.4142
```

```
SetRandomSeed 0.5              // For reproducible randomness
Make/O/N=1024 gdata = gnoise(1)
Make/O/N=20 gdata_Hist
Histogram/C/N/B=4 gdata,gdata_Hist
Display gdata_Hist
ModifyGraph mode=3,marker=8
```

```
CurveFit gauss  gdata_Hist /D
CurveFit gauss  gdata_Hist /W=W_SqrtN /I=1 /D
```

Note the addition of "`/W=W_SqrtN /I=1`" addition to the second CurveFit command. This adds the weighting using the weighting wave created by the Histogram operation. Also, /B=4 was used to have the Histogram operation set the number of bins and bin range appropriately for the input data.

The results from the unweighted fit:

```
y0    =  -3.3383 ± 2.98
A     =  133.26 ± 3.51
x0    =  0.024088 ± 0.0252
width =  1.5079 ± 0.0578
```

And from the weighted fit:

```
y0    =  0.33925 ± 0.804
A     =  135.21 ± 5.25
x0    =  0.0038416 ± 0.031
width =  1.3604 ± 0.0405
```

The results are clearly and significantly different. Since we created fake data we know what to expect the results to be; the weighted fit also is closer to our expectation.

There are some possible objections to this weighted fit, that may or may not be important to you.

The weighted fit is only an approximate solution to the fact that the actual errors follow a Poisson distribution. The truly correct way to fit count data is to do a maximum likelihood fit. Igor does not directly support maximum likelihood fitting.

When using the square root approximation of the standard deviation of the Poisson distribution, the fitted model is a better approximation of the actual counts than each individual original data points. But Igor doesn't have a way to replace the weighting based on the current iteration, so the only way to do that is to do the fit, re-compute the weighting wave and do the fit again. Repeat long enough that it doesn't change much any more.

The shape of the Poisson distribution is well-approximated by a Gaussian only for large numbers of counts. In practice, "large" may mean "more than 5 or so". In our example, only five points out on the tails have five or fewer counts, but we started with over 1000 points. You may not be so lucky!

Iterative Weighted Fit

This section is for advanced users only. It provides an example of an iterative fit with corrected weighting wave.

```
Function FitGaussHistogram(Wave histwave, Wave InSqrtNwave)
   // Make a copy so that we don't change the input wave
   Duplicate/FREE InSqrtNwave, sqrtNwave

   // Get a first cut at the correct fit using sqrtNwave provided by Histogram
   CurveFit/Q gauss  histwave /W=sqrtNwave /I=1
   Wave W_coef

   // Save the fit solution for comparison in the loop
   Duplicate/FREE W_coef, lastCoef

   // Compute the length of the initial solution to use in the stopping criterion
   MatrixOp/FREE/O length = sqrt(sum(magSqr(W_coef)))
   Variable initialLength = length[0]

   // Now loop and re-compute the weighting wave for each iteration
   // Do a new fit with the new weighting
   do
```

```
      // New weighting is square root of the expected Y values from the fit
      sqrtNwave = sqrt(Gauss1D(W_coef, pnt2x(histwave, p)))

      // Do a new fit
      CurveFit/Q gauss  histwave /W=sqrtNwave /I=1
      Wave W_coef

      // Compute the length of the difference between this fit and the last
      MatrixOp/FREE/O delta = sqrt(sum(magSqr(W_coef - lastCoef)))

      // Uncomment these lines if you would like to see the progress
      // Print delta
      // Print W_coef

      // Rather arbitrary stopping criterion
      if (delta[0] < 1e-8*initialLength)
         break
      endif

      lastCoef = W_coef
   while(1)

   Wave W_sigma
   Print "y0\t=\t",W_coef[0], "±", W_sigma[0]
   Print "A\t=\t",W_coef[1], "±", W_sigma[1]
   Print "x0\t=\t",W_coef[2], "±", W_sigma[2]
   Print "width\t=\t",W_coef[3], "±", W_sigma[3]
End
```

To try it, paste the code into the procedure window, compile, and execute this on the command line:

```
FitGaussHistogram(gdata_Hist, W_SqrtN)
```

The result is:

```
y0    =  -0.985823 ± 1.22856
A     =  138.462 ± 5.31568
x0    =  -0.00928052 ± 0.0313763
width =  1.45437 ± 0.0521087
```

This result from this example is within one standard deviation of the weighted fit using the weighting wave generated by the histogram.

Computing an “Integrating” Histogram

In a histogram, each bin of the destination wave contains a count of the number of occurrences of values in the source that fell within the bounds of the bin. In an *integrating* histogram, instead of *counting* the occurrences of a value within the bin, we *add* the value itself to the bin. When we’re done, the destination wave contains the sum of all values in the source which fell within the bounds of the bin.

Igor comes with an example experiment called “Integrating Histogram” that illustrates how to do this with a user function. To see the example, choose File→Example Experiments→Analysis→Integrating Histogram.

Sorting

The **Sort** operation (see page V-883) sorts one or more 1D numeric or text waves in ascending or descending order.

The Sort operation is often used to prepare a wave or an XY pair for subsequent analysis. For example, the interp function assumes that the X input wave is monotonic.

There are other sorting-related operations: MakeIndex and IndexSort. These are used in rare cases and are described the section **MakeIndex and IndexSort** on page III-134. The **SortColumns** operation sorts columns of multidimensional waves. Also see the **SortList** function for sorting string lists.

To use the Sort operation, choose Sort from the Analysis menu.

The sort key wave controls the reordering of points. However, the key wave itself is not reordered unless it is also selected as a destination wave in the “Waves to Sort” list.

The number of points in the destination wave or waves must be the same as in the key wave. When you select a wave from the dialog’s Key Wave list, Igor shows only waves with the same number of points in the Waves to Sort list.

The key wave can be a numeric or text wave, but it must not be complex. The destination wave or waves can be text, real or complex except for the MakeIndex operation in which case the destination must be text or real.

The number of destination waves is constrained by the 2500 byte limit in Igor’s command buffer. To sort a very large number of waves, use several Sort commands in succession, being careful not to sort the key wave until the very last.

By default, text sorting is case-insensitive. Use the /C flag with the Sort operation to make it case-sensitive.

Simple Sorting

In the simplest case, you would select a single wave as both the source and the destination. Then Sort would merely sort that wave.

If you want to sort an XY pair such that the X wave is in order, you would select the X wave as the source and both the X and Y waves as the destination.

Sorting to Find the Median Value

The following user-defined function illustrates a simple use of the Sort operation to find the median value of a wave.

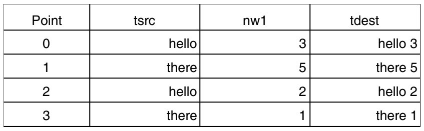
```
Function FindMedian(w, x1, x2)// Returns median value of wave w
   Wave w
   Variable x1, x2            // Range of interest

   Variable result

   Duplicate/R=(x1,x2)/FREE w, medianWave // Make a clone of wave
   Sort tempMedianWave, medianWave        // Sort clone
   SetScale/P x 0,1,medianWave
   result = medianWave((numpnts(medianWave)-1)/2)

   return result
End
```

It is easier and faster to use the built-in **median** function to find the median value in a wave.

Multiple Sort Keys

If the key wave has two or more identical values, you may want to use a secondary source to determine the order of the corresponding points in the destination. This requires using multiple sort keys. The Sorting dialog does not provide a way to specify multiple sort keys but the Sort operation does. Here is an example demonstrating the difference between sorting by single and by multiple keys. Notice that the sorted wave (`tdest`) is a text wave, and the sort keys are text (`tsrc`) and numeric (`nw1`):

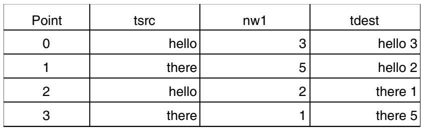
```
Make/O/T tsrc={"hello","there","hello","there"}
Duplicate/O tsrc,tdest
Make nw1= {3,5,2,1}
tdest= tsrc + " " + num2str(nw1)
Edit tsrc,nw1,tdest
```



Point	tsrc	nw1	tdest
0	hello	3	hello 3
1	there	5	there 5
2	hello	2	hello 2
3	there	1	there 1

Single-key text sort:

```
Sort tsrc,tdest             // nw1 not used
```



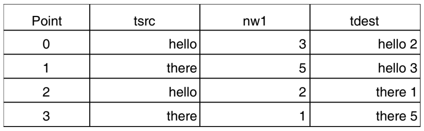
Point	tsrc	nw1	tdest
0	hello	3	hello 3
1	there	5	hello 2
2	hello	2	there 1
3	there	1	there 5

Execute this to scramble tdest again:

```
tdest= tsrc + " " + num2str(nw1)
```

Execute this to see a two key sort (nw1 breaks ties):

```
Sort {tsrc,nw1},tdest
```



Point	tsrc	nw1	tdest
0	hello	3	hello 2
1	there	5	hello 3
2	hello	2	there 1
3	there	1	there 5

The reason that “hello 3” sorts after “hello 2” is because nw1[0] = 3 is greater than nw1[2] = 2.

You can sort by more than two keys by specifying more than two waves inside the braces.

MakeIndex and IndexSort

The MakeIndex and IndexSort operations are infrequently used. You will normally use the Sort operation.

Applications of MakeIndex and IndexSort include:

- Sorting large quantities of data
- Sorting individual waves from a group one at a time
- Accessing data in sorted order without actually rearranging the data
- Restoring data to the original ordering

The MakeIndex operation creates a set of index numbers. IndexSort can then use the index numbers to rearrange data into sorted order. Together they can be used to sort just like the Sort operation but with an extra wave and an extra step.

The advantage is that once you have the index wave you can quickly sort data from a given set of waves at any time. For example, if you have hundreds of waves you can not use the normal sort operation on a single command line. Also, when writing procedures it is sometimes more convenient to loop through a set of waves one at a time than to try to generate a single command line with multiple waves.

You can also use the index values to access data in sorted order without using the IndexSort operation. For example, if you have data and index waves named wave1 and wave1index, you can access the data in sorted order on the right hand side of a wave assignment like so:

```
wave1[wave1index[p]]
```

If you create an index wave, you can undo a sort and restore data to the original order. To do this, simply use the Sort operation with the index wave as the sort key.

Like the Sort operation, the MakeIndex operation can handle multiple sort keys.

The output wave from MakeIndex contains indices which can be used to access the elements of the input wave in sorted order. These indices can be used later with IndexSort to sort the input wave and/or to reorder other waves. For example:

```
Make/O data0 = {1,9,2,3}
Make/O/N=4 index
MakeIndex data0, index
Print index             // Prints 0, 2, 3, 1
```

The output values 0, 2, 3, and 1 mean that, to sort data0, you would access element 0, then element 2, then element 3, then element 1. For example, this prints the values of data0 in order:

```
Print data0[0], data0[2], data0[3], data0[1]
```

You can now apply that sequence of access using IndexSort:

```
IndexSort index, data0
Print data0             // Prints 1, 2, 3, 9
```

You can apply the same reordering to another wave using IndexSort:

```
Make/O data1 = {0,1,2,3}
IndexSort index, data1
Print data1             // Prints 0, 2, 3, 1
```

Decimation

If you have a large data set it may be convenient to deal with a smaller but representative number of points. In particular, if you have a graph with millions of points, it probably takes a long time to draw or print the graph. You can do without many of the data points without altering the graph much. Decimation is one way to accomplish this.

There are at least two ways to decimate data:

1. Keep only every nth data value. For example, keep the first value, discard 9, keep the next, discard 9 more, etc. We call this **Decimation by Omission** (see page III-135).
2. Replace every nth data value with the result of some calculation such as averaging or filtering. We call this **Decimation by Smoothing** (see page III-136).

Decimation by Omission

To decimate by omission, create the smaller output wave and use a simple assignment statement (see **Waveform Arithmetic and Assignments** on page II-74) to set their values. For example, If you are decimating by a factor of 10 (omitting 9 out of every 10 values), create an output wave with 1/10th as many points as the input wave.

For example, make a 1000 point test input waveform:

```
Make/O/N=1000 wave0
SetScale x 0, 5, wave0
wave0 = sin(x) + gnoise(.1)
```

Now, make a 100 point waveform to contain the result of the decimation:

```
Make/O/N=100 decimated
SetScale x 0, 5, decimated    // preserve the x range
decimated = wave0[p*10]       // for(p=0;p<100;p+=1) decimated[p]= wave0[p*10]
```

Decimation by omission can be obtained more easily using the Resample operation and dialog by using an interpolation factor of 1 and a decimation factor of (in this case) 10, and a filter length of 1.

```
Duplicate/O wave0, wave0Resampled
Resample/DOWN=10/N=1 wave0Resampled
```

Decimation by Smoothing

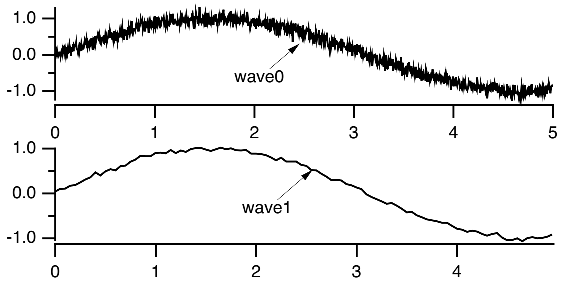
While decimation by omission completely discards some of the data, decimation by smoothing combines all of the data into the decimated result. The smoothing can take many forms: from simple averaging to various kinds of lowpass digital filtering.

The simplest form of smoothing is averaging (sometimes called "boxcar" smoothing). You can decimate by averaging some number of points in your original data set. If you have 1000 points, you can create a 100 point representation by averaging every set of 10 points down to one point. For example, make a 1000 point test waveform:

```
Make/O/N=1000 wave0
SetScale x 0, 5, wave0
wave0 = sin(x) + gnoise(.1)
```

Now, make a 100 point waveform to contain the result of the decimation:

```
Make/O/N=100 wave1
SetScale x 0, 5, wave1
wave1 = mean(wave0, x, x+9*deltax(wave0))
```



Notice that the output wave, wave1, has one tenth as many points as the input wave.

The averaging is done by the waveform assignment

```
wave1 = mean(wave0, x, x+9*deltax(wave0))
```

This evaluates the right-hand expression 100 times, once for each point in wave1. The symbol "x" returns the X value of wave1 at the point being evaluated. The right-hand expression returns the average value of wave0 over the segment that corresponds to the point in wave1 being evaluated.

It is essential that the X values of the output wave span the same range as the X values of the input range. In this simple example, the SetScale commands satisfy this requirement.

Results similar to the example above can be obtained more easily using the **Resample** operation (page V-803) and dialog.

Resample is based on a general sample rate conversion algorithm that optionally interpolates, low-pass filters, and then optionally decimates the data by omission. The lowpass filter can be set to "None" which averages an odd number of values centered around the retained data points. So decimation by a factor of 10 would involve averaging 11 values centered around every 10th point.

The decimation by averaging above can be changed to be 11 values centered around the retained data point instead 10 values from the beginning of the retained data point this way:

```
Make/O/N=100 wave1Centered
SetScale x 0, 5, wave1Centered
wave1Centered = mean(wave0, x-5*deltax(wave0), x+5*deltax(wave0))
```

Each decimated result (each average) is formed from different values than wave1 used, but it isn't any less valid as a representation of the original data.

Using the Resample operation:

```
Duplicate/O wave0, wave2
Resample/DOWN=10/WINF=None/N=11 wave2       // no /UP means no interpolation
```

gives nearly identical results to the wave1Centered = mean(…) computation, the exceptions being only the initial and final values, which are simple end-effect variations.

The /WINF and /N flags of Resample define simple low-pass filtering options for a variety of decimation-by-smoothing choices. The default /WINF=Hanning window gives a smoother result than /WINF=None. See the **WindowFunction** operation (page V-1097) for more about these window options.

See **Multidimensional Decimation** on page II-98 for a discussion of decimating 2D and higher dimension waves.

Miscellaneous Operations

WaveTransform

When working with large amounts of data (many waves or multiple large waves), it is frequently useful to replace various wave assignments with wave operations which execute significantly faster. The **WaveTransform** operation (see page V-1090) is designed to help in these situations. For example, to flip the data in a 1D wave you can execute the following code:

```
Function flipWave(inWave)
   wave inWave

   Variable num=numPnts(inWave)
   Variable n2=num/2
   Variable i,tmp
   num-=1
   Variable j
   for(i=0;i<n2;i+=1)
      tmp=inWave[i]
      j=num-i
      inWave[i]=inWave[j]
      inWave[j]=tmp
   endfor
End
```

You can obtain the same result much faster using the command:

```
WaveTransform/O flip, waveName
```

In addition to “flip”, WaveTransform can also fill a wave with point index or the inverse point index, shift data points, normalize, convert to complex-conjugate, compute the squared magnitude or the phase, etc.

For multi-dimensional waves, use MatrixOp instead of WaveTransform. See **Using MatrixOp** on page III-140 for details.

The Compose Expression Dialog

The Compose Expression item in the Analysis menu brings up the Compose Expression dialog.

This dialog generates a command that sets the value of a wave, variable or string based on a numeric or string expression created by pointing and clicking. Any command that you can generate using the dialog could also be typed directly into the command line.

The command that you generate with the Compose Expression dialog consists of three parts: the destination, the assignment operator and the expression. The command resembles an equation and is of the form:

```
<destination> <assignment-operator> <expression>
```

For example:

```
wave1 = K0 + wave2            // a wave assignment command
K0 += 1.5 * K1                // a variable assigment command
str1 = "Today is" + date()    // a string assignment command
```

Table Selection Item

The Destination Wave pop-up menu contains a “_table selection_” item. When you choose “_table selection_”, Igor assigns the expression to whatever is selected in the table. This could be an entire wave or several entire waves, or it could be a subset of one or more waves.

To use this feature, start by selecting in a table the numeric wave or waves to which you want to assign a value. Next, choose Compose Expression from the Analysis menu. Choose “_table selection_” in the Destination Wave pop-up menu. Next, enter the expression that you want to assign to the waves. Notice the command that Igor has created which is displayed in the command box toward the bottom of the dialog. If you have selected a subset of a wave, Igor will generate a command for that part of the wave only. Finally, click Do It to execute the command.

Create Formula Checkbox

The Create Formula checkbox in the Compose Expression dialog generates a command using the := operator rather than the = operator. The := operator establishes a dependency such that, if a wave or variable on the right hand side of the assignment statement changes, Igor will reassign values to the destination (left hand side). We call the right hand side a formula. Chapter IV-9, **Dependencies**, provides details on dependencies and formulas.

Matrix Math Operations

There are four basic methods for performing matrix calculations: normal wave expressions, the Matrix*XXX* operations, the **MatrixOp** operation, and the **MatrixSparse** operation.

Normal Wave Expressions

You can add matrices to other matrices and scalars using normal wave expressions. You can also multiply matrices by scalars. For example:

```
Make matA={{1,2,3},{4,5,6}}, matB={{7,8,9},{10,11,12}}
matA = matA+0.01*matB
```

gives new values for

```
matA = {{1.07,2.08,3.09},{4.1,5.11,6.12}}
```

Matrix*XXX* Operations

A number of matrix operations are implemented in Igor. Most have names starting with the word “Matrix”. For example, you can multiply a series of matrices using the **MatrixMultiply** operation (page V-548). This operation. The /T flag allows you to specify that a given matrix’s data should be transposed before being used in the multiplication.

Many of Igor’s matrix operations use the LAPACK library. To learn more about LAPACK see:

LAPACK Users’ Guide, 3rd ed., SIAM Publications, Philadelphia, 1999.

or the LAPACK web site:

```
http://www.netlib.org/lapack/lug/lapack_lug.html
```

Unless noted otherwise, LAPACK routines support real and complex, IEEE single-precision and double-precision matrix waves. Most matrix operations create the variable V_flag and set it to zero if the operation is successful. If the flag is set to a negative number it indicates that one of the parameters passed to the LAPACK routines is invalid. If the flag value is positive it usually indicates that one of the rows/columns of the input matrix caused the problem.

MatrixOp Operation

The **MatrixOp** operation (page V-550) improves the execution efficiency and simplifies the syntax of matrix expressions. For example, the command

```
MatrixOp matA = (matD - matB x matC) x matD
```

is equivalent to matrix multiplications and subtraction following standard precedence rules.

See **Using MatrixOp** on page III-140 for details.

MatrixSparse Operation

The **MatrixSparse** operation (page V-580) can improve performance and reduce memory utilization for calculations involving large matrices the elements of which are mostly 0. See **Sparse Matrices** on page III-151 for details.

Matrix Commands

Here are the matrix math operations and functions.

General:

```
MatrixCondition(matrixA)
MatrixConvolve coefMatrix, dataMatrix
MatrixCorr [flags] waveA [, waveB]
MatrixDet(matrixA)
MatrixDot(waveA, waveB)
MatrixFilter [flags] Method  dataMatrix
MatrixGLM [/Z] matrixA, matrixB, waveD
MatrixMultiply matrixA[/T], matrixB[/T] [, additional matrices]
MatrixOp [/O] destwave = expression
MatrixRank(matrixA [, maxConditionNumber])
MatrixTrace(matrixA)
MatrixTranspose [/H] matrix
```

EigenValues, eigenvectors and decompositions:

```
MatrixBalance [flags] srcWave
MatrixEigenV [flags] matrixWave
MatrixFactor [flags] srcWave
MatrixGLM matrixA, matrixB, waveD
MatrixInverse [flags] srcWave
MatrixLUD matrixA
MatrixLUDTD srcMain, srcUpper, srcLower
MatrixReverseBalance [flags] scaleWave, eigenvectorsWave
MatrixSchur [/Z] srcMatrix
MatrixSVD matrixA
```

Linear equations and least squares:

```
MatrixGaussJ matrixA, vectorsB
MatrixLinearSolve [flags] matrixA, matrixB
MatrixLinearSolveTD [/Z] upperW, mainW, lowerW, matrixB
MatrixLLS [flags] matrixA, matrixB
MatrixLUBkSub matrtixL, matrixU, index, vectorB
MatrixSolve method, matrixA, vectorB
MatrixSVBkSub matrixU, vectorW, matrixV, vectorB
```

Sparse matrices:

```
MatrixSparse
```

Using MatrixOp

When performing mathematical calculations your first choice may be to write standard wave assignments such as:

```
wave1 = wave2 + wave3
```

When the expression on the right-hand-side (RHS) is complicated or when the waves are large you can gain a significant performance improvement using MatrixOp (short for "matrix operation"). MatrixOp was originally conceived to perform calculations on 2D waves and was later extended to waves of any dimension.

To use the basic form of **MatrixOp** simply prepend its name to the assignment statement:

```
MatrixOp wave1 = wave2 + wave3
```

Although the two commands appear similar, having the general form:

```
destWave = expression
```

they are different in behavior.

A significant difference is that MatrixOp operates on pure array elements without regard to wave scaling and does not support indexing using x, y, z, t, p, q, r, or s on the RHS.

The wave assignment requires that the destination wave exist at the time of execution but MatrixOp creates its destination wave automatically. In these commands:

```
Make/O wave2 = 1/(p+1), wave3 = 1
MatrixOp/O wave1 = wave2 + wave3
```

MatrixOp creates a single precision (SP) wave named wave1 and stores the result in it.

When MatrixOp creates a destination wave, its data type and dimensions depend on the expression on the RHS. In the preceding example, MatrixOp created wave1 as SP because wave2 and wave3 are SP. The rules that MatrixOp uses to determine the data type and dimensions of the destination are discussed below under **MatrixOp Data Promotion Policy** on page III-145.

These rules leads to the following difference between MatrixOp and wave assignments:

```
Make/O wave1 = 42       // wave1 is 128 points wave, all points are set to 42

MatrixOp/O wave1 = 42   // wave1 is now 1 point wave set to 42
```

Assignment statements evaluate the RHS and store the result in a point-by-point manner. Because of this, you sometimes get unexpected results if the destination wave appears on the RHS. In this example, `WaveMin(wave2)` changes during the evaluation of the assignment statement:

```
wave2 = wave2 - WaveMin(wave2)
```

By contrast, MatrixOp evaluates the RHS for all points in the destination before storing anything in the destination wave. Consequently this command behaves as expected:

```
MatrixOp/O wave2 = wave2 - minVal(wave2)
```

Another important difference appears when the RHS involves complex numbers. If you try, for example,

```
Make/O/C cwave2
Make/O wave1, wave3
wave1 = cwave2 + wave3
```

you get an error, "Complex wave used in real expression".

By contrast, MatrixOp creates the destination as complex so this command works without error:

```
MatrixOp/O wave1 = cwave2 + wave3
```

In this example the RHS mixes real and complex waves, something MatrixOp handles without problems.

In addition to the standard math operators, MatrixOp supports a list of functions that can be applied to operands on the RHS. Examples are `abs`, `sin`, `cos`, `mean`, `minVal` and `maxVal`. There are many more described in the **MatrixOp** reference documentation and listed below under **MatrixOp Functions by Category** on page III-149.

MatrixOp Data Tokens

A MatrixOp command has the general form:

```
MatrixOp [flags] destWave = expression
```

The expression on the RHS is a combination of data tokens with MatrixOp operators and MatrixOp functions. A data token is one of the following:

- A literal number
- A numeric constant declared with the Constant keyword
- A numeric local variable
- A numeric global variable
- A numeric wave
- A numeric wave layer
- The result from a MatrixOp function

You can not call a user-defined function or external function from *expression*.

This function illustrates each type of data token:

```
Constant kConstant = 234
Function Demo()
   // Literal number
   MatrixOp/O dest = 123

   // Constant
   MatrixOp/O dest = kConstant

   // Local Variable
   Variable localVariable = 345
   MatrixOp/O dest = localVariable

   // Global Variable
   Variable/G root:gVar = 456
   NVAR globalVariable = root:gVar
   MatrixOp/O dest = globalVariable

   // Wave
   Make/O/N=(3,3) mat = p + 10*q
   MatrixOp/O tMat = mat^t

   // Wave layer
   Make/O/N=(3,3,3) w3D = p + 10*q + 100*r
   MatrixOp/O layer1 = w3D[][][1]

   // Output from a MatrixOp function
   MatrixOp/O invMat = inv(mat)
End
```

MatrixOp Wave Data Tokens

MatrixOp uses only the numeric data and the dimensions of waves. All other properties, in particular wave scaling, are ignored. If wave scaling is important in your application, follow MatrixOp with **CopyScales** or **SetScale** or use a wave assignment statement instead of MatrixOp.

From the command line you can reference a wave in *expression* using the simple wave name, the partial data folder path to the wave or the full data folder path to a wave. In a user-defined function you can reference a wave using a simple wave name or a wave reference variable pointing to an existing wave. We call all of these methods of referencing waves “wave references”.

A wave data token consists of a wave reference optionally qualified by indices that identify a subset of the wave. For example:

```
Function Demo()
   // Creates automatic wave reference for w3D
   Make/O/N=(3,4,5) w3D = p + 10*q + 100*r

   MatrixOp/O dest = w3D            // dest is a 3D wave
   MatrixOp/O dest = w3D[0][1][2]   // dest is a 1D wave with 1 point
   MatrixOp/O dest = w3D[0][1][]    // dest is a 2D wave with 1 layer
End
```

MatrixOp supports only two kinds of subsets of waves:

- A subset that evaluates to a scalar (a single value)
- A subset that evaluates to one or more layers (2D arrays)

Subsets that evaluate to a row (`w3D[1][][2]`) or a column (`w3D[][1][2]`) are not supported.

Scalars: The following expressions evaluate to scalars:

`wave1d[a]` — *a* is a literal number or a local variable. MatrixOp clips the index *a* to the valid range for *wave1d*. The expression evaluates to a scalar equal to the referenced wave element.

`wave2d[a][b]` — *a* and *b* are literal numbers or local variables. MatrixOp clips the indices to the valid range of the corresponding dimensions. The expression evaluates to a scalar equal to the referenced wave element.

`wave3d[a][b][c]` — *a*, *b* and *c* are literal numbers or local variables. MatrixOp clips the indices to the valid range of the corresponding dimensions. The expression evaluates to a scalar equal to the referenced wave element.

Layers: The following evaluate to one or more 2D layers:

`wave3d[][][a]` — Layer *a* from the 3D wave is treated as a 2D matrix. The first and second bracket pairs must be empty. MatrixOp clips *a* to the valid range of layers. The result is a 2D wave.

`wave3d[][][a,b]` — *expression* is evaluated for all layers between layer *a* and layer *b*. MatrixOp clips *a* and *b* to the valid range of layers. The result is a 3D wave.

`wave3d[][][a,b,c]` — *expression* is evaluated for layers starting with layer *a* and increasing to layer *b* incrementing by *c*. Layers are clipped to the valid range and *c* must be a positive integer. The result is a 3D wave.

You can use waves of any dimensions as parameters to MatrixOp functions. For example:

```
Make/O/N=128 wave1d = x
MatrixOp/O outWave = exp(wave1d)
```

MatrixOp does not support expressions that include the same 3D wave on both sides of the equation:

```
MatrixOp/O wave3D = wave3D + 3   // Not allowed
```

Instead use:

```
MatrixOp/O newWave3D = wave3D + 3
```

You can use any combination of data types for operands. In particular, you can mix real and complex types in *expression*. MatrixOp determines data types of inputs and the appropriate output data type at runtime without regard to any type declaration such as Wave/C.

MatrixOp functions can create certain types of 2D data tokens. Such tokens are used on the RHS of the assignment. The const and zeroMat functions create matrices of specified dimensions containing fixed values. The identity function creates identity matrixes and triDiag creates a tridiagonal matrix from 1D input waves. The **rowRepeat** and colRepeat functions also construct matrices from 1D waves. The setRow, setCol and setOffDiag functions transfer data from 1D waves to elements of 2D waves.

MatrixOp and Wave Dimensions

MatrixOp was designed to optimize operations on 2D arrays (matrices) but it also supports other wave dimensions in various expressions. However it does not support zero point waves.

A 1D wave is treated as a matrix with one column. 3D and 4D waves are treated as arrays of layers so their assignments are processed on a layer-by-layer basis. For example:

```
Make/O/N=(4,5) wave2 = q
Make/O/N=(4,5,6) wave3 = r
MatrixOp/O wave1 = wave2 * wave3       // 2D multiplies 3D one layer at a time
```

wave1 winds up as a 3D wave of dimensions (4,5,6). Each layer of wave1 contains the contents of the corresponding layer of wave3 multiplied on an element-by-element basis by the contents of wave2.

Exceptions for layer-by-layer processing are special MatrixOp functions such as beam and transposeVol.

Some binary operators have dimensional restrictions for their wave operands. For example, matrix multiplication (wave2 x wave3) requires that the number of columns in wave2 be equal to the number of rows in wave3. Regular multiplication (wave2*wave3) requires that the number of rows and the columns of the two be equal. For example,

```
Make/O/N=(4,6) wave2
Make/O/N=(3,8) wave3
MatrixOp/O wave1 = wave2 * wave3       // Error: "Matrix Dimensions Mismatch"
```

The exception to this rule is when the righthand operand is a single wave element:

```
MatrixOp/O wave1 = wave2 * wave3[2][3] // Equivalent to scalar multiplication
```

Some MatrixOp functions operate on each column of the matrix and result in a 1 row by N column wave which may be confusing when used elsewhere in Igor. If what you need is an N row 1D wave you can redimension the wave or simply include a transpose operator in the MatrixOp command:

```
Make/O/N=(4,6) wave2=gnoise(4)
MatrixOp/O wave1=sumCols(wave2)^t      // wave1 is a 1D wave with 6 rows
```

MatrixOp Operators

MatrixOp defines 8 binary operators and two postfix operators which are available for use in the RHS expression. The operators are listed in the Operators section of the reference documentation for **MatrixOp** on page V-550.

MatrixOp does not support operator combinations such as +=.

The basic operators, +, -, *, and /, operate in much the same way as they would in a regular wave assignment with one exception: when the - or / operation involves a matrix and a scalar, it is only supported if the matrix is the first operand and the scalar is the second. For example:

```
Make/O wave2 = 1
Variable var1 = 7
MatrixOp/O wave1 = wave2 - var1  // OK: Subtract var1 to each point in wave2
```

```
MatrixOp/O wave1 = var1 - wave2  // Error: Can't subtract a matrix from a scalar
MatrixOp/O wave1 = var1 / wave2  // Error: Can't divide a scalar by a matrix
```

Division of a scalar by a matrix can be accomplished using the reciprocal function as in:

```
MatrixOp/O wave1 = scalar * rec(wave2)
```

The dot operator . is used to compute the generalized dot product:

```
MatrixOp/O wave1 = wave2 . wave3    // Spaces around '.' are optional
```

The x operator designates matrix-matrix multiplication.

```
MatrixOp/O wave1 = wave2 x wave3    // Spaces around 'x' are required!
```

The logical operators && and `||` are restricted to real-valued data tokens and produce unsigned byte numeric tokens with the value 0 or 1.

The two postfix operators ^t (matrix transpose) and ^h (Hermitian transpose) operate on the first token to their left:

```
MatrixOp/O wave1 = wave2^t + wave3
```

Tthe transpose operation has higher precedence and therefore executes before the addition.

The left token may be a compound token as in:

```
MatrixOp/O wave1 = sumCols(wave2)^t
```

MatrixOp supports the ^ character only in the two postfix operators ^t and ^h. For exponentiation use the *powR* and `powC` functions.

MatrixOp Multiplication and Scaling

MatrixOp provides a number of multiplication and scaling capabilities. They include matrix-scalar multiplication:

```
MatrixOp/O wave1 = wave2 * scalar   // Equivalent to a wave assignment
```

and wave-wave multiplication:

```
MatrixOp wave1 = wave2 * wave3      // Also includes layer-by-layer support
```

and matrix-matrix multiplication:

```
MatrixOp wave1 = wave2 x wave3
```

The latter is equivalent to the **MatrixMultiply** operation with the convenience of allowing you to write and execute complicated compound expressions in one line.

MatrixOp adds two specialized scaling functions that are frequently used. The `scaleCols` function multiplies each column by a different scalar and the `scale` function scales its input to a specified range.

MatrixOp Data Rearrangement and Extraction

MatrixOp is often used to rearrange data in a matrix or to extract a subset of the data for further calculation. You can extract an element or a layer from a wave using square-bracket indexing:

```
// Extract scalar from point a of the wave
MatrixOp destWave = wave1d[a]

// Extract scalar from element a,b of the wave
MatrixOp destWave = wave2d[a][b]

// Extract scalar from element a,b,c of the wave
MatrixOp destWave = wave3d[a][b][c]
```

```
// Extract layer a from the 3D wave
MatrixOp destWave = wave3d[][][a]
```

You can also extract layers from a 3D wave starting with layer a and increasing the layer number up to layer b using increments c:

```
MatrixOp destWave = wave3d[][][a,b,c]
```

a, b and c must be scalars. Layers are clipped to the valid range and c must be a positive integer.

If you need to loop over rows, columns, layers or chunks it is often useful to extract the relevant data using the MatrixOp functions `row`, `col`, `layer` and `chunk`. You can extract matrix diagonals using `getDiag`. You can use `subRange` to extract more than one row or more than one column.

The **Rotate** operation works on 1D waves. MatrixOp extends this to higher dimensions with the functions: `rotateRows`, `rotateCols`, `rotateLayers`, and `rotateChunks`.

MatrixOp `transposeVol` is similar to **ImageTransform** `transposeVol` but it supports complex data types.

The MatrixOp `redimension` function is designed to convert 1D waves into 2D arrays but it can also extract data from higher-dimensional waves.

MatrixOp Data Promotion Policy

MatrixOp chooses the data type of the destination wave based on the data types and operations used in the expression on the RHS. Here are some examples:

```
Make/O/B/U wave2, wave3             // Create two unsigned byte waves

MatrixOp/O wave1 = wave2            // wave1 is unsigned byte

MatrixOp/O wave1 = wave2 + wave3    // wave1 is unsigned word (16 bit)

MatrixOp/O wave1 = wave2 * wave3    // wave1 is unsigned word (16 bit)

MatrixOp/O wave1 = wave2 / wave3    // wave1 is SP

MatrixOp/O wave1 = magsqr(wave2)    // wave1 is SP
```

The examples show the destination wave's data type changing to represent the range of possible results of the operation. MatrixOp does not inspect the data in the waves to determine if the promotion is required for the specific waves.

Igor variables are usually treated as double precision data tokens. MatrixOp applies special rules for variables that contain integers. Here is an example:

```
Make/O/B/U wave2                    // Create unsigned byte wave
Variable vv = 2.1                   // Variable containing DP value
MatrixOp/O wave1 = wave2*vv         // wave1 is DP
```

Now change the variable to store various integers:

```
vv = 2
MatrixOp/O wave1 = wave2*vv         // wave1 is unsigned word (16 bit)
vv = 257
MatrixOp/O wave1 = wave2*vv         // wave1 is unsigned integer (32 bit)
vv = 65538

MatrixOp/O wave1 = wave2*vv         // wave1 is DP
```

These examples show that MatrixOp represents the variable `vv` as a token with a data type that fits it contents. If the variable does not contain an integer value the token is treated as DP.

MatrixOp is convenient for complex math calculations because it automatically creates real or complex destination waves as appropriate. For example:

```
Make/O/C wave2, wave3                        // SP complex
Make/O wave4                                 // SP real
MatrixOp/O wave1 = wave2 * wave3 * wave4     // wave1 is SP complex
MatrixOp/O wave1 = abs(wave2*wave3) * wave4  // wave1 is SP real
```

An exception to the complex policy is the `sqrt` function which returns NaN when its input is real and negative.

If you want to restrict data type promotion you can use the /NPRM flag:

```
Make/O/B/U wave2, wave3
MatrixOp/O wave1 = wave2 * wave3    // wave1 is unsigned word (16 bit).
```

A number of MatrixOp functions convert integer tokens to SP prior to processing and are not affected by the /NPRM flag. These include all the trigonometric functions, `chirp` and `chirpZ`, `fft`, `ifft`, normalization functions, `sqrt`, `log`, `exp` and forward/backward substitutions.

MatrixOp Compound Expressions

You can use compound expressions with most MatrixOp functions that operate on regular data tokens. In a compound expression you pass the result from one MatrixOp operator or function as the input to another. For example:

```
MatrixOp/O wave1 = sum(abs(wave2-wave3))
```

This is particularly convenient for transformations and filtering:

```
MatrixOp/O wave1 = IFFT(FFT(wave1,2)*filterWave,3)
```

Some MatrixOp functions, such as `beam` and `transposeVol`, do not support compound expressions because they require input that has more than two-dimensions while compound expressions are evaluated on a layer by layer basis:

```
Make/O/N=(10,20,30) wave3
MatrixOp/O wave1 = beam(wave3+wave3,5,5)  // Error: Bad MatrixOp token
MatrixOp/O wave1 = beam(wave3,5+1,5)      // Compound expression allowed here
```

MatrixOp Quaternion Data Tokens

A quaternion {w,x,y,z} consists of an amplitude w and three vector components x, y, and z where:

w = a, x = bi, y = cj, z = dk

a, b, c and d are real numbers. i, j, and k are the fundamental quaternion units, analogous to i in complex numbers.

Quaternions are often used to represent rotation in 3D graphics. Igor Pro 8 added support for quaternions via MatrixOp functions.

Quaternion arithmetic has its own rules. For example, multiplication is not commutative (ij is not equal to ji).

You can do quaternion arithmetic with MatrixOp by creating and then manipulating quaternion tokens. For example:

```
Function QuaternionMultiplicationDemo()
   // Create waves containing the w, x, y, and z values of quaternions
   Make/FREE wR = {1, 0, 0, 0}
   Make/FREE wI = {0, 1, 0, 0}
   Make/FREE wJ = {0, 0, 1, 0}
   Make/FREE wK = {0, 0, 0, 1}

   MatrixOp/O rW = quat(wR) * quat(wI)     // rW is the result wave
   Printf "1i = {%g,%g,%g,%g} (i)\r", rW[0], rW[1], rW[2], rW[3]

   MatrixOp/O rW = quat(wI) * quat(wI)
   Printf "ii = {%g,%g,%g,%g} (-1)\r", rW[0], rW[1], rW[2], rW[3]

   MatrixOp/O rW = quat(wI) * quat(wJ)
   Printf "ij = {%g,%g,%g,%g} (k)\r", rW[0], rW[1], rW[2], rW[3]

   MatrixOp/O rW = quat(wJ) * quat(wI)
   Printf "ji = {%g,%g,%g,%g} (-k)\r", rW[0], rW[1], rW[2], rW[3]
End
```

In this example, wR, wI, wJ and wK are waves containing the w, x, y, and z values of quaternions. rW is the result wave from MatrixOp. quat(wR), quat(wI) and quat(wJ) are MatrixOp quaternion tokens. They exist only during the execution of a MatrixOp command. MatrixOp uses quaternion arithmetic to evaluate expressions containing quaternion tokens. After the righthand side has been evaluated, MatrixOp stores the result in a wave. Functions that return quaternion coordinates, such as quat and matrixToQuat, return a 4-element wave containing the w, x, y and z values.

You can also create a quaternion token from a scalar or from a wave containing x, y, and z values:

```
Function QuaternionTokenDemo()
   Make/FREE wI = {0, 1, 0, 0}       // w, x, y, z
   Make/FREE wXYZ = {1, 0, 0}        // x, y, z

   MatrixOp/O rW = quat(wI) * quat(wXYZ)
   Printf "ii = {%g,%g,%g,%g} (-1)\r", rW[0], rW[1], rW[2], rW[3]

   MatrixOp/O rW = quat(wI) * quat(wXYZ) * quat(3)
   Printf "3ii = {%g,%g,%g,%g} (-3)\r", rW[0], rW[1], rW[2], rW[3]
End
```

quat(wXYZ) converts the wave containing x, y, and z to a pure imaginary quaternion whose w component is 0. wXZY can be a 3x1 wave as in this example or it can be a 1x3 wave; both produce the same quaternion token.

quat(3) converts the scalar value 3 to a real quaternion whose x, y and z components are 0.

The input to quat must be real, not complex.

The quat function does not normalize the quaternion. It is normalized when it is used as an input to another MatrixOp quaternion operation.

MatrixOp supports the following quaternion functions:

quat, quatToMatrix, quatToAxis, axisToQuat, quatToEuler, and slerp

These are documented in the reference documenation for MatrixOp.

This example shows how to use quatToEuler to generale Euler angles from a quaternion:

```
Function QuaternionEulerDemo()
   Make/FREE wI = {0, 1, 0, 0}
   MatrixOp/O rW = quatToEuler(quat(wI), 123)
   Printf "Euler angles = {%g,%g,%g}\r", rW[0], rW[1], rW[2]
End
```

MatrixOp Multithreading

Common CPUs are capable of running multiple threads. Some calculations are well suited to run in parallel. There are a several ways to take advantage of multithreading using MatrixOp:

- User-created preemptive threads

  MatrixOp is thread-safe so you can call it from preemptive threads. See **ThreadSafe Functions and Multitasking** on page IV-329 for details.
- Layer threads

  If you are evaluating expressions that involve multiple layers you can use the /NTHR flag to run each layer calculation in a separate thread. When you account for thread overhead it makes sense to use /NTHR when the per-layer calculations are on the order of 1 million CPU cycles or more.
- Internal multithreading of operations or functions

  Some MatrixOp functions are automatically multithreaded for SP and DP data. These include matrix-matrix multiplication, trigonometric functions, `hypot`, `sqrt`, `erf`, `erfc`, `inverseErf`, and `inverseErfc`. The **MultiThreadingControl** operation provides fine-tuning of automatic multithreading but you normally do not need to tinker with it.

MatrixOp Performance

In most situations MatrixOp is faster than a wave assignment statement or **FastOp**. However, for small waves the extra overhead may make it slower.

MatrixOp works fastest on floating point data types. For maximum speed, convert integer waves to single-precision floating point before calling MatrixOp.

Some MatrixOp expressions are evaluated with automatic multithreading. See **MatrixOp Multithreading** on page III-148 for details.

MatrixOp Optimization Examples

The section shows examples of using MatrixOp to improve performance.

- Replace matrix manipulation code with MatrixOp calls. For example, replace this:

```
Make/O/N=(vecSize,vecSize) identityMatrix = p==q ? 1 : 0
MatrixMultiply matB, matC
identityMatrix -= M_Product
MatrixMultiply identityMatrix, matD
MatrixInverse M_Product
Rename M_Inverse, matA
```

  with:

```
MatrixOp matA = Inv((Identity(vecSize) - matB x matC) x matD)
```

- Replace waveform assignment statements with MatrixOp calls. For example, replace this:

```
Duplicate/O wave2,wave1
wave1 = wave2*2
```

  with:

```
MatrixOp/O wave1 = wave2*2
```

- Factor and compute only once any repeated sub-expressions. For example, replace this:

```
MatrixOp/O wave1 = var1*wave2*wave3
MatrixOp/O wave4 = var2*wave2*wave3
```

  with:

```
MatrixOp/O/FREE tmp = wave2*wave3   // Compute the product only once
MatrixOp/O wave1 = var1*tmp
MatrixOp/O wave4 = var2*tmp
```

- Replace instances of the ?: conditional operator in wave assignments with MatrixOp calls. For example, replace this:

  ```
  wave1 = wave1[p]==0 ? NaN : wave1[p]
  ```

  with:

  ```
  MatrixOp/O wave1 = setNaNs(wave1,equal(wave1,0))
  ```

MatrixOp Functions by Category

This section lists the MatrixOp functions by category to help you select the appropriate function.

Numbers and Arithmetic

e inf Pi nan maxAB minAB
maxMagAB minMagAB mod

Trigonometric

acos asin atan atan2 cos hypot
phase sin sqrt tan

Exponential

acosh asinh atanh cosh exp expIntegralE1
expm ln log powC powR
sinh tanh

Complex

cmplx conj imag magSqr p2Rect phase
powC r2Polar real

Rounding and Truncation

abs ceil clip floor mag round

Conversion

cmplx
fp32 fp64
int8 int16 int32 uint8 uint16 uint32

Data Properties

numCols numPoints numRows numType
waveChunks waveLayers wavePoints
waveX waveY waveZ waveT

Data Characterization

averageCols crossCovar chol det frobenius
integrate indexCols indexRows

intMatrix	maxCols	maxRows	maxVal		
mean	minCols	minRows	minVal		
normP	oneNorm				
productCol	productCols	productDiagonal	productRows		
sgn					
sum	sumBeams	sumCols	sumND	sumRows	sumSqr
trace	varBeams	varCols			

Data Creation and Extraction

beam	catCols	catRows	col	colRepeat	rowRepeat
chunk	const	decimateMinMax	getDiag	identity	insertMat
inv	layer	layerStack	rec		
subRange	subWaveC	subWaveR			
tridiag	waveIndexSet	waveMap	zeroMat		

Data Transformation

addCols	addRows	bitReverseCol	diagonal	diagRC
normalize	normalizeCols	normalizeRows		
redimension	replace	replaceNaNs		
reverseCol	reverseCols	reverseRow	reverseRows	
rotateChunks	rotateCols	rotateLayers	rotateRows	
scale	scaleCols	spliceCols		
setCol	setColsRange	setNaNs	setOffDiag	setRow
shiftVector	subtractMean	transposeVol		
zapINFs	zapNaNs			

Time Domain

asyncCorrelation	convolve	correlate	limitProduct	syncCorrelation

Frequency Domain

chirpZ	chirpZf	fft	ifft		
FST	FCT	FSST	FSCT	FSST2	FSCT2

Matrix

backwardSub	chol	covariance	det	diagonal	diagRC
forwardSub	frobenius	getDiag	identity	inv	kronProd
outerProduct	setOffDiag	tensorProduct	trace		

Special Functions

erf	erfc	gamma	gammaln	inverseErf	inverseErfc

Logical

equal	greater	within

Bitwise

bitAnd	bitOr	bitShift	bitXOR	bitNot

Quaternions

quat	axisToQuat	quatToAxis	matrixToQuat	quatToMatrix	slerp

Sparse Matrices

Some applications require manipulation of large matrices the elements of which are mostly 0. In these applications, the use of sparse matrices can improve performance and reduce memory utilization.

Igor supports sparse matrices through the **MatrixSparse** operation which was added in Igor Pro 9.00. It uses the Intel Math Kernel Library Sparse BLAS routines and employs the libraries terminology and conventions.

Sparse Matrix Concepts

In this section we discuss some basic sparse matrix concepts as they apply to Igor. For a general primer on sparse matrices, see <https://en.wikipedia.org/wiki/Sparse_matrix>.

A normal matrix in Igor is a 2D wave in which each element of the matrix is stored in memory in a regular pattern of rows and columns. In the context of sparse matrices, we use the term "dense matrix" to refer to a normal matrix.

A sparse matrix in Igor is represented by a set of three 1D waves which define the non-zero elements of the matrix. Igor supports three formats, described below, for sparse matrix representation. The formats are called COO (coordinate format), CSC (compressed column format), and CSR (compressed row format). In the following sections, we use the term "sparse matrix" to mean a matrix defined by three 1D waves according to one of these formats.

You can create a sparse matrix equivalent to a dense matrix using the MatrixSparse TOCOO, TOCSR, and TOCSC conversion operations. Alternatively you can create it directly by creating three 1D waves and storing the appropriate values in them. You can create a dense matrix equivalent to a sparse matrix using TODENSE conversion operation.

MatrixSparse supports a number of math operations such as ADD (add two sparse matrices), MV (multiply a sparse matrix by a vector), SMSM (multiply two sparse matrices), and TRSV (solve a system of linear equations).

Sparse matrix operations work on single-precision and double-precision real and complex data. They do not support INFs or NaNs.

Sparse Matrix Formats

A sparse matrix in Igor is represented by a set of three 1D waves which define the non-zero elements of the matrix. Igor supports three sparse matrix storage formats named COO (coordinate format), CSR (compressed row format), and CSC (compressed column format).

MatrixSparse uses the CSR format except for operations that convert between formats.

For the purpose of illustrating these formats, we will use the example dense matrix from the Wikipedia sparse matrix web page:

```
0   0   0   0
5   8   0   0
0   0   3   0
0   6   0   0
```

The term nnz is short for "number of non-zero values" and appears below and in the documentation for MatrixSparse. In this example, nnz = 4.

COO Sparse Matrix Storage Format

"COO" is the shorthand name for "coordinate format". It is conceptually the simplest format and stores each non-zero matrix value along with the corresponding zero-based row and column indices.

In Igor terminology, COO format uses the following three waves:

W_COOValues stores each non-zero value in the matrix.

W_COORows stores the zero-based row index for each non-zero value in the matrix.

W_COOColumns stores the zero-based column index for each non-zero value in the matrix.

Our example matrix is represented in COO format like this:

```
W_COOValues:    5   8   3   6
W_COORows:      1   1   2   3
W_COOColumns:   0   1   2   1
```

These values can be read vertically as a set of ordered triplets: (5,1,0), (8,1,1), (3,2,2), and (6,3,1). The last of these tells us that the value 6 is the value of row 3, column 1.

The wave names W_COOValues, W_COORows, and W_COOColumns are used by MatrixSparse when it creates an output sparse matrix in COO format. When you specify an input sparse matrix, you can use any wave names.

CSR Sparse Matrix Storage Format

"CSR" is the shorthand name for "compressed sparse row". It is widely used because it is more efficient in terms of memory use and computational speed than COO.

MatrixSparse uses the CSR format except for operations that convert between formats.

In CSR format, the three 1D waves store the non-zero values, the zero-based column indices, and a "pointer" vector which is used to determine in which row each value appears.

In Igor terminology, CSR format uses the following three waves:

W_CSRValues stores each non-zero value in the matrix.

W_CSRColumns stores the zero-based column index for each non-zero value in the matrix.

W_CSRPointerB stores indices into W_CSRValues which are used to determine in which row a particular value appears.

Our example matrix is represented in CSR format like this:

```
W_CSRValues:    5   8   3   6
W_CSRColumns:   0   1   2   1
W_CSRPointerB:  0   0   2   3   4
```

Unlike the case of COO format, these values can not be read vertically as a set of ordered triplets. CSR is a bit more complicated.

The first two of these waves can be read as ordered pairs: (5,0), (8,1), (3,2), and (6,1). Each ordered pair specifies a value (e.g., 6) and a column (e.g., 1) but does not tell us in which row the value appears.

The third wave, W_CSRPointerB, is used to determine in which row a given value appears. It contains one index for each row in the represented matrix plus an additional index which is nnz, the number of non-zero values in the matrix. W_CSRPointerB[i] is the zero-based index in W_CSRValues of the first non-zero value in row i.

In this example, we can interpret the values of W_CSRPointerB as follows:

```
W_CSRPointerB[0] = 0    // First value for row 0 is at index 0 in W_CSRValues *
W_CSRPointerB[1] = 0    // First value for row 1 is at index 0 in W_CSRValues
W_CSRPointerB[2] = 2    // First value for row 2 is at index 2 in W_CSRValues
W_CSRPointerB[3] = 3    // First value for row 3 is at index 3 in W_CSRValues
W_CSRPointerB[4] = 4    // Number of non-zero values in W_CSRValues is 4
```

* There are no non-zero values in row 0 so W_CSRPointerB[0] is the same as W_CSRPointerB[1].

When you specify a sparse matrix in CSR format to the MatrixSparse operation, the last element of W_CSRPointerB, specifying nnz, is optional and you can omit it.

The wave names W_CSRValues, W_CSRColumns, and W_CSRPointerB are used by MatrixSparse when it creates an output sparse matrix in CSR format. When you specify an input sparse matrix, you can use any wave names.

CSC Sparse Matrix Storage Format

"CSC" is the shorthand name for "compressed sparse column". It is more efficient in terms of memory use and computational speed than COO.

In CSC format, the three 1D waves store the non-zero values, the zero-based row indices, and a "pointer" vector which is used to determine in which column each value is to be stored.

In Igor terminology, CSC format uses the following three waves:

W_CSCValues stores each non-zero value in the matrix.

W_CSCRows stores the zero-based row indices for each non-zero value in the matrix.

W_CSCPointerB stores indices into W_CSCValues which are used to determine in which column a particular value appears.

The W_CSCPointerB wave works in CSC in a manner analogous to how W_CSRPointerB in CSR.

When you specify a sparse matrix in CSC format to the MatrixSparse operation, the last element of W_CSCPointerB, specifying nnz, is optional and you can omit it.

The wave names W_CSCValues, W_CSCRows, and W_CSCPointerB are used by MatrixSparse when it creates an output sparse matrix in CSC format. When you specify an input sparse matrix, you can use any wave names.

Sparse Matrix Example

To help you get a feel for how the MatrixSparse operation works, here is a simple example showing multiplication of a sparse matrix by a vector using the MatrixSparse MV operation.

```
Function DemoSparseMatrixMV()
   // Define Wikipedia example sparse matrix in CSR format
   Make/FREE/D values = {5, 8, 3, 6}   // Double-precision floating point
   Make/FREE/L columns = {0, 1, 2, 1}  // 64-bit signed integer
   Make/FREE/L ptrB = {0, 0, 2, 3, 4}  // 64-bit signed integer

   // Create a vector
   Make/FREE/D vector = {1, 1, 1, 1}   // Double-precision floating point

   // Multiply the sparse matrix by the vector
   MatrixSparse rowsA=4, colsA=4, csrA={values,columns,ptrB}, vectorX=vector,
            operation=MV

   // Create wave references for output sparse matrix
   WAVE W_MV      // Output from MV

   Print          // Prints W_MV[0]= {0,13,3,6}
End
```

The three Make commands at the start define a sparse matrix in CSR format using free waves. The values wave must be single-precision or double-precision floating point, real or complex, without INFs or NaNs. The waves containing indices, columns and ptrB in this case, must be 64-bit signed integer.

The next Make statement creates a vector which must have the same data type as the values wave.

The sparse input matrix is defined by the rowsA, colsA, and csrA keywords.

The input vector is specified by the vectorX keyword.

The operation keyword specifies the operation to be performed, MV in this case.

The output in this case is a wave named W_MV created in the current directory. It is a vector with the same data type as the values wave.

Different MatrixSparse operations require different inputs and create different outputs.

List of MatrixSparse Operations

Here are the operations supported by MatrixSparse.

Operation	What It Does
ADD	Adds two sparse matrices producing a sparse output matrix. See **MatrixSparse ADD** on page III-157 for details.
MM	Computes the product of a sparse matrix and a dense matrix producing a dense output matrix. See **MatrixSparse MM** on page III-157 for details.
MV	Computes the product of a sparse matrix and a vector producing a sparse output matrix. See **MatrixSparse MV** on page III-158 for details.
SMSM	Computes the product of two sparse matrices producing a sparse output matrix. See **MatrixSparse SMSM** on page III-158 for details.
TOCOO	Produces a sparse output matrix in COO format equivalent to the input matrix which may be in dense, CSC, or CSR format. See **MatrixSparse TOCOO** on page III-159 for details.
TOCSC	Produces a sparse output matrix in CSC format equivalent to the input matrix which may be in dense, COO, or CSR format. See **MatrixSparse TOCSC** on page III-159 for details.
TOCSR	Produces a sparse output matrix in CSR format equivalent to the input matrix which may be in dense, COO, or CSC format. See **MatrixSparse TOCSR** on page III-160 for details.
TODENSE	Produces a dense output matrix equivalent to the sparse input matrix which may be in COO, CSC, or CSR format. See **MatrixSparse TODENSE** on page III-160 for details.
TRSV	Solves a system of linear equations for a triangular sparse input matrix. See **MatrixSparse TRSV** on page III-161 for details.

MatrixSparse Inputs

Inputs to the MatrixSparse operation are documented and can be understood in terms of the following conceptual matrix, vector, and scalar inputs:

Sparse matrix A is defined by the rowsA and colsA keywords and by one of the cooA, csrA or cscA keywords. In this command, from the DemoSparseMatrixMV example above, the keywords specifying sparse matrix A are highlighted in red:

```
MatrixSparse rowsA=4, colsA=4, csrA={values,columns,ptrB}, vectorX=vector, operation=MV
```

Sparse matrix A is used in all MatrixSparse operations that take one or more sparse matrix inputs. MatrixSparse math operations (ADD, MV, MM, SMSM, TRSV) require that input sparse matrices be in CSR format.

Sparse matrix G is defined by the rowsG and colsG keywords and by one of the cooG, csrG or cscG keywords. Sparse matrix G is used only in MatrixSparse operations that take two sparse matrix inputs (ADD and SMSM). These operations require that input sparse matrices be in CSR format.

Matrix B is defined by the matrixB keyword and is used in MatrixSparse operations that take one or more dense matrix inputs (currently just the MM operation).

Matrix C is defined by the matrixC keyword and is used in MatrixSparse operations that take two dense matrix inputs (currently just the MM operation).

Vector X is defined by the vectorX keyword and is used in MatrixSparse operations that take one or more vector inputs (currently the MM and TRSV operations).

Vector Y is defined by the vectorY keyword and is used in MatrixSparse operations that take two vector inputs (currently just the MV operation).

Alpha and ***alphai*** are defined by the alpha and alphai keywords and are used in all MatrixSparse operations that take one or more scalar inputs (currently the MM, MV, and TRSV operations). Alphai is used only when operating on complex data.

Beta and ***betai*** are defined by the beta and betai keywords and are used in MatrixSparse operations that take two scalar inputs(currently the MM and MV operations). Betai is used only when operating on complex data.

MatrixSparse Operation Data Type

The operation data type is the data type required for all data waves participating in the MatrixSparse command. Here "data waves" means the values wave in the representation of a sparse matrix and the matrix wave representing a dense matrix.

If a given MatrixSparse command takes sparse matrix A as an input then the values wave (the first wave specified by the cooA, cscA, or csrA keywords) determines the operation data type. If the command does not take sparse matrix A then the matrix B wave (specified by the matrixB keyword) determines the operation data type.

If there are multiple input data waves, such as for the ADD operation which adds two sparse matrices, the data type of all input data waves must be the same.

The operation data type must be single-precision or double-precision floating point and can be real or complex. MatrixSparse does not support waves containing NaNs or INFs.

Output waves are created using the operation data type.

MatrixSparse math operations (ADD, MV, MM, SMSM, TRSV) require that input sparse matrices be in CSR format. The math operations that return sparse matrices (ADD, SMSM) create output sparse matrices in CSR format.

The conversion operations (TOCOO, TOCSC, TOCSR, TODENSE) accept inputs in COO, CSC, CSR, or dense formats.

MatrixSparse Index Data Type

Index waves, containing row or column indices or indices into values waves (i.e., pointer waves), must be signed 64-bit integer waves which are typically created using Make/L.

MatrixSparse Transformations

You can optionally tell MatrixSparse to operate on a transformed version of an input sparse matrix. The available transformations are named T for transpose, H for Hermitian, and N for no transformation (default).

The opA keyword tells MatrixSparse to operate on a transformed version of sparse matrix A. For example this command:

```
MatrixSparse rowsA=4, colsA=4, csrA={values,columns,ptrB}, opA=T,
            vectorX=vector, operation=MV
```

operates on a transposed version of sparse matrix A.

The opG keyword tells MatrixSparse to operate on a transformed version of sparse matrix G.

Optional Sparse Matrix Information

The MatrixSparse sparseMatrixType keyword allows you to provide optional information characterizing the sparse matrix inputs. If you know the characteristics of a sparse matrix input, you can use sparseMatrixType to pass this information to MatrixSparse. This can improve performance.

The syntax of the sparseMatrixType keyword is:

```
sparseMatrixType={smType,smMode,smDiag}
```

All of the parameters are keywords.

smType: GENERAL, SYMMETRIC, HERMITIAN, TRIANGULAR, DIAGONAL, BLOCK_TRIANGULAR, or BLOCK_DIAGONAL.

smMode: LOWER or UPPER.

smDiag: DIAG or NON_DIAG.

MatrixSparse Operations

This section documents each of the operations supported by **MatrixSparse**. It is assumed that you have read and understood the background material presented under **Sparse Matrices** on page III-151.

The following sections use these abbreviations:

Symbol	Stands For	Specified By Keywords
smA	Sparse matrix A	rowsA, colsA, csrA (1)
smG	Sparse matrix G	rowsG, colsG, csrG (1)
dmB	Dense matrix B	matrixB
dmC	Dense matrix C	matrixC
vX	Vector X	vectorX
vY	Vector Y	vectorY
alpha	Scalar value alpha	alpha and, for complex input, alphai

Symbol	Stands For	Specified By Keywords
beta	Scalar value beta	beta and, for complex input, betai
smOut	Output sparse matrix	N/A (2)

(1) For the matrix format conversion operations TOCOO, TOCSC, TOCSR, and TODENSE, you can also use the cooA and cscA keywords to specify the input sparse matrix in COO or CSC format. For all other operations you must use csrA and csrG to specify the input matrices in CSR format.

(2) The output sparse matrix, smOut, is represented in CSR format by waves W_CSRValues, W_CSRColumns, and W_CSRPointerB.

MatrixSparse ADD

ADD computes the sum of sparse matrices A and G which must be in CSR format. Symbolically:

```
smOut = smA + smG
```

Inputs: Sparse matrix A and sparse matrix G, both in CSR format.

Output: A sparse matrix in CSR format represented by W_CSRValues, W_CSRColumns, and W_CSRPointerB.

MatrixSparse ADD Example

```
Function DemoMatrixSparseADD()
   // Create smA in CSR format
   Make/FREE/D/N=(11) valuesA = {1,25,26,44,16,22,28,5,11,36,42}
   Make/FREE/L/N=(11) columnsA = {0,4,4,7,2,3,4,0,1,5,6}
   Make/FREE/L/N=(6) ptrBA = {0,2,4,4,7,9}

   // Create smG in CSR format
   Make/FREE/D/N=(3) valuesG ={1,2,3}
   Make/FREE/L/N=(3) columnsG ={0,1,2}
   Make/FREE/L/N=(4) ptrBG = {0,1,2,3,3,3,3,3,3}

   // Compute smA + smG
   MatrixSparse rowsA=6, colsA=8, csrA={valuesA,columnsA,ptrBA}, rowsG=6,
            colsG=8, csrG={valuesG,columnsG,ptrBG}, operation=ADD
   WAVE W_CSRValues, W_CSRColumns, W_CSRPointerB   // Outputs from ADD

   // Print the 1D waves representing the output CSR sparse matrix
   Print W_CSRValues
   Print W_CSRColumns
   Print W_CSRPointerB
End
```

MatrixSparse MM

MM computes the product of a sparse matrix and a dense matrix. Symbolically:

```
M_MMOut = alpha*smA*dmB + beta*dmC
```

Inputs: alpha, sparse matrix A in CSR format, dense matrix B, and optionally beta and dense matrix C.

If you leave beta with its default value of 0 by omitting the beta keyword, the beta*dmC term is not computed and you do not need to specify the dmC input.

Output: Dense matrix M_MMOut.

MatrixSparse MM Example

```
Function DemoMatrixSparseMM()
   // Define sparse matrix in CSR format
   Make/FREE/D values = {5, 8, 3, 6}
   Make/FREE/L columns = {0, 1, 2, 1}
   Make/FREE/L ptrB = {0, 0, 2, 3, 4}

   // Create a dense matrix
   Make/FREE/D matrix = { {1,0,0,0}, {0,1,0,0}, {0,0,1,0}, {0,0,0,1} }

   // Multiply the sparse matrix by the dense matrix
   MatrixSparse rowsA=4, colsA=4, csrA={values,columns,ptrB}, matrixB=matrix,
            operation=MM

   // Create wave reference for output dense matrix
   WAVE M_MMOut         // Output from MV

   Print M_MMOut
End
```

MatrixSparse MV

MV computes the product of a sparse matrix which must be in CSR format and a vector producing an output vector. Symbolically:

```
W_MV = alpha*smA*vX + beta*vY
```

Inputs: alpha, sparse matrix A in CSR format, vector X, and optionally beta and vector Y.

If you leave beta with its default value of 0 by omitting the beta keyword, the beta*vY term is not computed and you do not need to specify the vY input.

Output: Vector W_MV.

MatrixSparse MV Example

```
Function DemoMatrixSparseMV()
   // Define sparse matrix in CSR format
   Make/FREE/D values = {5, 8, 3, 6}
   Make/FREE/L columns = {0, 1, 2, 1}
   Make/FREE/L ptrB = {0, 0, 2, 3, 4}

   // Create a vector
   Make/FREE/D vector = {1, 1, 1, 1}

   // Multiply the sparse matrix by the vector
   MatrixSparse rowsA=4, colsA=4, csrA={values,columns,ptrB}, vectorX=vector,
            operation=MV

   // Create wave reference for output vector
   WAVE W_MV      // Output from MV

   Print W_MV
End
```

MatrixSparse SMSM

SMSM computes the product of a two sparse matrices. Symbolically:

```
smOut = smA * smG
```

Inputs: Sparse matrix A in CSR format, sparse matrix G in CSR format.

Output: A sparse matrix in CSR format represented by W_CSRValues, W_CSRColumns, and W_CSRPointerB.

MatrixSparse SMSM Example

```
Function DemoMatrixSparseSMSM()
   // Create sparse matrix A in CSR format
   Make/FREE/D/N=(11) valuesA = {1,25,26,44,16,22,28,5,11,36,42}
   Make/FREE/L/N=(11) columnsA = {0,4,4,7,2,3,4,0,1,5,6}
   Make/FREE/L/N=(6) ptrBA = {0,2,4,4,7,9}

   // Create sparse matrix G in CSR format
   Make/FREE/D/N=(8) valuesG = {1,10,26,6,14,38,15,23}
   Make/FREE/L/N=(8) columnsG = {0,1,3,0,1,4,1,2}
   Make/FREE/L/N=(8) ptrBG = {0,1,3,3,3,3,6,8}

   // Compute product MatrixA x MatrixG
   MatrixSparse rowsA=6, colsA=8, csrA={valuesA,columnsA,ptrBA}, rowsG=8,
            colsG=5, csrG={valuesG,columnsG,ptrBG}, operation=SMSM
   WAVE W_CSRValues, W_CSRColumns, W_CSRPointerB   // Outputs from SMSM

   // Print the 1D waves representing the output CSR sparse matrix
   Print W_CSRValues
   Print W_CSRColumns
   Print W_CSRPointerB
End
```

MatrixSparse TOCOO

TOCOO produces a sparse output matrix in COO format equivalent to the input matrix which may be in dense, CSC, or CSR format.

Inputs: A dense matrix specified by the matrixB keyword or a sparse matrix specified by the cscA or csrA keywords.

Output: A sparse matrix in COO format represented by W_COOValues, W_COORows, and W_COOColumns.

MatrixSparse TOCOO Example

```
Function DemoMatrixSparseTOCOO()
   // Create the Wikipedia example 4x4 matrix in CSR format
   Make/FREE/D values = {5, 8, 3, 6}
   Make/FREE/L columns = {0, 1, 2, 1}
   Make/FREE/L ptrB = {0, 0, 2, 3, 4}

   // Create a sparse matrix in COO format from the CSR matrix
   MatrixSparse rowsA=4, colsA=4, csrA={values,columns,ptrB}, operation=TOCOO
   WAVE W_COOValues, W_COORows, W_COOColumns    // Outputs from TOCOO

   // Print the 1D waves representing the COO sparse matrix
   Print W_COOValues
   Print W_COORows
   Print W_COOColumns
End
```

MatrixSparse TOCSC

TOCSC produces a sparse output matrix in CSC format equivalent to the input matrix which may be in dense, COO, or CSR format.

Inputs: A dense matrix specified by the matrixB keyword or a sparse matrix specified by the cooA or csrA keywords.

Output: A sparse matrix in CSC format represented by W_CSCValues, W_CSCRows, and W_CSCPointerB.

MatrixSparse TOCSC Example

```
Function DemoMatrixSparseTOCSC()
   // Create the Wikipedia example 4x4 matrix in dense format
   Make/FREE/D/N=(4,4) dense
   dense[0][0] = {0,5,0,0}
   dense[0][1] = {0,8,0,6}
   dense[0][2] = {0,0,3,0}
   dense[0][3] = {0,0,0,0}

   // Create a sparse matrix in CSC format from the dense matrix
   // MatrixSparse requires rowsA and colsA even though they are not used here
   MatrixSparse rowsA=4, colsA=4, matrixB=dense, operation=TOCSC
   WAVE W_CSCValues, W_CSCRows, W_CSCPointerB      // Outputs from TOCSC

   // Print the 1D waves representing the CSC sparse matrix
   Print W_CSCValues
   Print W_CSCRows
   Print W_CSCPointerB
End
```

MatrixSparse TOCSR

TOCSR produces a sparse output matrix in CSR format equivalent to the input matrix which may be in dense, COO, or CSC format.

Inputs: A dense matrix specified by the matrixB keyword or a sparse matrix specified by the cooA or cscA keywords.

Output: A sparse matrix in CSR format represented by W_CSRValues, W_CSRColumns, and W_CSRPointerB.

MatrixSparse TOCSR Example

```
Function DemoMatrixSparseTOCSR()
   // Create the Wikipedia example 4x4 matrix in dense format
   Make/FREE/D/N=(4,4) dense
   dense[0][0] = {0,5,0,0}
   dense[0][1] = {0,8,0,6}
   dense[0][2] = {0,0,3,0}
   dense[0][3] = {0,0,0,0}

   // Create a sparse matrix in CSR format from the dense matrix
   // MatrixSparse requires rowsA and colsA even though they are not used here
   MatrixSparse rowsA=4, colsA=4, matrixB=dense, operation=TOCSR
   WAVE W_CSRValues, W_CSRColumns, W_CSRPointerB   // Outputs from TOCSR

   // Print the 1D waves representing the CSR sparse matrix
   Print W_CSRValues
   Print W_CSRColumns
   Print W_CSRPointerB
End
```

MatrixSparse TODENSE

TODENSE produces a dense output matrix equivalent to the sparse input matrix which may be in COO, CSC, or CSR format.

Inputs: A sparse matrix specified by the cooA, cscA, or csrA keywords.

Output: Dense output matrix M_cooToDense, M_cscToDense, or M_csrToDense.

MatrixSparse TODENSE Example

```
Function DemoMatrixSparseTODENSE()
   // Create the Wikipedia example 4x4 matrix in COO format
   Make/FREE/D values = {5, 8, 3, 6}
   Make/FREE/L rows = {1, 1, 2, 3}
   Make/FREE/L columns = {0, 1, 2, 1}

   // Create a dense matrix from the sparse matrix
   MatrixSparse rowsA=4, colsA=4, cooA={values,rows,columns}, operation=TODENSE
   WAVE M_cooToDense          // Output from TODENSE

   // Print the dense output matrix
   Print M_cooToDense
End
```

MatrixSparse TRSV

TRSV solves a system of linear equations for the triangular sparse matrix A. Symbolically it solves for output matrix M_TRSVOut where:

```
smA*M_TRSVOut = alpha*vX
```

Inputs: Sparse matrix A in CSR format, alpha, vector X.

Output: Dense matrix M_TRSVOut.

MatrixSparse TRSV Example

```
// This is based on the example in the Row Reduction section
// of the Wikipedia page
// https://en.wikipedia.org/wiki/System_of_linear_equations
// The system of equations is:
//    x + 3y - 2z = 5
//    3x + 5y + 6z = 7
//    2x + 47 + 3z = 8
// This gives the following augmented matrix:
//    1  3  -2 5
//    3  5  6  7
//    2  4  3  8
// The solution is: x=-15, y=8, z=2
// Because the MatrixSparse TRSV operation requires a triangularized coefficient
// matrix, we start with the upper-triangular version of the augmented matrix
// obtained using Gauss-Jordan elimination:
//    1  3  -2 5
//    0  1  -3 2
//    0  0  1  2
// We create an equivalent sparse matrix using MatrixSparse TOCSR.
// We then create the corresponding solution vector {5, 5, 2}.
// We then call MatrixSparse TRSV an obtain the solution set {-15, 8, 2}
Function DemoMatrixSparseTRSV()
   // Create dense upper-triangular matrix representing coefficients
   Make/FREE/D/N=(3,3) utMat
   utMat[0][0] = {1, 0, 0}       // Column 0
   utMat[0][1] = {3, 1, 0}       // Column 1
   utMat[0][2] = {-2, -3, 1}     // Column 2

   // Create sparse version of upper triangular matrix in CSR format
   MatrixSparse rowsA=3, colsA=3, matrixB=utMat, operation=TOCSR
   WAVE values = W_CSRValues
   WAVE columns = W_CSRColumns
   WAVE ptrB = W_CSRPointerB
   // Create solution vector
   Make/FREE/D/N=(3) vector = {5,2,2}

   // Solve system of linear equations
   // Adding sparseMatrixType={TRIANGULAR,UPPER,NON_DIAG} may improve speed
   MatrixSparse rowsA=3, colsA=3, csrA={values,columns,ptrB}, vectorX=vector,
         operation=TRSV
   WAVE M_TRSVOut       // Output from TRSV

   Print M_TRSVOut      // Should be {-15, 8, 2}
End
```

Clustering

The following operations perform cluster analysis of various kinds:

FastGaussTransform, **FPClustering**, **ImageThreshold**, **KMeans**, **HCluster**

For hierarchical clustering, see the next section.

Hierarchical Clustering

Hierarchical clustering builds a hierarchy of clusters which provides the information needed to create a cluster dendrogram. You can perform hierarchical clustering using the **HCluster** operation added in Igor Pro 9.00. HCluster, based on code developed by Daniel Müllner, uses an agglomerative hierarchical clustering algorithm.

You provide as input either a square dissimilarity matrix or a matrix in which rows represent vectors in some data space as the input wave. A dissimilarity matrix contains values measuring the degree of difference between every pair of vectors in the original data set. It is common to use "distance" instead of "dissimilarity" because for most purposes Euclidean distance is used to measure dissimilarity. We use "dissimilarity" below.

HCluster creates one or two output waves, depending on the output type that you request. The output waves are a square dissimilarity matrix wave and a dendrogram wave. The format of the dendrogram wave is discussed below under **Dendrogram Wave Format**.

Agglomerative hierarchical clustering proceeds by iteratively finding the pair of nodes that have the minimum dissimilarity amongst all pairs. The node pair with the smallest value of dissimilarity is joined into a single replacement node. A node can represent a single original data vector or a set of vectors that have already been joined into a cluster. This process is repeated until all original data vectors are represented by a single cluster. The output dendrogram wave describes a tree identifying the nodes that were combined and the dissimilarity between those nodes. This description is sufficient for drawing a dendrogram illustrating the clusters.

If the input wave is a dissimilarity matrix, then those dissimilarities are used to form pair-wise nodes in a dendrogram expressing the degree of dissimilarity between pairs of nodes in the tree.

If the input wave is a matrix of raw data vectors, the HCluster operation can be used to either create a dissimilarity matrix or to create a dendrogram directly, without outputting a dissimilarity matrix. It is also possible to get both the dissimilarity matrix and the dendrogram wave from a single call. If you don't need the dissimilarity matrix output, in certain cases vector data can be processed using an algorithm that minimizes memory use.

There are two types of dissimilarity that HCluster need to calculate:

- The dissimilarity between two input vectors
  We call this a "vector dissimilarity" calculation.
  You specify how to calculate vector dissimilarity using the HCluster /DISS flag.

The most common method for calculating vector dissimilarity is the Euclidean metric, the familiar square root of the summed squared differences of the vector elements. See **HCluster Vector Dissimilarity Calculation Methods** on page III-163 for a description of the vector dissimilarity metrics offered by the HCluster operation.

If you use /ITYP=DMatrix, you can prepare your own dissimilarity matrix using whatever method you wish for measuring the dissimilarity between vectors. Your dissimilarity metric must return a positive number. Identical vectors, such as comparing a vector with itself, have a dissimilarity of zero.

- The dissimilarity between a vector and a previously-determined cluster or between two previously-determined clusters

  We call this a "linkage" calculation.

  You specify how to calculate linkage using the HCluster /LINK flag.

  See **HCluster Linkage Calculation Methods** on page III-166 dfor a description of the linkage calculation methods offered by the HCluster operation. The linkage method that you choose can have a very strong effect on the resulting dendrogram.

HCluster Vector Dissimilarity Calculation Methods

You use the HCluster /DISS=dm flag to specify the dissimilarity metric between two data vectors. Our definitions of dissimilarity follows Python scipy.spatial.distance.pdist.

The following values are supported for the *dm* keyword. If you omit /DISS, HCluster defaults to the Euclidean method.

***dm* = Euclidean**

This is the usual way to measure the dissimilarity between two vectors, the two-norm or L_2 norm. It is simply the Euclidean distance. This is the default.

$$d(u,v) = \|u - v\|_2 = \sqrt{\sum_j (u_j - v_j)^2}$$

***dm* = SquaredEuclidean**

Just like Euclidean, but omits taking the square root. May be needed to reproduce some results from R or Python. Results in the same clustering as Euclidean, but exaggerates larger differences.

$$d(u,v) = \sum_j (u_j - v_j)^2$$

***dm* = SEuclidean**

Standardized Euclidean. Euclidean distance in which the dimensions are scaled by V_j, which is usually the variance of the j-th element of all the vectors.

$$d(u,v) = \sqrt{\sum_j (u_j - v_j)^2 / V_j}$$

Specify a wave giving the V_j vector using the /VARW flag.

***dm* = Cityblock**

Manhattan distance or L_1 norm.

$$d(u,v) = \sum_j |u_j - v_j|$$

Cityblock gives the same value of 2 for vectors (0,2), (2,0), and (1,1). Euclidean distance gives a smaller value, sqrt(2), for the vector (1,1). This can affect the resulting clusters.

dm **= Chebychev**

Supremum or L_∞ norm.

$$d(u,v) = \max_j |u_j - v_j|$$

dm **= Minkowski**

The L_p norm.

$$d(u,v) = \left(\sum_j |u_j - v_j|^p \right)^{1/p}$$

The value of p is specified using the HCluster /P flag.

p = 1 makes Minkowski equivalent to Cityblock.

p = 2 makes Minkowski equivalent to Euclidean.

p = Inf makes Minkowski equivalent to Chebychev.

dm **= Cosine**

$$d(u,v) = 1 - \frac{\langle u, v \rangle}{\|u\| \cdot \|v\|} = 1 - \frac{\sum_j u_j v_j}{\sqrt{\sum_j u_j^2 \cdot \sum_j v_j^2}}$$

dm **= Canberra**

$$d(u,v) = \sum_j \frac{|u_j - v_j|}{|u_j| + |v_j|}$$

Terms in which uj = vj = 0 contribute 0 to the sum.

dm **= BrayCurtis**

$$d(u,v) = \sum_j \frac{|u_j - v_j|}{|u_j + v_j|}$$

Terms in which uj = vj = 0 contribute 0 to the sum.

In the following, the notation |{...}| indicates the count of true boolean values.

dm **= Hamming**

$$d(u,v) = |\{j | u_j \neq v_j\}|$$

Hamming is actually intended to be used with binary data, but the definition will test a "1" and "2" as being different. See Matching below, which tests each vector element for u_j != 0. For data that is all ones or zeroes, Hamming and Matching give the same results.

dm **= Jaccard**

Like Hamming, Jaccard is intended for boolean data, but tests for "not equal".

$$d(u,v) = \frac{|\{j|u_j \neq v_j\}|}{|\{j|u_j \neq 0 \; or \; v_j \neq 0\}|}$$

HCluster Vector Dissimilarity Calculation Methods for Boolean Data

The remaining metrics are for boolean data. Any data is accepted and each value is converted to a boolean value by testing for u_j != 0.

The following definitions are used:

$$a = |\{j \mid u_j \wedge v_j\}|$$

$$b = |\{j \mid u_j \wedge (\neg v_j)\}|$$

$$c = |\{j \mid (\neg u_j) \wedge v_j\}|$$

$$d = |\{j \mid (\neg u_j) \wedge (\neg v_j)\}|$$

D is the vector length, $D = a + b + c + d$.

***dm* = Yule**

$$d\left(u,v\right) = \frac{2bc}{ad + bc}$$

***dm* = Dice**

$$d\left(u,v\right) = \frac{b + c}{2a + b + c} \;,\; d\left(0,0\right) = 0$$

***dm* = RogersTanimoto**

$$d\left(u,v\right) = \frac{2\left(b + c\right)}{b + c + D}$$

***dm* = RusselRao**

$$d\left(u,v\right) = \frac{b + c + d}{D}$$

***dm* = SokalSneath**

$$d\left(u,v\right) = \frac{2(b + c)}{a + 2(b + c)} \;,\; d(0,0) = 0$$

***dm* = Kulsinski**

$$d\left(u,v\right) = \frac{1}{2} \cdot \left(\frac{b}{a + b} + \frac{c}{a + c}\right)$$

***dm* = Matching**

This metric is the same as Hamming, if Hamming is given only boolean data, that is, only ones and zeroes.

$$d\left(u,v\right)=\frac{b+c}{D}$$

HCluster Linkage Calculation Methods

You use the HCluster /LINK=*linkMethod* flag to specify the method used to determine the dissimilarity between nodes in the dendrogram that represent more than one data vector. This is also referred to as the "linkage" method. Our definitions of node dissimilarity follows Python scipy.cluster.hierarchy.linkage.

linkMethod is a keyword identifying the method to use. Each of these keywords, described below, selects a method for measuring the dissimilarity between two clusters, *A* and *B*, with individual dissimilarities *a* and *b*. The symbol | A | denotes the number of elements in a cluster. It is not always possible to give an expression in this form.

Alternately, dissimilarity between a new node and other nodes in the tree can be computed from the dissimilarities of the two nodes being combined. That is, if nodes I and J are combined to form K, then it is possible to compute the dissimilarity from K to any other node L in terms of I and J.

Each of the following linkage method descriptions includes two equations. The first describes the method for computing dissimilarity between nodes given the dissimilarities of each of the component dissimilarities. The second describes the method for computing the dissimilarity to another node from a node that combines I and J.

The following values are supported for *linkMethod* keyword. If you omit /LINK, HCluster defaults to the average method.

***linkMethod* = single**

This is the default dissimilarity metric.

$$d(A,B)=\min_{a\in A,b\in B}d(a,b)$$

That is, the dissimilarity is the dissimilarity between the two nearest elements of each cluster. Also called Nearest Point algorithm.

$$d(K,L)=min(d(I,L),d(J,L))$$

***linkMethod* = complete**

$$d(A,B)=\max_{a\in A,b\in B}d(a,b)$$

Also called Farthest Point algorithm.

$$d(K,L)=max(d(I,L),d(J,L))$$

***linkMethod* = average**

$$d(A,B)=\frac{1}{|A|\,|B|}\sum_{a\in A,b\in B}d(a,b)$$

That is, the average of all the pairwise dissimilarities between points in each cluster.

$$d(K,L) = \frac{|I| \cdot d(I,L) + |J| \cdot d(J,L)}{|I| + |J|}$$

***linkMethod* = weighted**

There is no general expression for the dissimilarity between clusters as it depends on the order in which the clusters are merged. At each step, new dissimilarities are computed as:

$$d(K,L) = \frac{d(I,L) + d(J,L)}{2}$$

***linkMethod* = centroid**

This method is suitable only for use with the Euclidean dissimilarity metric.

$$d(A,B) = \|\vec{c}_A - \vec{c}_B\|$$

where $\vec{c}$ c indicates the centroid of a cluster.

$$d(K,L) = \sqrt{\frac{|I| \cdot d(I,L)^2 + |J| \cdot d(J,L)^2}{|I| + |J|} - \frac{|I| \cdot |J| \cdot d(I,J)^2}{(|I| + |J|)^2}}$$

***linkMethod* = median**

This method is suitable only for use with the Euclidean dissimilarity metric.

assigns d(K,L) like the centroid method. When two clusters *I* and *J* are combined into a new cluster *K*, the average of centroids *i* and *j* give the new centroid *k*. Or,

$$d(K,L) = \sqrt{\frac{d(I,L)^2}{2} + \frac{d(J,L)^2}{2} - \frac{d(I,J)^2}{4}}$$

***linkMethod* = ward**

This method is suitable only for use with the Euclidean dissimilarity metric.

$$d(A,B) = \sqrt{\frac{2\,|A|\,|B|}{|A| + |B|}} \cdot \|\vec{c}_A - \vec{c}_B\|$$

The Ward variance minimization algorithm

$$d(K,L) = \sqrt{\frac{(|I| + |L|) \cdot d(I,L)^2 + (|J| + |L|) \cdot d(J,L)^2 - |L| \cdot d(I,J)^2}{|I| + |J| + |L|}}$$

Dendrogram Wave Format

The HCluster operation optionally produces a dendrogram output wave that can be used to create a dendrogram plot. This section describes the format of the dendrogram output wave.

The dendrogram wave contains all the information about the node pairs that are combined by HCluster, and the dissimilarity between the nodes. It also has a list of the indices into the original data in the order in which, for instance, labels should be drawn to avoid node connector lines that cross. The wave has four columns and N rows, where N is the number of original data vectors.

Column 0 and Column 1 contain the node numbers that are combined to form the node represented by a given row in the dendrogram wave. If a node number is negative, it represents the row in the original vector data. If a node number is positive, it represents the row number within the dendrogram wave of a combined node. Since negative numbers cannot represent zero-based indices, the row numbers are 1-based. If the value in column 0 or 1 is A, then the zero-based row number is abs(A)-1.

Column 2 contains the dissimilarity value between the nodes represented by columns 0 and 1. If you are drawing a dendrogram tree, a tie bar should be drawn in a location proportional to this dissimilarity.

Column 3 contains ordering information for the original data. This allows you to draw lines that don't cross. Labels for vector rows should also use this ordering. These numbers are zero-based indices into the original data. If you have a text wave with labels for each of the rows in your vector data, you can use this column to access the correct label for a given row in the dendrogram. If you include a false-color image (heat map) of the dissimilarity matrix or the vector data, the rows of the heat map data should be re-ordered based on column 3.

A data set with N vector rows has N-1 dissimilarity values. The information for column 3 has N values, one for each of the vector rows, so column 0 and column 1 contain NaN in row N to show that those cells are not used. The last row in column 2 has zero, which is a nonsense dissimilarity value. Only column 3 has useful data in the last row.

Dendrogram Example

Create a matrix wave containing fake data representing XY pairs with the X values in column 0 of the matrix and the Y values in column 1:

```
Make/N=(10,2)/O vectors
vectors[0][0]= {6.04379,-1.40976,5.32518,-0.140695,1.94087,0.971393,5.29093,-
                0.953138,5.58752,4.16877}
vectors[0][1]= {6.2448,-0.914027,4.36285,1.45288,-
                1.21538,1.21118,5.86274,1.37278,4.43314,4.40538}
```

Each row in vectors, that is, each XY pair, is one input vector to HCluster.

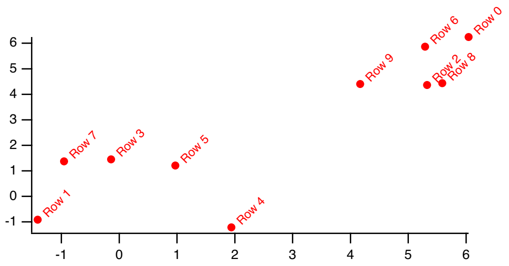
Create a text wave with labels for each of the vector rows:

```
Make/N=10/T/O labels
labels[0]= {"Row0","Row1","Row2","Row3","Row4","Row5","Row6","Row7","Row8","Row9"}
```

Make a graph of the XY pairs:

```
Display vectors[*][1] vs vectors[*][0]       // For displaying markers
AppendToGraph vectors[*][1] vs vectors[*][0] // For displaying labels
ModifyGraph margin(top)=27
ModifyGraph mode=3
ModifyGraph marker(vectors)=19
ModifyGraph textMarker(vectors#1)={labels,"default",0,45,5,15.00,12.00}
```

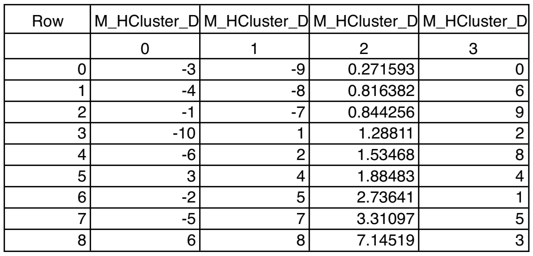
Since this is fake data, we have manipulated it so that there are two clear clusters. We used text markers to show which vector row each XY pair represents:



Invoke HCluster to analyze the vectors and create a dendrogram output wave:

```
HCluster/OTYP=Both vectors
```

The resulting dendrogram wave, with the default name M_HCluster_Dendrogram, has these contents:

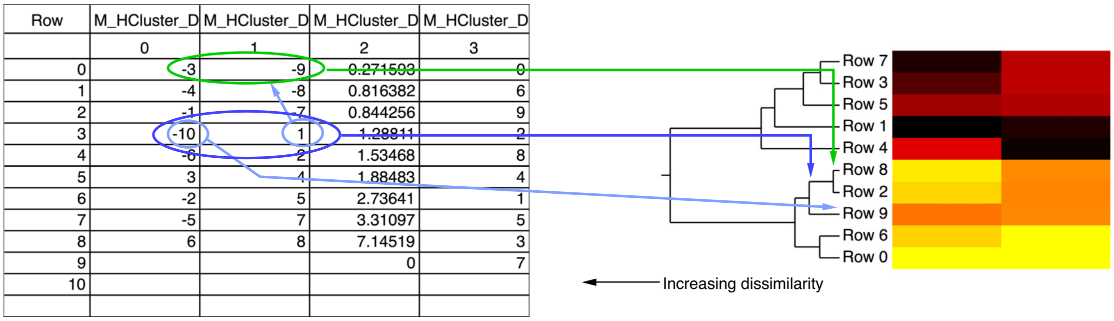


Row	M_HCluster_D	M_HCluster_D	M_HCluster_D	M_HCluster_D
	0	1	2	3
0	-3	-9	0.271593	0
1	-4	-8	0.816382	6
2	-1	-7	0.844256	9
3	-10	1	1.28811	2
4	-6	2	1.53468	8
5	3	4	1.88483	4
6	-2	5	2.73641	1
7	-5	7	3.31097	5
8	6	8	7.14519	3

The first row represents the smallest dissimilarity (0.2716). The first two columns indicate that a node has been formed by combining original vector rows 2 and 8 (that is abs(-3)-1 and abs(-9)-1). If we are drawing a dendrogram, we would position rows 2 and 8 at a position 4 and 5 up from the bottom (or down from the top). We see that from the fact that column 3 has row 2 and row 8 listed in those positions. We would draw a tie line between 2 and 8 at a position on the dendrogram proportional to a distance of 0.2716.

The first three rows have negative numbers in columns 0 and 1, showing that they all combine nodes representing the original vectors. If we have text labels for the rows, these nodes would all tie directly to those labels.

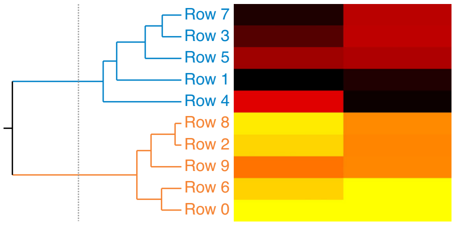
Row 3 lists nodes -10 and 1, indicating that it combines an original vector row 9 with the already combined node from row 0 of the dendrogram wave. The relationships between a dendrogram with a heat map of the original vector data, and the dendrogram wave is illustrated here:



Row	M_HCluster_D	M_HCluster_D	M_HCluster_D	M_HCluster_D
	0	1	2	3
0	-3	-9	0.271593	0
1	-4	-8	0.816382	6
2	-1	-7	0.844256	9
3	-10	1	1.28811	2
4	-6	2	1.53468	8
5	3	4	1.88483	4
6	-2	5	2.73641	1
7	-5	7	3.31097	5
8	6	8	7.14519	3
9			0	7
10				

Here we have ordered the vector rows from bottom to top following the ordering given in column 3 of the dendrogram wave. Rows 2 and 8 are tied together with the tie line least distant from the labels. Then Rows 3 and 7 and Rows 0 and 6 are each a bit more distant. Row 3 of the dendrogram wave indicates the tie between the label "Row 9" and the combined node from Row 2 and Row 8.

The two clusters are clearly indicated by the large distance between the most extreme tie line and the next most distant tie line. We might add a "cut line" in that space, and group the two clusters. To show it more clearly, we could color the two clusters differently, possibly like this:



We created this dendrogram with heat map using the Hierarchical Clustering package which you can access by choosing Analysis→Packages→Hierarchical Clustering.

Analysis Programming

This section contains data analysis programming examples. There are many more examples in the WaveMetrics Procedures, Igor Technical Notes, and Examples folders.

Passing Waves to User Functions and Macros

As you look through various examples you will notice two different ways to pass a wave to a function: using a Wave parameter or using a String parameter.

Using a Wave Parameter	Using a String Parameter

`Function Test1(w)`
`   Wave w`
	`Function Test2(wn)`
	`   String wn`

Usable in functions, not in macros.	Usable in functions and macros.
w is a “formal” name. Use it just as if it were the name of an actual wave.	Use the $ operator to convert from a string to wave name.

The string method is used in macros and in user functions for passing the name of a wave that may not yet exist but will be created by the called procedure. The wave parameter method is used in user functions when the wave will always exist before the function is called. For details, see **Accessing Waves in Functions** on page IV-82.

Returning Created Waves from User Functions

A function can return a wave as the function result. For example:

```
Function Test()
   Wave w = CreateNoiseWave(5, "theNoiseWave")
   WaveStats w
   Display w as "Noise Wave"
End

Function/WAVE CreateNoiseWave(noiseValue, destWaveName)
   Variable noiseValue
   String destWaveName

   Make/O $destWaveName = gnoise(noiseValue)
   Wave w = $destWaveName
   return w
End
```

If the returned wave is intended for temporary use, you can create it as a free wave:

```
Function Test()
   Wave w = CreateFreeNoiseWave(5)// w is a free wave
   WaveStats w
   // w is killed when the function exist
End

Function/WAVE CreateFreeNoiseWave(noiseValue)
   Variable noiseValue

   Make/O/FREE aWave = gnoise(noiseValue)
   return aWave
End
```

To return multiple waves, you can return a "wave reference wave". See **Wave Reference Waves** on page IV-77 for details.

Writing Functions that Process Waves

The user function is a powerful, general-purpose analysis tool. You can do practically any kind of analysis. However, complex analyses require programming skill and patience.

It is useful to think about an analysis function in terms of its input parameters, its return value and any side effects it may have. By return value, we mean the value that the function directly returns. For example, a function might return a mean or an area or some other characteristic of the input. By side effects, we mean changes that the function makes to any objects. For example, a function might change the values in a wave or create a new wave.

This table shows some of the common types of analysis functions. Examples follow.

Input Parameters	Return Value	Side Effects	Example Function
A source wave	A number	None	WaveSum
A source wave	Not used	The source wave is modified	RemoveOutliers
A source wave	Wave reference	A new destination wave is created	LogRatio
A source wave and a destination wave	Not used	The destination wave is modified	
An array of wave references	A number	None	WavesMax
An array of wave references	Wave reference	A new wave is created	WavesAverage

The following example functions are intended to show you the general form for some common analysis function types. We have tried to make the examples useful while keeping them simple.

WaveSum Example

Input:	Source wave
Return value:	Number
Side effects:	None

```
// WaveSum(w)
// Returns the sum of the entire wave, just like Igor’s sum function.
Function WaveSum(w)
   Wave w

   Variable i, n=numpnts(w), total=0
   for(i=0;i<n;i+=1)
      total += w[i]
   endfor

   return total
End
```

To use this, you would execute something like

```
Print "The sum of wave0 is:", WaveSum(wave0)
```

RemoveOutliers Example

Input:	Source wave
Return value:	Number
Side effects:	Source wave is modified

Often a user function used for number-crunching needs to loop through each point in an input wave. The following example illustrates this.

```
// RemoveOutliers(theWave, minVal, maxVal)
// Removes all points in the wave below minVal or above maxVal.
// Returns the number of points removed.
Function RemoveOutliers(theWave, minVal, maxVal)
   Wave theWave
   Variable minVal, maxVal

   Variable i, numPoints, numOutliers
   Variable val
   numOutliers = 0
   numPoints = numpnts(theWave)        // number of times to loop

   for (i = 0; i < numPoints; i += 1)
      val = theWave[i]
      if ((val < minVal) || (val > maxVal))  // is this an outlier?
         numOutliers += 1
      else                                   // if not an outlier
         theWave[i - numOutliers] = val      // copy to input wave
      endif
   endfor

   // Truncate the wave
   DeletePoints numPoints-numOutliers, numOutliers, theWave
   return numOutliers
End
```

To test this function, try the following commands.

```
Make/O/N=10 wave0= gnoise(1); Edit wave0
Print RemoveOutliers(wave0, -1, 1), "points removed"
```

RemoveOutliers uses the for loop to iterate through each point in the input wave. It uses the built-in numpnts function to find the number of iterations required and a local variable as the loop index. This is a very common practice.

The line "`if ((val < minVal) || (val > maxVal))`" decides whether a particular point is an outlier. `||` is the logical OR operator. It operates on the logical expressions "`(val < minVal)`" and "`(val > maxVal)`". This is discussed in detail under **Bitwise and Logical Operators** on page IV-41.

To use the WaveMetrics-supplied RemoveOutliers function, include the Remove Points.ipf procedure file:

```
#include <Remove Points>
```

See **The Include Statement** on page IV-166 for instructions on including a procedure file.

LogRatio Example

Input:	Source waves
Return value:	Wave reference
Side effects:	Output wave created

```
// LogRatio(source1, source2, outputWaveName)
// Creates a new wave that is the log of the ratio of input waves.
// Returns a reference to the output wave.
Function/WAVE LogRatio(source1, source2, outputWaveName)
   Wave source1, source2
   String outputWaveName

   Duplicate/O source1, $outputWaveName
   WAVE wOut = $outputWaveName
   wOut = log(source1/source2)
   return wOut
End
```

To call this from a function, you would execute something like:

```
Wave wRatio = LogRatio(wave0, wave1, "ratio")
Display wRatio
```

The LogRatio function illustrates how to treat input and output waves. Input waves must exist before the function is called and therefore are passed as wave reference parameters. The output wave may or may not already exist when the function is called. If it does not yet exist, it is not possible to create a wave reference for it. Therefore we pass to the function the desired name for the output wave using a string parameter. The function creates or overwrites a wave with that name in the current data folder and returns a reference to the output wave.

WavesMax Example

Input:	Array of wave references
Return value:	Number
Side effects:	None

```
// WavesMax(waves)
// Returns the maximum value in waves in the waves array.
Function WavesMax(waves)
   Wave/WAVE waves

   Variable theMax = -INF

   Variable numWaves = numpnts(waves)
   Variable i
   for(i=0; i<numWaves; i+=1)
      Wave w = waves[i]
      Variable tmp = WaveMax(w)
      theMax = max(tmp, theMax)
   endfor

   return theMax
End
```

This function illustrates how you might call WavesMax from another function:

```
Function DemoWavesMax()
   Make/FREE/N=10 w0 = p
   Make/FREE/N=10 w1 = p + 1

   Make/FREE/WAVE waves = {w0, w1}

   Variable theMax = WavesMax(waves)

   Printf "The maximum value is %g\r", theMax
End
```

WavesAverage Example

Input:	An array of wave references
Return value:	Wave reference
Side effects:	Creates output wave

```
// WavesAverage(waves, outputWaveName)
// Returns a reference to a new wave, each point of which contains the
// average of the corresponding points of a number of source waves.
// waves is assumed to contain at least one wave reference and
// all waves referenced by waves are expected to have the same
// number of points.
Function/WAVE WavesAverage(waves, outputWaveName)
   Wave/WAVE waves         // A wave containing wave references
   String outputWaveName
   // Make output wave based on the first source wave
   Wave first = waves[0]
   Duplicate/O first, $outputWaveName
   Wave wOut = $outputWaveName
   wOut = 0

   Variable numWaves = numpnts(waves)
   Variable i
   for(i=0; i<numWaves; i+=1)
      Wave source = waves[i]
      wOut += source          // Add source to output
   endfor

   wOut /= numWaves           // Divide by number of waves

   return wOut
End
```

This function shows how you might call WavesAverage from another function:

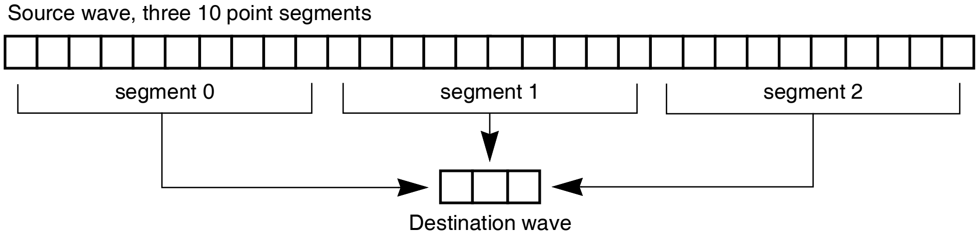
```
Function DemoWavesAverage()
   Make/FREE/N=10 w0 = p
   Make/FREE/N=10 w1 = p + 1

   Make/FREE/WAVE waves = {w0, w1}

   Wave wAverage = WavesAverage(waves, "averageOfWaves")
   Display wAverage
End
```

Finding the Mean of Segments of a Wave

An Igor user who considers each of his waves to consist of a number of segments with some number of points in each segment asked us how he could find the mean of each of these segments. We wrote the FindSegmentMeans function to do this.

```
Function/WAVE FindSegmentMeans(source, n)
   Wave source
   Variable n

   String dest                    // name of destination wave
   Variable segment, numSegments
   Variable startX, endX, lastX

   dest = NameOfWave(source)+"_m"  // derive name of dest from source
   numSegments = trunc(numpnts(source) / n)
   if (numSegments < 1)
      DoAlert 0, "Destination must have at least one point"
      return $""                     // Null wave reference
   endif

   Make/O/N=(numSegments) $dest
   WAVE destw = $dest
   for (segment = 0; segment < numSegments; segment += 1)
      startX = pnt2x(source, segment*n)      // start X for segment
      endX = pnt2x(source, (segment+1)*n - 1)// end X for segment
      destw[segment] = mean(source, startX, endX)
   endfor

   return destw
End
```

This diagram illustrates a source wave with three ten-point segments and a destination wave that will contain the mean of each of the source segments. The FindSegmentMeans function makes the destination wave.



To test FindSegmentMeans, try the following commands.

```
Make/N=100 wave0=p+1; Edit wave0
FindSegmentMeans(wave0,10)
Append wave0_m
```

The loop index is the variable "segment". It is the segment number that we are currently working on, and also the number of the point in the destination wave to set.

Using the segment variable, we can compute the range of points in the source wave to work on for the current iteration: segment*n up to (segment+1)*n - 1. Since the mean function takes arguments in terms of a wave's X values, we use the pnt2x function to convert from a point number to an X value.

If it is guaranteed that the number of points in the source wave is an integral multiple of the number of points in a segment, then the function can be speeded up and simplified by using a waveform assignment statement in place of the loop. Here is the statement.

```
destw = mean(source, pnt2x(source,p*n), pnt2x(source,(p+1)*n-1))
```

The variable p, which Igor automatically increments as it evaluates successive points in the destination wave, takes on the role of the segment variable used in the loop. Also, the startX, endX and lastX variables are no longer needed.

Using the example shown in the diagram, p would take on the values 0, 1 and 2 as Igor worked on the destination wave. n would have the value 10.

Working with Mismatched Data

Occasionally, you may find yourself with several sets of data each sampled at a slightly different rate or covering a different range of the independent variable (usually time). If all you want to do is create a graph showing the relationship between the data sets then there is no problem.

However, if you want to subtract one from another or do other arithmetic operations then you will need to either:

- Create representations of the data that have matching X values. Although each case is unique, usually you will want to use the **Interpolate2** operation (see **Using the Interpolate2 Operation** on page III-111) or the interp function (see **Using the Interp Function** on page III-110) to create data sets with common X values. You can also use the **Resample** to create a wave to match another.
- Properly set each wave's X scaling, and perform the waveform arithmetic using X scaling values and Igor's automatic linear interpolation. See **Mismatched Waves** on page II-83.

The WaveMetrics procedure file Wave Arithmetic Panel uses these techniques to perform a variety of operations on data in waves. You can access the panel by choosing Packages→Wave Arithmetic from the Analysis menu. This will open the procedure file and display the control panel. Click the help button in the panel to learn how to use it.

References

Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

Reinsch, Christian H., Smoothing by Spline Functions, *Numerische Mathematic, 10*, 177-183, 1967.

Chapter

III-8

Curve Fitting

Overview

Igor Pro’s curve fitting capability is one of its strongest analysis features. Here are some of the highlights:

- Linear and general nonlinear curve fitting.
- Fit by ordinary least squares, or by least orthogonal distance for errors-in-variables models.
- Fit to implicit models.
- Built-in functions for common fits.
- Automatic initial guesses for built-in functions.
- Fitting to user-defined functions of any complexity.
- Fitting to functions of any number of independent variables, either gridded data or multicolumn data.
- Fitting to a sum of fit functions.
- Fitting to a subset of a waveform or XY pair.
- Produces estimates of error.
- Supports weighting.

The idea of curve fitting is to find a mathematical model that fits your data. We assume that you have theoretical reasons for picking a function of a certain form. The curve fit finds the specific coefficients which make that function match your data as closely as possible.

You cannot use curve fitting to find which of thousands of functions fit a data set.

People also use curve fitting to merely show a smooth curve through their data. This sometimes works but you should also consider using smoothing or interpolation, which are described in Chapter III-7, **Analysis**.

You can fit to three kinds of functions:

- Built-in functions.
- User-defined functions.
- External functions (XFUNCs).

The built-in fitting functions are line, polynomial, sine, exponential, double-exponential, Gaussian, Lorentzian, Hill equation, sigmoid, lognormal, log, Gauss2D (two-dimensional Gaussian peak) and Poly2D (two-dimensional polynomial).

You create a user-defined function by entering the function in the New Fit Function dialog. Very complicated functions may have to be entered in the Procedure window.

External functions, XFUNCs, are written in C or C++. To create an XFUNC, you need the optional Igor XOP Toolkit and a C/C++ compiler. You don’t need the toolkit to *use* an XFUNC that you get from WaveMetrics or from another user.

Curve fitting works with equations of the form $y = f(x_1, x_2, \ldots, x_n)$; although you can fit functions of any number of independent variables (the x_n’s) most cases involve just one. For more details on multivariate fitting, see **Fitting to a Multivariate Function** on page III-200.

You can also fit to implicit functions; these have the form$f(x_1, x_2, \ldots, x_n) = 0$. See **Fitting Implicit Functions** on page III-242.

You can do curve fits with linear constraints (see **Fitting with Constraints** on page III-227).

Curve Fitting Terminology

Built-in fits are performed by the CurveFit operation. User-defined fits are performed by the FuncFit or FuncFitMD operation. We use the term “curve fit operation” to stand for CurveFit, FuncFit, or FuncFitMD, whichever is appropriate.

Fitting to an external function works the same as fitting to a user-defined function (with some caveats concerning the Curve Fitting dialog — see **Fitting to an External Function (XFUNC)** on page III-194).

If you use the Curve Fitting dialog, you don't really need to know much about the distinction between built-in and user-defined functions. You may need to know a bit about the distinction between external functions and other types. This will be discussed later.

We use the term "coefficients" for the numbers that the curve fit is to find. We use the term "parameters" to talk about the values that you pass to operations and functions.

Overview of Curve Fitting

In curve fitting we have raw data and a function with unknown coefficients. We want to find values for the coefficients such that the function matches the raw data as well as possible. The "best" values of the coefficients are the ones that minimize the value of Chi-square. Chi-square is defined as:

$$\sum_i \left(\frac{y - y_i}{\sigma_i}\right)^2$$

where y is a fitted value for a given point, y_i is the measured data value for the point and σ_i is an estimate of the standard deviation for y_i.

The simplest case is fitting to a straight line: $y = ax + b$. Suppose we have a theoretical reason to believe that our data should fall on a straight line. We want to find the coefficients a and b that best match our data.

For a straight line or polynomial function, we can find the best-fit coefficients in one step. This is noniterative curve fitting, which uses the singular value decomposition algorithm for polynomial fits.

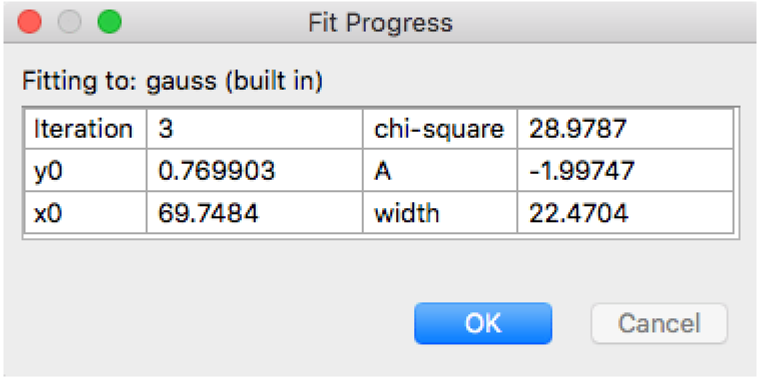
Iterative Fitting

For the other built-in fitting functions and for user-defined functions, the operation is iterative as the fit tries various values for the unknown coefficients. For each try, it computes chi-square searching for the coefficient values that yield the minimum value of chi-square.

The Levenberg-Marquardt algorithm is used to search for the coefficient values that minimize chi-square. This is a form of nonlinear, least-squares fitting.

As the fit proceeds and better values are found, the chi-square value decreases. The fit is finished when the rate at which chi-square decreases is small enough.

During an iterative curve fit, you will see the Curve Fit progress window. This shows you the function being fit, the updated values of the coefficients, the value of chi-square, and the number of passes.



Fitting to: gauss (built in)

Iteration	3	chi-square	28.9787
y0	0.769903	A	-1.99747
x0	69.7484	width	22.4704

Normally you will let the fit proceed until completion when the Quit button is disabled and the OK button is enabled. When you click OK, the results of the fit are written in the history area.

If the fit has gone far enough and you are satisfied, you can click the Quit button, which finishes the iteration currently under way and then puts the results in the history area as if the fit had completed on its own.

Sometimes you can see that the fit is not working, e.g., when chi-square is not decreasing or when some of the coefficients take on very large nonsense values. You can abort it by pressing the **User Abort Key Combinations**, which discards the results of the fit. You will need to adjust the fitting coefficients and try again.

Initial Guesses

The Levenberg-Marquardt algorithm is used to search for the minimum value of chi-square. Chi-square defines a surface in a multidimensional error space. The search process involves starting with an initial guess at the coefficient values. Starting from the initial guesses, the fit searches for the minimum value by travelling down hill from the starting point on the chi-square surface.

We want to find the deepest valley in the chi-square surface. This is a point on the surface where the coefficient values of the fitting function minimize, in the least-squares sense, the difference between the experimental data and fit data. Some fitting functions may have only one valley. In this case, when the bottom of the valley is found, the best fit has been found. Some functions, however, may have multiple valleys, places where the fit is better than surrounding values, but it may not be the best fit possible.

When the fit finds the bottom of a valley it concludes that the fit is complete even though there may be a deeper valley elsewhere on the surface. Which valley is found first depends on the initial guesses.

For built-in fitting functions, you can automatically set the initial guesses. If this produces unsatisfactory results, you can try manual guesses. For fitting to user-defined functions you must supply manual guesses.

Termination Criteria

A curve fit will terminate after 40 passes in searching for the best fit, but will quit if 9 passes in a row produce no decrease in chi-square. This can happen if the initial guesses are so good that the fit starts at the minimum chi-square. It can also happen if the initial guesses are way off or if the function does not fit the data at all.

You can change the 40-pass limit. See the discussion of V_FitMaxIters under **Special Variables for Curve Fitting** on page III-232. Usually needing more than 40 passes is a sign of trouble with the fit. See **Identifiability Problems** on page III-226.

Unless you know a great deal about the fitting function and the data, it is unwise to assume that a solution is a good one. In almost all cases you will want to see a graph of the solution to compare the solution with the data. You may also want to look at a graph of the residuals, the differences between the fitted model and the data. Igor makes it easy to do both in most cases.

Errors in Curve Fitting

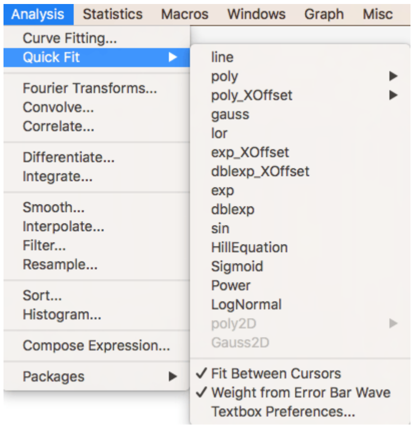
In certain cases you may encounter a situation in which it is not possible to decide where to go next in searching for the minimum chi-square. This results in a “singular matrix” error. This is discussed under **Singularities in Curve Fitting** on page III-265. **Curve Fitting Troubleshooting** on page III-266 can help you find the solution to the problem.

Data for Curve Fitting

You must have measured values of both the dependent variable (usually called “y”) and the independent variables (usually called “x” especially if there is just one). These are sometimes called the “response variable” and “explanatory variables.” You can do a curve fit to waveform data or to XY data. That is, you can fit data contained in a single wave, with the data values in the wave representing the Y data and the wave’s X scaling representing equally-spaced X data. Or you can fit data from two (or more) waves in which the data values in one wave represent the Y values and the data values in another wave represent the X data. In this case, the data do not need to be equally spaced. In fact, the X data can be in random order.

You can read more about waveform and XY data in Chapter II-5, **Waves**.

Curve Fitting Using the Quick Fit Menu



The Quick Fit menu is the easiest, fastest way to do a curve fit.

The Quick Fit menu gives you quick access to curve fits using the built-in fitting functions. The data to be fit are determined by examining the top graph; if a single trace is found, the graphed data is fit to the selected fitting function. If the graph contains more than one trace a dialog is presented to allow you to select which trace should be fit.

The graph contextual menu also gives access to the Quick Fit menu. If you Control-click (*Macintosh*) or right-click (*Windows*) on a trace in a graph, you will see a Quick Fit item at the bottom of the resulting contextual menu. When you access the Quick Fit menu this way, it automatically fits to the trace you clicked on. This gives you a way to avoid the dialog that Quick Fit uses to select the correct trace when there is more than one trace on a graph.

When you use the Quick Fit menu, a command is generated to perform the fit and automatically add the model curve to the graph. By default, if the graph cursors are present, only the data between the cursors is fit. You can do the fit to the entire data set by selecting the Fit Between Cursors item in the Quick Fit menu in order to uncheck the item. When unchecked, fits are done disregarding the graph cursors.

If the trace you are fitting has error bars and the data for the error bars come from a wave, Quick Fit will use the wave as a weighting wave for the fit. Note that this assumes that your error bars represent one standard deviation. If your error wave represents more than one standard deviation, or if it represents a confidence interval, you should not use it for weighting. You can select the Weight from Error Bar Wave item to unmark it, preventing Igor from using the error bar wave for weighting.

By default, a report of curve fit results is printed to the history. If you select Textbox Preferences, the Curve Fit Textbox Preferences dialog is displayed. It allows you to specify that a textbox be added to your graph containing most of the information that is printed in the history. You can select various components of the information by selecting items in the Dialog.

In the screen capture above, the poly2D and Gauss2D fit functions are not available because the top graph does not contain a contour plot or image plot, in which case the fitting functions would be available.

For a discussion of the built-in fit functions, see **Built-in Curve Fitting Functions** on page III-206.

Limitations of the Quick Fit Menu

The Quick Fit menu does not give you access to the full range of curve fitting options available to you. It does not give you access to user-defined fitting functions, automatic residual calculation, masking, or confidence interval analysis. A Quick Fit always uses automatic guesses; if the automatic guesses don't work, you must use the Curve Fitting dialog to enter manual guesses.

If your graph displays an image that uses auxiliary X and Y waves to set the image pixel sizes, Quick Fit will not be able to do the fit. This is because these waves for an image plot have an extra point that makes them unsuitable for fitting. A contour plot uses X and Y waves that set the centers of the data, and these can be used for fitting. Quick Fit will do the right thing with such a contour plot.

Using the Curve Fitting Dialog

If you want options that are not available via the Quick Fit menu, the next easiest way to do a fit is to choose Curve Fitting from the Analysis menu. This displays the Curve Fitting dialog, which presents an interface for selecting a fitting function and data waves, and for setting various curve fitting options. You can use the dialog to enter initial guesses if necessary. The Curve Fitting dialog can also be used to create a new user-defined fitting function.

Most curve fits can be accomplished using the Curve Fitting dialog. If you need to do many fits using the same fit function fitting to numerous data sets you will probably want to write a procedure in Igor's programming language to do the job.

The facility for creating a user-defined fitting function using the Curve Fitting dialog will handle most common cases, but is probably not the best way to create a very complex fitting function. In such cases, you will need to write a fitting function in a procedure window. This is described later under **User-Defined Fitting Functions** on page III-250.

Some very complicated user-defined fitting functions may not work well with the Curve Fitting dialog. In some cases, you may need to write the fitting function in the Procedure window, and then use the dialog to set up and execute the fit. In other cases it may be necessary to enter the operation manually using either a user procedure or by typing on the command line. These cases should be quite rare.

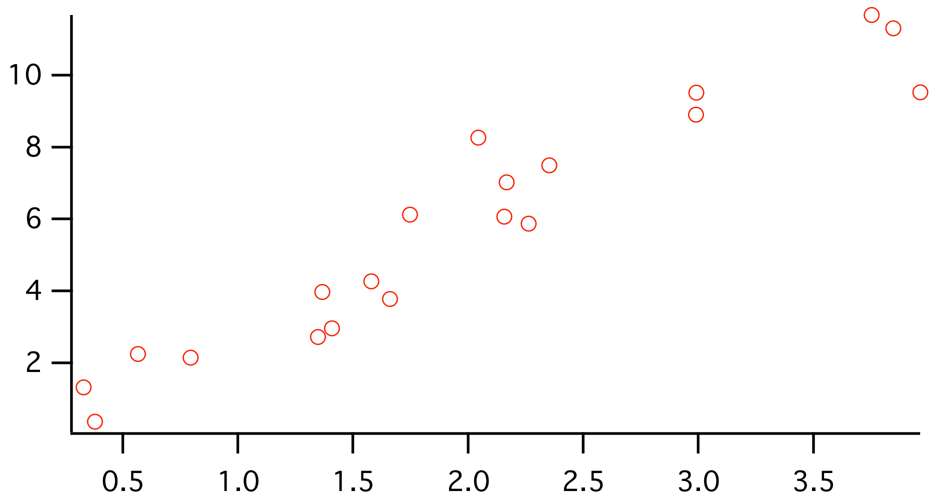
A Simple Case — Fitting to a Built-In Function: Line Fit

To get started, we will cover fitting to a simple built-in fit: a line fit. You may have a theoretical reason to believe that your data should be described by the function $y = ax + b$. You may simply have an empirical observation that the data appear to fall along a line and you now want to characterize this line. It's better if you have a theoretical justification, but we're not all that lucky.

The Curve Fitting dialog is organized into four tabs. Each tab contains controls for some aspect of the fitting operation. Simple fits to built-in functions using default options will require only the Function and Data tab.

We will go through the steps necessary to fit a line to data displayed in a graph. Other built-in functions work much the same way.

You might have data displayed in a graph like this:



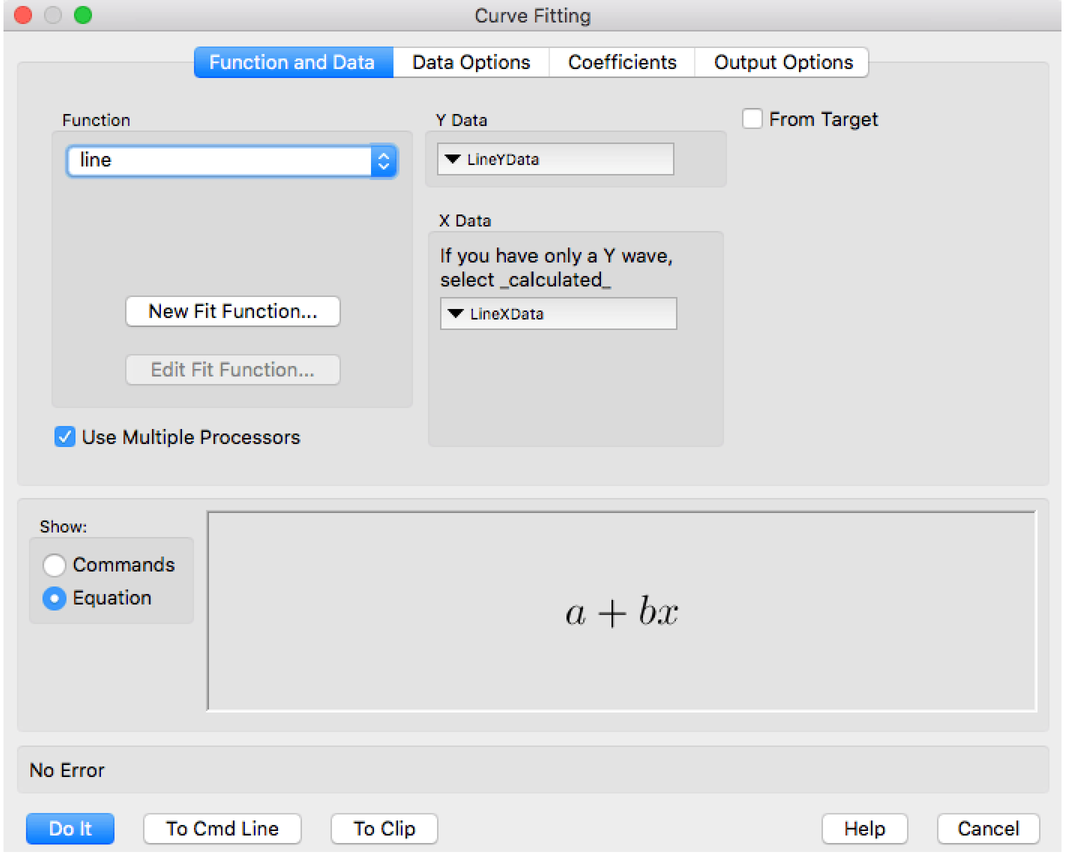
Now you wish to find the best-fitting line for this data. The following commands will make a graph like this one. The SetRandomSeed command is used so that the "random" scatter produced by the enoise function will be the same as shown above. If you would like to perform the actions yourself as you read the manual, you can make the data shown here and the graph by typing these commands on the command line:

```
Make/N=20/D LineYData, LineXData
SetRandomSeed 0.5   // So the example always makes the same "random" numbers
LineXData = enoise(2)+2             // enoise makes random numbers
LineYData = LineXData*3+gnoise(1)   // so does gnoise
Display LineYData vs LineXData
ModifyGraph mode=3,marker=8
```

The first line makes two waves to receive our “data”. The second line sets the seed for Igor's pseudo-random number generators, resulting in reproducible noise. The third line fills the X wave with uniformly-distributed random numbers in the range of zero to four. The fourth line fills the Y wave with data that falls on a line having a slope of three and passing through the origin, with some normally-distributed noise added. The final two lines make the graph and set the display to markers mode with open circles as the marker.

Choosing the Function and Data

You display the Curve Fitting dialog by choosing Curve Fitting from the Analysis menu. If you have not used the dialog yet, it looks like this, with the Function and Data tab showing:



Function: line

Y Data: LineYData

☐ From Target

X Data

If you have only a Y wave, select _calculated_

LineXData

☑ Use Multiple Processors

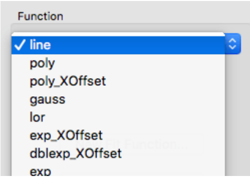
Show:

○ Commands

● Equation

$$a + bx$$

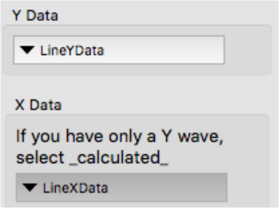
No Error



The first step in doing a curve fit is to choose a fit function. We are doing a simple line fit, so pop up the Function menu and choose “line”.

Select the Y data from the Y Data menu. If you have waveform data, be sure that the X data menu has “_Calculated_” selected.

If you have separate X and Y data waves, you must select the X wave in the X Data menu. Only waves having the same number of data points as the Y wave are shown in this menu. A mismatch in the number of points is usually the problem if you don’t see your X wave in the menu.



Y Data: LineYData

X Data

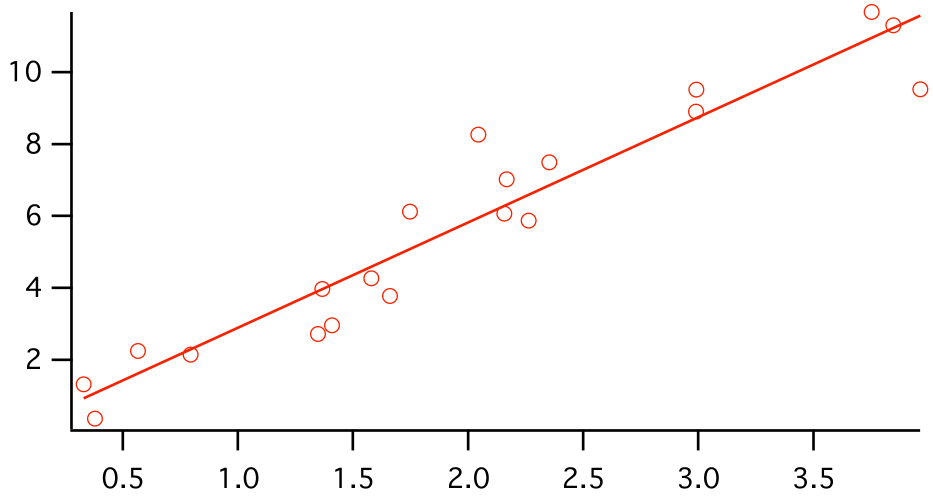
If you have only a Y wave, select _calculated_

LineXData

For the line fit example, we select LineYData from the Y Data menu, and LineX-Data from the X Data menu.

If you have a large number of waves in your experiment, it may be easier if you select the From Target checkbox. When it is selected only waves from the top graph or table are shown in the Y and X wave menus, and an attempt is made to select wave pairs used by a trace on the graph.

At this point, everything is set up to do the fit. For this simple case it is not necessary to visit the other tabs in the dialog. When you click Do It, the fit proceeds. The line fit example graph winds up looking like this:



In addition to the model line shown on the graph, various kinds of information appears in the history area:

```
•CurveFit line LineYData /X=LineXData /D
   fit_LineYData= W_coef[0]+W_coef[1]*x
   W_coef={-0.037971,2.9298}
   V_chisq= 18.25; V_npnts= 20; V_numNaNs= 0; V_numINFs= 0;
   V_startRow= 0;V_endRow= 19;V_q= 1;V_Rab= -0.879789;
   V_Pr= 0.956769;V_r2= 0.915408;
   W_sigma={0.474,0.21}
   Coefficient values ± one standard deviation
      a    =-0.037971 ± 0.474
      b    =2.9298 ± 0.21
```

Command line generated by the dialog.

This line can be copied and used to reevaluate the model curve.

Fit coefficients as a wave.

Standard deviations of the Fit coefficients as a wave.

Coefficient values in a list using the names shown in the dialog.

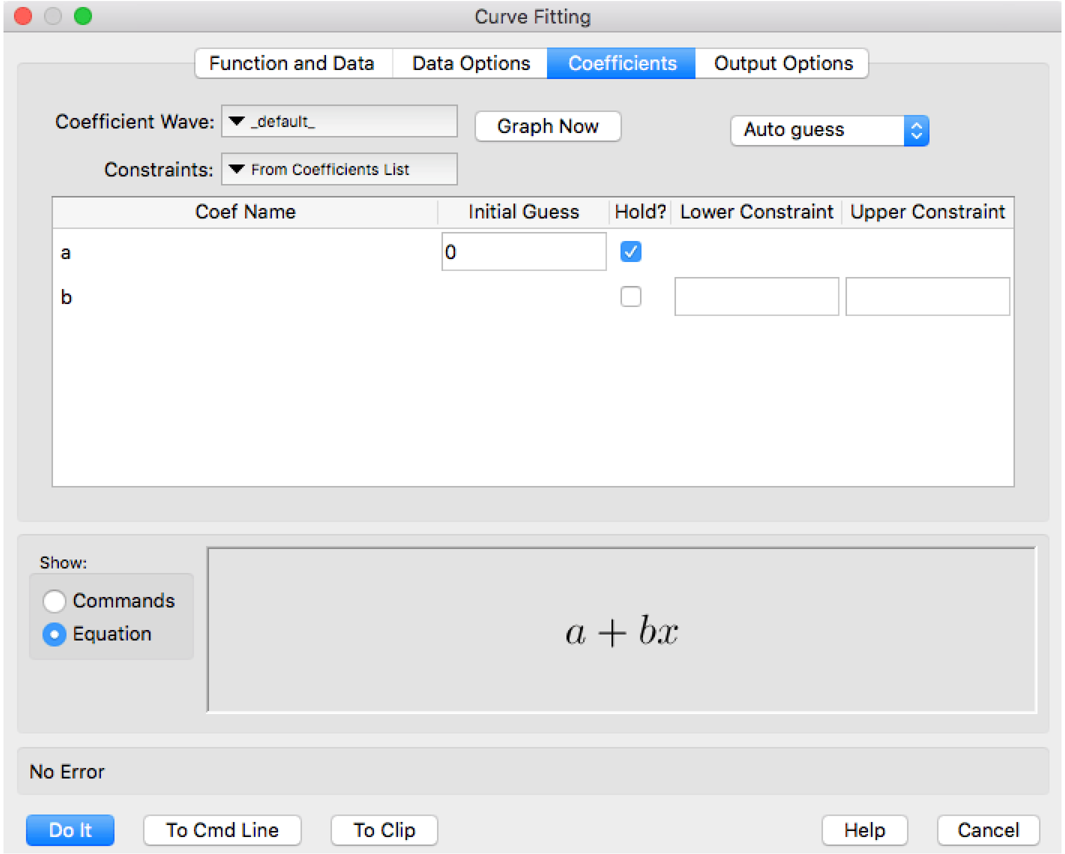
Two Useful Additions: Holding a Coefficient and Generating Residuals

Well, you've done the fit and looked at the graph and you decide that you have reason to believe that the line should go through the origin. Because of the scatter in the measured Y values, the fit line misses the origin. The solution is to do the fit again, but with the Y intercept coefficient held at a value of zero.

You might also want to display the residuals as a visual check of the fit.

Bring up the dialog again. The dialog remembers the settings you used last time, so the line fit function is already chosen in the Function menu, and your data waves are selected in the Y Data and X Data menus.

Select the Coefficients tab. Each of the coefficients has a row in the Coefficients list:



Coefficient Wave: _default_

Constraints: From Coefficients List

Auto guess

Coef Name	Initial Guess	Hold?	Lower Constraint	Upper Constraint
a	0	☑		
b		□		

Show:

○ Commands

● Equation

$$a + bx$$

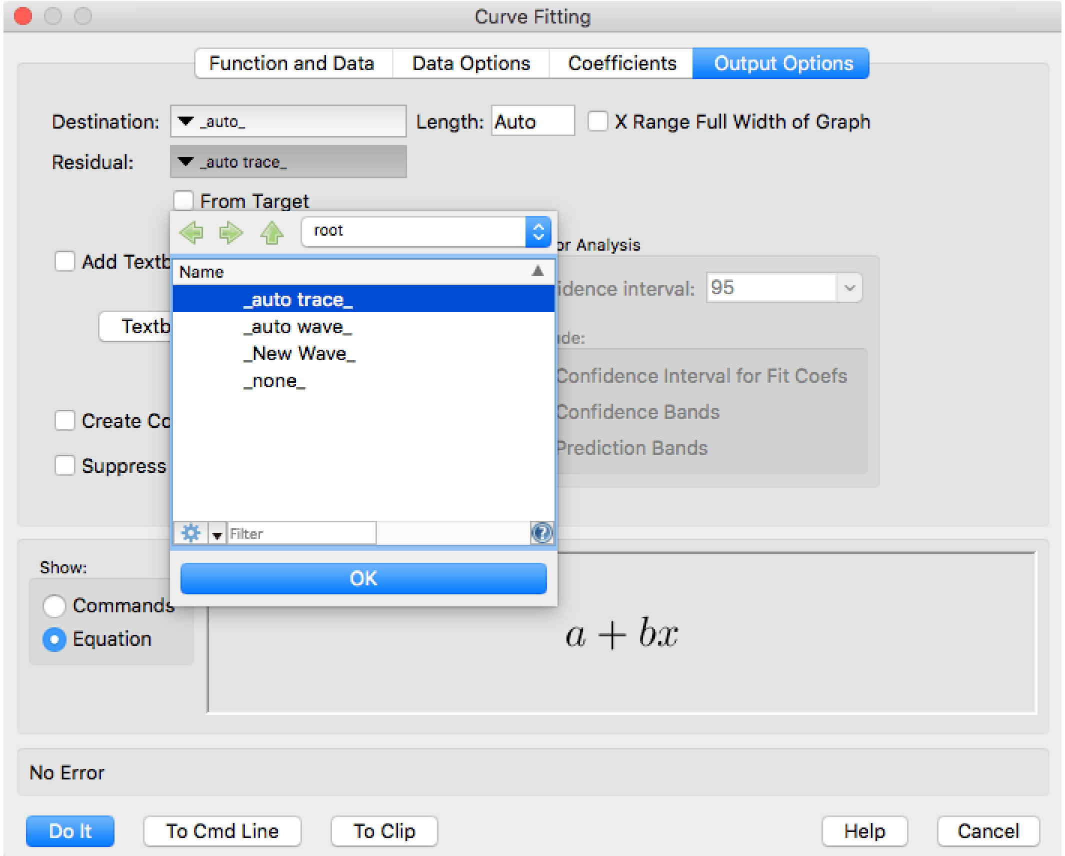
No Error

Click the checkbox in the column labeled “Hold?” to hold the value of that coefficient.

To specify a coefficient value, fill in the corresponding box in the Initial Guess column. Until you select the Hold box the initial guess box is not available because built-in fits don’t require initial guesses.

To fill in a value, click in the box. You can now type a value. When you have finished, press Enter (*Windows*) or Return (*Macintosh*) to exit editing mode for that box.

Now we want to calculate the fit residuals and add them to the graph. Click the Output Options tab and choose _auto trace_ from the Residual menu:



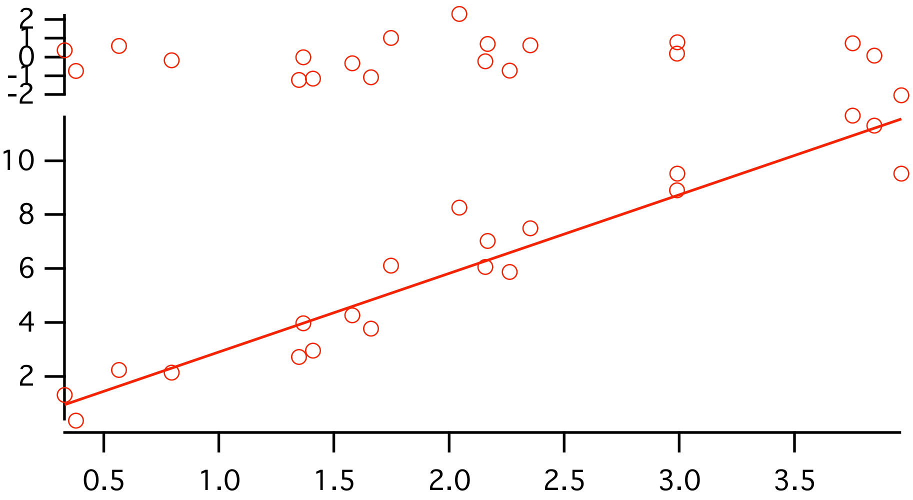
Destination: _auto_
Length: Auto
☐ X Range Full Width of Graph
Residual: _auto trace_
☐ From Target
Confidence interval: 95
○ Commands
● Equation

$a + bx$

No Error

There are a number of options for the residual. We chose _auto trace_ to calculate the residual and add it to the graph. You may not always want the residuals added to your graph; choose _auto wave_ to automatically calculate the residuals but *not* display them on your graph. Both _auto trace_ and _auto wave_ create a wave with the same name as your Y wave with “Res_” prefixed to the name. Choosing _New Wave_ generates commands to make a new wave with your choice of name to fill with residuals. It is not added to your graph.

Now when we click Do It, the fit is recalculated with *a* held at zero so that the line passes through the origin. Residuals are calculated and added to the graph:



Note that the line on the graph doesn't cross the vertical axis at zero, because the horizontal axis doesn't extend to zero.

Holding *a* at zero, the result of the fit printed in the history is:

The /H flag shows that one or more coefficients are held.

```
•K0 = 0;
•CurveFit/H="10" line LineYData /X=LineXData /D /R
   fit_LineYData= W_coef[0]+W_coef[1]*x
   Res_LineYData= LineYData[p] - (W_coef[0]+W_coef[1]*LineXData[p])
   W_coef={0,2.915}
   V_chisq= 18.2565; V_npnts= 20; V_numNaNs= 0; V_numINFs= 0;
   V_startRow= 0; V_endRow= 19; V_q= 1; V_Rab= 0; V_Pr= 0.956769;
   V_r2= 0.906186;
   W_sigma={0,0.0971}
   Coefficient values ± one standard deviation
      a    =   0 ± 0
      b    =   2.915 ± 0.0971
```

a is zero because it was held.

Automatic Guesses Didn't Work

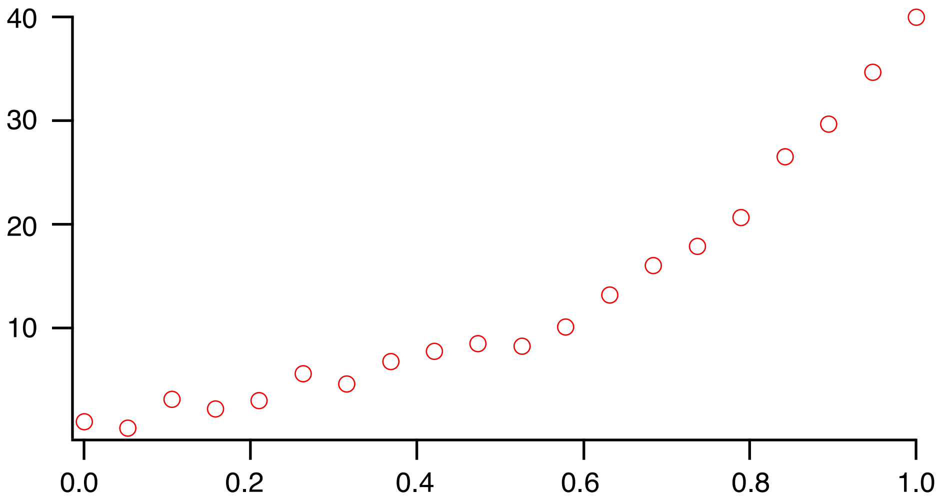
Most built-in fits will work just like the line fit. You simply choose a function from the Function menu, choose your data wave (or waves if you have both X and Y waves) and select output options on the Output Options tab. For built-in fits you don't need the Coefficients tab unless you want to hold a coefficient.

In a few cases, however, automatic guesses don't work. Then you must use the Coefficient tab to set your own initial guesses. One important case in which this is true is if you are trying to fit a growing exponential, $y = ae^{bx}$, where *b* is positive.

Here are commands to create an example for this section. Once again, you may wish to enter these commands on the command line to follow along:

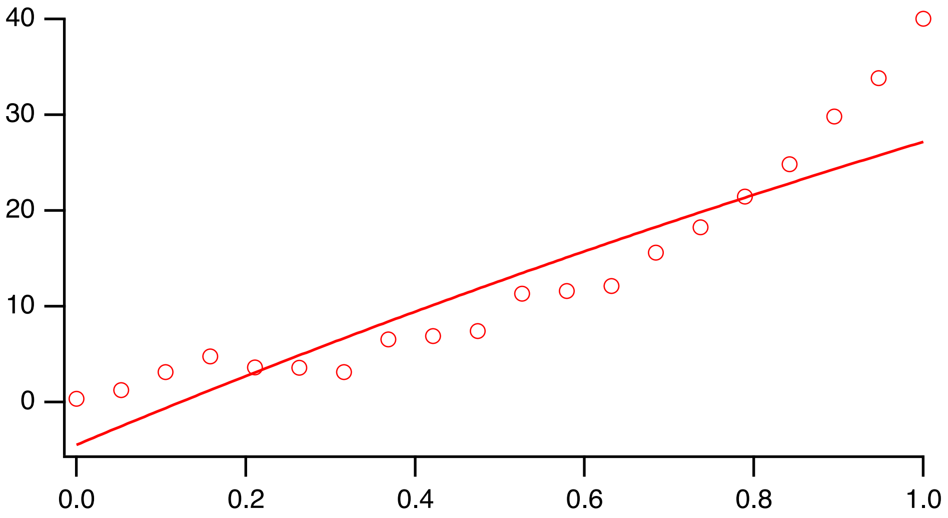
```
Make/N=20 RisingExponential
SetScale/I x 0,1,RisingExponential
SetRandomSeed 0.5
RisingExponential = 2*exp(3*x)+gnoise(1)
Display RisingExponential
ModifyGraph mode=3,marker=8
```

These commands make a 20-point wave, set its X scaling to cover the range from 0 to 1, fill it with exponential values plus a bit of noise, and make a graph:



The first-cut trial to fitting an exponential function is to select exp from the Function menu and the RisingExponential wave in the Y Data menu (if you are continuing from the previous section, you may need to go to the Coefficients tab and un-hold the y0 coefficient, and to the Output Options tab and de-select _auto trace_ in the Residual menu). Automatic guesses assume that the exponential is well described by a negative coefficient in the exponential, so the fit doesn't work:

```
•CurveFit exp RisingExponential /D
   Fit converged properly
   fit_RisingExponential= W_coef[0]+W_coef[1]*exp(-W_coef[2]*x)
   W_coef={108.87,-113.32,0.32737}
   V_chisq= 423.845;V_npnts= 20;V_numNaNs= 0;V_numINFs= 0;
   V_startRow= 0;V_endRow= 19;
   W_sigma={255,252,0.863}
   Coefficient values ± one standard deviation
      y0    =108.87 ± 255
      A     =-113.32 ± 252
      invTau=0.32737 ± 0.863
```



In addition to the fact that the graphed fit curve doesn't follow the data points, the estimated uncertainty for the fit coefficients is unreasonably large.

The solution is to provide your own initial guesses. Click the Coefficients tab and choose Manual Guesses in the menu in the upper-right.

The Initial Guesses column in the Coefficients list is now available for you to type your own initial guesses, including a *negative* value for invTau. In this case, we might enter the following initial guesses:

```
y0       0
A        2
invTau-  3
```

In response, Igor generates some extra commands for setting the initial guesses and this time the fit works correctly:

```
•K0 = 0; K1 = 2; K2 = -3;
•CurveFit/G exp RisingExponential /D
   Fit converged properly
   fit_RisingExponential= W_coef[0]+W_coef[1]*exp(-W_coef[2]*x)
   W_coef={-0.81174,2.2996,-2.8742}
   V_chisq= 15.6292;V_npnts= 20;V_numNaNs= 0;V_numINFs= 0;
   V_startRow= 0;V_endRow= 19;
   W_sigma={0.798,0.41,0.171}
   Coefficient values ± one standard deviation
      y0    =-0.81174 ± 0.798
      A     =2.2996 ± 0.41
      invTau=-2.8742 ± 0.171
```

It may well be that finding a set of initial guesses from scratch is difficult. Automatic guesses might be a good starting point which will provide adequate initial guesses when modified. For this the dialog provides the Only Guess mode.

When Only Guess is selected, click Do It to create the automatic initial guesses, and then stop without trying to do the fit. Now, when you bring up the Curve Fitting dialog again, you can choose the coefficient wave created by the auto guess (W_coef if you chose _default_ in the Coefficient Wave menu). Choosing this wave will set the initial guesses to the automatic guess values. Now choose Manual Guesses and modify the initial guesses. The Graph Now button may help you find good initial guesses (see **Coefficients Tab for a User-Defined Function** on page III-192).

Fits with Constants

A few of the built-in fit functions include constants that are not varied during the fit. They enter only to provide, for instance, a constant X offset to improve numerical stability. One such built-in fit function is the exp_XOffset fit function. It fits this equation:

$$y_0 + A\exp\left(\frac{x-x_0}{\tau}\right)$$

Here, y0, A and tau are fit coefficients - they are varied during iterative fitting, and their final values are the solution to the fit. On the other hand, x0 is a constant - it is not varied, rather you give it any value you wish as part of the fit setup. In the case of the exp_XOffset fit function, if you do not set it yourself, it will be set by default to the minimum X value in your input data. For fits to data far from the origin, this improves numerical stability. Naturally, it affects the value of A in the final solution.

In the Curve Fitting dialog, when you select a built-in fit function that uses a constant, an additional edit box appears below the fit function menu where you can set the value of the constant. Setting it to the default value Auto causes Igor to set the constant to some reasonable value based on your input data.

When you do a fit using a built-in fit function that uses a constant, the output includes the wave W_fitConstants. Each element of this wave holds the value of one constant for the equation. At present, the wave will have just one element because there are no built-in fit functions that use more than one constant.

See **Built-in Curve Fitting Functions** on page III-206 for details on the constants used with specific fit functions.

Fitting to a User-Defined Function

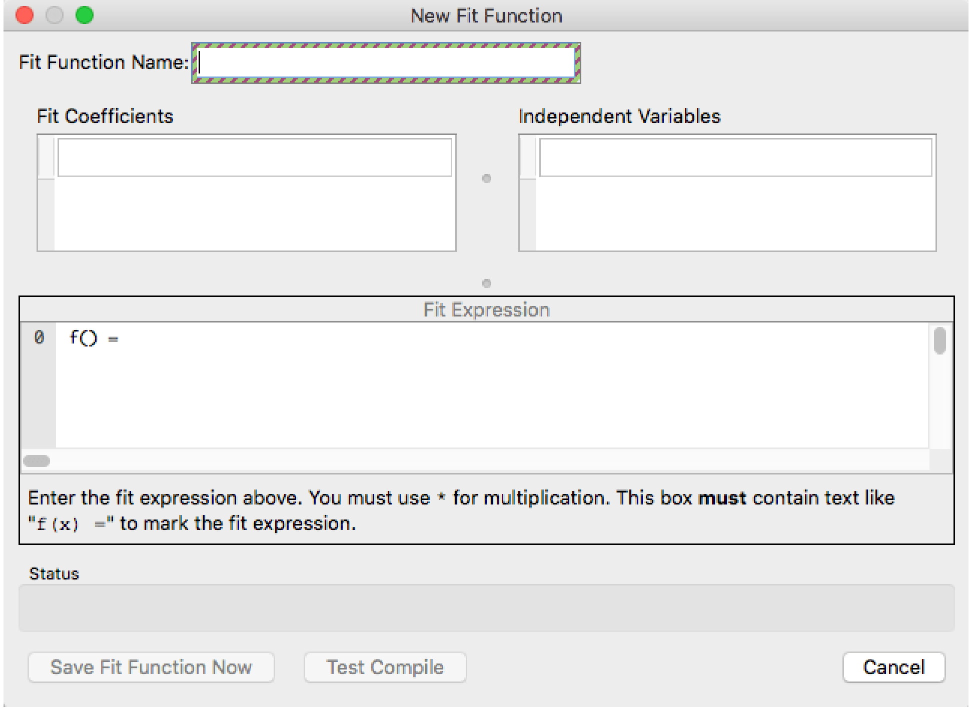
Fitting to a user-defined function is much like fitting to a built-in function, with two main differences:

- You must define the fitting function.
- You must supply initial guesses.

To illustrate the creation of a user-defined fit function, we will create a function to fit a log function: $y = C_1 + C_2\ln(x)$.

Creating the Function

To create a user-defined fitting function, click the New Fit Function button in the Function and Data tab of the Curve Fitting Dialog. The New Fit Function dialog is displayed:



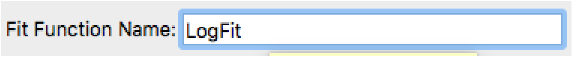
```
0  f() =
```

Enter the fit expression above. You must use * for multiplication. This box **must** contain text like "f(x) =" to mark the fit expression.

You must fill in a name for your function, fill in the Fit Coefficients list with names for the coefficients, fill in the Independent Variables list with names for independent variables, and then enter a fit expression in the Fit Expression window.

The function name must conform to nonliberal naming rules. That means it must start with a letter, contain only letters, numbers or underscore characters, and it must be 255 or fewer bytes in length (see **Object Names** on page III-501). It must not be the same as the name of another object like a built-in function, user procedure, or wave name.

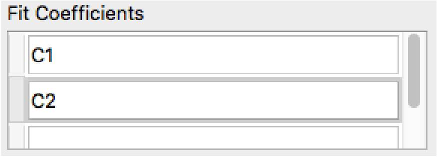
For the example log function, we enter "LogFit":



Fit Function Name: LogFit

Press Tab to move to the first entry in the Fit Coefficient list. There is always one blank entry in the list where you can add a new coefficient name; since we haven't entered anything yet, there is only one blank entry.

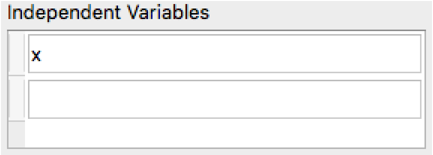
Each time you enter a name, press Return (*Macintosh*) or Enter (*Windows*). Igor accepts that name and makes a new blank entry where you can enter the next name. We enter C1 and C2 as the names:



C1

C2

Click in the first blank entry in the Independent Variables list. Most fit functions will require just a single independent variable. We choose to name our independent variable x:

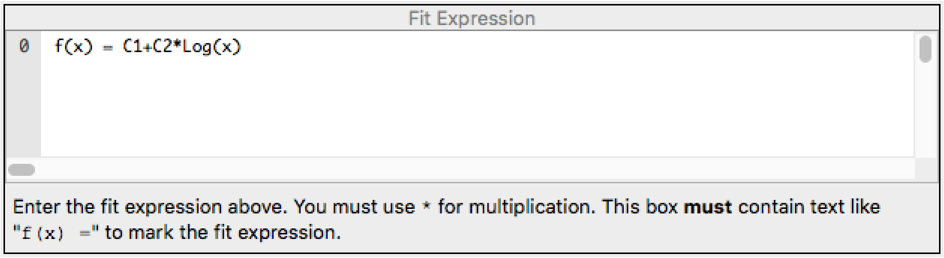


x

It is now time to enter the fit expression. You will notice that when you have entered a name in the Independent Variables list, some text is entered in the expression window. The return value of the fit function (the Y value in most cases) is marked with something like "f(x) = ". If you had entered "temperature" as the independent variable, it would say "f(temperature) = ".

This "f() = " text is required; otherwise the return value of the function will be unknown.

The fit expression is not an algebraic expression. It must be entered in the same form as a command on the command line. If you need help constructing a legal expression, you may wish to read **Assignment Statements** on page IV-4. The expression you need to type is simply the right-hand side of an assignment statement. The log expression in our example will look like this:



```
f(x) = C1+C2*Log(x)
```

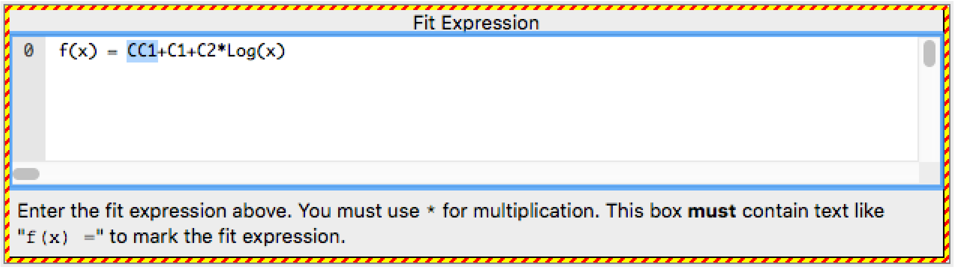
Enter the fit expression above. You must use * for multiplication. This box **must** contain text like "f(x) =" to mark the fit expression.

Multiplication requires an explicit *.

The dialog will check the fit expression for simple errors. For instance, it will not make the Save Fit Function Now button available if any of the coefficients or independent variables are missing from the expression.

The dialog cannot check for correct expression syntax. If all the easily-checked items are correct, the Save Fit Function Now and Test Compile buttons are made available. Clicking either of them will enter a new function in the Procedure window and attempt to compile procedures. If you click the Save Fit Function Now button and compilation is successful, you are returned to the Curve Fitting dialog with the new function chosen in the Function menu.

If compile errors occur, the compiler's error message is displayed in the status box, and the offending part of your expression is highlighted. A common error might be to misspell a coefficient name somewhere in your expression. For instance, if you had typed CC1 instead of C1 somewhere you might see something like this:



```
f(x) = CC1+C1+C2*Log(x)
```

Enter the fit expression above. You must use * for multiplication. This box **must** contain text like "f(x) =" to mark the fit expression.

Note that C1 appears in the expression. Otherwise, the dialog would show that C1 is missing.

When everything is ready to go, click the Save Fit Function Now button to construct a function in the Procedure window. It includes comments in the function code that identify various kinds of information for the dialog. Our example function looks like this:

```
Function LogFit(w,x) : FitFunc
    WAVE w
    Variable x

    //CurveFitDialog/ These comments were created by the Curve Fitting dialog. Altering them will
    //CurveFitDialog/ make the function less convenient to work with in the Curve Fitting dialog.
    //CurveFitDialog/ Equation:
    //CurveFitDialog/ f(x) = C1+C2*log(x)
    //CurveFitDialog/ End of Equation
    //CurveFitDialog/ Independent Variables 1
    //CurveFitDialog/ x
    //CurveFitDialog/ Coefficients 2
    //CurveFitDialog/ w[0] = C1
    //CurveFitDialog/ w[1] = C2

    return w[0]+w[1]*log(x)
End
```

You shouldn't have to deal with the code in the Procedure window unless your function is so complex that the dialog simply can't handle it. You can look at **User-Defined Fitting Functions** on page III-250 for details on how to write a fitting function in the Procedure window.

Having entered the fit expression correctly, click the Save Fit Function Now button, which returns you to the main Curve Fitting dialog. The Function menu will now have LogFit chosen as the fitting function.

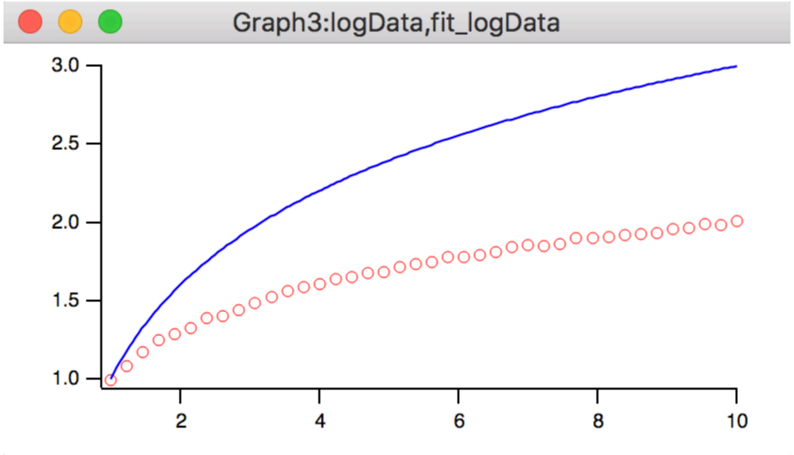
Coefficients Tab for a User-Defined Function

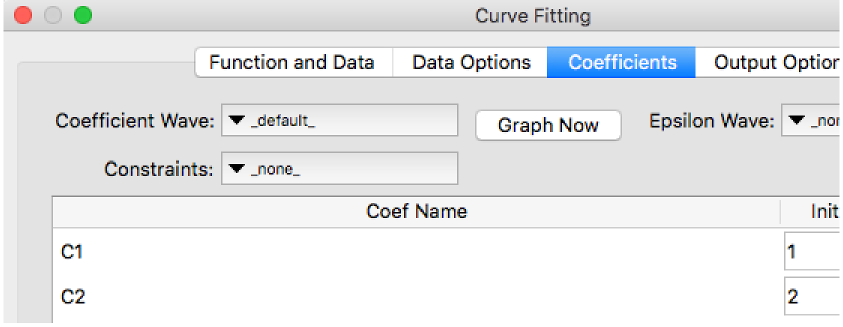
To fit a user-defined function, you will need to enter initial guesses in the Coefficients tab.

Having created a user-defined fitting function (or simply having selected a preexisting one) you will find that the error message window at the bottom of the dialog now states: "You have selected a user-defined fit function so you must enter an initial guess for every fit coefficient. Go to the Coefficients Tab to do this."

When you have selected a user-defined fitting function, the Initial Guess column of the Coefficients List is available. You must enter a number in each row. Some functions may be difficult to fit; in such a case the initial guess may have to be pretty close to the final solution.

To help you find good initial guesses, the Coefficients tab includes the Graph Now button. This button will add a trace to the top graph showing your fitting function using the initial guesses you have entered. For instance:





Coefficient Wave: _default_

Constraints: _none_

Epsilon Wave: _nor

Coef Name	Init
C1	1
C2	2

You can change the values in the initial guess column and click the Graph Now button as many times as you wish. The trace will be updated with the changes each time.

The Graph Now button works as describe in **The Destination Wave** on page III-196, with one exception: if you selected _none_ in the Destination pop-up menu of the Output Options tab, the Graph Now button works as if you selected _auto_. The Graph Now button honors your choices for destination wave style, and it makes a new wave if you selected _New Wave_. Unless you change the destination wave settings before clicking Do It, the wave set by Graph Now will be overwritten by the fit.

On the Coefficients tab you have the option of selecting an “epsilon” wave. An epsilon wave contains one epsilon value for each point in your coefficients wave. By default the Epsilon menu is set to _none_ indicating that the epsilon values are set to the default values.

Each epsilon value is used to calculate partial derivatives with respect to the fit coefficients. The partial derivatives are used to determine the search direction for coefficients that give the smallest chi-square.

In most cases the epsilon values are not critical. However, you can supply your own if you have reason to believe that the default epsilon values are not providing acceptable partial derivatives (sometimes a singular matrix error can be avoided by using custom epsilon values). To specify epsilon values, either select a pre-existing wave from the Epsilon Wave menu, or select _New Wave_. Only waves having a number of points equal to the number of fit coefficients are shown in the menu. Either choice causes the Epsilon column to be shown in the Coefficients list, where you can enter values for epsilon.

If you select a wave from the Epsilon menu, the values in that wave are entered in the list. If you select _New Wave_, the dialog generates commands to create an epsilon wave and fill it with the values in the Epsilon column.

For more information about the epsilon wave and what it does, see **The Epsilon Wave** on page III-267.

Making a User-Defined Function Always Available

Note that, because the fitting function is created in the Procedure window, it is stored as part of the experiment file. That means that it will be available for fitting while you are working on the experiment in which it was created, but will not be available when you work on other experiment files.

You can make the fit function available whenever you start up Igor Pro. Make a new procedure window using the Procedure item under New in the Windows menu. Find the fit function in the Procedure window, select all the text from `Function` through `End` and choose Cut from the Edit menu. Paste the code into your new procedure window. Finally, choose Save Procedure Window from the File menu and save in"Igor Pro User Files/Igor Procedures" (see **Igor Pro User Files** on page II-31 for details). The next time you start Igor Pro you will find that the function is available in all your experiments.

Removing a User-Defined Fitting Function

To remove a user-defined fitting function that you don't want any more, choose Procedure Window from the Windows menu. Find the function in the Procedure window (you can use Find from the Edit menu and search for the name of your function). Select all of the function definition text from the word "`Function`" through the word "`End`" and delete the text.

If you have followed the directions in the section above for making the function always available, find the procedure file in "Igor Pro User Files/Igor Procedures", remove it from the folder, and then restart.

User-Defined Fitting Function Details

The New Fit Function dialog is the easiest way to enter a user-defined fit function. But if your fit expression is very long, or it requires multiple lines with local variables or conditionals, the dialog can be cumbersome. Certain special situations may call for a format that is not supported by the dialog.

For a complete discussion of user-defined fit function formats and the uses for different formats, see **User-Defined Fitting Functions** on page III-250.

Fitting to an External Function (XFUNC)

An external function, or XFUNC, is a function provided via an Igor extension or plug-in. A programmer uses the XOP Toolkit to build an XFUNC. You don't need the toolkit to use one. An XFUNC must be installed before it can be used. See **Igor Extensions** on page III-511.

An XFUNC can speed up curve fitting greatly if your fitting function requires a great deal of computation. The speed of fitting is usually dominated by other kinds of overhead and the effort of writing an XFUNC is not justified.

Fitting to an external function is just like fitting to a user-defined function, except that the Curve Fitting dialog has no way to find out how many fit coefficients are required. When you switch to the Coefficients tab, you will see an alert telling you of that fact. The solution to this problem is to select a coefficient wave with the correct number of points. You must create the wave before entering the Curve Fitting dialog.

When you select a coefficient wave the contents of the wave are used to build the Coefficients list. The wave values are entered in the Initial Guess column. If you change an initial guess, the dialog will generate the commands necessary to enter the new values in the wave.

The Coefficient Wave menu normally shows only those waves whose length is the same as the number of fit coefficients required by the fitting function. When you choose an XFUNC for fitting, the menu shows all waves. You have to know which one to select. We suggest using a wave name that identifies what the wave is for.

Igor doesn't know about coefficient names for an XFUNC. Coefficient names will be derived from the name of the coefficient wave you select. That is, if your coefficient wave is called "coefs", the coefficient names will be "coefs_0", "coefs_1", etc.

Of course, implementing your function in C or C++ is more time-consuming and requires both the XOP Toolkit from WaveMetrics, and a software development environment. See **Creating Igor Extensions** on page IV-208 for details on using the XOP Toolkit to create your own external function.

The Coefficient Wave

When you fit to a user-defined function, your initial guesses are transmitted to the curve fitting operation via a coefficient wave. The coefficients that result from the fit are output in a coefficient wave no matter what kind of function you select. For the most part, the Curve Fitting dialog hides this from you.

When you create a user-defined function, the dialog creates a function that takes a wave as the input containing the fit coefficients. But through special comments in the function code, the dialog gives names to each of the coefficients. A built-in function has names for the coefficients stored internally. Using these names, the dialog is able to hide from you some of the complexities of using a coefficient wave.

In the history printout following a curve fit, the coefficient values are reported both in the form of a wave assignment, using the actual coefficients wave, and as a list using the coefficient names. For instance, here is the printout from the example user-defined fit earlier (**Fitting to a User-Defined Function** on page III-190):

```
•FuncFit LogFit W_coef logData /D
  Fit converged properly
  fit_logData= LogFit(W_coef,x)
  W_coef={1.0041,0.99922}         ———————— Fit Coefficient values as a wave assignment.
  V_chisq= 0.00282525; V_npnts= 30; V_numNaNs= 0; V_numINFs= 0;
  W_sigma={0.00491,0.00679}       ——————— Fit Coefficient sigmas as a wave assignment.
  Coefficient values ± one standard deviation
   C1 =  1.0041± 0.491     | Fit Coefficient values and sigmas in a list using the coefficient names.
   C2 =  0.99922± 0.679    |
```

The wave assignment version can be copied to the command line and executed, or it can be used as a command in a user procedure. The list version is easier to read.

You control how to handle the coefficients wave using the Coefficient Wave menu on the Coefficients tab. Here are the options.

Default

When _default_ is chosen it creates a wave called W_coef. For built-in fits this wave is used only for output. For user-defined fits it is also input. The dialog generates commands to put your initial guesses into the wave before the fit starts.

Explicit Wave

The Coefficient Wave menu lists any wave whose length matches the number of fit coefficients. If you select one of these waves, it is used for input and output from any fit.

When you choose a wave from the menu the data in the wave is used to fill in the Initial Guess column in the Coefficients list. This can be used as a convenient way to enter initial guesses. If you choose an explicit wave and then edit the initial guesses, the dialog generates commands to change the values in the selected coefficient wave before the fit starts. To avoid altering the contents of the wave, after selecting a wave with the initial guesses you want, you can choose _default_ or _New Wave_. The initial guesses will be put into the new or default coefficients wave.

New Wave

A variation on the explicit coefficients wave is _New Wave_. This works just like an explicit wave except that the dialog generates commands to make the wave before the fit starts, so you don't have to remember to make it before entering the dialog. The wave is filled in with the values from the Initial Guess column.

The _New Wave_ option is convenient if you are doing a number of fits and you want to save the fit coefficients from each fit. If you use _default_ the results of a fit will overwrite the results from any previous fit.

Errors

Estimates of fitting errors (the estimated standard deviation of the fit coefficients) are automatically stored in a wave named W_sigma. There is no user choice for this output.

The Destination Wave

When performing a curve fit, it will calculate the model curve corresponding to the fit coefficients. As with most results, the model curve is stored as an array of numbers in a wave. This wave is the “destination wave”.

The main purpose of the destination wave is to show what the fit function looks like with various coefficients during the fit operation and with the final fit coefficients when the operation is finished. You can choose no destination wave, an explicit destination wave or you can use the auto-trace feature.

You choose the destination wave option on the Output Options tab of the dialog. Here are the options.

No Destination

You would choose no destination wave if you don’t want graphic feedback during the fitting process and you don’t need to graphically compare your raw data to the fitting function. This might be the case if you are batch fitting a large number of data sets. Choose _none_ in the Destination menu.

Auto-Trace

In most cases, auto-trace is recommended; choose _auto_ from the Destination menu. When you choose this, it automatically creates a new wave, sets its X scaling appropriately, and appends it to the top graph if the Y data wave is displayed in it. The name of the new wave is generated by prepending “fit_” to the name of the Y data wave. If a wave of that name already exists, it is overwritten. If the name exceeds the 255 character maximum for a wave, the name is truncated.

The number of points in the destination wave depends on the number of independent variables. For the most common case of a univariate fit, the default is 200 points. The rest of this discussion assumes you are fitting univariate data.

If you want to fit more than one function to the same raw data using auto-trace, you should rename the auto-trace wave after the fit so that it will not be overwritten by a subsequent fit. You can rename it using the Rename item in the Data menu or by executing a Rename command directly from the command line. You may also want to save the W_coef and W_sigma waves using the same technique.

Usually the auto-trace wave is set up as a waveform even if you are fitting to XY data. The X scaling of the auto-trace wave is set to spread the 200 points evenly over the X range of the raw data. When preferences are on, the auto-trace wave appended to the graph has the preferred line style (color, size, etc.) *except* that the line mode is always set to “lines between points”, which is best suited to showing the curve fitting results.

Evenly-spaced data are not well suited to displaying a curve on a logarithmic axis. If your data are displayed using a log axis, the fit will create an XY pair of waves. The X wave will be named by prepending “fitX_” to the name of the Y data wave. This X wave is filled with exponentially-spaced X values spread out over the X range of the fit data. Of course, if you subsequently change the axis to a linear axis, the point spacing will not look right.

With _auto_ chosen in the Destination menu, the dialog displays a box labelled Length. Use this to change the number of points in the destination wave. You can set this to any number greater than 3. The more points, the smoother the curve (up to a point). More points will also take longer to draw so the fit will be slower.

Explicit Destination

You can specify an explicit destination wave rather than using auto-trace. Use this if you want a model value at the X location of each of your input data points.

An explicit destination wave must have the same number of points as the Y data wave, so you should create it using the Duplicate operation. The Destination menu shows only waves that have the same number of points as the selected Y Data wave.

The explicit destination wave is not automatically appended to the top graph. Therefore, before executing the curve fit operation, you would normally execute commands like:

```
Duplicate/O yData, yDataFit
AppendToGraph yDataFit vs xData
```

If you are fitting waveform data rather than XY data, you would omit "vs xData" from the AppendToGraph command.

New Wave

As a convenience, the Curve Fitting dialog can create a destination wave for you if you choose _New Wave_ from the Destination menu. It does this by generating a Duplicate command to duplicate your Y data wave and then uses it just like any other explicit destination wave. The new wave is not automatically appended to your graph, so you will have to do that yourself after the fit is completed.

Fitting a Subset of the Data

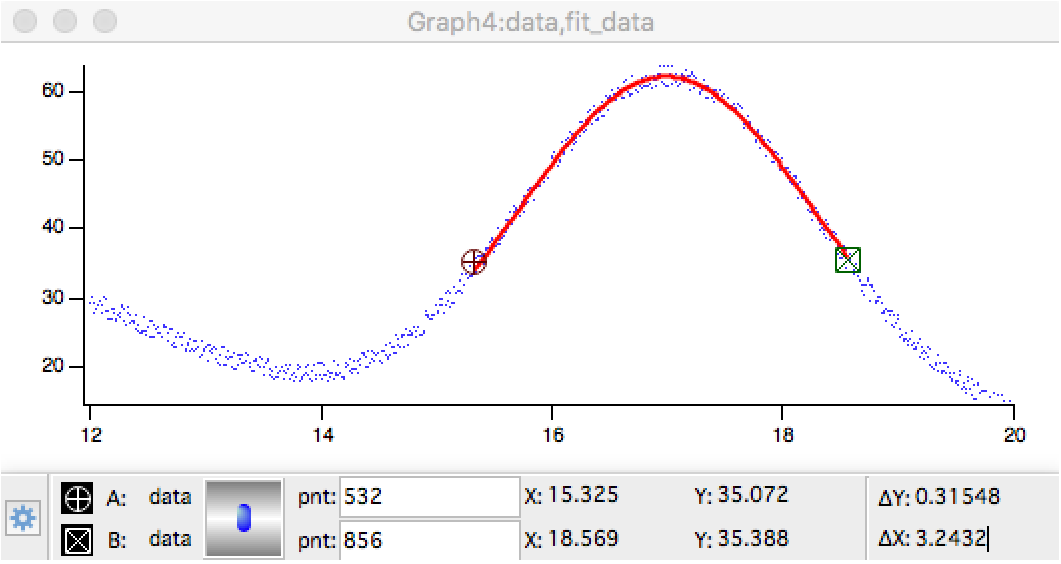
A common problem is that you don't want to include all of the points in your data set in a curve fit. There are two methods for restricting a fit to a subset of points. You will find these options on the Data Options tab of the Curve Fitting dialog.

Selecting a Range to Fit

You can select a contiguous range of points in the Range box. The values that you use to specify the range for a curve fit are in terms of point or row numbers of the Y data wave. Note that if you are fitting to an XY pair of waves and your X values are in random order, you will not be selecting a contiguous range as it appears on a graph.

To simplify selecting the range, you can use graph cursors to select the start and end of the range. To use cursors, display your raw data in a graph. Display the info panel in the graph by selecting ShowInfo from the Graph menu. Drag the cursors onto your raw data. Then use the Cursors button in the Curve Fitting dialog to generate a command to fit the cursor range.

Here is what a graph might look like after a fit over a subrange of a data set:



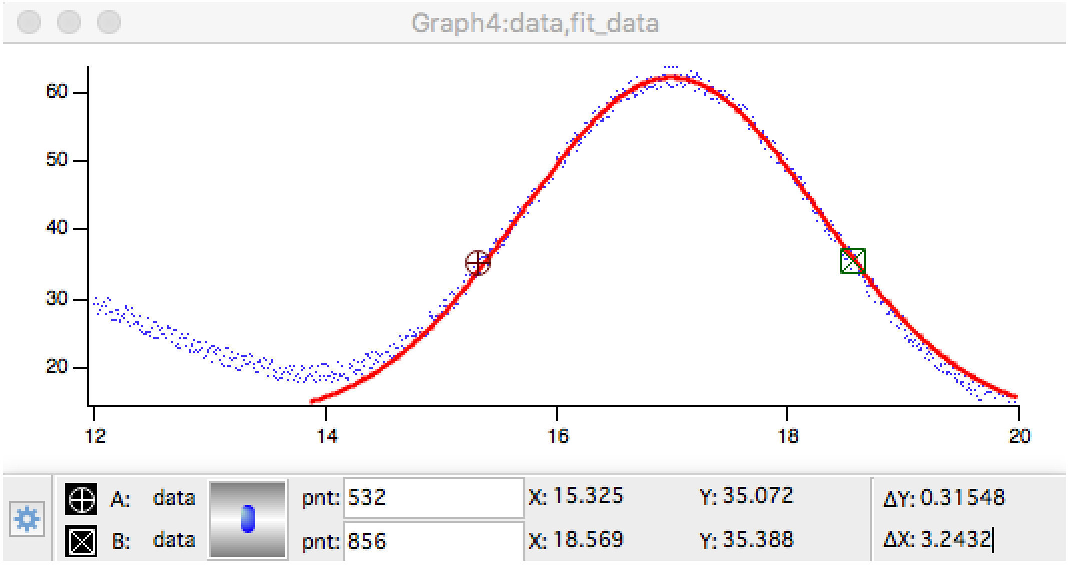
A: data pnt: 532 X: 15.325 Y: 35.072 ΔY: 0.31548
B: data pnt: 856 X: 18.569 Y: 35.388 ΔX: 3.2432

In this example, we used auto-trace for the destination. Notice that the trace appears over the selected range only. If we want to show the destination over a wider range, we need to change the destination wave's X scaling. These commands change the destination wave to show more points over a wider range:

```
Redimension/N=500 fit_data              // change to 500 points
SetScale x 13, 20, fit_data             // set domain from 13 to 20
fit_data= W_coef[0]+W_coef[1]/((x-W_coef[2])^2+W_coef[3])
```

The last line was copied from the history area, where it was displayed after the fit.

This produces the following graph:



A: data pnt: 532 X: 15.325 Y: 35.072 ΔY: 0.31548
B: data pnt: 856 X: 18.569 Y: 35.388 ΔX: 3.2432

If you use an explicit destination wave rather than auto-trace, it is helpful to set the destination wave to blanks (NaN) before performing the fit. As the fit progresses, it will store new values only in the destination wave range that corresponds to the range being fit. Also, it stores into the destination wave only at points where the source wave is not NaN or INF. If you don't preset the destination wave to blanks, you will wind up with a combination of new and old data in the destination wave.

These commands illustrate presetting the destination wave and then performing a curve fit to a range of an XY pair.

```
Duplicate/O yData, yDataFit          // make destination
yDataFit = NaN                       // preset to blank (NaN)
AppendToGraph yDataFit vs xData
CurveFit lor yData(xcsr(A),xcsr(B)) /D=yDataFit
```

Another way to make the fit curve cover a wider range is to select the checkbox labelled X Range Full Width of Graph. You will find the checkbox on the Output Options tab of the Curve Fitting dialog.

Using a Mask Wave

Sometimes the points you want to exclude are not contiguous. This might be the case if you are fitting to a data set with occasional bad points. Or, in spectroscopic data you may want to fit regions of baseline and exclude points that are part of a spectroscopic peak. You can achieve the desired result using a mask wave.

The mask wave must have the same number of points as the Y Data wave. You fill in the mask wave with a NaN (Not-a-Number, blank cell) or zero corresponding to data points to exclude and nonzero for data points you want to include. You must create the mask wave before bringing up the Curve Fitting dialog. You may want to edit the mask wave in a table.

Enter a NaN in a table by typing "NaN" and pressing Return or Enter. Having entered on NaN, you can copy it to the clipboard to paste into other cells.

You can also use a wave assignment on the command line. If your data set has a bad data point at point 4, a suitable command to set point four in the mask wave would be:

```
BadPointMask[4] = NaN
```

When you have a suitable mask wave, you choose it from the Data Mask menu on the Data Options tab.

You can use a mask with NaN points to suppress display of the masked points in a graph if you select the mask wave as an f(z) wave in the Modify Trace Appearance dialog. You could also use the same wave with the ModifyGraph mask keyword.

Weighting

You may provide a weighting wave if you want to assign greater or lesser importance to certain data points. You would do so for one of two reasons:

- To get a better, more accurate fit.
- To get more accurate error estimates for the fit coefficients.

The weighting wave is used in the calculation of chi-square. chi-square is defined as

$$\sum_i \left(\frac{y - y_i}{w_i}\right)^2$$

where y is a fitted value for a given point, y_i is the original data value for the point and w_i is the standard error for the point. The weighting wave provides the $\mathrm{w_i}$ values. The values in the weighting wave can be either $1/\sigma_\mathrm{i}$ or simply σ_i, where σ_i is the standard deviation for each data value. If necessary, Igor takes the inverse of the weighting value before using it to perform the weighting.

You specify the wave containing your weighting values by choosing it from the Weighting menu in the Data Options tab. In addition you must specify whether your wave has standard deviations or inverse standard deviations in it. You do this by selecting one of the buttons below the menu:

- Standard Deviations
- 1/Standard Deviations

Usually you would use standard deviations. Inverse standard deviations are permitted for historical reasons.

There are several ways in which you might obtain values for σ_i. For example, you might have *a priori* knowledge about the measurement process. If your data points are average values derived from repeated measurements, then the appropriate weight value is the standard error. That is the standard deviation of the repeated measurements divided by $\mathrm{N}^{1/2}$. This assumes that your measurement errors are normally distributed with zero mean.

If your data are the result of counting, such as a histogram or a multichannel detector, the appropriate weighting is $(\sqrt{y})$. This formula, however, makes infinite weighting for zero values, which isn't correct and will eliminate those points from the fit. It is common to substitute a value of 1 for the weights for zero points.

You can use a value of zero to completely exclude a given point from the fit process, but it is better to use a data mask wave for this purpose.

If you do not provide a weighting wave, then unity weights are used in the fit and the covariance matrix is normalized based on the assumption that the fit function is a good description of the data. The reported errors for the coefficients are calculated as the square root of the diagonal elements of the covariance matrix and therefore the normalization process will provide valid error estimates only if all the data points have roughly equal errors and if the fit function is, in fact, appropriate to the data.

If you do provide a weighting wave then the covariance matrix is not normalized and the accuracy of the reported coefficient errors rests on the accuracy of your weighting values. For this reason you should not use arbitrary values for the weights.

Proportional Weighting

In some cases, it is desirable to use weighting but you know only proportional weights, not absolute measurement errors. In this case, you can use weighting and after the fit is done, calculate reduced chi-square. The reduced chi-square can be used to adjust the reported error estimates for the fit coefficients. When you do this, the resulting reduced chi-square cannot be used to test goodness of fit.

For example, a data set having Gaussian errors that are proportional to X:

```
Make data = exp(-x/10) + gnoise((x+1)/1000)
Display data
```

Prepare a weighting wave that has weights proportional to X, but are not equal to the true measurement errors:

```
Duplicate data, data_wt
data_wt = x+1        // Right proportionality with X, but 1000 times too big
```

The fit has pretty meaningless coefficient errors because the weights provided were proportional to the true measurement errors, but 1000 times too big:

```
CurveFit/NTHR=0 exp data /W=data_wt /I=1 /D
Fit converged properly
fit_data= W_coef[0]+W_coef[1]*exp(-W_coef[2]*x)
W_coef={0.0044805,0.99578,0.10063}
V_chisq= 0.000117533;V_npnts= 128;V_numNaNs= 0;V_numINFs= 0;
V_startRow= 0;V_endRow= 127;
W_sigma={5.78,5.74,1.15}
Coefficient values ± one standard deviation
   y0    =0.0044805 ± 5.78
   A     =0.99578 ± 5.74
   invTau=0.10063 ± 1.15
```

So now we compute reduced errors based on reduced chi-square:

```
Variable reducedChiSquare = V_chisq/(V_npnts - numpnts(W_coef))
Duplicate W_sigma, reducedSigma
reducedSigma = W_sigma*sqrt(reducedChiSquare)
```

The resulting errors are more reasonable:

```
Print reducedSigma
reducedSigma[0]= {0.00560293,0.0055649,0.00111907}
```

Note that, as with any non-linear fitting, Gaussian statistics like this are not really applicable. The results should be used with caution.

Fitting to a Multivariate Function

A multivariate function is a function having more than one independent variable. This might arise if you have measured data over some two-dimensional area. You might measure surface temperature at a variety of locations, resulting in temperature as a function of both X and Y. It might also arise if you are trying to find dependencies of a process output on the various inputs, perhaps initial concentrations of reagents, plus temperature and pressure. Such a case might have a large number of independent variables.

Fitting a multivariate function is pretty much like fitting to a function with a single independent variable. This discussion assumes that you have already read the instructions above for fitting a univariate function.

You can create a new multivariate user-defined function by clicking the New Fit Function button. In the Independent Variables list, you would enter more than one variable name. You can use as many independent variables as you wish (within generous limits set by the length of a line in the procedure window).

A univariate function usually is written as $y = f(x)$, and the Curve Fitting dialog reflects this in using “Y Data” and “X Data” to label the menus where you select the input data.

Multivariate data isn’t so convenient. Functions intended to fit spatial data are often written as $z = f(x,y)$; volumetric data may be $g = f(x,y,z)$. Functions of a large number of independent variables are often written as $y = f(x_1,x_2,\ldots)$. To avoid confusion, we just keep the Y Data and X Data labels and use them to mean dependent variable and independent variables.

The principle difference between univariate and multivariate functions is in the selection of input data. If you have four or fewer independent variables, you can use a multidimensional wave to hold the Y values. This would be appropriate for data measured on a spatial grid, or any other data measured at regularly-spaced intervals in each of the independent variables. We refer to data in a multidimensional wave as “gridded data.”

Alternately, you can use a 1D wave to hold the Y values. The independent variables can then be in N 1D waves, one wave for each independent variable, or a single N-column matrix wave. The X wave or waves must have the same number of rows as the Y wave.

Selecting a Multivariate Function

When the Curve Fitting dialog is first used, multivariate functions are not listed in the Function menu. The first thing you must do is to turn on the listing of multivariate functions. You do this by choosing Show Multivariate Functions from the Function menu. This makes two built-in multivariate functions, poly2D and Gauss2D, as well as suitable user-defined functions, appear in the menu.

The Show Multivariate Functions setting is saved in the preferences. Unless you turn it off again, you never need select it again.

Now you can select your multivariate function from the menu.

Selecting Fit Data for a Multivariate Function

When you have selected a multivariate function, the Y Data menu is filled with 1D waves and any multidimensional waves that match the number of independent variables required by the fit function.

Selecting X Data for a 1D Y Data Wave

If your Y data are in a 1D wave, you must select an X wave for each independent variable. There is no way to store X scaling data in the Y wave for more than one independent variable, so there is no _calculated_ item.

With a 1D wave selected in the Y Data menu, the X Data menu lists both 1D waves and 2D waves with N columns for a function with N independent variables.

As you select X waves, the wave names are transferred to a list below the menu. When you have selected the right number of waves the X Data menu is disabled. The order in which you select the X waves is important. The first selected wave gives values for the first independent variable, etc.

If you need to remove an X Data wave from the list, simply click the wave name and press Backspace (*Windows*) or Delete (*Macintosh*). To change the order of X Data waves, select one or more waves in the list and drag them into the proper order.

Selecting X Data for Gridded Y Data

When you select a multidimensional Y wave, the independent variable values can come from the dimension scaling of the Y wave or from 1D waves containing values for the associated dimensions of the Y wave. That is, if you have a 2D matrix Y wave, you could select a wave to give values for the X dimension and a wave to give values for the Y dimension. The Independent Variable menus list only waves that match the given dimension of the Y wave.s

Fitting a Subrange of the Data for a Multivariate Function

Selecting a subrange of data for a 1D Y wave is just like selecting a subrange for a univariate function. Simply enter point numbers in the Start and End range boxes in the Data Options tab.

If you are fitting gridded Y data, the Data Options tab displays eight boxes to set start and end ranges for each dimension of a multidimensional wave. Enter row, column, layer or chunk numbers in these boxes:

If your Y wave is a matrix wave displayed in a graph as an image, you can use the cursors to select a subset of the data. With the graph as the target window, clicking the Cursors button will enter text in the range boxes to do this.

Using cursors with a contour plot is not straightforward, and the dialog does not support it.

You can also select data subranges using a data mask wave (see **Using a Mask Wave** on page III-198). The data mask wave must have the same number of points and dimensions as the Y Data wave.

Model Results for Multivariate Fitting

As with fitting to a function of one independent variable, Igor creates waves containing the model output and residuals automatically. This is done if you choose _auto_ for the destination and _auto trace_ for the residual on the Output Options tab. There are some differences in detail, however.

By default, the model curve for a univariate fit is a smooth curve having 200 points to display the model fit. This depends on being able to sensibly interpolate between successive values of the independent variable. Multicolumn independent variables, on the other hand, probably don't have successive values of all the independent variables in sequential order, so it is not possible to do this. Consequently, it calculates a model point for each point in the dependent variable data wave. If the data wave is displayed as a simple 1D trace in the top graph window, the fit results will be appended to the graph.

Residuals are always calculated on a point-for-point basis, so calculating residuals for a multicolumn multivariate fit is just like a univariate fit.

Displaying results of fitting to a multidimensional wave is more problematic. If the dependent variable has three or more dimensions, it is not easy to display the results. The model and residual waves will be created and the results calculated but not displayed. You can make a variety of 3D plots using Gizmo: just choose the appropriate plot type from the Windows→New→3D Plots menu.

Fits to a 2D matrix wave are displayed on the top graph if the Y Data wave is displayed there as either an image or a contour. The model results are plotted as a contour regardless of whether the data are displayed as an image or a contour. Model results contoured on top of data displayed as an image can be a very powerful visualization technique.

Residuals are displayed in the same manner as the data in a separate, automatically-created graph window. The window size will be the same as the window displaying the data.

Time Required to Update the Display

Because contours and images can take quite a while to redraw, the time to update the display at every iteration may cause fits to contour or image data to be very slow. To suppress the updates, click the Suppress Screen Updates checkbox on the Output Options tab.

Multivariate Fitting Examples

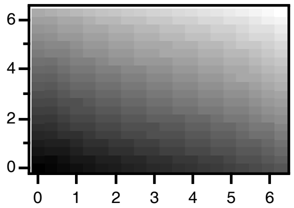
Here are two examples of fitting to a multivariate function — the first uses the built-in poly2D function to fit a plane to a gridded dataset to remove a planar trend from the data. The second defines a simplified 2D gaussian function and uses it to define the location of a peak in XY space using random XYZ data.

Example One — Remove Planar Trend Using Poly2D

Here is an example in which a matrix is filled with a two-dimensional sinusoid with a planar trend that overwhelms the sinusoid. The example shows how you might fit a plane to the data to remove the trend. First, make the data matrix, fill it with values, and display the matrix as an image:

```
Make/O/N=(20,20) MatrixWave
SetScale/I x 0,2*pi,MatrixWave
SetScale/I y 0,2*pi,MatrixWave
MatrixWave = sin(x) + sin(y) + 5*x + 10*y
Display;AppendImage MatrixWave
```

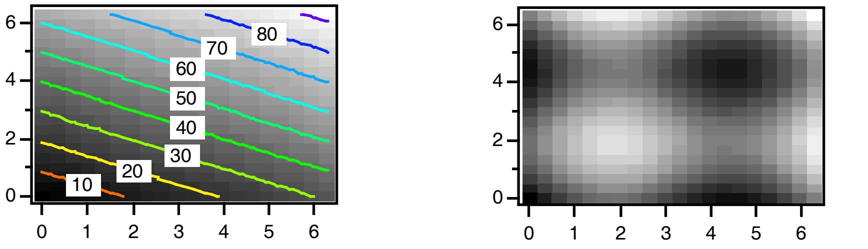
These commands make a graph displaying an image like the one that follows. Note the gradient from the lower left to the upper right:



We are ready to do the fit.

1. Choose Curve Fitting from the Analysis menu to bring up the Curve Fitting dialog.
2. If you have not already done so, choose Show Multivariate Functions from the Function menu.
3. Choose Poly2D from the Function menu.
4. Make sure the 2D Polynomial Order is set to 1.
5. Choose MatrixWave from the Y Data menu.
6. Click the Output Options tab.
7. Choose _auto trace_ from the Residual menu.
8. Click Do It.

The result is the original graph with a contour plot showing the fit to the data, and a new graph of the residuals, showing the sinusoidal signal left over from the fit:



Original graph

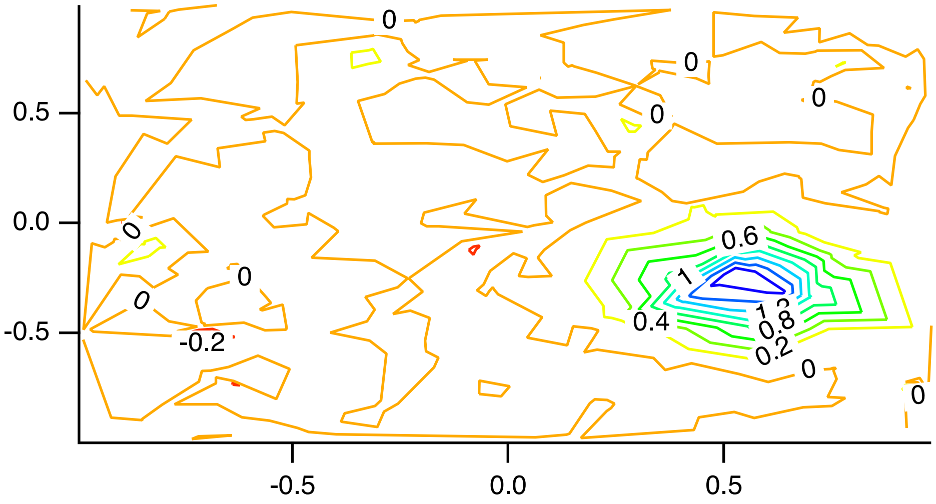
Residual graph showing sinusoidal signal

Similarly, you can use the ImageRemoveBackground operation, which provides a one-step operation to do the same fit. With an image plot as the top window, you will find Remove Background in the Image menu.

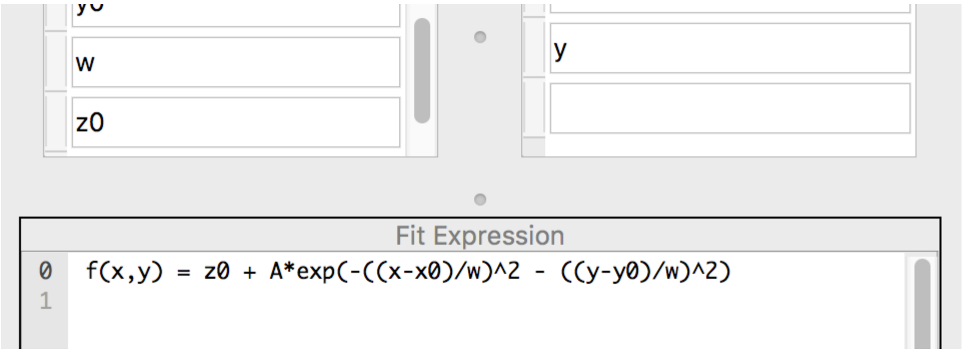
Example Two — User-Defined Simplified 2D Gaussian Fit

In this example, we have data defining a spot which we wish to fit with a 2D Gaussian to find the center of the spot. For some reason this data is in the form of XYZ triplets with random X and Y coordinates. These commands will generate the example data:

```
Make/N=300 SpotXData, SpotYData, SpotZData
SetRandomSeed 0.5
SpotXData = enoise(1)
SpotYData = enoise(1)
// make a gaussian centered at {0.55, -0.3}
SpotZData = 2*exp(-((SpotXData-.55)/.2)^2 -((SpotYData+.3)/.2)^2)+gnoise(.1)
Display; AppendXYZContour SpotZData vs {SpotXData,SpotYData}
```



Now bring up the Curve Fitting dialog and click the New Fit Function button so that you can enter your user-defined fit function. We have reason to believe that the spot is circular so the gaussian can use the same width in the X and Y directions, and there is no need for the cross-correlation term. Thus, the new function has a z0 coefficient for the baseline offset, A for amplitude, x0 and y0 for the X and Y location and w for width. Here is what it looks like in the New Fit Function dialog:



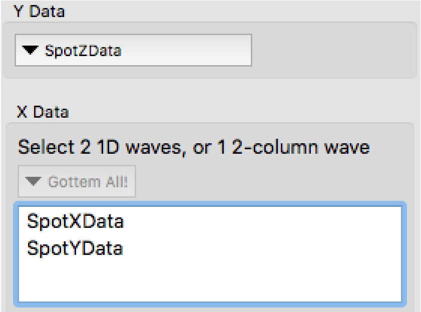
```
f(x,y) = z0 + A*exp(-((x-x0)/w)^2 - ((y-y0)/w)^2)
```

Click Save Fit Function Now to save the function in the Procedure window and return to the Curve Fitting dialog. The new function is selected in the Function menu automatically.

To perform the fit choose:

1. SpotZData in the Y Data menu.
2. SpotXData in the X Data menu.
3. SpotYData in the X Data menu.

At this point, the data selection area of the Function and Data tab looks like this:



Y Data

SpotZData

X Data

Select 2 1D waves, or 1 2-column wave

SpotXData
SpotYData

4. Click the Coefficients tab (the error box at the bottom shows that we must enter initial guesses).
5. Enter initial guesses: we set z0 to 0, A to 2, x0 to 0.5, y0 to -0.3, and width to 0.5.
6. For our problem, residuals and destination aren't really important since we just want to know the coordinates of the spot center. We click Do It and get this in history:

```
FuncFit SimpleGaussian W_coef SpotZData /X={SpotXData,SpotYData} /D
  Fit converged properly
  fit_SpotZData= SimpleGaussian(W_coef,x,y)
  Res_SpotZData= SpotZData[p] - SimpleGaussian(W_coef,SpotXData[p],SpotYData[p])
  W_coef={1.9797,0.54673,-0.2977,0.19962,-0.0067009}
  V_chisq= 3.06428;V_npnts= 300;V_numNaNs= 0;V_numINFs= 0;
  V_startRow= 0;V_endRow= 299;
  W_sigma={0.065,0.00559,0.00553,0.00529,0.00622}
  Coefficient values ± one standard deviation
   A  =1.9797 ± 0.065
   x0 =0.54673 ± 0.00559
   y0 =-0.2977 ± 0.00553
   w  =0.19962 ± 0.00529
   z0 =-0.0067009 ± 0.00622
```

The output shows that the fit has determined that the center of the spot is {0.54697, -0.30557}.

Problems with the Curve Fitting Dialog

Occasionally you may find that things don't work the way you expect when using the Curve Fitting dialog. Common problems are:

- You can't find your user-defined function in the Function menu.

  This usually happens for one of two reasons: either your function is a multivariate function or it is an old-style function. The problem is solved by choosing Show Multivariate Functions or Show Old-Style Functions from the Function menu on the Function and Data tab.

  If you find that choosing Show Old-Style Functions makes your fit function appear, you may want to consider clicking the Edit Fit Function button, which makes the Edit Fit Function dialog appear. Part of the initialization for the dialog involves revising your fit function to make it conform to current standards. While you're there you can give your fit coefficients mnemonic names.

- You get a message that "Igor can't determine the number of coefficients…".

  This happens when you click the Coefficients tab when you are using an external function or a user-defined function that is so complicated that the dialog can't parse the function code to determine how many coefficients are required.

  The only way to get around this is to choose an explicit coefficient wave (**The Coefficient Wave** on page III-195). The dialog will then use the number of points in the coefficient wave to determine the number of coefficients.

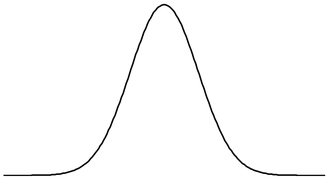
Built-in Curve Fitting Functions

For the most part you will get good results using automatic guesses. A few require additional input beyond what is summarized in the preceding sections. This section contains notes on the fitting functions that give a bit more detail where it may be helpful.

gauss

Fits a Gaussian peak.

$$y_0 + A\exp\left[-\left(\frac{x - x_0}{width}\right)^2\right]$$

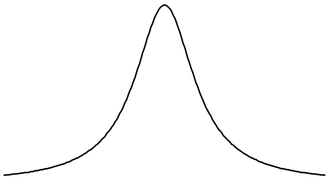


Note that the width parameter is sqrt(2) times the standard deviation of the peak. This is different from the w_i parameter in the Gauss function, which is simply the standard deviation.

lor

Fits a Lorentzian peak.

$$y_0 + \frac{A}{(x - x_0)^2 + B}$$



Voigt

Fits a Voigt peak, a convolution of a Lorentzian and a Gaussian peak. By changing the ratio of the widths of the Lorentzian and Gaussian peak components the shape can grade between a Lorentzian shape and a Gaussian shape.

$$y_0 + \frac{2Area}{W_G}\sqrt{\frac{\ln(2)}{\pi}} \bullet Voigt\left[\frac{2\sqrt{\ln(2)}}{W_G}(x - x_0), Shape \bullet 2\sqrt{\ln(2)}\right]$$

The Voigt function is a normalized Voigt peak shape function. It is the same as the built-in function **VoigtFunc**.

The premultiplier and arguments to the Voigt function result in a peak shape in which fit coefficients are:

Area	The area under the peak excluding the vertical offset
y0	The vertical offset
WG	The Gaussian full width at half maximum (FWHM)
Shape	The ratio of the Lorentzian and Gaussian components, W_L/W_G

There is no analytic expression to compute the height of a Voigt peak.

The FWHM of the Voigt peak can be approximated as

$$W_V = \frac{W_L}{2} + \sqrt{\frac{W_L^2}{4} + W_G^2}$$

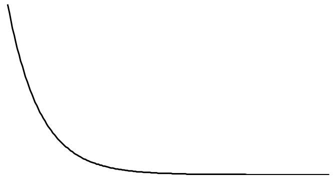
where W_L = Shape * W_G.

Fitting peaks with extreme values of Shape is not recommended as it suffers from numerical instability. Fit the end-point shapes Lor (Shape → ∞) or Gauss (Shape = 0) instead.

exp_XOffset

Fits a decaying exponential.

$$y_0 + A\exp\left(\frac{x-x_0}{\tau}\right)$$



In this equation, x_0 is a constant, not a fit coefficient. During generation of automatic guesses, x_0 will be set to the first X value in your fit data. This eliminates problems caused by floating-point roundoff.

You can set the value of x_0 using the /K flag with the CurveFit operation, but it is recommended that you accept the automatic value. Setting x_0 to a value far from the initial X value in your input data is guaranteed to cause problems.

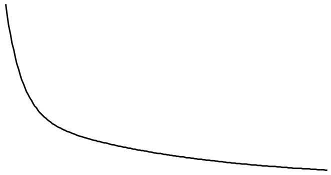
Note: The fit coefficient τ is the inverse of the equivalent coefficient in the exp function. It is actually the decay constant, not the inverse decay constant.

Automatic guesses don't work for growing exponentials (negative τ). To fit a negative value of τ, use Manual Guess on the Coefficients tab, or CurveFit/G on the command line.

dblexp_XOffset

Fits a sum of two decaying exponentials.

$$y_0 + A_1\exp\left(\frac{x-x_0}{\tau_1}\right) + A_2\exp\left(\frac{x-x_0}{\tau_2}\right)$$

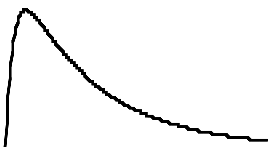


In this equation, x_0 is a constant, not a fit coefficient. During generation of automatic guesses, x_0 wall be set to the smallest X value in your fit data. This eliminates problems caused by floating-point roundoff.

You can set the value of x_0 using the /K flag with the CurveFit operation, but it is recommended that you accept the automatic value. Setting x_0 to a value far from the initial X value in your input data is guaranteed to cause problems.

Note: The fit coefficients τ_1 and τ_2 are the inverse of the equivalent coefficients in the dblexp function. They are actual decay constants, not inverse decay constants.

See the notes for **exp_XOffset** on page III-207 for growing exponentials. You will also need to use manual guesses if the amplitudes have opposite signs:



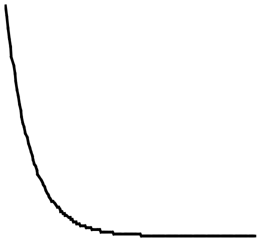
See also the dblexp_peak fit function described below.

If the two decay constants (τ_1 and τ_2) are not quite distinct you may not get accurate results.

exp

Fits a decaying exponential. Similar to exp_XOffset, but not as robust. Included for backward compatibility; in new work you should use exp_Xoffset.

$$y_0 + A\exp(-Bx)$$



Note that offsetting your data in the X direction will cause changes in *A*. Use exp_XOffset for a result that is independent of X position.

Note: The fit coefficient *B* is the inverse decay constant.

Automatic guesses don't work for growing exponentials (negative *B*). To fit a negative value of *B*, use Manual Guess on the Coefficients tab, or CurveFit/G on the command line.

Floating-point arithmetic overflows will cause problems when fitting exponentials with large X offsets. This problem often arises when fitting decays in time as times are often large. The best solution is to use the exp_XOffset fit function. Otherwise, to fit such data, the X values must be offset back toward zero.

You could simply change your input X values, but it is usually best to work on a copy. Use the Duplicate command on the command line, or the Duplicate Waves item in the Data menu to copy your data.

For an XY pair, execute these commands on the command line (these commands assume that you have made a duplicate wave called `myXWave_copy`):

```
Variable xoffset = myXWave_copy[0]
myWave_copy[0] -= xoffset
```

Note that these commands assume that you are fitting data from the beginning of the wave. If you are fitting a subset, replace `[0]` with the point number of the first point you are fitting. If you are using graph cursors to select the points, substitute `[pcsr(A)]`. This assumes that the round cursor (cursor A) marks the beginning of the data.

If you are fitting to waveform data (you selected _calculated_ in the X Data menu) then you need to set the x0 part of the wave scaling to offset the data. If you are fitting the entire wave, simply use the Change Wave Scaling dialog from the Data menu to set the x0 part of the scaling to zero. If you are fitting a subset selected by graph cursors, it is easier to change the scaling on the command line:

```
SetScale/P x leftx(myWave_copy)-xcsr(A), deltax(myWave_copy), myWave_copy
```

This command assumes that you have used the round cursor (cursor A) to mark the beginning of the data.

Subtracting an X offset will change the amplitude coefficient in the fit. Often the only coefficient of interest is the decay constant (invTau) and the change in the amplitude can be ignored. If that is not the case, you can calculate the correct amplitude after the fit is done:

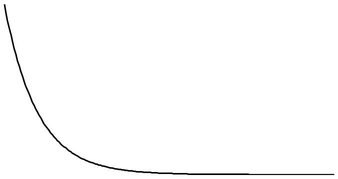
```
W_coef[1] = W_coef[1]*exp(W_coef[2]*xoffset)
```

If you are fitting waveform data, the value of xoffset would be `-leftx(myWave_copy)`.

dblexp

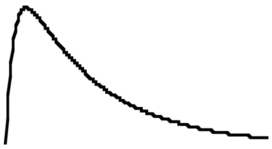
Fits a sum of decaying exponentials. Similar to dblexp_XOffset, but suffers from floating-point roundoff problems if the data do not start quite close to x=0. Included for backward compatibility; in new work you should use exp_Xoffset.

$$y_0 + A_1\exp(-B_1x) + A_2\exp(-B_2x)$$



Note that offsetting your data in the X direction will cause changes in A_1 and A_2. Use dblexp_XOffset for a result that is independent of X position.

Note: The fit coefficients B_1 and B_2 are inverse decay constants.



See the notes for exp for growing exponentials. You will also need to use manual guesses if the amplitudes have opposite signs:

See also the dblexp_peak fit function described below.

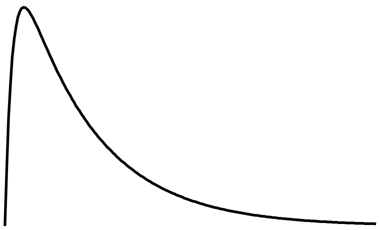
If the two decay constants (B_1 and B_2) are not quite distinct you may not get accurate results.

Fitting data with a large X offset will have the same sort of troubles as when fitting with exp. The best solution is to use the dblexp_XOffset fit function; you can also solve the problem using a procedure similar to the one outlined for **exp** on page III-207.

dblexp_peak

Fits the sum of two exponential terms of opposite sign, making a peak.

$$y_0 + A \cdot \left\{ -exp\left[\frac{-(x-x_0)}{tau_1}\right] + exp\left[\frac{-(x-x_0)}{tau_2}\right]\right\}$$



The location of the peak is given by

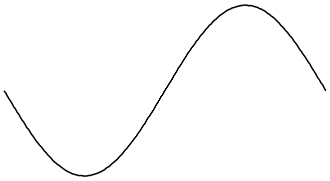
$$X_{peak} = \frac{tau_1\ tau_2\ ln\left(\frac{tau_1}{tau_2}\right)}{tau_1 - tau_2} + x_0$$

If you need individual control of the amplitudes of the two terms, use the dblexp_XOffset function. In that case, you will need to use manual guesses.

sin

Fits a sinusoid.

$$(y_0 + A\sin(fx + \phi))$$



phi is in radians. To convert to degrees, multiply by 180/Pi.

A sinusoidal fit takes an additional parameter that sets the approximate frequency of the sinusoid. This is entered in terms of the approximate number of data points per cycle. When you choose `sin` from the Function menu, a box appears where you enter the expected number of points per cycle.

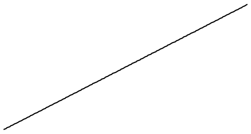
If you enter a number less than 6, the default value will be 7. It may be necessary to try various values to get good results. You may want to simply use manual guesses.

The nature of the sin function makes it impossible for a curve fit to distinguish phases that are different by 2π. It is probably easier to subtract 2nπ than to try to get the fit to fall in the desired range.

line

Fit a straight line through the data.

$$(a + bx)$$



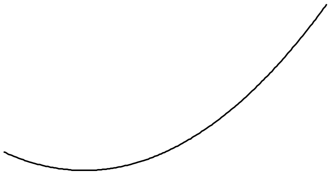
Never requires manual guesses.

If you want to fit a line through the origin, in the Coefficients tab select the Hold box for coefficient *a* and set the Initial Guess value to zero.

poly n

Fits a polynomial with n terms, or order n-1.

$$(K_0 + K_1 x + K_2 x^2 + \ldots)$$



A polynomial fit takes an additional parameter that sets the number of polynomial terms. When you choose `poly` from the Function menu, a box appears where you enter that value.

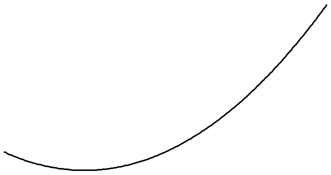
The minimum value of n is 1 corresponding to taking the mean of the Y value. Most Igor users won't need anything less than 3, a quadratic polynomial.

A polynomial fit never requires manual guesses.

poly_XOffset n

Fits a polynomial with n terms, or order n-1. The constant x_0 is not an adjustable fit coefficient; it allows you to place the polynomial anywhere along the X axis. This would be particularly useful for situations in which the X values are large, for instance when dealing with date/time data.

$$(K_0 + K_1(x - x_0) + K_2(x - x_0)^2 + \ldots)$$



The poly_XOffset fit function takes additional parameters that set the number of polynomial terms and the value of x_0. When you select poly_XOffset from the Function menu, boxes appear where you enter these values.

The minimum value of n is 1 corresponding to taking the mean of the Y value. Most Igor users won't need anything less than 3, a quadratic polynomial.

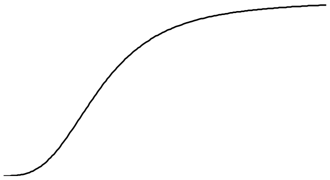
When Set Constant X0 is set to Auto, x_0 is set to the minimum X value found in the data you are fitting. You can set x_0 to any value you wish. Values far from the X values in your data set will cause numerical problems.

A polynomial fit never requires manual guesses.

HillEquation

Fits Hill’s Equation, a sigmoidal function.

$$base + \frac{(max - base)}{1 + \left(\frac{x_{1/2}}{x}\right)^{rate}}$$



The coefficient *base* sets the y value at small X, *max* sets the y value at large X, *rate* sets the rise rate and $x_{1/2}$ sets the X value at which Y is at (*base* + *max*)/2.

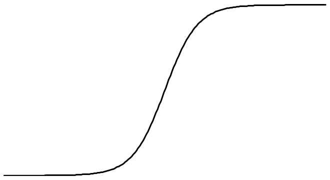
Note that X values must be greater than 0. Including a data point at $X \leq 0$ will result in a singular matrix error and the message, "The fitting function returned NaN for at least one X value."

You can reverse the values of *base* and *max* to fit a falling sigmoid.

sigmoid

Fits a sigmoidal function with a different shape than Hill's equation.

$$base + \frac{max}{1 + \exp\left(\frac{x_0 - x}{rate}\right)}$$

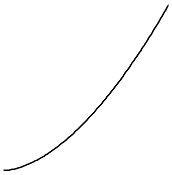


The coefficient *base* sets the Y value at small X. The Y value at large X is *base*+*max*. x0 sets the X value at which Y is at (*base*+*max*/2) and *rate* sets the rise rate. Smaller *rate* causes a faster rise, specifically, the slope at x=x0 is *max*/(4**rate*).

power

Fits a power law.

$$(y_0 + Ax^{pow})$$



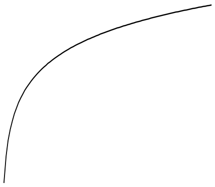
May be difficult to fit, requiring good initial guesses, especially for *pow* > 1 or *pow* close to zero.

Note that X values must be greater than 0. Including a data point at $X \leq 0$ will result in a singular matrix error and the message, "The fitting function returned NaN for at least one X value."

log

Fits a logarithmic curve.

$$a + b\log(x)$$



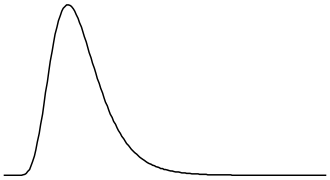
log is very forgiving of bad initial guesses. It was added in Igor Pro 9.00.

X values must be greater than 0. Including a data point with X<=0 results in a singular matrix error and the message "The fitting function returned NaN for at least one X value."

lognormal

Fits a lognormal peak shape. This function is gaussian when plotted on a log X axis.

$$y_0 + A\exp\left[-\left(\frac{\ln(x/x_0)}{width}\right)^2\right]$$



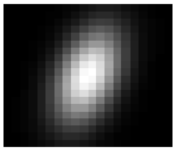
Coefficient y_0 sets the baseline, *A* sets the amplitude, x_0 sets the peak position in X and *width* sets the peak width.

Note that X values must be greater than 0. Including a data point at $X \leq 0$ will cause a singular matrix error and the message, “The fitting function returned NaN for at least one X value.”

gauss2D

Fits a Gaussian peak in two dimensions.

$$z_0 + A\exp\left[\frac{-1}{2(1-cor^2)}\left(\left(\frac{x-x_0}{xwidth}\right)^2 + \left(\frac{y-y_0}{ywidth}\right)^2 - \frac{2cor(x-x_0)(y-y_0)}{xwidth \cdot ywidth}\right)\right]$$



Coefficient *cor* is the cross-correlation term; it must be between -1 and 1 (the small illustration was done with *cor* equal to 0.5). A constraint automatically enforces this range. If you know that a value of zero for this term is appropriate, you can hold this coefficient. Holding *cor* at zero usually speeds up the fit quite a bit.

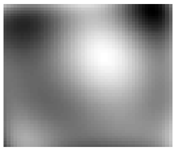
In contrast with the gauss fit function, xWidth and yWidth are standard deviations of the peak.

Note that the Gauss function lacks the cross-correlation parameter *cor*.

poly2D n

Fits a polynomial of order n in two dimensions.

$$(C_0 + C_1x + C_2y + C_3x^2 + C_4xy + C_5y^2 + \ldots)$$



A poly2D fit takes an additional parameter that specifies the order of the polynomial. When you choose poly2D from the Function menu, a box appears where you enter that value.

The minimum value is 1, corresponding to a first-order polynomial, a plane. The coefficient wave for poly2D has the constant term (C_0) in point zero, and following points contain groups of increasing order. There are two first-order terms, C_1*x and C_2*y, then three second-order terms, etc. The total number of terms is (N+1)(N+2)/2, where N is the order.

Poly2d never requires manual guesses.

Inputs and Outputs for Built-In Fits

There are a number of variables and waves that provide various kinds of input and output to a curve fit. Usually you will use the Curve Fitting dialog and the dialog will make it clear what you need, and detailed descriptions are available in various places in this chapter. For a quick introduction, here is a table that lists the waves and variables used for fitting to a built-in function.

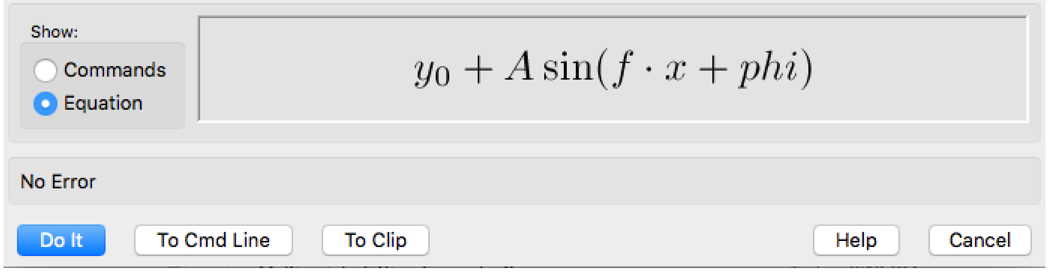
Wave or Variable	Type	What It Is Used For
Dependent variable data wave	Input	Contains measured values of the dependent variable of the curve to fit. Often referred to as “Y data”.
Independent variable data wave	Input	Contains measured values of the independent variable of the curve to fit. Often referred to as “X data”.
Destination wave	Optional output	For graphical feedback during and after the fit. The destination wave continually updates during the fit to show the fit function evaluated with the current coefficients.
Residual wave	Optional output	Difference between the data and the model.
Weighting wave	Optional input	Used to control how much individual Y data points contribute to the search for the output coefficients.
System variables K0, K1, K2 …	Input and output	*Built-in fit functions only.* Optionally takes initial guesses from the system variables and updates them at the end of the fit.
Coefficients wave By default, W_coef.	Input and Output	Takes initial guesses from the coefficients wave, updates it during the fit and leaves final coefficients in it. See the reference for CurveFit and FuncFit for additional options.
Epsilon wave	Optional input	*User-defined fit functions only.* Used by the curve fitting algorithm to calculate partial derivatives with respect to the coefficients.
W_sigma	Output	Creates this wave and stores the estimates of error for the coefficients in it.
W_fitConstants	Output	Created when you do a fit using a built-in fit function containing a constant. Igor creates this wave and stores the values of any constants used by the fit equation. For details, see **Fits with Constants** on page III-189. For notes on constants used in specific fit functions, see **Built-in Curve Fitting Functions** on page III-206.
V_<xxx>	Input	There are a number of special variables, such as V_FitOptions, that you can set to tweak the behavior of the curve fitting algorithms.
V_<xxx>	Output	Creates and sets a number of variables such as V_chisq and V_npnts. These contain various statistics found by the curve fit.
M_Covar	Output	Optionally creates a matrix wave containing the “covariance matrix”. It can be used to generate advanced statistics.
Other waves	Optional input and output	User-supplied or automatically generated waves for displaying confidence and prediction bands, and for specifying constraints on coefficient values.

Curve Fitting Dialog Tabs

This section describes the controls on each tab and on the main pane of the Curve Fitting dialog.

Global Controls

The controls at the bottom of the dialog are always available:



Show:

○ Commands

● Equation

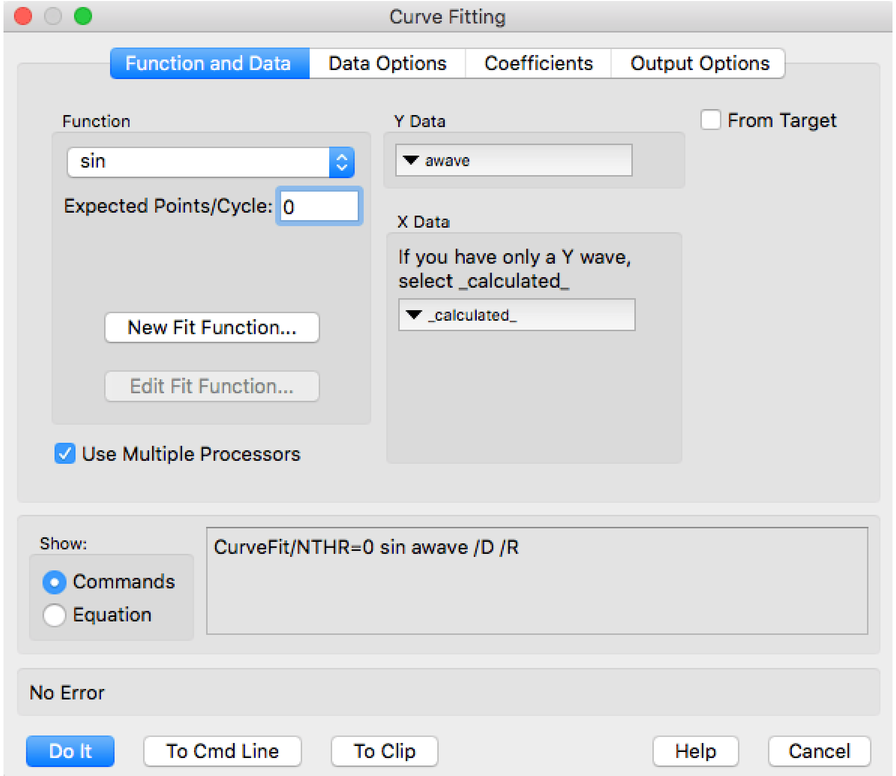
$$y_0 + A\sin(f \cdot x + phi)$$

No Error

If you click the Commands radio button, Igor displays the commands generated by the dialog instead of the equation.

Function and Data Tab

The Function and Data tab has a variety of appearances depending on the function chosen in the Function menu. Here is what it looks like when the built-in sin function is chosen:



Function: sin

Expected Points/Cycle: 0

Y Data: awave

□ From Target

X Data

If you have only a Y wave, select _calculated_

calculated

☑ Use Multiple Processors

Show:

● Commands

○ Equation

```
CurveFit/NTHR=0 sin awave /D /R
```

No Error

Function menu: The main purpose is to choose a function to fit.

The menu also has two items that control what functions are displayed in the menu.

Choose Show Multivariate functions to include functions of more than one independent variable.

Choose Show Old-Style Functions to display functions that lack the FitFunc keyword. See **User-Defined Fitting Functions** on page III-250 for details on the FitFunc keyword. You may need to choose this item if you have fitting functions from a version of Igor Pro older than version 4.

Some of the fit functions require additional information that is collected by customized items that appear when you select the function.

Polynomial Terms: Appears when you choose the poly function to fit a polynomial. Note that the number of terms is one greater than the degree — set this to three for a quadratic polynomial.

Set Constant X0: Appears when you select the exp_XOffset , dblexp_XOffset or poly_XOffset function. X0 is a constant subtracted from the X values to move the X range for fitting closer to zero. This eliminates numerical problems that arise when evaluating exponentials over X ranges even moderately far from zero. Setting X0 to Auto causes it to be set to the minimum X value in your input data. Setting to a number overrides this default behavior.

Expected Points/Cycle: Appears when you choose the sin function. Use it to set the approximate number of data points in one cycle of the sin function. Helps the automatic initial guess come up with good guesses. If it is set to less than four, Igor will use the default of seven.

2D Polynomial Order: Appears when you choose the Poly2D function to fit a two-dimensional polynomial (a multivariate function — only appears in the menu when you have chosen Show Multivariate Functions). Sets the *order* of the polynomial, not the number of terms. Because a given order includes a term for X and a term for Y plus cross terms, the number of terms is (N+1)(N+2)/2 where N is the order.

New Fit Function: Click this button to bring up a dialog in which you can define your own fitting function.

Edit Fit Function: This button is available when you have chosen a user-defined function from the Function menu. Brings up a dialog in which you can edit your user-defined function.

Y Data: Select a wave containing the dependent variable data to fit. When a multivariate function is chosen in the Function menu, the Y Data menu shows1D waves and waves with dimensions matching the number of independent variables used by the function.

X Data: This area changes depending on the function and Y Data wave chosen.

With a **univariate function**, just the X data menu is shown. Choose _calculated_ if you have just a Y wave; the X values will come from the Y wave's X scaling. The X Data menu shows only waves with the same number of points as the Y wave.

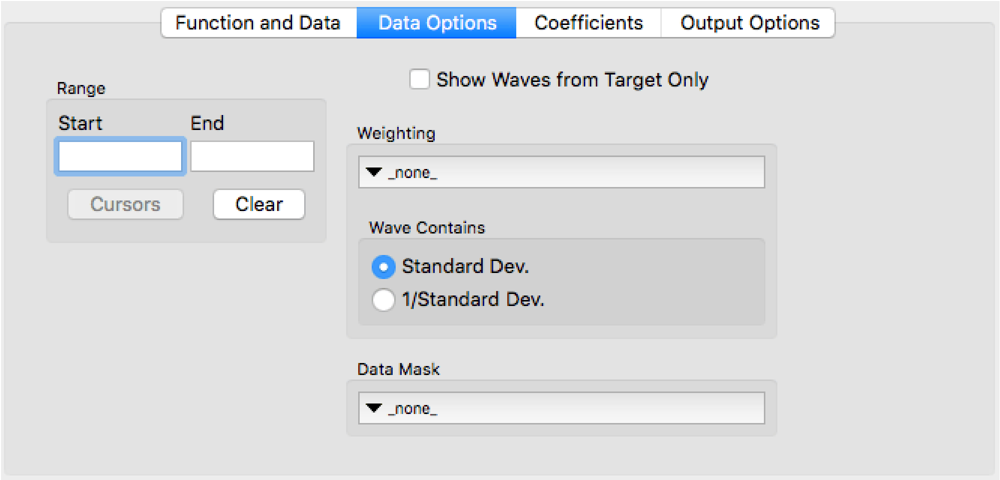
When a **multivariate function** and a **1D Y wave** are selected, it adds a list box below the X wave menu. You must select one X wave for each independent variable used by the fit function, or a single multicolumn wave with the correct number of columns. As you select X waves, they are added to the list. The order of waves in the list is important — it determines which is identified with each of the function's independent variables.

Remove waves from the list by highlighting a wave and pressing Delete (*Macintosh*) or Backspace (*Windows*).

With a **multivariate function** and a **multidimensional Y wave** selected, it displays four independent variable wave menus, one for each dimension that a wave can have. Each menu displays waves of length matching the corresponding dimension of the Y Data wave. Choose "_calculated_" if the independent variable values will come from the Y wave's dimension scaling.

From Target: When From Target is selected, the Y Data and X Data menus show only waves that are displayed in the top table or graph. Makes it easier to find waves in the menus when you are working on an experiment with many waves. When you click the From Target checkbox, it will attempt to select appropriate waves for the X and Y data.

Data Options Tab



Range: Enter point numbers for starting and ending points when fitting to a subset of the Y data.

Historical Note: These boxes used to require X values, they now require point numbers.

Cursors: Available when the top window is a graph displaying the Y data and the graph has the graph cursors on the Y data trace. Click this button to enter text in the Start and End range boxes that will restrict fitting to the data between the graph cursors.

Note: If the data use both a Y wave and an X wave, and the X values are in random order, you won't get the expected result.

Clear: Click this button to remove text from the Start and End range boxes.

The **range** box changes if you have selected a multivariate function and multidimensional Y wave. The dialog presents Start and End range boxes for each dimension of the Y wave.

Weighting: select a wave that contains weighting values. Only waves that match the Y wave in number of points and dimensions are shown in this menu. See **Weighting** on page III-199 for details.

Wave Contains: select Standard Deviation if the weighting wave contains values of standard deviation for each Y data point. A larger value decreases the influence of a point on the fit.

Select 1/Standard Deviation if your weighting wave contains values of the reciprocal of the standard deviation. A larger value increases the influence of the point on the fit.

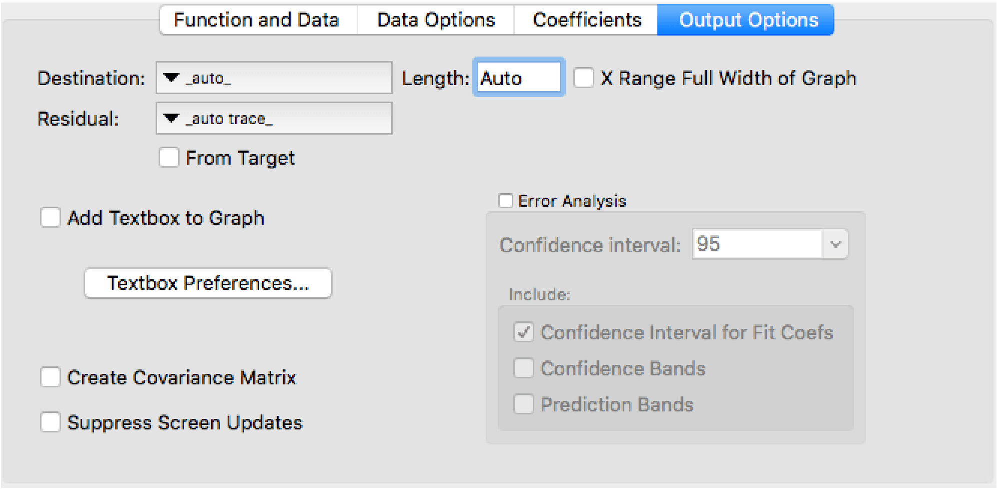
Data Mask: select a wave that contains ones and zeroes or NaN's indicating which Y Data points should be included in the fit. Only waves that match the Y wave in number of points and dimensions are shown in this menu. A one indicates a data point that should be included, a zero or NaN (Not a Number or blank in a table) indicates a point that should be excluded.

Coefficients Tab

The coefficients tab is quite complex. It is completely explained in the various sections on how to do a fit. See **Two Useful Additions: Holding a Coefficient and Generating Residuals** on page III-184, **Automatic Guesses Didn't Work** on page III-187, **Coefficients Tab for a User-Defined Function** on page III-192, and **The Coefficient Wave** on page III-195.

Output Options Tab

The output options tab has settings that control the reporting and display of fit results:



Destination: _auto_ Length: Auto □ X Range Full Width of Graph

Residual: _auto trace_

□ From Target

□ Add Textbox to Graph

□ Error Analysis

Confidence interval: 95

Include:

☑ Confidence Interval for Fit Coefs

□ Confidence Bands

□ Prediction Bands

□ Create Covariance Matrix

□ Suppress Screen Updates

Destination: Select a wave to receive model values from the fit, or select _auto_ to have Igor create an evenly-spaced auto-destination wave for the model values. Updated on each iteration so you can follow the fit progress by the graph display. See **The Destination Wave** on page III-196 for details on the Destination menu, the Length box shown above and on the New Wave box that isn't shown above.

X Range Full Width of Graph: If you have restricted the range of the fit using graph cursors, the auto destination wave will cover only the range selected. Select this checkbox to make the auto destination cover the full width of the graph.

Residual: Select a wave to receive calculated values of residuals, or the differences between the model and the data. See **Computing Residuals** on page III-217 for details on residuals and on the various selections you can make from this menu.

Error Analysis: Selects various kinds of statistical error analysis. See **Confidence Bands and Coefficient Confidence Intervals** on page III-223 for details.

Add Textbox to Graph: When selected, a textbox with information about the fit will be added to the graph containing the Y data. Click the **Textbox Preferences** button to display a dialog in which you can select various bits of information to be included in the text box.

Create Covariance Matrix: When this is selected, the dialog generates the command to create a covariance matrix for the fit. See **Covariance Matrix** on page III-226 for details on the covariance matrix.

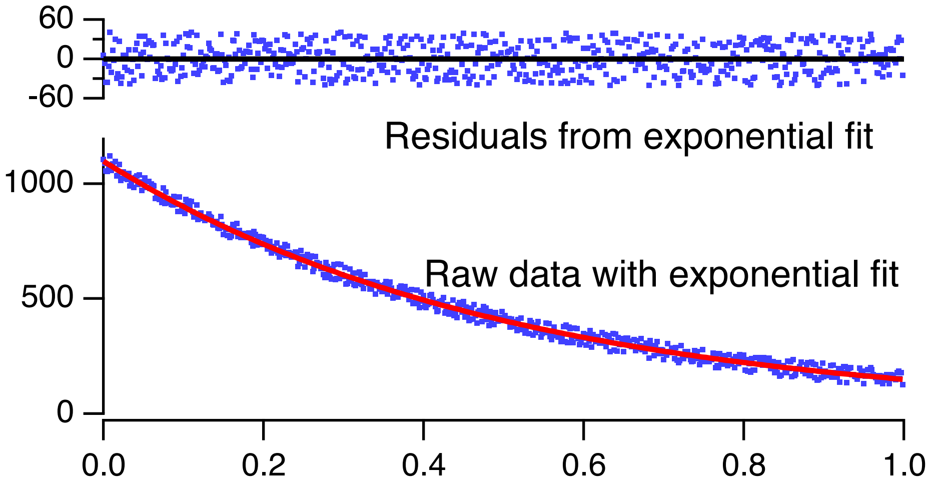
Suppress Screen Updates: When this is selected, graphs and tables are not updated while the fit progresses. This can greatly speed up the fitting process, especially if the fit involves a contour or image plot, but reduces the feedback you get during the fit.

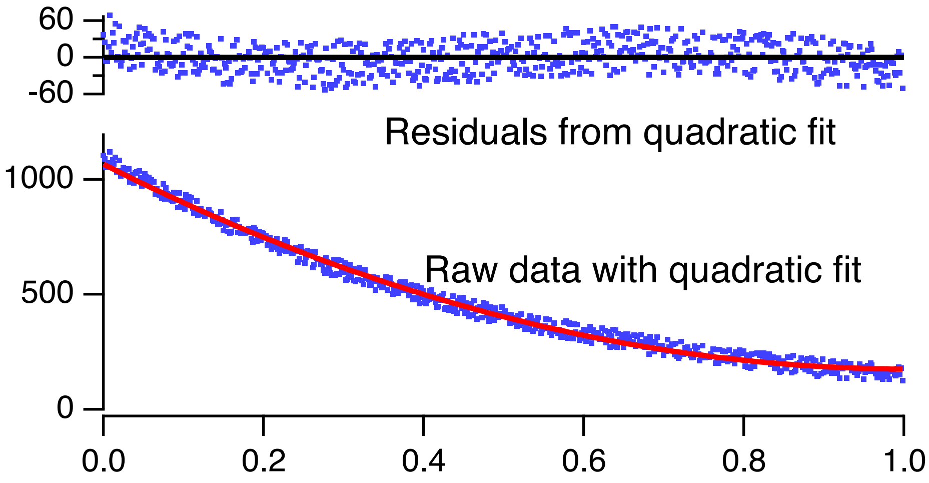
Computing Residuals

A residual is what is left when you subtract your fitting function model from your raw data.

Ideally, your raw data is equal to some known function plus random noise. If you subtract the function from your data, what's left should be noise. If this is not the case, then the function doesn't properly fit your raw data.

The graphs below illustrate some exponential raw data fitted to an exponential function and to a quadratic (3 term polynomial). The residuals from the exponential fit are random whereas the residuals from the quadratic display a trend overlaid on the random scatter. This indicates that the quadratic is not a good fit for the data.





The easiest way to make a graph such as these is to let it proceed automatically using the Residual pop-up menu in the Output Options tab of the Curve Fitting dialog. The graphs above were made this way with some minor tweaks to improve the display.

The residuals are recalculated at every iteration of a fit. If the residuals are displayed on a graph, you can watch the residuals change as the fit proceeds.

In addition to providing an easy way to compute residuals and add the residual plot to a graph, it prints the wave assignment used to create the residuals into the history area as part of the curve fitting process. For instance, this is the result of a line fit to waveform data:

```
•CurveFit line LineYData /X=LineXData /D /R
   fit_LineYData= W_coef[0]+W_coef[1]*x
   Res_LineYData= LineYData[p] - (W_coef[0]+W_coef[1]*LineXData[p])
   W_coef={-0.037971,2.9298}
   V_chisq= 18.25;V_npnts= 20;V_numNaNs= 0;V_numINFs= 0;
   V_startRow= 0;V_endRow= 19;V_q= 1;V_Rab= -0.879789;
   V_Pr= 0.956769;V_r2= 0.915408;
   W_sigma={0.474,0.21}
   Coefficient values ± one standard deviation
      a  =-0.037971 ± 0.474
      b  =2.9298 ± 0.21
```

In this case, a wave called “LineYData” was fit with a straight line model. _auto trace_ was selected for both the destination (/D flag) and the residual (/R flag), so the waves fit_LineYData and Res_LineYData were created. The third line above gives the equation for calculating the residuals. You can copy this line if you wish to recalculate the residuals at a later time. You can edit the line if you want to calculate the residuals in a different way. See **Calculating Residuals After the Fit** on page III-220.

Residuals Using Auto Trace

If you choose _auto trace_ from the Residual menu, it will automatically create a wave for residual values and, if the Y data are displayed in the top graph, it will append the residual wave to the graph.

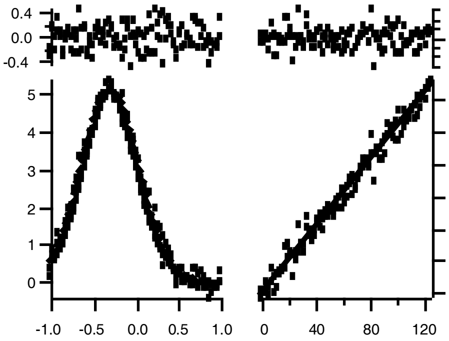
The automatically-created residual wave is named by prepending “Res_” to the name of the Y data wave. If the resulting name is too long, it will be truncated. If a wave with that name already exists, it will be reused, overwriting the values already in the wave. If a wave does not exist already, the newly-created wave is automatically filled with NaN. The residual wave is made with the same number of points as the data wave, with one residual value calculated for each data point.

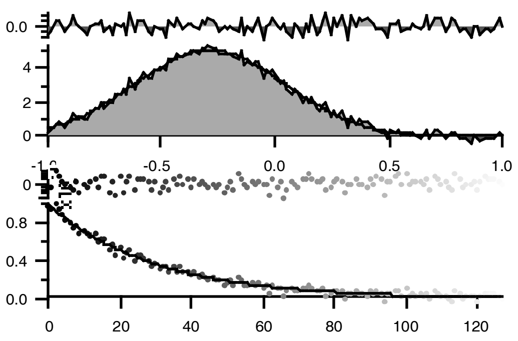
Residual values are stored only for data points that actually participate in fitting. That is, if you fit to a subrange of the data, or use a mask wave to eliminate some points from fitting, residuals are stored only for the data used in fitting and the other points are not disturbed. This allows you to use the auto-residual option to build up the residuals from a piece-wise fit, where parts of the data are fit in succession. It also means that sometimes Igor will leave behind stale values. To prevent that, pre-fill your residual wave with a value of your choice.

If you fit the same data again, the same residual wave will be used. Thus, to preserve a previous result, you must rename the residual wave. This can be done using the Rename item in the Data menu, or the Rename command on the command line.

Residuals are displayed on a stacked graph like those above. This is done by shortening the axis used to display the Y data, and positioning a new free axis above the axis used for the Y data. The residual plot occupies the space made by shortening the data axis. The axis that Igor creates is given a name derived from the axis used to display the Y data, either “Res_Left” or “Res_Right” if the Y data are displayed on the standard axes. If the data axis is a named free axis, the automatically-generated axis is given a name created by prepending “Res_” to the name of the data axis.

Igor tries hard to make the appended residual plot look nice by copying the formatting of the Y data trace and making the residual trace identical so that you can identify which residual trace goes with a given Y data trace. The axis formatting for the residual axis is copied from the vertical axis used for the data trace, and the residual axis is positioned directly above the data axis. Any other vertical axes that might be in the way are also shortened to make room. Here are two examples of graphs that have been modified by the auto-trace residuals:





It is likely that the automatic formatting won’t be quite what you want, but it will probably be close enough that you can use the Modify Axis and Modify Trace Appearance dialogs to tweak the formatting. For details see Chapter II-13, **Graphs**, especially the sections **Creating Graphs with Multiple Axes** on page II-324 and **Creating Stacked Plots** on page II-324.

Removing the Residual Auto Trace

When an auto-trace residual plot is added to a graph, it modifies the axis used to plot the original Y data. If you remove the auto-trace residual from the graph, the residual axis is removed and in most cases the Y data axis is restored to its previous state.

In some complicated graphs the restoration of the data axis isn't done correctly. To restore the graph, double-click the Y data axis to bring up the Modify Axes dialog and select the Axis tab. You will find two settings labeled "Draw between … and … % of normal". Typically, the correct settings will be 0 and 100.

Residuals Using Auto Wave

Because the changes to the graph formatting are substantial, you may want the automatic residual wave created and filled in but not appended to the graph. To accomplish this, simply choose _Auto Wave_ from the Residual pop-up menu in the Curve Fitting dialog. Once the wave is made by the curve fitting process, you can append it to a graph, or use it in any other way you wish.

Residuals Using an Explicit Residual Wave

You can choose a wave from the Residual pop-up menu and that wave will be filled with residual values. In this case, it does not append the wave to the top graph. You can use this technique to make graphs with the formatting completely under your own control, or to use the residuals for some other purpose.

Only waves having the same number of points as the Y data wave are listed in the menu. If you don't want to let the dialog create the wave, you would first create a suitable wave by duplicating the Y data wave using the Duplicate Waves item in the Data menu, or using the Duplicate command:

```
Duplicate yData, residuals_poly3
```

It is often a good idea to set the wave to NaN, especially when fitting to a subrange of the data:

```
residuals_poly3=NaN
```

After the wave is duplicated, it would typically be appended to a graph or table before it is used in curve fitting.

Explicit Residual Wave Using New Wave

The easiest way to make an explicit residual wave is to let the Curve Fitting dialog do it for you. You do this by choosing _New Wave_ in the Residual menu. A box appears allowing you to enter a name for the new wave. The dialog will then generate the required commands to make the wave before the curve fit starts. The wave made this way will not be added to a graph. You will need to do that yourself after the fit is finished.

Calculating Residuals After the Fit

You may wish to avoid the overhead of calculating residuals during the fitting process. This section describes ways to calculate the residuals after a curve fit without depending on automatic wave creation.

Graphs similar to the ones at the top of this section can be made by appending a residuals wave using a free left axis. Then, under the Axis tab of the Modify Axes dialog, the distance for the free axis was set to zero and the axis was set to draw between 80 and 100% of normal. The normal left axis was set to draw between 0 and 70% and axis standoff was turned off for both left and bottom axes.

Here are representative commands used to accomplish this.

```
// Make sample data
Make/N=500 xData, yData
xData = x/500 + gnoise(1/1000)
yData = 100 + 1000*exp(-.005*x) + gnoise(20)

// Do exponential fit with auto-trace
CurveFit exp yData /X=xData /D
Rename fit_yData, fit_yData_exp

// Calculate exponential residuals using interpolation in
// the auto trace wave to get model values
```

```
Duplicate yData, residuals_exp
residuals_exp = yData - fit_yData_exp(xData)

// Do polynomial fit with auto-trace
CurveFit poly 3, yData /X=xData /D
Rename fit_yData fit_yData_poly3

// Find polynomial residuals
Duplicate yData, residuals_poly3
residuals_poly3 = yData - fit_yData_poly3(xData)
```

Calculating model values by interpolating in the auto trace wave may not be sufficiently accurate. Instead, you can calculate the exact value using the actual fitting expression. When the fit finishes, it prints the equation for the fit in the history. With a little bit of editing you can create a wave assignment for calculating residuals. Here are the assignments from the history for the above fits:

```
fit_yData= poly(W_coef,x)
fit_yData= W_coef[0]+W_coef[1]*exp(-W_coef[2]*x)
```

We can convert these into residuals calculations like this:

```
residuals_poly3 = yData - poly(W_coef,xData)
residuals_exp = yData - (W_coef[0]+W_coef[1]*exp(-W_coef[2]*xData))
```

Note that we replaced “x” with “xData” because we have tabulated x values. If we had been fitting equally spaced data then we would not have had a wave of tabulated x values and would have left the “x” alone.

This technique for calculating residuals can also be used if you create and use an explicit destination wave. In this case the residuals are simply the difference between the data and the destination wave. For example, we could have done the exp fit and residual calculations as follows:

```
Duplicate yData, yDataExpFit,residuals_exp
// explicit destination wave using /D=wave
CurveFit exp yData /X=xData /D=yDataExpFit
residuals_exp = yData - yDataExpFit
```

Coefficient of Determination or R-Squared

When you do a line fit, y = a + bx, Igor prints a variety of statistics, including "r-squared", also known as the "coefficient of determination". The value of r-squared is stored in the automatically-created variable V_r2.

The literature on regression shows differences of opinion on the interpretation of r-squared. Because it is commonly reported, Igor provides the value.

There are two ways to compute r-squared that reflect slightly different interpretations:

$$r^2 = 1 - \frac{SS_{res}}{SS_{tot}} \qquad (1)$$

$$r^2 = \frac{SS_{reg}}{SS_{tot}} \qquad (2)$$

where

$$SS_{tot} = \sum_{i=1}^{n} (y_i - \overline{y})^2 \qquad \text{(3 - total sum of squares)}$$

$$SS_{res} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \qquad \text{(4 - residual sum of squares)}$$

$$SS_{reg} = \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2 \qquad \text{(5 - regression sum of squares)}$$

Here, y_i is the i-th Y data point, $\hat{y}_i$ is i-th predicted or model Y value, and $\bar{y}_i$ is the average of the Y data. When you do a line fit, the fit line is guaranteed to pass through the data average, and you can be sure that

SStot = SSres + SSreg (6)

Consequently, equations (1) and (2) are equivalent, with equation (2) emphasizing the interpretation of r-squared as "explained variance". That is, it is the fraction of the data variance that is "explained" or accounted for by the fit line. The r-squared value reported by Igor generally agrees with values from Excel and other applications.

Igor also reports the Pearson correlation coeffient via the automatically-created variable V_Pr. In the case of a line fit, V_Pr^2 = V_r2.

R-Squared and Fits Through the Origin

The situation changes if you fit a line that is constrained to pass through the origin, that is, you set the value of the fit coefficient a to zero and hold it. This is usually done when common sense or theory demands that the quantity of interest must be zero at X=0.

There is no consensus on the correct way to compute something like r-squared for a line fit through the origin. Such a fit does not, in general, pass through the average data value, and equation (6) does not hold, so equations (1) and (2) do not give the same value.

We prefer the interpretation of r-squared as "explained variance" so we use equation (2) for fits through the origin. This has some consequences. One is that Igor's r-squared will not necessarily agree with other applications, notably Excel. Another is that it is guaranteed to be smaller, some would say, "less good", than a line fit to the same data that is not constrained to pass through the origin.

A more surprising consequence is that it is possible for r-squared to be larger than 1. This has a simple interpretation: your fit line has larger variance than the data! Quite possibly a fit through the origin is simply not justified for your data.

Given the lack of consensus and uncertain interpretation of r-squared for a fit through the origin, we cannot recommend citing it, or using it as any sort of indication of goodness of fit, for such fits.

Estimates of Error

Igor automatically calculates the estimated error (standard deviation) for each of the coefficients in a curve fit. When you perform a curve fit, it creates a wave called W_sigma. Each point of W_sigma is set to the estimated error of the corresponding coefficients in the fit. The estimated errors are also indicated in the history area, along with the other results from the fit. If you don’t provide a weighting wave, the sigma values are estimated from the residuals. This implicitly assumes that the errors are normally distributed with zero mean and constant variance and that the fit function is a good description of the data.

The coefficients and their sigma values are estimates (usually remarkably good estimates) of what you would get if you performed the same fit an infinite number of times on the same underlying data (but with different noise each time) and then calculated the mean and standard deviation for each coefficient.

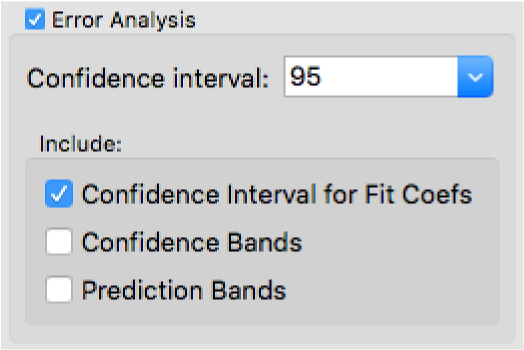
Confidence Bands and Coefficient Confidence Intervals

You can graphically display the uncertainty of a model fit to data by adding confidence bands or prediction bands to your graph. These are curves that show the region within which your model or measured data are expected to fall with a certain level of probability. A confidence band shows the region within which the model is expected to fall while a prediction band shows the region within which random samples from that model plus random errors are expected to fall.

You can also calculate a confidence interval for the fit coefficients. A confidence interval estimates the interval within which the real coefficient will fall with a certain probability.

Note: Confidence and prediction bands are not available for multivariate curve fits.

You control the display of confidence and prediction bands and the calculation of coefficient confidence intervals using the Error Analysis section of the Output Options tab of the Curve Fitting dialog:



☑ Error Analysis

Confidence interval: 95

Include:

☑ Confidence Interval for Fit Coefs

☐ Confidence Bands

☐ Prediction Bands

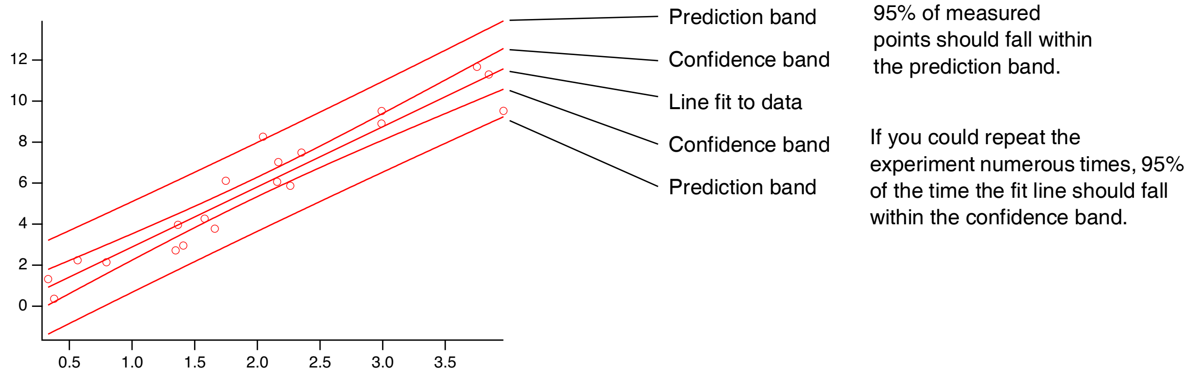
Using the line fit example at the beginning of this chapter (see **A Simple Case — Fitting to a Built-In Function: Line Fit** on page III-182), we set the confidence level to 95% and selected all three error analysis options to generate this output and graph:

Dialog has added the /F with parameters to select error analysis options.

```
•CurveFit line LineYData /X=LineXData /D /F={0.950000, 7}
   fit_LineYData= W_coef[0]+W_coef[1]*x
   W_coef={-0.037971,2.9298}
   V_chisq= 18.25; V_npnts= 20; V_numNaNs= 0; V_numINFs= 0;
   V_startRow= 0; V_endRow= 19; V_q= 1; V_Rab= -0.879789;
   V_Pr= 0.956769;V_r2= 0.915408;
   W_sigma={0.474,0.21}
   Fit coefficient confidence intervals at 95.00% confidence level:
   W_ParamConfidenceInterval={0.995,0.441,0.95}
   Coefficient values ± 95.00% Confidence Interval
      a   =   -0.037971 ± 0.995
      b   =   2.9298 ± 0.441
```

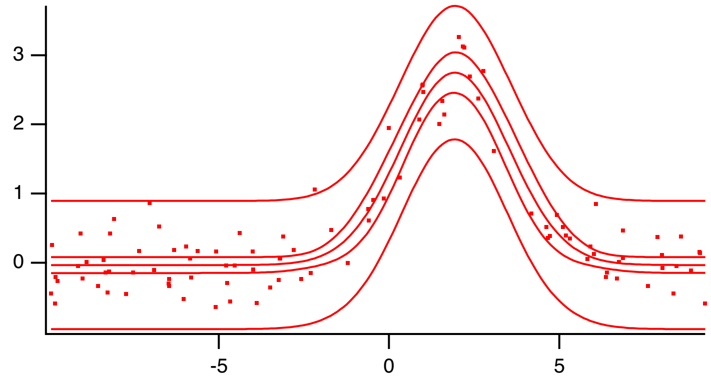
When confidence intervals are available they are listed here instead of the standard deviation.

Coefficient confidence intervals are stored in the wave W_ParamConfidenceInterval. Note that the last point in the wave contains the confidence level used in the calculation.



You can do this with nonlinear functions also, but be aware that it is only an approximation for nonlinear functions:

```
Make/O/N=100 GDataX, GDataY                       // waves for data
GDataX = enoise(10)                               // Random X values
GDataY = 3*exp(-((GDataX-2)/2)^2) + gnoise(0.3)  // Gaussian plus noise
Display GDataY vs GDataX                          // graph the data
ModifyGraph mode=2,lsize=2                        // as dots
CurveFit Gauss GDataY /X=GDataX /D/F={.99, 3}
```



The dialog supports only automatically-generated waves for confidence bands. The CurveFit and FuncFit operations support several other options including an error bar-style display. See **CurveFit** on page V-124 for details.

Calculating Confidence Intervals After the Fit

You can use the values from the W_sigma wave to calculate a confidence interval for the fit coefficients after the fit is finished. Use the StudentT function to do this. The following information was printed into the history following a curve fit:

```
  Fit converged properly
  fit_junk= f(coeffs,x)
  coeffs={4.3039,1.9014}
  V_chisq= 101695; V_npnts= 128; V_numNaNs= 0; V_numINFs= 0;
  W_sigma={4.99,0.0679}
```

To calculate the 95 per cent confidence interval for fit coefficients and deposit the values into another wave, you could execute the following lines:

```
Duplicate W_sigma, ConfInterval
ConfInterval = W_sigma[p]*StudentT(0.95, V_npnts-numpnts(coeffs))
```

Naturally, you could simply type “126” instead of “V_npnts-numpnts(coeffs)”, but as written the line will work unaltered for any fit. When we did this following the fit in the example, these were the results:

```
ConfInterval = {9.86734,0.134469}
```

Clearly, coeffs[0] is not significantly different from zero.

Confidence Band Waves

New waves containing values required to display confidence and prediction bands are created by the curve fit if you have selected these options. The number of waves and the names depend on which options are selected and the style of display. For a contour band, such as shown above, there are two waves: one for the upper contour and one for the lower contour. Only one wave is required to display error bars. For details, see the **CurveFit** operation on page V-124.

Some Statistics

Calculation of the confidence and prediction bands involve some statistical assumptions. First, of course, the measurement errors are assumed to be normally distributed. Departures from normality usually have to be fairly substantial to cause real problems.

If you don’t supply a weighting wave, the distribution of errors is estimated from the residuals. In making this estimate, the distribution is assumed to be not only normal, but also uniform with mean of zero. That is, the error distribution is centered on the model and the standard deviation of the errors is the same for all values of the independent variable. The assumption of zero mean requires that the model be correct; that is, it assumes that the measured data truly represent the model plus random normal errors.

Some data sets are not well characterized by the assumption that the errors are uniform. In that case, you should specify the error distribution by supplying a weighting wave (see **Weighting** on page III-199). If you do this, your error estimates are used for determining the uncertainties in the fit coefficients, and, therefore, also in calculating the confidence band.

The confidence band relies only on the model coefficients and the estimated uncertainties in the coefficients, and will always be calculated taking into account error estimates provided by a weighting wave. The prediction band, on the other hand, also depends on the distribution of measurement errors at each point. These errors are not taken into account, however, and only the uniform measurement error estimated from the residuals are used.

The calculation of the confidence and prediction bands is based on an estimate of the variance of a predicted model value:

$$V(\hat{Y}) = a^T C a$$
$$a = \delta F / \delta p|_x$$

Here, $\hat{Y}$ is the predicted value of the model at a given value of the independent variable X, $\boldsymbol{a}$ is the vector of partial derivatives of the model with respect to the coefficients evaluated at the given value of the independent variable, and $\boldsymbol{C}$ is the covariance matrix. Often you see the $\boldsymbol{a}^T\boldsymbol{Ca}$ term multiplied by σ^2, the sample variance, but this is included in the covariance matrix. The confidence interval and prediction interval are calculated as:

$$CI = t(n-p,\, 1-\alpha/2)[V(\hat{Y})]^{1/2} \text{ and } PI = t(n-p, 1-\alpha/2)[\sigma^2 + V(\hat{Y})]^{1/2}.$$

The quantities calculated by these equations are the magnitudes of the intervals. These are the values used for error bars. These values are added to the model values ($\hat{Y}$) to generate the waves used to display the bands as contours. The function $t(n-p,\, 1-\alpha/2)$ is the point on a Student’s t distribution having probability $1-\alpha/2$, and σ^2 is the sample variance. In the calculation of the prediction interval, the value used for σ^2 is the uniform value estimated from the residuals. This is not correct if you have supplied a weighting wave with nonuniform values because there is no information on the correct values of the sample variance for arbitrary values of the independent variable. You can calculate the correct prediction interval using the **StudentT** function. You will need a value of the derivatives of your fitting function with respect to the fitting coefficients. You can either differentiate your function and write another function to provide derivatives, or you can use a numerical approximation. Igor uses a numerical approximation.

Confidence Bands and Nonlinear Functions

Strictly speaking, the discussion and equations above are only correct for functions that are linear in the fitting coefficients. In that case, the vector **a** is simply a vector of the basis functions. For a polynomial fit, that means 1, x, x^2, etc. When the fitting function is nonlinear, the equation results from an approximation that uses only the linear term of a Taylor expansion. Thus, the confidence bands and prediction bands are only approximate. It is impossible to say how bad the approximation is, as it depends heavily on the function.

Covariance Matrix

If you select the Create Covariance Matrix checkbox in the Output Options tab of the Curve Fitting dialog, it generates a covariance matrix for the curve fitting coefficients. This is available for all of the fits except the straight-line fit. Instead, a straight-line fit generates the special output variable V_rab giving the coefficient of correlation between the slope and Y intercept.

By default (if you are using the Curve Fitting dialog) it generates a matrix wave having N rows and columns, where N is the number of coefficients. The name of the wave is M_Covar. This wave can be used in matrix operations. If you are using the CurveFit, FuncFit or FuncFitMD operations from the command line or in a user procedure, use the /M=2 flag to generate a matrix wave.

Originally, curve fits created one 1D wave for each fit coefficient. The waves taken all together made up the covariance matrix. For compatibility with previous versions, the /M=1 flag still produces multiple 1D waves with names W_Covar*n*. Please don’t do this on purpose.

The diagonal elements of the matrix, M_Covar[i][i], are the variances for parameter i. The variance is the square of sigma, the standard deviation of the estimated error for that parameter.

The covariance matrix is described in detail in *Numerical Recipes in C*, edition 2, page 685 and section 15.5. Also see the discussion under **Weighting** on page III-199.

Correlation Matrix

Use the following commands to calculate a correlation matrix from the covariance matrix produced during a curve fit:

```
Duplicate M_Covar, CorMat  // You can use any name instead of CorMat
CorMat = M_Covar[p][q]/sqrt(M_Covar[p][p]*M_Covar[q][q])
```

A correlation matrix is a normalized form of the covariance matrix. Each element shows the correlation between two fit coefficients as a number between -1 and 1. The correlation between two coefficients is perfect if the corresponding element is 1, it is a perfect inverse correlation if the element is -1, and there is no correlation if it is 0.

Curve fits in which an element of the correlation matrix is very close to 1 or -1 may signal “identifiability” problems. That is, the fit doesn’t distinguish between two of the parameters very well, and so the fit isn’t very well constrained. Sometimes a fit can be rewritten with new parameters that are combinations of the old ones to get around this problem.

Identifiability Problems

In addition to off-diagonal elements of the correlation matrix that are near 1 or -1, symptoms of identifiability problems include fits that require a large number of iterations to converge, or fits in which the estimated coefficient errors (W_sigma wave) are unreasonably large.

The phrase "identifiability problems" describes a situation in which two or more of the fit coefficients trade off in a way that makes it nearly impossible to solve for the values of both at once. They are correlated in a way that if you adjust one coefficient, you can find a value of the other that makes a fit that is nearly as good.

When the correlation is too strong, the fitting algorithm doesn't know where to go and therefore wanders around in a coefficient space in which a broad range of values all seem about as good. That is, broad regions in chi-square space provide very little variation in chi-square. The usual result is apparent convergence but with large estimated values in W_sigma, or a singular matrix error.

The error estimates are based on the curvature of the chi-square surface around the solution point. A flat-bottomed chi-square surface, such as results from having many solutions that are nearly as good, results in large errors. The flat bottom of the chi-square surface also results in small derivatives with respect to the coefficients that don't give a good indication of where the fit should go next, so iterations wander around, giving rise to fits that require many iterations to converge.

If you see a fit with unreasonably large error estimates, or that take many iterations to converge, compute the correlation matrix and look for off-diagonal values near 1 or -1. In our experience, values about 0.9 are probably OK. Values near 0.99 are suspicious but can be acceptable. Values around 0.999 are almost certainly an indication of problems.

Unfortunately, there is little you can do about identifiability problems. It is a mathematical characteristic of your fitting function. Sometimes a model has regions in coefficient space where two coefficients have similar effects on a fit, and expanding the range of the independent variable can alleviate the problem. Occasionally some feature controlled by a coefficient might be very narrow and you can fix the problem with higher sampling density.

Fitting with Constraints

It is sometimes desirable to restrict values of the coefficients to a fitting function. Sometimes fitting functions may allow coefficient values that, while fine mathematically, are not physically reasonable. At other times, some ranges of coefficient values may cause mathematical problems such as singularities in the function values, or function values that are so large that they overflow the computer representation. In such cases it is often desirable to apply constraints to keep the solution out of the problem areas. It could be that the final solution doesn’t involve any active constraints, but the constraints prevent termination of the fit on an error caused by wandering into bad regions on the way to the solution.

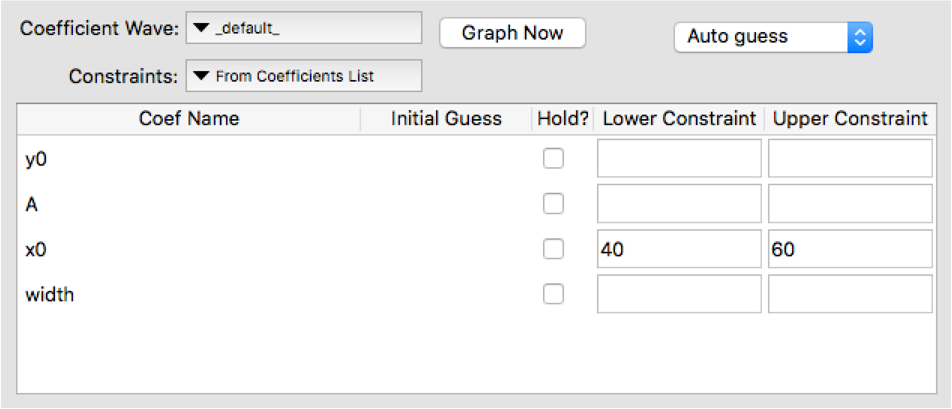
Curve fitting supports constraints on the values of any linear combination of the fit coefficients. The Curve Fitting dialog supports constraints on the value of individual coefficients.

The algorithm used to apply constraints is quite forgiving. Your initial guesses do not have to be within the constraint region (that is, initial guesses can violate the constraints). In most cases, it will simply move the parameters onto a boundary of the constraint region and proceed with the fit. Constraints can even be contradictory (“infeasible” in curve fitting jargon) so long as the violations aren’t too severe, and the fit will simply “split the difference” to give you coefficients that are a compromise amongst the infeasible constraints.

Constraints are not available for the built-in line, poly and poly2D fit functions. To apply constraints to these fit functions you must create a user-defined fit function.

Constraints Using the Curve Fitting Dialog

The Coefficients Tab of the Curve Fitting dialog includes a menu to enable fitting with constraints. When you select the checkbox, the constraints section of the Coefficients list becomes available:



Coefficient Wave: _default_

Constraints: From Coefficients List

Coef Name	Initial Guess	Hold?	Lower Constraint	Upper Constraint
y0		□		
A		□		
x0		□	40	60
width		□		

Filling in a value in the Lower Constraint column causes the dialog to generate a command to constrain the corresponding coefficient to values greater than the value you enter. Filling in a value in the Upper Constraint column constrains the corresponding coefficient to values less than the value you enter. A box that is left empty does not generate any constraint.

The figure above was made with the gauss function chosen. The following commands are generated by the dialog:

```
Make/O/T/N=2 T_Constraints
T_Constraints[0] = {"K2 > 40","K2 < 60"}
CurveFit gauss aa /D /C=T_Constraints
```

A Make command to make a text wave to contain constraint expressions.

A wave assignment to put constraint expressions into the wave.

/C parameter added to CurveFit command line to request a constrained fit.

More complicated constraints are possible, but cannot be entered in the Curve Fitting dialog. This requires that you make a constraints wave before you enter the dialog. Then choose the wave from the Constraints menu. See the following sections to learn how to construct the constraints wave.

Complex Constraints Using a Constraints Wave

You can constrain the values of linear combinations of coefficients, but the Curve Fitting dialog provides support only for simple constraints. You can construct an appropriate text wave with constraints before entering the Curve Fitting dialog. Select the wave from the Constraints menu in the Coefficients tab. You can also use the CurveFit or FuncFit commands on the command line with a constraints wave.

Each element of the text wave holds one constraint expression. Using a text wave makes it easy to edit the expressions in a table. Otherwise, you must use a command line like the second line in the example shown above.

Constraint Expressions

Constraint expressions can be arbitrarily complex, and can involve any or all of the fit coefficients. Each expression must have an inequality symbol (“<”, “<=”, “>”, or “>=”). In the expressions the symbol Kn (K0, K1, etc.) is used to represent the nth fitting coefficient. This is like the Kn system variables, but they are merely symbolic place holders in constraint expressions. Expressions can involve sums of any combination of the Kn’s and factors that multiply or divide the Kn’s. Factors may be arbitrarily complex, even nonlinear, as long as they do not involve any of the Kn’s. The Kn’s cannot be used in a call to a function, and cannot be involved in a nonlinear expression. Here are some legal constraint expressions:

```
K0 > 5
K1+K2 < numVar^2+2       // numVar is a global numeric variable
K0/5 < 2*K1
(numVar+3)*K3 > K1+K2/(numVar-2)
log(numVar)*K3 > 5       // nonlinear factor doesn’t involve K3
```

These are not legal:

```
K0*K1 > 5          // K0*K1 is nonlinear
1/K1 < 4           // This is nonlinear: division by K1
ln(K0) < 1         // K0 not allowed as parameter to a function
```

When constraint expressions are parsed, the factors that multiply or divide the Kn's are extracted as literal strings and evaluated separately. Thus, if you have <expression>*K0 or K0/<expression>, <expression> must be executable on its own.

You cannot use a text wave with constraint expressions for fitting from a threadsafe function. You must use the method described in **Constraint Matrix and Vector** on page III-230.

Equality Constraint

You may wish to constrain the value of a fit coefficient to be equal to a particular value. The constraint algorithm does not have a provision for equality constraints. One way to fake this is to use two constraints that require a coefficient to be both greater than and less than a value. For instance, "K1 > 5" and "K1 < 5" will require K1 to be equal to 5.

If it is a single parameter that is to be held equal to a value, this isn't the best method. You are much better off holding a parameter. In the Curve Fitting dialog, simply select the Hold box in the Coefficients list on the Coefficients tab and enter a value in the Initial Guess column. If you are using a command line to do the fit,

```
FuncFit/H="01"…
```

will hold K1 at a particular value. Note that you have to set that value before starting the fit.

Example Fit with Constraints

The examples here are available in the Curve Fitting help file where they can be conveniently executed directly from the help window.

This example fits to a sum of two exponentials, while constraining the sum of the exponential amplitudes to be less than some limit that might be imposed by theoretical knowledge. We use the command line because the constraint is too complicated to enter into the Curve Fitting dialog.

First, make the data and graph it:

```
Make/O/N=50 expData= 3*exp(-0.2*x) + 3*exp(-0.03*x) + gnoise(.1)
Display expData
ModifyGraph mode=3,marker=8
```

Do a fit without constraints:

```
CurveFit dblExp expData /D/R
```

The following command makes a text wave with a single element containing the string "K1 + K3 < 5" which implements a restriction on the sum of the individual exponential amplitudes.

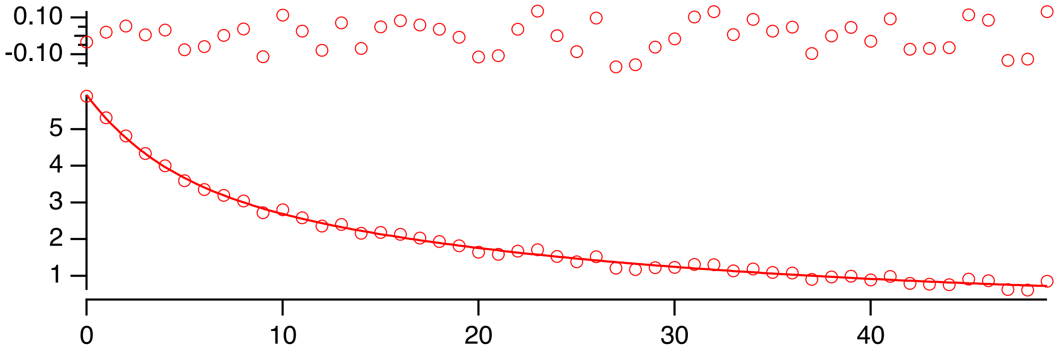
```
Make/O/T CTextWave={"K1 + K3 < 5"}
```

The wave is made using commands so that it could be written into this help file. It may be easier to use the Make Waves item from the Data menu to make the wave, and then display the wave in a table to edit the expressions. Make sure you make Text wave. Do not leave any blank lines in the wave.

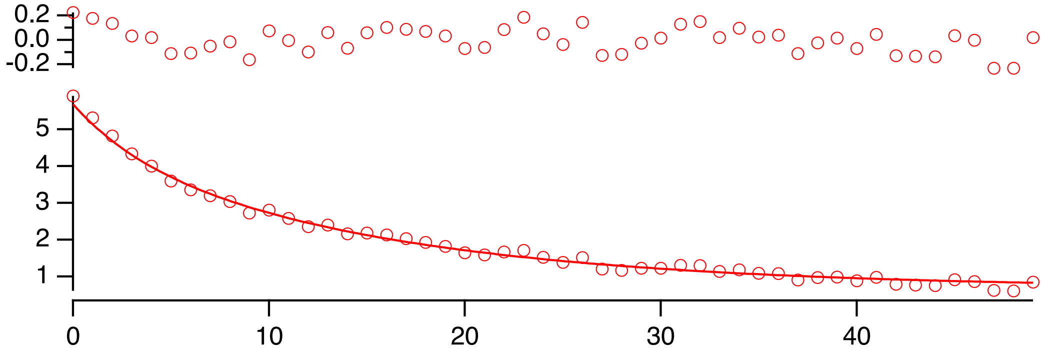
Now do the fit again with constraints:

```
CurveFit dblExp expData /D/R/C=CTextWave
```

In this case, the difference is slight; in the graph of the fit with constraints, notice that the fit line is slightly lower at the left end and slightly higher at the right end than in the standard curve fit, and that difference is reflected in the residual values at the ends:



Standard Curve Fit



Curve Fit with Constraints

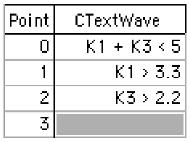
The output from a curve fit with constraints includes these lines reporting on the fact that constraints were used, and that the constraint was active in the solution:

```
  --Curve fit with constraints--
     Active Constraint: Desired:  K1+K3<5  Achieved:  K1+K3=5
```

In most cases you will see a message similar to this one. If you have conflicting constraints, it is likely that one or more constraints will be violated. In that case, you will get a report of that fact. The following commands add two more constraints to the example. The new constraints require values for the individual amplitudes that sum to a number greater than 5, while still requiring that the sum be less than 5 (so these are “infeasible constraints”):

```
Make/O/T CTextWave={"K1 + K3 < 5", "K1 > 3.3", "K3 > 2.2"}
CurveFit dblExp expData /D/R/C=CTextWave
```

In most cases, you would have added the new constraints by editing the constraint wave in a table.



Point	CTextWave
0	K1 + K3 < 5
1	K1 > 3.3
2	K3 > 2.2
3	

The curve fit report shows that all three constraints were violated to achieve a solution:

```
--Curve fit with constraints--
   Constraint VIOLATED:  Desired:  K1>3.3  Achieved:  K1=3.06604
   Constraint VIOLATED:  Desired:  K3>2.2  Achieved:  K3=1.93381
```

Constraint Matrix and Vector

When you do a constrained fit, it parses the constraint expressions and builds a matrix and a vector that describe the constraints.

Each constraint expression is parsed to form a simple expression like $C_0K_0 + C_1K_1 + \ldots \le D$, where the K_i’s are the fit coefficients, and the C_i’s and D are constants. The constraints can be expressed as the matrix operation **CK** ≤ **D**, where **C** is a matrix of constants, **K** is the fit coefficient vector, and **D** is a vector of limits. The matrix has dimensions N by M where N is the number of constraint expressions and M is the number of fit coefficients. In most cases, almost all the elements of the matrix are zeroes. In the previous example, the **C** matrix and **D** vector are:

C matrix					**D vector**
0	1	0	1	0	5
0	-1	0	0	0	-3.3
0	0	0	-1	0	-2.2

This is the form used internally by the curve-fitting code. If you wish, you can build a matrix wave and one-dimensional wave containing the *C* and *D* constants. Instead of using /C=*textwave* to request a fit with constraints, use `/C={matrix, 1Dwave}`.

In general, it is confusing and error-prone to create the right waves for a given set of constraints. However, it is not allowed to use the textwave method for curve fitting from a threadsafe function (see **Constraints and ThreadSafe Functions** on page III-250). Fortunately, Igor can create the necessary waves when it parses the expressions in a text wave.

You can create waves containing the **C** matrix and **D** vector using the /C flag (note that this flag goes right after the CurveFit command name, not at the end). If you have executed the examples above, you can now execute the following commands to build the waves and display them in a table:

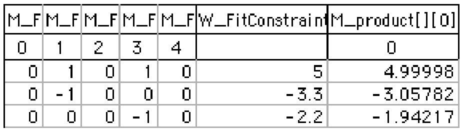
```
CurveFit/C dblExp expData /D/R/C=CTextWave
Edit M_FitConstraint, W_FitConstraint
```

The **C** matrix is named M_FitConstraint and the **D** vector is named W_FitConstraint (by convention, matrix names start with “M_”, and 1D wave names start with “W_”).

In addition to their usefulness for specifying constraints in a threadsafe function, you can use these waves later to check the constraints. The following commands multiply the generated fit coefficient wave (W_coefs) by the constraint matrix and appends the result to the table made previously:

```
MatrixMultiply M_FitConstraint, W_coef
AppendToTable M_Product
```

The result of the MatrixMultiply operation is the matrix wave M_Product. Note that the values of M_Product[1] and M_Product[2] are larger than the corresponding values in W_FitConstraint because the constraints were infeasible and resulted in constraint violations.



M_F	M_F	M_F	M_F	M_F	W_FitConstraint	M_product[][0]
0	1	2	3	4		0
0	1	0	1	0	5	4.99998
0	-1	0	0	0	-3.3	-3.05782
0	0	0	-1	0	-2.2	-1.94217

Constrained Curve Fitting Pitfalls

Bad Values on the Constraint Boundary

The most likely problem with constrained curve fitting is a value on a constraint boundary that produces a singular matrix error during the curve fit. For instance, it would be reasonable in many applications to require a pre-exponential multiplier to be positive, in this case K1>0:

$$y = K_0 + K_1 e^{K_2 x}$$

It would be natural to write a constraint expression “$K_1 > 0$”, but this can lead to problems. If some iteration tries a negative value of K_1, the constraints will set K_1 to be exactly zero. But when K1 is zero, the function has no dependence on K_2 and a singular matrix error results. The solution is to use a constraint that requires K_1 to be greater than some small positive number: “$K_1 > 0.1$”, for instance. The value of the limit must be tailored to your application. If you expect the constraint to be inactive when the solution is found, and the fit reports that the constraint was active, you can decrease the limit and do the fit again.

Severe Constraint Conflict

Although the method used for applying the constraints can find a solution even when constraints conflict with each other (are infeasible), it is possible to have a conflict that is so bad that the method fails. This will result in a singular matrix error.

Constraint Region is a Poor Fit

It is possible that the region of coefficient space allowed by the constraints is such a bad fit to the data that a singular matrix error results.

Initial Guesses Far Outside the Constraint Region

Usually if the initial guesses for a fit lie outside the region allowed by the constraints, the fit coefficients will shift into the constraint region on the first iteration. It is possible, however, for initial guesses to be so far from the constraint region that the solution to the constraints fails. This will cause the usual singular matrix error.

Constraints Conflict with a Held Parameter

You cannot hold a parameter and apply a constraint to the same parameter. Thus, this is not allowed:

```
Make/T CWave="K1 > 5"
FuncFit/H="01" myFunc, myCoefs, myData /C=CWave
```

NaNs and INFs in Curve Fits

Curve fits ignore NaNs and INFs in the input data. This is a convenient way to eliminate individual data values from a fit. A better way is to use a data mask wave (see **Using a Mask Wave** on page III-198).

Special Variables for Curve Fitting

There are a number of special variables used for curve fitting input (to provide additional control of the fit) and for output (to provide additional statistics). Knowledgeable users can use the input variables to tweak the fitting process. However, this is usually not needed. Some output variables help users knowledgeable in statistics to evaluate the quality of a curve fit.

To use an input variable interactively, create it from the command line using the Variable operation before performing the fit.

Most of the output variables are automatically created by the CurveFit or FuncFit operations. Some, as indicated below, are not automatically created; you must create them yourself if you want the information they provide.

If you are fitting using a procedure, both the input and output variables can be local or global. It is best to make them local. See **Accessing Variables Used by Igor Operations** on page IV-123 for information on how to use local variables. In procedures, output-only variables are always as local variables.

If you perform a curve fit interactively via the command line or via the Curve Fitting dialog, the variables will be global. If you use multiple data folders (described in Chapter II-8, **Data Folders**), you need to remember that input and output variables are searched for or created in the current data folder.

The following table lists all of the input and output special variables. Some variables are discussed in more detail in sections following the table.

Variable	I/O	Meaning
V_FitOptions	Input	Miscellaneous options for curve fit.
V_FitTol	Input	Normally, an iterative fit terminates when the fractional decrease of chi-square from one iteration to the next is less than 0.001. If you create a global variable named V_FitTol and set it to a value between 1E-10 and 0.1 then that value will be used as the termination tolerance. Values outside that range will have no effect.
V_tol	Input	(poly fit only) The "singular value threshold". See **Special Considerations for Polynomial Fits** on page III-265.
V_chisq	Output	A measure of the goodness of fit. It has absolute meaning only if you've specified a weighting wave containing the reciprocal of the standard error for each data point.
V_q	Output	(line fit only) A measure of the believability of chi-square. Valid only if you specified a weighting wave.
V_siga, V_sigb	Output	(line fit only) The probable uncertainties of the intercept (K0 = a) and slope (K1 = b) coefficients for a straight-line fit (to y = a + bx).
V_Rab	Output	(line fit only) The coefficient of correlation between the uncertainty in a (the intercept, K0) and the uncertainty in b (the slope, K1).
V_Pr	Output	(line fit only) The linear correlation coefficient r (also called Pearson's r). Values of +1 or -1 indicate complete correlation while values near zero indicate no correlation.
V_r2	Output	(line fit only) The coefficient of determination, usually called simply "r-squared". See **Coefficient of Determination or R-Squared** on page III-221 for details.
V_npnts	Output	The number of points that were fitted. If you specified a weighting wave then points whose weighting was zero are not included in this count. Also not included are points whose values are NaN or INF.
V_nterms	Output	The number of coefficients in the fit.
V_nheld	Output	The number of coefficients held constant during the fit.
V_numNaNs	Output	The number of NaN values in the fit data. NaNs are ignored during a curve fit.
V_numINFs	Output	The number of INF values in the fit data. INFs are ignored during a curve fit.
V_FitError	Input/ Output	Used from a procedure to attempt to recover from errors during the fit.
V_FitQuitReason	Output	Provides additional information about why a nonlinear fit stopped iterating. You must create this variable; it is not automatically created.
V_FitIterStart	Output	Use of V_FitIterStart is obsolete; use all-at-once fit functions instead. See **All-At-Once Fitting Functions** on page III-256 for details. Set to 1 when an iteration starts. Identifies when the user-defined fitting function is called for the first time for a particular iteration. You must create this variable; it is not automatically created.

Variable	I/O	Meaning
V_FitMaxIters	Input	Controls the maximum number of passes without convergence before stopping the fit. By default this is 40. You can set V_FitMaxIters to any value greater than 0. If V_FitMaxIters is less than 1 the default value of 40 is used.
V_FitNumIters	Output	Number of iterations. You must create this variable; it is not automatically created.
S_Info	Output	Keyword-value pairs giving certain kinds of information about the fit. You must create this variable; it is not automatically created.

V_FitOptions

There are a number of options that you can invoke for the fitting process by creating a variable named V_FitOptions and setting various bits in it. Set V_FitOptions to 1 to set Bit 0, to 2 to set Bit 1, etc.

Bit 0: Controls X Scaling of Auto-Trace Wave

If V_FitOptions exists and has bit 0 set (`Variable V_fitOptions=1`) and if the Y data wave is on the top graph then the X scaling of the auto-trace destination wave is set to match the appropriate x axis on the graph. This is useful when you want to extrapolate the curve outside the range of x data being fit.

A better way to do this is with the /X flag (not parameter- this flag goes immediately after the CurveFit or FuncFit operation and before the fit function name). See **CurveFit** for details.

Bit 1: Robust Fitting

You can get a form of robust fitting where the sum of the absolute deviations is minimized rather than the squares of the deviations, which tends to deemphasize outlier values. To do this, create V_FitOptions and set bit 1 (`Variable V_fitOptions=2`).

Warning 1: No attempt to adjust the results returned for the estimated errors or for the correlation matrix has been made. You are on your own.

Warning 2: Don’t set this bit and then forget about it.

Warning 3: Setting Bit 1 has no effect on line, poly or poly2D fits.

Bit 2: Suppresses Curve Fit Window

Normally, an iterative fit puts up an informative window while the fit is in progress. If you don’t want this window to appear, create V_FitOptions and set bit 2 (`Variable V_fitOptions=4`). This may speed things up a bit if you are performing batch fitting on a large number of data sets.

A better way to do this is via the /W=2 flag. See **CurveFit** for details.

Bit 3: Save Iterates

It is sometimes useful to know the path taken by a curve fit getting to a solution (or failing to). To save his information, create V_FitOptions and set bit 3 (`Variable V_FitOptions=8`). This creates a matrix wave called M_iterates, which contains the values of the fit coefficients at each iteration. The matrix has a row for each iteration and a column for each fit coefficient. The last column contains the value of chi square for each iteration.

Bit 4: Suppress Screen Updates

Works just like setting the /N=1 flag. See **CurveFit** for details.

Added in Igor Pro 7.00.

Bit 5: Errors Only

When set, just like setting /O flag (Guess only) but for FuncFit also computes the W_sigma wave and optionally the covariance matrix (/M flag) for your set of coefficients. There is the possibility that setting this bit can generate a singular matrix error.

Added in Igor Pro 7.00.

V_chisq

V_chisq is a measure of the goodness of fit. It has absolute meaning only if you've specified a weighting wave. See the discussion in the section **Weighting** on page III-199.

V_q

V_q (straight-line fit only) is a measure of the believability of chi-square. It is valid only if you specified a weighting wave. It represents the quantity q which is computed as follows:

```
q = gammq((N-2)/2, chisq/2)
```

where gammq is the incomplete gamma function 1-P(a,x) and N is number of points. A q of 0.1 or higher indicates that the goodness of fit is believable. A q of 0.001 indicates that the goodness of fit may be believable. A q of less than 0.001 indicates systematic errors in your data or that you are fitting to the wrong function.

V_FitError and V_FitQuitReason

When an error occurs during a curve fit, it normally causes any running user-defined procedure to abort.

This makes it impossible for you to write a procedure that attempts to recover from errors. However, you can prevent an abort in the case of certain types of errors that arise from unpredictable mathematical circumstances. Do this creating a variable named V_FitError and setting it to zero before performing a fit. If an error occurs during the fit, it will set bit 0 of V_FitError. Certain errors will also cause other bits to be set in V_FitError:

Error	Bit Set
Any error	0
Singular matrix	1
Out of memory	2
Function returned NaN or INF	3
Fit function requested stop	4
Reentrant curve fitting	5

Reentrant curve fitting means that somehow a second curve fit started execution when there was already one running. That could happen if your user-defined fit function tried to do a curve fit, or if a button action procedure that does a fit responded too soon to another click.

There is more than one reason for a fit to stop iterating without an error. To obtain more information about the reason that a nonlinear fit stopped iterating, create a variable named V_FitQuitReason. After the fit, V_FitQuitReason is zero if the fit terminated normally, 1 if the iteration limit was reached, 2 if the user stopped the fit, or 3 if the limit of passes without decreasing chi-square was reached.

Other types of errors, such as missing waves or too few data points for the fit, are likely to be programmer errors. V_FitError does not catch those errors, but you can still prevent an abort if you wish, using the special function **AbortOnRTE** and Igor's **try-catch-endtry** construct. Here is an example function that attempts to do a curve fit to a data set that may contain nothing but NaNs:

```
Function PreventCurveFitAbort()
   Make/O test = NaN
   try
      CurveFit/N/Q line, test; AbortOnRTE
   catch
      if (V_AbortCode == -4)
         Print "Error during curve fit:"
         Variable cfError = GetRTError(1)    // 1 to clear the error
         Print GetErrMessage(cfError,3)
      endif
   endtry
End
```

If you run this function, the output is:

```
  Error during curve fit:
  You must have at least as many data points as fit parameters.
```

No error alert is presented because of the call to **GetRTError**. The error is reported to the user by getting the error message using **GetRTErrMessage** and then printing the message to the history area.

V_FitIterStart

V_FitIterStart provides a way for a user-defined function to know when the fitting routines are about to start a new iteration. The original, obsolete purpose of this is to allow for possible efficient computation of user-defined fit functions that involve convolution. Such functions should now use all-at-once fit functions. See **All-At-Once Fitting Functions** on page III-256 for details.

S_Info

If you create a string variable in a function that calls CurveFit or Funcfit, Igor will fill it with keyword-value pairs giving information about the fit:

Keyword	Information Following Keyword
DATE	The date of the fit.
TIME	The time of day of the fit.
FUNCTION	The name of the fitting function.
AUTODESTWAVE	If you used the /D parameter flag to request an autodestination wave, this keyword gives the name of the wave.
YDATA	The name of the Y data wave.
XDATA	A comma-separated list of X data waves, or "_calculated_" if there were no X waves. In most cases there is just one X wave.

Use **StringByKey** to get the information from the string. You should set keySepStr to "=" and listSepStr to ";".

Errors in Variables: Orthogonal Distance Regression

When you fit a model to data, it is usually assumed that all errors are in the dependent variable, and that independent variables are known perfectly (that is, X is set perfectly and Y is measured with error). This assumption is often not far from true, and as long as the errors in the dependent variable are much larger than those for the independent variable, it will not usually cause much difference to the curve fit.

When the errors are normally distributed with zero mean and constant variance, and the model is exact, then the standard least-squares fit gives the maximum-likelihood solution. This is the technique described earlier (see **Overview of Curve Fitting** on page III-179).

In some cases, however, the errors in both dependent and independent variables may be comparable. This situation has a variety of names including errors in variables, measurement error models or random regressor models. An example of a model that can result in similar errors in dependent and independent variables is fitting the track of an object along a surface; the variables involved would be measurements of cartesian coordinates of the object's location at various instants in time. Presumably the measurement errors would be similar because both involve spatial measurement.

Fitting such data using standard or ordinary least squares can lead to bias in the solution. To solve this problem, we offer Orthogonal Distance Regression (ODR). Rather than minimizing the sum of squared errors in the dependent variable, ODR minimizes the orthogonal distance from the data to the fitted curve by adjusting both the model coefficients and an adjustment to the values of the independent variable. This is also sometimes called "total least squares" or "reduced major axis" (RMA) fitting

For ODR curve fitting, Igor Pro uses a modified version of the freely available ODRPACK95. The CurveFit, FuncFit, and FuncFitMD operations can all do ODR fitting using the /ODR flag (see the documentation for the **CurveFit** operation on page V-124 for details on the /ODR flag, and the **Curve Fitting References** on page III-267 for information about the ODRPACK95 implementation of ODR fitting). Our version of ODRPACK95 has been modified to make it threadsafe and to do some automatic multithreading during single fits.

ODR Fitting is Threadsafe

As of Igor Pro 8.00, ODR fitting is threadsafe. You can perform an ODR fit from a threadsafe user-defined function and you can do multiple ODR fits simultaneously using Igor's preemptive threads.

Weighting Waves for ODR Fitting

Just as with ordinary least-squares fitting, you can provide a weighting wave to indicate the expected magnitude of errors for ODR fitting. But in ODR fitting, there are errors in both the dependent and independent variables, so ODR fits accept weighting waves for both. You use the /XW flag to specify weighting waves for the independent variable.

If you do not supply a weighting wave, it is assumed the errors have a variance of 1.0. This may be acceptable if the errors in the dependent and independent variables truly have similar magnitudes. But if the dependent and independent variables are of very different magnitudes, the chances are good that the errors are also of very different magnitudes, and weighting is essential for a proper fit. Unlike the case for ordinary least squares, where there is only a single weighting wave, ODR fitting depends on both the magnitude of the weights as well as the relative magnitudes of the X and Y weights.

ODR Initial Guesses

An ordinary least squares fit that is linear in the coefficients can be solved directly. No initial guess is required. The built-in line, poly, and poly2D curve fit functions are linear in the coefficients and do not require initial guesses.

An ordinary least-squares fit to a function that is nonlinear in the coefficients is an iterative process that requires a starting point. That means that you must provide an initial guess for the fit coefficients. The accuracy required of your initial guess depends greatly on the function you are fitting and the quality of your data. The built-in fit functions also attempt to calculate a good set of initial guesses, but for user-defined fits you must supply your own initial guesses.

An ODR fit introduces a nonlinearity into the fitting equations that requires iterative fitting and initial guesses even for fit functions that have linear fit coefficients. In the case of line, poly, and poly2D fit functions, ODR fitting uses an ordinary least squares fit to get an initial guess. For nonlinear built-in fit functions, the same initial guesses are used regardless of fitting method.

Because the independent variable is adjusted during the fit, an initial guess is also required for the adjustments to the independent variable. The initial guess is transmitted via one or more waves (one for each independent variable) specified with the /XR flag. The X residual wave is also an output- see the **ODR Fit Results** on page III-238.

In the absence of the /XR flag, initial guesses for the adjustments to the independent variable values are set to zero. This is usually appropriate; in areas where the fitting function is largely vertical, you may need nonzero guesses to fit successfully. One example of such a situation would be the region near a singularity.

Holding Independent Variable Adjustments

In some cases you may have reason to believe that you know some input values of the independent variables are exact (or nearly so) and should not be adjusted. To specify which values should not be adjusted, you supply X hold waves, one for each independent variable, via the /XHLD flag. These waves should be filled with zeroes corresponding to values that should be adjusted, or ones for values that should be held.

This is similar to the /H flag to hold fit coefficients at a set value during fitting. However, in the case of ODR fitting and the independent variable values, holds are specified by a wave instead of a string of ones and zeroes. This was done because of the potential for huge numbers of ones and zeroes being required. To save memory, you can use a byte wave for the holds. In the Make Waves dialog, you select Byte 8 Bit from the Type menu. Use the /B flag with the **Make** operation on page V-526.

ODR Fit Results

An ordinary least-squares fit adjusts the fit coefficients and calculates model values for the dependent variable. You can optionally have the fit calculate the residuals — the differences between the model and the dependent variable data.

ODR fitting adjusts both the fit coefficients and the independent variable values when seeking the least orthogonal distance fit. In addition to the residuals in the dependent variable, it can calculate and return to you a wave or waves containing the residuals in the independent variables, as well as a wave containing the adjusted values of the independent variable.

Residuals in the independent variable are returned via waves specified by the /XR flag. Note that the contents of these waves are inputs for initial guesses at the adjustments to the independent variables, so you must be careful — in most cases you will want to set the waves to zero before fitting.

The adjusted independent variable values are placed into waves you specify via the /XD flag.

Note that if you ask for an auto-destination wave (/D flag; see**The Destination Wave** on page III-196) the result is a wave containing model values at a set of evenly-spaced values of the independent variables. This wave will also be generated in response to the /D flag for ODR fitting.

You can also specify a specific wave to receive the model values (/D=*wave*). The values are calculated at the values of the independent variables that you supply as input to the fit. In the case of ODR fitting, to make a graph of the model, the appropriate X wave would be the output from the /XD flag, not the input X values.

Constraints and ODR Fitting

When fitting with the ordinary least-squares method (/ODR=0) you can provide a text wave containing constraint expressions that will keep the fit coefficients within bounds. These expressions can be used to apply simple bound constraints (keeping the value of a fit coefficient greater than or less than some value) or to apply bounds on linear combinations of the fit coefficients (constrain $a+b>1$, for instance).

When fitting using ODR (/ODR=1 or more) only simple bound constraints are supported.

Error Estimates from ODR Fitting

In a curve fit, the output includes an estimate of the errors in the fit coefficients. These estimates are computed from the linearized quadratic approximation to the chi-square surface at the solution. For a linear fit (line, poly, and poly2D fit functions) done by ordinary least squares, the chi-square surface is actually quadratic and the estimates are exact if the measurement errors are normally distributed with zero mean and constant variance. If the fitting function is nonlinear in the fit coefficients, then the error estimates are an approximation. The quality of the approximation will depend on the nature of the nonlinearity.

In an ODR fit, even when the fitting function is linear in the coefficients, the fitting equations themselves introduce a nonlinearity. Consequently, the error estimates from an ODR fit are always an approximation. See the **Curve Fitting References** on page III-267 for detailed information.

ODR Fitting Examples

A simple example: a line fit with no weighting. If you run these commands, the call to SetRandomSeed will make your “measurement error” (provided by **gnoise** function on page V-323) the same as the example shown here:

```
SetRandomSeed 0.5       // so that the "random" data will always be the same...
Make/N=10 YLineData, YLineXData
YLineXData = p+gnoise(1)      // gnoise simulates error in X values
YLineData = p+gnoise(1)       // gnoise simulates error in Y values
// make a nice graph with errors bars showing standard deviation errors
Display YLineData vs YLineXData
ModifyGraph mode=3,marker=8
ErrorBars YLineData XY,const=1,const=1
```

Now we’re ready to perform a line fit to the data. First, a standard curve fit:

```
CurveFit line, YLineData/X=YLineXData/D
```

This command results in the following history report:

```
fit_YLineData= W_coef[0]+W_coef[1]*x
W_coef={1.3711,0.78289}
V_chisq= 15.413; V_npnts= 10; V_numNaNs= 0; V_numINFs= 0;
V_startRow= 0; V_endRow= 9; V_q= 1; V_Rab= -0.797202; V_Pr= 0.889708;
V_r2= 0.791581;
W_sigma={0.727,0.142}
Coefficient values ± one standard deviation
    a = 1.3711 ± 0.727
    b = 0.78289 ± 0.142
```

Next, we will use /ODR=2 to request orthogonal distance fitting:

```
CurveFit/ODR=2 line, YLineData/X=YLineXData/D
```

which gives this result:

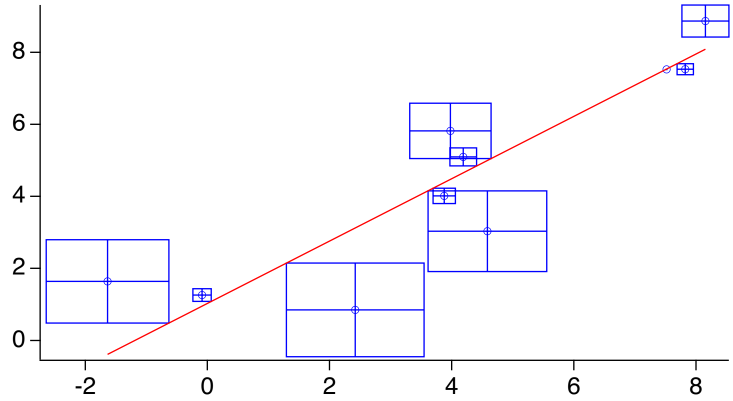
```
Fit converged properly
fit_YLineData= W_coef[0]+W_coef[1]*x
W_coef={1.0311,0.86618}
V_chisq= 9.18468; V_npnts= 10; V_numNaNs= 0; V_numINFs= 0;
V_startRow= 0; V_endRow= 9;
W_sigma={0.753,0.148}
Coefficient values ± one standard deviation
    a   = 1.0311 ± 0.753
    b   = 0.86618 ± 0.148
```

Add output of the X adjustments and Y residuals:

```
Duplicate/O YLineData, YLineDataXRes, YLineDataYRes
CurveFit/ODR=2 line, YLineData/X=YLineXData/D/XR=YLineDataXRes/R=YLineDataYRes
```

And a graph that uses error bars to show the residuals:

```
Display YLineData vs YLineXData
ModifyGraph mode=3,marker=8
AppendToGraph fit_YLineData
ModifyGraph rgb(YLineData)=(0,0,65535)
ErrorBars YLineData BOX,wave=(YLineDataXRes,YLineDataXRes),
wave=(YLineDataYRes,YLineDataYRes)
```



The boxes on this graph do not show error estimates, they show the residuals from the fit. That is, the differences between the data and the fit model. Because this is an ODR fit, there are residuals in both X and Y; error bars are the most convenient way to show this. Note that one corner of each box touches the model line.

In the next example, we do an exponential fit in which the Y values and errors are small compared to the X values and errors. The curve fit history report has been edited to include just the output of the solution. First, fake data and a graph:

```
SetRandomSeed 0.5   // so that the "random" data will always be the same…
Make/D/O/N=20 expYdata, expXdata
expYdata = 1e-6*exp(-p/2)+gnoise(1e-7)
expXdata = p+gnoise(1)
display expYdata vs expXdata
ModifyGraph mode=3,marker=8
```

A regular exponential fit:

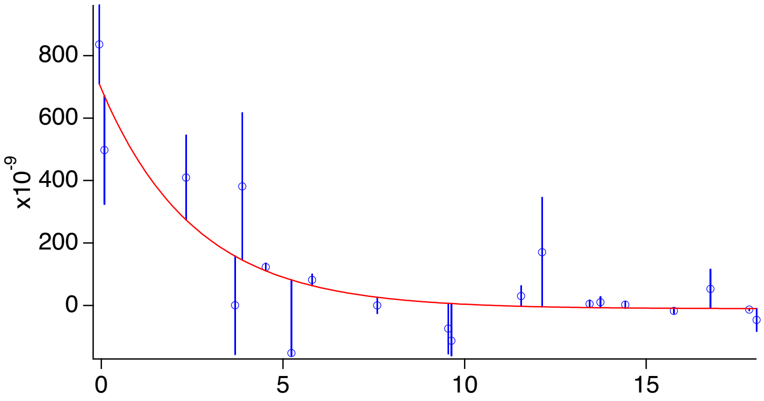
```
CurveFit exp, expYdata/X=expXdata/D

Coefficient values ± one standard deviation
    y0        =-1.0805e-08 ± 4.04e-08
    A         =7.0438e-07 ± 9.37e-08
    invTau    =0.38692 ± 0.116
```

An ODR fit with no weighting, with X and Y residuals:

```
Duplicate/O expYdata, expYdataResY, expYdataResX
expYdataResY=0
expYdataResX=0
CurveFit/ODR=2 exp, expYdata/X=expXdata/D/R=expYdataResY/XR=expYdataResX

Coefficient values ± one standard deviation
    y0     =-1.0541e-08 ± 4.03e-08
    A      =7.0443e-07 ± 9.37e-08
    invTau =0.38832 ± 0.116
```

And a graph:

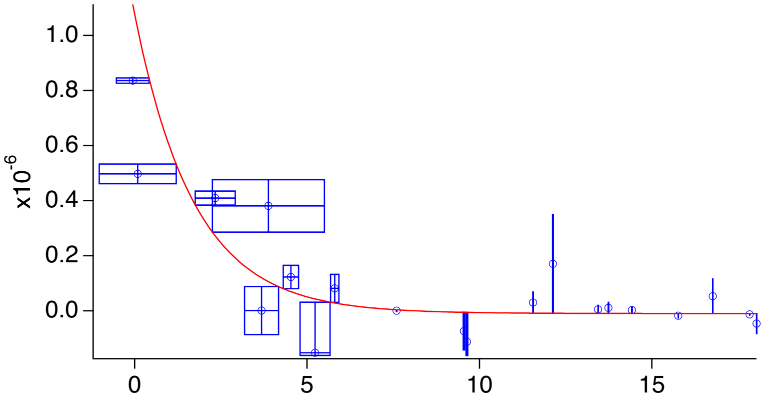
```
Display /W=(137,197,532,405) expYdata vs expXdata
AppendToGraph fit_expYdata
ModifyGraph mode(expYdata)=3
ModifyGraph marker(expYdata)=8
ModifyGraph lSize(expYdata)=2
ModifyGraph rgb(expYdata)=(0,0,65535)
ErrorBars expYdata
BOX,wave=(expYdataResX,expYdataResX),wave=(expYdataResY,expYdataResY)
```



Because the Y values are very small compared to the X values, and we didn't use weighting to reflect smaller errors in Y, the residual boxes are tall and skinny. If the vertical graph scale were the same as the horizontal scale, the boxes would be approximately square. The data line would be very nearly a horizonal line. One way to understand this is to remember that the ODR method is essentially geometric. In this example, the vertical scale on the graph has been expanded very greatly, but the ODR method works in an unexpanded scale where the perpendicular lines to the fit curve are very nearly exactly vertical.

Now we can add appropriate weighting. It's easy to decide on the correct weighting since we added "measurement error" using gnoise():

```
Duplicate/O expYdata, expYdataWY
expYdataWY=1e-7
Duplicate/O expYdata, expYdataWX
expYdataWX=1
// Caution: Next command wrapped to fit on page.
CurveFit/ODR=2 exp, expYdata/X=expXdata/D/R=expYdataResY/XR =expYdataResX/W=expYdataWY
/XW=expYdataWX/I=1

Coefficient values ± one standard deviation
    y0     =-9.8498e-09 ± 3e-08
    A      =1.0859e-06 ± 5.39e-07
    invTau =0.57731 ± 0.248
```



One way to think about the weighting waves for ODR fitting is that they provide geometric scaling. In this example, the vertical dimension is about 10^7 times smaller than the horizontal dimension. When the vertical dimension is scaled by the weighting waves, the dimensions are similar and the perpendicular distances from the fit curve to the data points are no longer merely vertical.

Fitting Implicit Functions

Occasionally you may need to fit data to a model that doesn't have a form that can be expressed as y=f(x). An example would be fitting a circle or ellipse using an equation like

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

Because this equation can't be expressed as y=f(x), you can't write a standard user-defined fitting function for it.

This problem is related to the errors in variables problem in which data has measurement errors in both the dependent and independent variables. You have two inputs, or independent variables (x and y), both with (probably) similar measurement errors. It differs from errors in variables fitting in that the function output is zero instead of being a dependent variable.

The ODRPACK95 package also supports fitting to implicit functions, which you can do with the /ODR=3 flag. You create a fitting function with multiple independent variables (in the case of the ellipse above, two for x and y). The fitting process will attempt to find x and y residuals and fit coefficients to minimize the distance from the data to the zero contour of the function.

It may be a good idea at this point to read the section about errors in variables fitting; see **Errors in Variables: Orthogonal Distance Regression** on page III-236; much of it applies also to implicit fits.

There are a few differences between regular fitting and implicit fitting. For implicit fits, there is no autodestination, and the report printed in the history does not include the wave assignment showing how to get values of the fitted model (doing that is actually not at all easy).

Because of the details of the way ODRPACK95 operates, when you do an implicit fit, the curve fit progress window does not update, and it appears that the fit takes just one iteration. That is because all the action takes place inside the call to ODRPACK95.

The fit function must be written to return zero when the function solution is found. So if you have an equation of the form f(x_i) = 0, you are ready to go; you simply create a fit function that implements f(x_i). If it is in the form f(x_i) = constant, you must move the constant to the other side of the equation: f(x_i)-constant = 0. If it is a form like f(x_i) = g(x_i), you must write your fitting function to return f(x_i) - g(x_i).

Example: Fit to an Ellipse

In this example we will show how to fit an equation like the one above. In the example the center of the ellipse will be allowed to be at nonzero x_0, y_0.

First, we must define a fitting function. The function includes special comments to get mnemonic names for the fit coefficients (see **The New Fit Function Dialog Adds Special Comments** on page III-253). To try the example, you will need to copy this function and paste it into the Procedure window:

```
Function FitEllipse(w,x,y) : FitFunc
	Wave w
	Variable x
	Variable y

	//CurveFitDialog/
	//CurveFitDialog/ Coefficients 4
	//CurveFitDialog/ w[0] = a
	//CurveFitDialog/ w[1] = b
	//CurveFitDialog/ w[2] = x0
	//CurveFitDialog/ w[3] = y0

	return ((x-w[2])/w[0])^2 + ((y-w[3])/w[1])^2 - 1
End
```

An implicit fit seeks adjustments to the input data and fit coefficients that cause the fit function to return zero. To implement the ellipse function above, it is necessary to subtract 1.0 to account for "= 1" on the right in the equation above.

The hard part of creating an example is creating fake data that falls on an ellipse. We will use the standard parametric equations to do the job (y = a*cos(theta), x=b*sin(theta)). So far, we will not add “measurement error” to the data so that you can see the ellipse clearly on a graph:

```
Make/N=20 theta,ellipseY,ellipseX
theta = 2*pi*p/20
ellipseY = 2*cos(theta)+2
ellipseX=3*sin(theta)+1
```

A graph of the data (you could use the Windows→New Graph menu, and then the Modify Trace Appearance dialog):

```
Display ellipseY vs ellipseX
ModifyGraph mode=3,marker=8
ModifyGraph width={perUnit,72,bottom},height={perUnit,72,left}
```

The last command line sets the width and height modes of the graph so that the ellipse is shown in its true aspect ratio. Now add some “measurement error” to the data:

```
SetRandomSeed 0.5       // so that the "random" data will always be the same…
ellipseY += gnoise(.3)
ellipseX += gnoise(.3)
```

Now you can see why we didn’t do that before — it’s a pretty lousy ellipse!

Now, finally, do the fit. That requires making a coefficient wave and filling it with the initial guesses, and making a pair of waves to receive estimated values of X and Y at the fit:

```
Duplicate ellipseY, ellipseYFit, ellipseXFit
Make/D/O ellipseCoefs={3,2,1,2}                  // a, b, x0, y0
FuncFit/ODR=3 FitEllipse, ellipseCoefs /X={ellipseX, ellipseY} /XD={ellipseXFit,ellipseYFit}
```

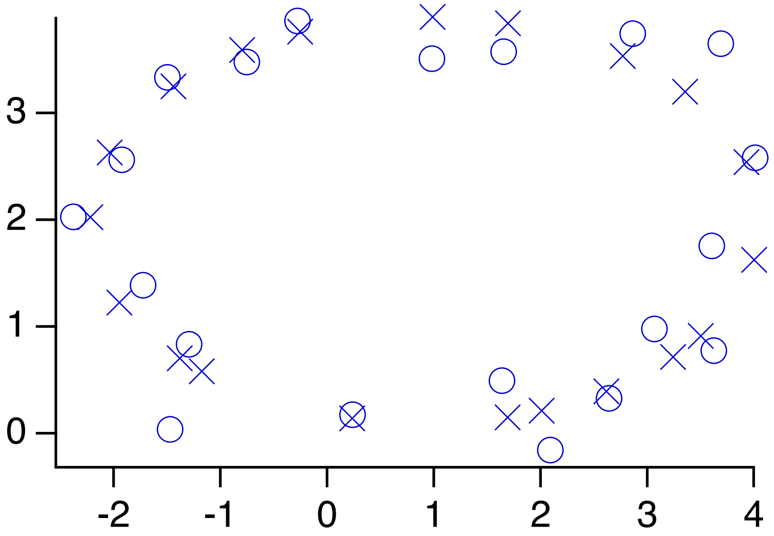
The call to the FuncFit operation has no Y wave specified (it would ordinarily go right after the coefficient wave, ellipseCoefs) because this is an implicit fit.

The results:

```
Fit converged properly
  ellipseCoefs={3.1398,1.9045,0.92088,1.9971}
  V_chisq= 1.74088; V_npnts= 20; V_numNaNs= 0; V_numINFs= 0;
  W_sigma={0.158,0.118,0.128,0.0906}
  Coefficient values ± one standard deviation
   a  =3.1398 ± 0.158
   b  =1.9045 ± 0.118
   x0 =0.92088 ± 0.128
   y0 =1.9971 ± 0.0906
```

And add the destination waves (the ones specified with the /XD flag) to the graph:

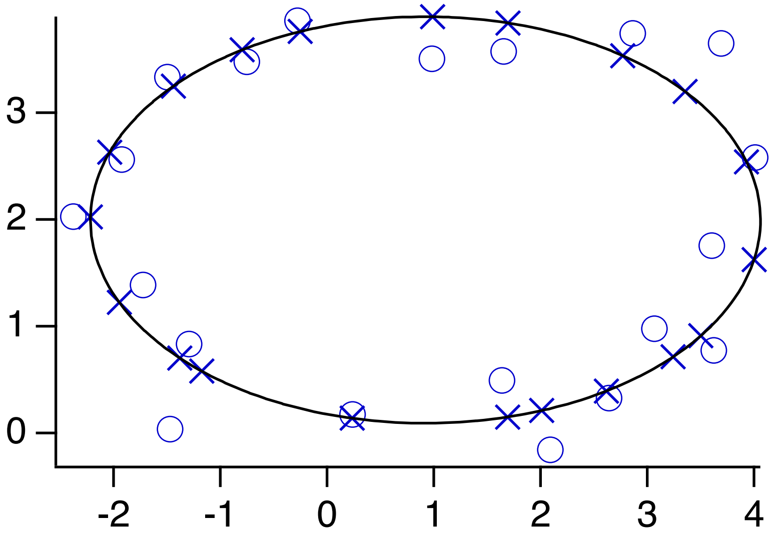
```
AppendToGraph ellipseYFit vs ellipseXFit
ModifyGraph mode=3,marker(ellipseYFit)=1
ModifyGraph rgb=(1,4,52428)
```



It is difficult to compute the model representing the solution, as it requires finding roots of the implicit function. A quick way to add a smooth model curve to a graph is to fill a matrix with values of the fit function at a range of X and Y and then add a contour plot of the matrix to your graph. Then modify the contour plot to show only the zero contour.

Here are commands to add a contour of the example function to the graph above:

```
Make/N=(100,100) ellipseContour
SetScale/I x -3,4.5,ellipseContour
SetScale/I y -.2, 5, ellipseContour
ellipseContour = FitEllipse(ellipseCoefs, x, y)
AppendMatrixContour ellipseContour
ModifyContour ellipseContour labels=0,autoLevels={*,*,0},moreLevels=0,moreLevels={0}
ModifyContour ellipseContour rgbLines=(0,0,0)
```



Fitting Sums of Fit Functions

Sometimes the appropriate model is a combination, typically a sum, of simpler models. It might be a sum of exponential decay functions, or a sum of several different peaks at various locations. Peak fitting can include a baseline function implemented as a separate user-defined fit function. With standard curve fitting, you have to write a user-defined function that implements the sum.

Instead, you can use an alternate syntax for the FuncFit operation to fit to a list of fit functions that are summed automatically. The results for each fit function are returned via separate coefficient waves, making it easy to keep the results of each term in the sum separate. The syntax is described in the **FuncFit** operation on page V-273.

The sum-of-fit-functions feature can mix any kind of fit function — built-in, or any of the variants described in **User-Defined Fitting Functions** on page III-250. However, even if you use only built-in fit functions you must provide initial guesses because our initial-guess generating code cannot discern how the various features of the data are partitioned amongst the list of fit functions.

Linear Dependency: A Major Issue

When you fit a list of fit functions you must be careful not to introduce linear dependencies between the coefficients of the list of functions. The most likely such dependency will arise from a constant Y offset used with all of the built-in fit functions, and commonly included in user functions. This Y offset accounts for any offset error that is common when making real-world measurements. For instance, the equation for the built-in exp function is:

$$y_0 + A\exp(-Bx)$$

If you sum two terms, you get two y_0's (primes used for the second copy of the function):

$$y_0 + A\exp(-Bx) + y_0' + A'\exp(-B'x)$$

As long as B and B' are distinct, the two exponential terms will not be a problem. But y_0 and y'_0 are linearly dependent — they cannot be distinguished mathematically. If you increase one and decrease the other the same amount, the result is exactly the same. This is sure to cause a singular matrix error when you try to do the fit.

The solution is to hold one of the y_0 coefficients. Because each fit function in the list has its own coefficient wave, and you can specify a hold string for each function, this is easy (see **Example: Summed Exponentials** on page III-246).

There are many ways to introduce linear dependence. They are not always so easy to spot!

Constraints Applied to Sums of Fit Functions

The sums of fit functions feature does not include a keyword for specifying constraints on a function-by-function basis. But you can use the normal constraint specifications - you just have to translate the constraint expressions to take account of the list of functions. (If you are interested in using constraints, but don't know about the syntax for constraints, see **Fitting with Constraints** on page III-227.)

Constraint expressions use K*n* to designate fit coefficients where *n* is simply the sequential number of the fit coefficient within the list of coefficients. n starts with zero. In order to reference fit coefficients for a fit function in a list of summed fit functions, you must account for all the fit coefficients in all the fit functions in the list before the fit function you are trying to constrain. For example, this is part of a command that will fit a sum of three exponential terms:

```
FuncFit {{exp,expTerm1},{exp,expTerm2,hold="1"},{exp,expTerm3,hold="1"}} ...
```

In order to constrain coefficients of the second exponential term, you must know first that the built-in exp fit function has three fit coefficients. Here is the equation of the exp fit function:

```
f(x) = y0 + A*exp(-x*invTau)
```

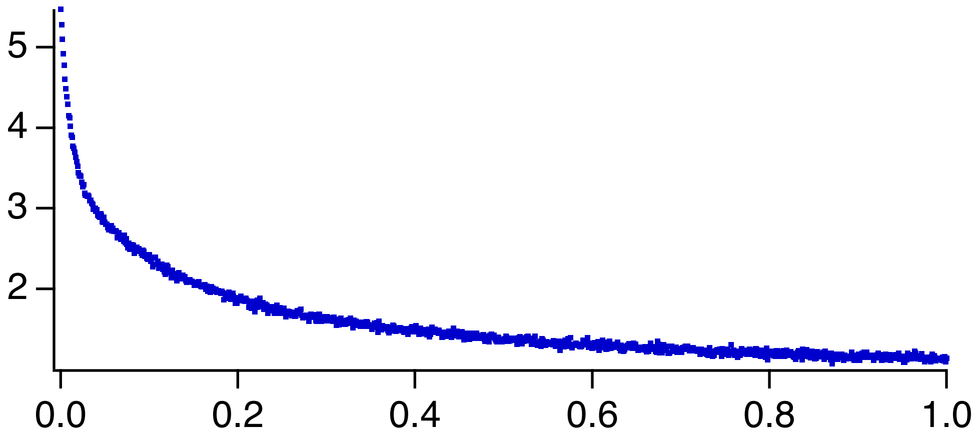
The vertical offset (y0) of the first summed exp term is represented in a constraint expression as "K0", the amplitude as "K1", and invTau as "K2". Constraint expressions for the second exp term continue the count starting with "K3" for y0 (but see the section **Linear Dependency: A Major Issue** on page III-245 and note that you are not allowed to hold and constrain a fit coefficient simultaneously), and use "K4" for the amplitude and "K5" for invTau.

Example: Summed Exponentials

This example fits a sum of three exponentials using the built-in exp fit function. For real work, we recommend the exp_XOffset function; it handles data that's not at X=0 better, and the decay constant fit coefficient is actually the decay constant. The exp fit function gives you the inverse of the decay constant.

First, make some fake data and graph it:

```
Make/D/N=1000 expSumData
SetScale/I x 0,1,expSumData
expSumData = 1 + exp(-x/0.5) + 1.5*exp(-x/.1) + 2*exp(-x/.01)+gnoise(.03)
Display expSumData
ModifyGraph mode=2,rgb=(1,4,52428)
```



The fake data was purposely made with a Y offset of 1.0 in order to illustrate how to handle the vertical offset terms in the fit function.

Next, we need to make a coefficient wave for each of the fit functions. In spite of the linear dependence between the vertical offset in each copy of the function, you must include all the fit coefficients in the coefficient waves. Otherwise when the function is evaluated the fit won't have all the needed information.

```
Make/D/O expTerm1 = {1, 1, 2}
Make/D/O expTerm2 = {0, 1.5, 10}
Make/D/O expTerm3 = {0, 2, 100}
```

Each of these lines makes one coefficient wave with three elements. The first element is y_0, the second is amplitude, and the third is the inverse of the decay constant. Since the data is fake, we have a pretty good idea of the initial guesses!

To reflect the baseline offset of 1.0, the y_0 coefficient for *only* the first exponential coefficient wave was set to 1. If y_0 were set to 1.0 for all three, the offset would be 3.0.

Now we can do the fit. A FuncFit command with a list of fit functions and coefficient waves can be pretty long:

```
FuncFit {{exp, expTerm1},{exp, expTerm2, hold="1"},{exp, expTerm3, hold="1"}}
expSumData/D
```

The entire list of functions is enclosed in braces and each fit function specification in the list is enclosed in braces as well. At a minimum you must provide a fit function and a coefficient wave for each function in the list, as was done here for the first exponential term.

The specification for each function can also contain various keywords; the second and third terms here contain the *hold* keyword in order to include a hold string. The string used will cause the fit to hold the y_0 coefficient for the second and third terms at zero. That prevents problems caused by linear dependence between the three fit functions. Since the coefficient waves for the second and third terms set their respective y_0 to zero, this puts all the vertical offset into the one coefficient for the first term.

In this example the hold strings are literal quoted strings. They can be any string expression but see below for restrictions if you use a function list contained in a string:

```
String holdStr = "1"
FuncFit {{exp,expTerm1},{exp,expTerm2,hold=holdStr},
            {exp,expTerm3,hold="1"}} expSumData/D     // All on one line
```

The history report for a sum of fit functions is enhanced to show all the functions:

```
Fit converged properly
fit_expSumData= Sum of Functions(,x)
expTerm1={1.027,1.045,2.2288}
expTerm2={0,1.4446,10.416}
expTerm3={0,2.0156,102.28}
V_chisq= 0.864844; V_npnts= 1000; V_numNaNs= 0; V_numINFs= 0;
V_startRow= 0; V_endRow= 999; V_startCol= 0; V_endCol= 0;
V_startLayer= 0; V_endLayer= 0; V_startChunk= 0; V_endChunk= 0;
W_sigma={0.0184,0.0484,0.185,0,0.052,0.495,0,0.0239,2.34}
For function 1: exp:
Coefficient values ± one standard deviation
   y0       = 1.027 ± 0.0184
   A        = 1.045 ± 0.0484
   invTau   = 2.2288 ± 0.185
For function 2: exp:
Coefficient values ± one standard deviation
   y0       = 0 ± 0
   A        = 1.4446 ± 0.052
   invTau   = 10.416 ± 0.495
For function 3: exp:
Coefficient values ± one standard deviation
   y0       = 0 ± 0
   A        = 2.0156 ± 0.0239
   invTau   = 102.28 ± 2.34
```

Example: Function List in a String

The list of functions in the FuncFit command in the first example is moderately long, but it could be much longer. If you wanted to sum 20 peak functions and a baseline function, the list could easily exceed the limit of 2500 bytes in a command line.

Fortunately, you can build the function specification in a string variable and use the string keyword. Doing this on the command line for the first example looks like this:

```
String myFunctions="{exp, expTerm1}"
myFunctions+="{exp, expTerm2, hold=\"1\"}"
myFunctions+="{exp, expTerm3, hold=\"1\"}"
FuncFit {string = myFunctions} expSumData/D
```

These commands build the list of functions one function specification at a time. The second and third lines use the += assignment operator to add additional functions to the list. Each function specification includes its pair of braces.

Notice the treatment of the hold strings — in order to include quotation marks in a quoted string expression, you must escape the quotation marks. Otherwise the command line parser thinks that the first quote around the hold string is the closing quote.

The function list string is parsed at run-time outside the context in which FuncFit is running. Consequently, you cannot reference local variables in a user-defined function. The hold string may be either a quoted literal string, as shown here, or it can be a reference to a global variable, including the full data folder path:

```
String/G root:myDataFolder:holdStr="1"
String myFunctions="{exp, expTerm1}"
myFunctions+="{exp, expTerm2, hold=root:myDataFolder:holdStr}"
myFunctions+="{exp, expTerm3, hold=root:myDataFolder:holdStr}"
FuncFit {string = myFunctions} expSumData/D
```

Fitting with Complex-Valued Functions

Prior to Igor Pro 9.00, to fit complex-valued functions to complex-valued data required writing a real-valued fit function that used a special organization of the data to pack real and imaginary parts into a single real-valued wave. Now Igor supports fitting to complex-valued fitting functions directly.

The basic requirement for fitting complex-valued functions is to write a fit function that returns a complex result. Igor supports the traditional format for a user-defined fit function (see **Format of a Basic Fitting Function** on page III-251) and all-at-once format (**All-At-Once Fitting Functions** on page III-256), but the return type of a basic fit function or the Y wave in an all-at-once fit function must be complex. Complex fitting functions are not supported by structure fit functions or by the sum-of-fit-functions format.

No Dialog Support for Complex Fitting

The Curve Fitting dialog does not allow you to select complex waves, or fit functions that return complex values. It may be possible to use a real-valued function and waves to set up a fit in the dialog, then click the To Cmd Line button to copy the generated command to the command line where you can edit the command. It's probably easier to read the reference documentation for the **FitFunc** operation and simply compose a command.

Complex Basic Fitting Function

The basic format for a complex-valued function looks like this:

```
Function/C F(Wave w, Variable xx) : FitFunc
   <body of function>
   <return statement>
End
```

The /C in "Function/C" tells Igor that the function returns a complex value. As shown above, the function takes a real-valued coefficient wave and a real-valued independent variable. If your particular function needs the coefficient wave and/or independent variable to be complex-valued, add /C to the parameter declaration:

```
Function/C F(Wave/C w, Variable/C xx) : FitFunc
   <body of function>
   <return statement>
End
```

We show here both the coefficient wave and the independent variable being complex; either one or both can be complex.

If you specify a complex independent variable, then you must supply a complex-valued X wave. Since Igor's wave scaling cannot be complex, if your fitting function requires a complex independent variable then you must have a separate X wave as input to the **FitFunc** operation.

Complex Basic Multivariate Fitting Function

As with normal real-valued fitting functions, you can write multivariate fitting functions that return complex values:

```
Function/C F(Wave w, Variable x1, Variable x2) : FitFunc
   <body of function>
   <return statement>
End
```

As before, the coefficient wave can be either real or complex, and the independent variables can be real or complex. The independent variables must be either all real-valued or all complex-valued.

Complex All-At-Once Fitting Function

In certain cases, a fit function may need to produce all model values at one time. You can write an all-at-once fitting functions that implement complex-valued functions:

```
Function AAOFitFunc(Wave pw, Wave/C yw, Wave xw) : FitFunc
   yw = <expression involving pw and xw>
End
```

Unlike the basic format, the function does not return a complex value and is not declared with Function/C. Instead, the all-at-once fit function returns the complex model values to Igor via the yw wave which you must declare as complex using /C.

As with the basic format, you can make the coefficient wave pw complex or real, and you can make the independent variable wave xw complex or real, as long as the type matches the waves you pass to the FuncFit operation.

All-at-once fitting functions can be multivariate functions, too, by adding more x input waves. As with the basic format, independent variable inputs must be all either real or complex.

Other Waves

A variety of other waves may be used during a fit: weighting, masking, epsilon, and residuals waves, and a destination wave to receive the model values of the fit solution. All of these waves must be complex or real depending on the nature of the fitting function.

Weighting applied to the dependent variable values must always be complex because a complex fitting function by definition returns complex dependent variable values. Residual and destination waves must be complex for the same reason.

If you use an epsilon wave, it must match your coefficient wave.

Mask waves must be real; they select data points, and cannot select just the real or imaginary part of your input data.

If you are fitting with errors in both dependent and independent variables, the X weighting waves must match the type of your independent variable inputs.

Curve Fitting with Multiple Processors

If you are using a computer with multiple processors, you no doubt want to take advantage of them. Curve fitting can take advantage of multiple processors in two ways:

- Automatic multithreading
- Programmed multithreading

Curve Fitting with Automatic Multithreading

If you use the built-in fit functions or thread-safe user-defined fit functions in the basic or all-at-once formats, Igor automatically uses multiple processors if your data set is “large enough”. The number of points required to be large enough is set by the **MultiThreadingControl** operation.

There are two keywords used by MultiThreadingControl for curve fitting. The `CurveFit1` keyword controls automatic multithreading with built-in and basic format user-defined fit functions. The `CurveFitAllAtOnce` keyword controls automatic multithreading with user-defined fit functions in the all-at-once format. The appropriate setting of these limits depends on the complexity of the fitting function and the nature of your problem. It can be best determined only by experimentation.

The automatic multithreading added in Igor7 makes the CurveFit /NTHR flag obsolete. The automatic multithreading is more effective than the threading available in Igor6.

Automatic multithreading when doing ODR fits (see **Errors in Variables: Orthogonal Distance Regression** on page III-236) is most effective with fit functions having large numbers of fit coefficients and with user-defined fit functions that are expensive to compute. Surprisingly, it may be more effective with fewer input data points.

If you use the ThreadSafe keyword with your user-defined fit function and Igor's automatic multithreading is enabled for your fit, your function must not access waves or global variables. Therefore your function must not use WAVE, NVAR, or SVAR declarations.

Curve Fitting with Programmed Multithreading

The CurveFit, FuncFit, and FuncFitMD operations are threadsafe. Consequently another way to use multiple processors is to do more than one curve fit simultaneously, with each fit using a different processor. This requires threadsafe programming, as described under **ThreadSafe Functions and Multitasking** on page IV-329, and requires at least intermediate-level Igor programming skills.

The fitting function can be either a built-in fit function or a threadsafe user-defined fit function.

For an example, choose File→Example Experiments→Curve Fitting→MultipleFitsInThreads.

Constraints and ThreadSafe Functions

The usual way to specify constraints to a curve fit is via expressions in a text wave (see **Fitting with Constraints** on page III-227). As part of the process of parsing these expressions and getting ready to use them in a curve fit involves evaluating part of the expressions. That, in turn, requires sending them to Igor's command-line interpreter, in a process very similar to the way the **Execute** operation works. Unfortunately, this is not threadsafe.

Instead, you must use the method described in **Constraint Matrix and Vector** on page III-230. Unfortunately, it is hard to understand, inconvenient to set up, and easy to make mistakes. The best way to do it is to set up your constraint expressions using the normal text wave method (see **Constraint Expressions** on page III-228) and use the /C flag with a trial fit. Igor will generate the matrix and vector required.

In most cases, the basic expressions will not change from one fit to another, just the limits of the constraints will change. If that is the case, you can use the matrix provided by Igor, and alter the numbers in the vector to change the constraint limits.

User-Defined Fitting Functions

When you use the New Fit Function dialog to create a user-defined function, the dialog uses the information you enter to create code for your function in the Procedure window. Using the New Fit Function dialog is the easiest way to create a user-defined fitting function, but it is possible also to write the function directly in the Procedure window.

Certain kinds of complexities will *require* that you write the function yourself. It may be that the easiest way to create such a function is to create a skeleton of the function using the dialog, and then modify it by editing in the procedure window.

This section describes the format of user-defined fitting functions so that you can understand the output of the New Fit Function dialog, and so that you can write one yourself.

You can use a variety of formats for user-defined fit functions tailored to different situations, and involving varying degrees of complexity to write and use. The following section describes the simplest format, which is also the format created by the New Fit Function dialog.

User-Defined Fitting Function Formats

You can use three formats for user-defined fitting functions: the Basic format discussed above, the All-At-Once format, and Structure Fit Functions, which use a structure as the only input parameter. Additionally, Structure Fit Functions come in basic and all-at-once variants. Each of these formats address particular situations.

The basic format (see **Format of a Basic Fitting Function** on page III-251) was the original format. It returns just one model value at a time.

The all-at-once format (see **All-At-Once Fitting Functions** on page III-256) addresses problems in which the operations involved, such as convolution, integration, or FFT, naturally calculate all the model values at once. Because of reduced function-call overhead, it is somewhat faster than the basic format for large data sets.

The structure-based format (see **Structure Fit Functions** on page III-261) uses a structure as the only function parameter, allowing arbitrary information to be transmitted to the function during fitting. This makes it very flexible, but also makes it necessary that FuncFit be called from a user-defined function.

Basic Fit Function	All-At-Once Function	Structure Function
Can be selected, created, and edited within the Curve Fitting dialog.	Can be selected, but *not* created or edited, within the Curve Fitting dialog.	Cannot be used from the Curve Fitting dialog.
With appropriate comments, mnemonic coefficient names.	No mnemonic coefficient names.	Must be used with FuncFit called from a user-defined function.
Straight-forward programming: one X value, one return value.	Programming requires a good understanding of wave assignment; there are some issues that can be difficult to avoid.	Hardest to program: requires both an understanding of structures and writing a driver function that calls FuncFit.
Not an efficient way to write a fit function that uses convolution, integration, FFT, or any operation that uses all the data values in a single operation.	Most efficient for problems involving operations like convolution, integration, or FFT. Often much faster than the Basic format, even for problems that don't require it.	Very flexible: any arbitrary information can be transmitted to the fit function. More information about the fit progress transmitted via the structure.
See **Format of a Basic Fitting Function** on page III-251.	See **All-At-Once Fitting Functions** on page III-256.	See **Structure Fit Functions** on page III-261.

Format of a Basic Fitting Function

A basic user-defined fitting function has the following form:

```
Function F(w, x) : FitFunc
   WAVE w; Variable x

   <body of function>
   <return statement>
End
```

You can choose a more descriptive name for your function.

The function must have exactly two parameters in the univariate case shown above. The first parameter is the coefficients wave, conventionally called w. The second parameter is the independent variable, conventionally called x. If your function has this form, it will be recognized as a curve fitting function and will allow you to use it with the FuncFit operation.

The FitFunc keyword marks the function as being intended for curve fitting. Functions with the FitFunc keyword that have the correct format are included in the Function menu in the Curve Fitting dialog.

The FitFunc keyword is not required. The FuncFit operation will allow any function that has a wave and a variable as the parameters. In the Curve Fitting dialog you can choose Show Old-Style Functions from the Function menu to display a function that lacks the FitFunc keyword, but you may also see functions that just happen to match the correct format but aren't fitting functions.

Note that the function does not know anything about curve fitting. All it knows is how to compute a return value from its input parameters. The function is called during a curve fit when it needs the value for a certain X and a certain set of fit coefficients.

Here is an example of a user-defined function to fit a log function. This might be useful since log is not one of the functions provided as a built-in fit function.

```
Function LogFit(w,x) : FitFunc
   WAVE w
   Variable x

   return w[0]+w[1]*log(x)
End
```

In this example, two fit coefficients are used. Note that the first is in the zero element of the coefficient wave. You cannot leave out an index of the coefficient wave. Igor uses the size of your coefficient wave to determine how many coefficients need to be fit. Consequently an unused element of the wave results in a singular matrix error.

Intermediate Results for Very Long Expressions

The body of the function is usually fairly simple but can be arbitrarily complex. If necessary, you can use local variables to build up the result, piece-by-piece. You can also call other functions or operations and use loops and conditionals.

If your function has many terms, you might find it convenient to use a local variable to store intermediate results rather than trying to put the entire function on one line. For example, instead of:

```
return w[0] + w[1]*x + w[2]*x^2
```

you could write:

```
Variable val         // local variable to accumulate result value
val = w[0]
val += w[1]*x
val += w[2]*x^2
return val
```

We often get fitting functions from our customers that use extremely long expressions derived by Mathematica. These expressions are hard to read and understand, and highly resistant to debugging and modification. It is well worth some effort to break them down using assignments to intermediate results.

Conditionals

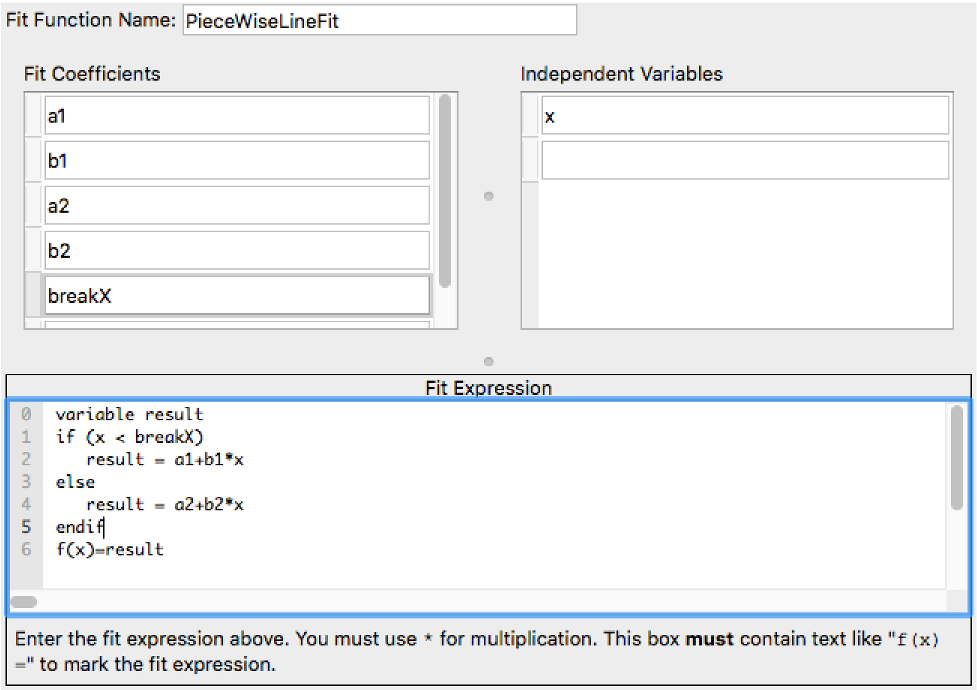
Flow control statements including `if` statements are allowed in fit functions. You could use this to fit a piece-wise function, or to control the return value in the case of a singularity in the function.

Here is an example of a function that fits two lines to different sections of the data. It uses one of the parameters to decide where to switch from one line to the other:

```
Function PieceWiseLineFit(w,x) : FitFunc
   WAVE w
   Variable x

   Variable result
   if (x < w[4])
      result = w[0]+w[1]*x
   else
      result = w[2]+w[3]*x
```

```
    endif
    return result
End
```

This function can be entered into the New Fit Function dialog. Here is what the dialog looked like when we created the function above:



Fit Function Name: PieceWiseLineFit

Fit Coefficients

- a1
- b1
- a2
- b2
- breakX

Independent Variables

- x

Fit Expression

```
variable result
if (x < breakX)
    result = a1+b1*x
else
    result = a2+b2*x
endif
f(x)=result
```

Enter the fit expression above. You must use * for multiplication. This box **must** contain text like "f(x)=" to mark the fit expression.

The New Fit Function Dialog Adds Special Comments

In the example above of a piece-wise linear fit function, the New Fit Function dialog uses coefficient names instead of indexing a coefficient wave, but there isn't any way to name coefficients in a fit function. The New Fit Function dialog adds special comments to a fit function that contain extra information. For instance, the PieceWiseLineFit function as created by the dialog looks like this:

```
Function PieceWiseLineFit(w,x) : FitFunc
    WAVE w
    Variable x

    //CurveFitDialog/ These comments were created by the Curve Fitting dialog. Alteri
    //CurveFitDialog/ make the function less convenient to work with in the Curve Fit
    //CurveFitDialog/ Equation:
    //CurveFitDialog/ variable result
    //CurveFitDialog/ if (x < breakX)
    //CurveFitDialog/  result = a1+b1*x
    //CurveFitDialog/ else
    //CurveFitDialog/  result = a2+b2*x
    //CurveFitDialog/ endif
    //CurveFitDialog/ f(x) = result
    //CurveFitDialog/ End of Equation
    //CurveFitDialog/ Independent Variables 1
    //CurveFitDialog/ x
    //CurveFitDialog/ Coefficients 5
    //CurveFitDialog/ w[0] = a1
    //CurveFitDialog/ w[1] = b1
    //CurveFitDialog/ w[2] = a2
    //CurveFitDialog/ w[3] = b2
    //CurveFitDialog/ w[4] = breakX
```

Special comments give the Curve Fitting dialog extra information about the fit function.

The function code as it appears in the text window of the New Fit Function dialog.

Independent variable name (or names, for a multivariate function).

Coefficient names.

This prefix in the comment identifies the comment as belonging to the curve fitting dialog.

```
   variable result
   if (x < w[4])
    result = w[0]+w[1]*x
   else
    result = w[2]+w[3]*x
   endif
   return result
End
```

The actual function code.

If you click the Edit Fit Function button, the function code is analyzed to determine the number of coefficients. If the comments that name the coefficients are present, the dialog uses those names. If they are not present, the coefficient wave name is used with the index number appended to it as the coefficient name.

Having mnemonic names for the fit coefficients is very helpful when you look at the curve fit report in the history area. The minimum set of comments required to have names appear in the dialog and in the history is a lead-in comment line, plus the Coefficients comment lines. For instance, the following version of the function above will allow the Curve Fitting dialog and history report to use coefficient names:

```
Function PieceWiseLineFit(w,x) : FitFunc
   WAVE w
   Variable x

   //CurveFitDialog/
   //CurveFitDialog/ Coefficients 5
   //CurveFitDialog/ w[0] = a1
   //CurveFitDialog/ w[1] = b1
   //CurveFitDialog/ w[2] = a2
   //CurveFitDialog/ w[3] = b2
   //CurveFitDialog/ w[4] = breakX

   Variable result
   if (x < w[4])
    result = w[0]+w[1]*x
   else
    result = w[2]+w[3]*x
   endif
   return result
End
```

The blank comment before the line with the number of coefficients on it is required- the parser that looks at these comments needs one lead-in line to throw away. That line can contain anything as long as it includes the lead-in "`//CurveFitDialog/`".

Functions that the Fit Function Dialog Doesn't Handle Well

In the example functions it is quite clear by looking at the function code how many fit coefficients a function requires, because the coefficient wave is indexed with a literal number. The number of coefficients is simply one more than the largest index used in the function.

Occasionally a fit function uses constructions other than a literal number for indexing the coefficient wave. This will make it impossible for the Curve Fitting dialog to figure out how many coefficients are required. In this case, the Coefficients tab can't be constructed until you specify how many coefficients are needed. You do this by choosing a coefficient wave having the right number of points from the Coefficient Wave menu.

You cannot edit such a function by clicking the Edit Fit Function button. You must write and edit the function in the Procedure window.

Here is an example function that can fit an arbitrary number of Gaussian peaks. It uses the length of the coefficient wave to determine how many peaks are to be fit. Consequently, it uses a variable (`cfi`) rather than a literal number to access the coefficients:

```
Function FitManyGaussian(w, x) : FitFunc
   WAVE w
   Variable x

   Variable returnValue = w[0]
```

The first coefficient is a baseline offset.

```
   Variable i
   Variable numPeaks = floor((numpnts(w)-1)/3)
   Variable cfi

   for (i = 0; i < numPeaks; i += 1)
      cfi = 3*i+1
      returnValue += w[cfi]*exp(-((x-w[cfi+1])/w[cfi+2])^2)
   endfor
   return returnValue
End
```

Each peak takes three coefficients: amplitude, x position and width.

Calculate index of amplitude for this peak.

Expression of a single Gaussian peak.

Loop over the peaks, calculating them one at a time.

Each peak is added to the result.

Format of a Multivariate Fitting Function

A multivariate fitting function has the same form as a univariate function, but has more than one independent variable:

```
Function F(w, x1, x2, ...) : FitFunc
   WAVE w;
   Variable x1
   Variable x2
   Variable ...

   <body of function>
   <return statement>
End
```

A function to fit a planar trend to a data set could look like this:

```
Function Plane(w, x1, x2) : FitFunc
   WAVE w
   Variable x1, x2

   return w[0] + w[1]*x1 + w[2]*x2
End
```

There is no limit on the number of independent variables, with the exception that the entire Function declaration line must fit within a single command line of 2500 bytes.

The New Fit Function dialog will add the same comments to a multivariate fit function as it does to a basic fit function. The `Plane()` function above might look like this (we have truncated the first two special comment lines to make them fit):

```
Function Plane(w,x1,x2) : FitFunc
   WAVE w
   Variable x1
   Variable x2

   //CurveFitDialog/ These comments were created by the Curve...
   //CurveFitDialog/ make the function less convenient to work...
   //CurveFitDialog/ Equation:
   //CurveFitDialog/ f(x1,x2) = A + B*x1 + C*x2
   //CurveFitDialog/ End of Equation
   //CurveFitDialog/ Independent Variables 2
   //CurveFitDialog/ x1
   //CurveFitDialog/ x2
   //CurveFitDialog/ Coefficients 3
   //CurveFitDialog/ w[0] = A
   //CurveFitDialog/ w[1] = B
   //CurveFitDialog/ w[2] = C

   return w[0] + w[1]*x1 + w[2]*x2
End
```

All-At-Once Fitting Functions

The scheme of calculating one Y value at a time doesn't work well for some fitting functions. This is true of functions that involve a convolution such as might arise if you are trying to fit a theoretical signal convolved with an instrument response. Fitting to a solution to a differential equation might be another example.

For this case, you can create an "all at once" fit function. Such a function provides you with an X and Y wave. The X wave is input to the function; it contains all the X values for which your function must calculate Y values. The Y wave is for output — you put all your Y values into the Y wave.

Because an all-at-once function is called only once for a given set of fit coefficients, it will be called many fewer times than a basic fit function. Because of the saving in function-call overhead, all-at-once functions can be faster even for problems that don't require an all-at-once function.

Here is the format of an all-at-once fit function:

```
Function myFitFunc(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   yw = <expression involving pw and xw>
End
```

Note that there is no return statement because the function result is put into the wave yw. Even if you include a return statement the return value is ignored during a curve fit.

The X wave contains all the X values for your fit, whether you provided an X wave to the curve fit or not. If you did not provide an X wave, xw simply contains equally-spaced values derived from your Y wave's X scaling.

There are some restrictions on all-at-once fitting functions:

1) You can't create or edit an all-at-once function using the Curve Fitting dialog. You must create it by editing in the Procedure window. All-at-once functions are, however, listed in the Function menu in the Curve Fitting dialog.
2) There is no such thing as an "old-style" all-at-once function. It must have the FitFunc keyword.
3) You don't get mnemonic coefficient names.

Here is an example that fits an exponential decay using an all-at-once function. This example is silly — there is no reason to make this an all-at-once function. It is simply an example showing how a real function works without the computational complexities. Here it is, as an all-at-once function:

```
Function allatonce(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   // a wave assignment does the work
   yw = pw[0] + pw[1]*exp(-xw/pw[2])
End
```

This is the same function written as a standard user fitting function:

```
Function notallatonce(pw, x) : FitFunc
   WAVE pw
   Variable x

   return pw[0] + pw[1]*exp(-x/pw[2])
End
```

In the all-at-once function, the argument of exp() includes xw, a wave. The basic format uses the input parameter x, which is a variable containing a single value. The use of xw in the all-at-once version of the function uses the implied point number feature of wave assignments (see **Waveform Arithmetic and Assignments** on page II-74).

There are some things to watch out for when creating an all-at-once fitting function:

1. You must not change the yw wave except for assigning values to it. You must not resize it. This will get you into trouble:

```
   Function allatonce(pw, yw, xw) : FitFunc
      WAVE pw, yw, xw

      Redimension/N=2000 yw                  // BAD!
      yw = pw[0] + pw[1]*exp(-xw/pw[2])
   End
```

2. You may not get the same number of points in yw and xw as you have in the waves you provide as the input data. If you fit to a restricted range, if there are NaNs in the input data, or if you use a mask wave to select a subset of the input data, you will get a wave with a reduced number of points. Your fitting function must be written to handle that situation or you must not use those features.
3. It is tricky, but not impossible, to write an all-at-once fitting function that works correctly with the auto-destination feature (that is, _auto_ in the Destination menu, or /D by itself in a FuncFit command).
4. The xw and yw waves are *not* your data waves. They are created by Igor during the execution of CurveFit, FuncFit or FuncFitMD and filled with data from your input waves. They will be destroyed when fitting is finished. For debugging, you can use Duplicate to save a copy of the waves. Altering the contents of the xw wave will have no effect.

The example above uses xw as an argument to the exp function, and it uses the special features of a wave assignment statement to satisfy point 2.

The next example fits a convolution of a Gaussian peak with an exponential decay. It fits a baseline offset, amplitude and width of the Gaussian peak and the decay constant for the exponential. This might model an instrument with an exponentially decaying impulse response to recover the width of an impulsive signal.

```
Function convfunc(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   // pw[0] = gaussian baseline offset
   // pw[1] = gaussian amplitude
   // pw[2] = gaussian position
   // pw[3] = gaussian width
   // pw[4] = exponential decay constant of instrument response

   // Make a wave to contain an exponential with decay
   // constant pw[4]. The wave needs enough points to allow
   // any reasonable decay constant to get really close to zero.
   // The scaling is made symmetric about zero to avoid an X
   // offset from Convolve/A
   Variable dT = deltax(yw)
   Make/D/O/N=201 expwave           // Long enough to allow decay to zero
   SetScale/P x -dT*100,dT,expwave

   // Fill expwave with exponential decay
   expwave = (x>=0)*pw[4]*dT*exp(-pw[4]*x)

   // Normalize exponential so that convolution doesn't change
   // the amplitude of the result
   Variable sumexp
   sumexp = sum(expwave)
   expwave /= sumexp

   // Put a Gaussian peak into the output wave
   yw = pw[1]*exp(-((x-pw[2])/pw[3])^2)

   // Convolve with the exponential. NOTE /A.
   Convolve/A expwave, yw

   // Add the vertical offset AFTER the convolution to avoid end effects
   yw += pw[0]
End
```

Some things to be aware of with regard to this function:

1. A wave is created inside the function to store the exponential decay. Making a wave can be a time-consuming operation; it is less convenient for the user of the function, but can save computation time if you make a suitable wave ahead of time and then simply reference it inside the function.
2. The wave containing the exponential decay is passed to the convolve operation. The use of the /A flag prevents the convolve operation from changing the length of the output wave yw. You may wish to read the section on **Convolution** on page III-284.
3. The output wave yw is used as a parameter to the convolve operation. Because the convolve operation assumes that the data are evenly-space, this use of yw means that the function *does not satisfy* points 2) or 3) above. If you use input data to the fit that has missing points or unevenly-spaced X values, this function *will fail*.

You may also find the section **Waveform Arithmetic and Assignments** on page II-74 helpful.

Here is a much more complicated version of the fitting function that solves these problems, and also is more robust in terms of accuracy. The comments in the code explain how the function works.

Note that this function makes at least two different waves using the Make operation, and that we have used Make/D to make the waves double-precision. This can be crucial. To improve performance and reduce clutter in your Igor experiment file, we create the intermediate waves as free waves

```
Function convfunc(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   // pw[0] = gaussian baseline offset
   // pw[1] = gaussian amplitude
   // pw[2] = gaussian position
   // pw[3] = gaussian width
   // pw[4] = exponential decay constant of instrument response

   // Make a wave to contain an exponential with decay
   // constant pw[4]. The wave needs enough points to allow
   // any reasonable decay constant to get really close to zero.
   // The scaling is made symmetric about zero to avoid an X
   // offset from Convolve/A.

   // resolutionFactor sets the degree to which the exponential
   // will be over-sampled with regard to the problem's parameters.
   // Increasing this number increases the number of time
   // constants included in the calculation. It also decreases
   // the point spacing relative to the problem's time constants.
   // Increasing will also increase the time required to compute

   Variable resolutionFactor = 10
   // dt contains information on important time constants.
   // We wish to set the point spacing for model calculations
   // much smaller than the exponential time constant or gaussian width.
   // Use absolute values to prevent failures if an iteration strays
   // into unmeaningful but mathematically allowed negative territory.

   Variable absDecayConstant = abs(pw[4])
   Variable absGaussWidth = abs(pw[3])
   Variable dT = min(absDecayConstant/resolutionFactor, absGaussWidth/resolutionFactor)

   // Calculate suitable number points for the exponential. Length
   // of exponential wave is 10 time constants; doubled so
   // exponential can start in the middle; +1 to make it odd so
   // exponential starts at t=0, and t=0 is exactly the middle
```

```
    // point. That is better for the convolution.

    Variable nExpWavePnts = round((10*absDecayConstant)/dT)*2 + 1

    // Important: Make double-precision waves.
    // We make free waves to improve performance and so that
    // the intermediate waves clean themselves up at the end of
    // function execution.

    Make/D/FREE/O/N=(nExpWavePnts) expwave// Double-precision free wave

    // In this version of the function, we make a y output wave
    // ourselves, so that we can control the resolution and
    // accuracy of the calculation. It also will allow us to use
    // a wave assignment later to solve the problem of variable
    // X spacing or missing points.

    Variable nYPnts = max(resolutionFactor*numpnts(yw), nExpWavePnts)
    Make/D/FREE/O/N=(nYPnts) yWave         // Double-precision free wave

    // This wave scaling is set such that the exponential will
    // start at the middle of the wave
    SetScale/P x -dT*(nExpWavePnts/2),dT,expwave

    // Set the wave scaling of the intermediate output wave to have
    // the resolution calculated above, and to start at the first
    // X value.
    SetScale/P x xw[0],dT, yWave

    // Fill expwave with exponential decay, starting with X=0
    expwave = (x>=0)*dT/pw[4]*exp(-x/pw[4])

    // Normalize exponential so that convolution doesn't change
    // the amplitude of the result
    Variable sumexp
    sumexp = sum(expwave)
    expwave /= sumexp

    // Put a Gaussian peak into the intermediate output wave. We use
    // our own wave (yWave) because the convolution requires a wave
    // with even spacing in X, whereas we may get X values input that
    // are not evenly spaced.
    // Also, we do not add the vertical offset here - it will cause problems
    // with the convolution. Before the convolution Igor zero-pads the
    // wave, so the baseline here must be zero. Otherwise there is
    // a step function at the start of the convolution.
    yWave = pw[1]*exp(-((x-pw[2])/pw[3])^2)

    // Now convolve with the exponential. NOTE /A.
    Convolve/A expwave, yWave

    // Move appropriate values corresponding to input X data into
    // the output Y wave. We use a wave assignment involving the
    // input X wave. This will extract the appropriate values from
    // the intermediate wave, interpolating values where the
    // intermediate wave doesn't have a value precisely at an X
    // value that is required by the input X wave. This wave
    // assignment also solves the problem with auto-destination:
    // The function can be called with an X wave that sets any X
    // spacing, so it doesn't matter what X values are required.
    // This is the appropriate place to add the vertical offset.
```

```
   yw = yWave(xw[p]) + pw[0]
End
```

Note that all the waves created in this function are double-precision. If you don't use the /D flag with the Make operation, Igor makes single-precision waves. Curve fitting computations can be very sensitive to floating-point roundoff errors. We have solved many user problems by simply making intermediate waves double precision.

None of the computations involve the wave yw. That was done so that the computations could be done at finer resolution than the resolution of the fitted data. By making a separate wave, it is not necessary to modify yw.

The actual return values are picked out of yWave and stored in yw using the special X-scale-based indexing available for one-dimensional waves in a wave assignment. This allows any arbitrary X values in the xw input wave, so long as the intermediate computations are done over the full X range included in xw.

Multivariate All-At-Once Fitting Functions

A multivariate all-at-once fitting function accepts more than one independent variable. To make one, simply add additional waves after the xw wave:

```
Function MultivariateAllAtOnce(pw, yw, xw1, xw2) : FitFunc
   WAVE pw, yw, xw1, xw2

   yw = <expression involving pw and all the xw waves>
End
```

This example accepts two independent variables, xw1 and xw2. If you have more, add additional X wave parameters.

All the X waves are 1D waves with the same number of points as the Y wave, yw, even if your input data is in the form of a matrix wave. If you use **FuncFitMD** to fit an N row by M column matrix, Igor unrolls the matrix to form a 1D Y wave containing NxM points. This is passed to your function as the Y wave parameter. The X wave parameters are 1D waves containing NxM points with values that repeat for each column.

If you know that the input data fall into a fully-formed matrix wave, you may be able to temporarily re-fold the yw wave into a matrix using the Redimension operation:

```
Function MultivariateAllAtOnce(pw, yw, xw1, xw2) : FitFunc
   WAVE pw, yw, xw1, xw2

   Redimension/N=(rows, columns) yw
   MatrixOP yw = <matrix math expression>
   Redimension/N=(rows*columns) yw

   yw = <expression involving pw and all the xw waves>
End
```

Use this technique with great caution, though. In order to know the appropriate rows and columns for the Redimension command, you must make assumptions about the size of the original data set. If there are missing values (NaNs) in the original input matrix, these values will be missing from the Y wave, resulting in fewer points than you would expect.

If you use this technique, you will not be able to use the auto-destination feature (the /D flag with no destination wave) because the auto-destination wave is pretty much guaranteed to be a different size than you expect.

Similarly to the second example all-at-once function above, you can create an intermediate matrix wave and then use wave assignment to get values from the matrix to assign to the yw wave. It may be tricky to figure out how to extract the correct values from your intermediate matrix.

Structure Fit Functions

Sometimes you may need to transmit extra information to a fitting function. This might include constants that need to be set to reflect conditions in a run, but should not be fit; or a wave containing a look-up table or a measured standard sample that is needed for comparison. Perhaps your fit function is very complex and needs some sort of book-keeping data.

If you use a basic fit function or an all-at-once function, such information must be transmitted via global variables and waves. That, in turn, requires that this global information be looked up inside the fit function. The global data requirement makes it difficult to be flexible: the fit function is tied to certain global variables and waves that must be present for the function to work. In addition to adding complexity and difficulty to management of global information, it adds a possible time-consuming operation: looking up the global information requires examining a list of waves or variables, and comparing the names to the desired name.

Structure fit functions are ideal for such problems because they take a single parameter that is a structure of your own design. The first few members of the structure must conform to certain requirements, but the rest is up to you. You can include any kind of data in the structure. An added bonus is that you can include members in the structure that identify for the fit function which fit coefficient is being perturbed when calculating numerical derivatives, that signal when the fit function is being called to fill in the auto-destination wave, and identify when a new fit iteration is starting. You can pass back a flag to have FuncFit abandon fitting.

To use a structure fit function, you must do three things:

1. Define a structure containing certain standard items at the top.
2. Write a fitting function that uses that structure as its only parameter.
3. Write a wrapper function for FuncFit that creates an instance of your structure, initializes it and invokes FuncFit with the /STRC parameter flag.

You should familiarize yourself with the use of structures before attempting to write a structure fit function (see **Structures in Functions** on page IV-99).

Structure fit functions come in basic and all-at-once variants; the difference is determined by the members at the beginning of the structure. The format for the structure for a basic structure fit function is:

```
Structure myBasicFitStruct
   Wave coefw
   Variable x
   …
EndStructure
```

The name of the structure can be anything you like that conforms to Igor’s naming rules; all that is required is that the first two fields be a wave and a variable. By convention we name the wave coefw and the variable x to match the use of those members. These members of the structure are equivalent to wave and variable parameters required of a basic fit function.

You may wish to use an all-at-once structure fit function for the same reason you might use a regular all-at-once fit function. The same concerns apply to all-at-once structure fit functions; you should read and understand **All-At-Once Fitting Functions** on page III-256 before attempting to write an all-at-once structure fit function.

The structure for an all-at-once structure fit function is:

```
Structure myAllAtOnceFitStruct
   Wave coefw
   Wave yw
   Wave xw
   …
EndStructure
```

The first three members of the structure are equivalent to the pw, yw, and xw parameters required in a regular all-at-once fit function.

A simple example of fitting with a structure fit function follows. More examples are available in the File menu: select appropriate examples from Examples→Curve Fitting.

Basic Structure Fit Function Example

As an example of a basic structure fit function, we will write a fit function that fits an exponential decay using an X offset to compensate for numerical problems that can occur when the X range of an exponential function is small compared to the X values. The X offset must not be a fit coefficient because it is not mathematically distinguishable from the decay amplitude. We will write a structure that carries this constant as a custom member of a structure. This function will duplicate the built-in exp_XOffset function (see **Built-in Curve Fitting Functions** on page III-206).

First, we create fake data by executing these commands:

```
Make/D/O/N=100 expData,expDataX
expDataX = enoise(0.5)+100.5
expData = 1.5+2*exp(-(expDataX-100)/0.2) + gnoise(.05)
Display expData vs expDataX
ModifyGraph mode=3,marker=8
```

Now the code. Copy this code and paste it into your Procedure window:

```
// The structure definition
Structure expFitStruct
   Wave coefw           // required coefficient wave
   Variable x           // required X value input
   Variable x0          // constant
EndStructure

// The fitting function
Function fitExpUsingStruct(s) : FitFunc
   Struct expFitStruct &s

   return s.coefw[0] + s.coefw[1]*exp(-(s.x-s.x0)/s.coefw[2])
End

// The driver function that calls FuncFit:
Function expStructFitDriver(pw, yw, xw, xOff)
   Wave pw     // coefficient wave- pre-load it with initial guess
   Wave yw
   Wave xw
   Variable xOff
   Variable doODR

   // An instance of the structure. We initialize the x0 constant only,
   // Igor (FuncFit) will initialize coefw and x as required.
   STRUCT expFitStruct fs
   fs.x0 = xOff   // set the value of the X offset in the structure

   FuncFit fitExpUsingStruct, pw, yw /X=xw /D /STRC=fs

   // no history report for structure fit functions. We print our own
   // simple report here:
   print pw
   Wave W_sigma
   print W_sigma
End
```

Finally, make a coefficient wave loaded with initial guesses and invoke our driver function:

```
Make/D/O expStructCoefs = {1.5, 2, .2}
expStructFitDriver(expStructCoefs, expData, expDataX, 100)
```

This is a very simple example, intended to show only the most basic aspects of fitting with a structure fit function. An advanced programmer could add a control panel user interface, plus code to automatically calculate initial guesses and provide a default value of the x0 constant.

The WMFitInfoStruct Structure

In addition to the required structure members, you can include a WMFitInfoStruct structure member immediately after the required members. The WMFitInfoStruct structure, if present, will be filled in by FuncFit with information about the progress of fitting, and includes a member allowing you to stop fitting if your fit function detects a problem.

Adding a WMFitInfoStruct member to the structure in the example above:

```
Structure expFitStruct
   Wave coefw                      // Required coefficient wave.
   Variable x                      // Required X value input.
   STRUCT WMFitInfoStruct fi       // Optional WMFitInfoStruct.
   Variable x0                     // Constant.
EndStructure
```

And the members of the WMFitInfoStruct:

WMFitInfoStruct Structure Members

Member	Description
char IterStarted	Nonzero on the first call of an iteration.
char DoingDestWave	Nonzero when called to evaluate the autodestination wave.
char StopNow	Fit function sets this to nonzero to indicate that a problem has occurred and fitting should stop.
Int32 IterNumber	Number of iterations completed.
Int32 ParamPerturbed	Index of the fit coefficient being perturbed for the calculation of numerical derivatives. Set to -1 when evaluating a solution point with no perturbed coefficients.

The IterStarted and ParamPerturbed members may be useful in some obscure cases to short-cut lengthy computations. The DoingDestWave member may be useful in an all-at-once structure fit function.

Multivariate Structure Fit Functions

To fit multivariate functions (those having more than one dimension or independent variable) you simply use an array for the X member of the structure. For instance, for a basic 2D structure fit function:

```
Structure My2DFitStruct
   Wave coefw
   Variable x[2]
   …
EndStructure
```

Or a 2D all-at-once structure fit function:

```
Structure My2DAllAtOnceFitStruct
   Wave coefw
   Wave yw
   Wave xw[2]
   …
EndStructure
```

Curve Fitting Using Commands

A few curve fitting features are not completely supported by the Curve Fitting dialog, such as constraints involving combinations of fit coefficients, or user-defined fit functions involving more complex constructions. Also, you might sometimes want to batch-fit to a number of data sets without interacting with the dialog for each data set. These circumstances will require that you use command lines to do fits.

The easiest way to do this is to use the Curve Fitting dialog to generate command lines that do almost what you want. If you simply want to add a more complex feature, such as a more complicated constraint expression, click the To Cmd Line button, then edit the commands generated by the dialog to add the features you want.

If you are writing a user procedure that does curve fitting, you can click the To Clip button to copy the commands generated by the dialog. Then paste the commands into the Procedure window. Edit them as needed by your application.

Curve fitting is done by three operations — **CurveFit**, **FuncFit**, and **FuncFitMD**. You will find details on these operations in Chapter V-1, **Igor Reference**.

Batch Fitting

If you are doing batch fitting, you probably want to call a curve fitting operation inside a loop. In that case, you probably don't want a history report for every fit- it could possible make a very large amount of text in the history. You probably don't want the progress window during the fits- it slows down the fit and will make a flashing window as it appears and disappears. Finally, you probably don't want graphs and tables updating during the fits, as this can slow down computation considerably.

Here is an example function that will do all of this, plus it checks for an error during the fit. If the use of the $ operator is unfamiliar you will want to consult **Accessing Waves in Functions** on page IV-82**.**

```
Function FitExpToListOfWaves(theList)
   String theList

   Variable i=0
   string aWaveName = ""
   Variable V_fitOptions = 4      // suppress progress window
   Variable V_FitError = 0        // prevent abort on error
   do
      aWaveName = StringFromList(i, theList)
      WAVE/Z aWave = $aWaveName
      if (!WaveExists(aWave))
         break
      endif

      // /N suppresses screen updates during fitting
      // /Q suppresses history output during fitting
      CurveFit/N/Q exp aWave /D/R
      WAVE W_coef

      // save the coefficients
      Duplicate/O W_coef $("cf_"+aWaveName)
      // save errors
      Duplicate/O W_sigma, $("sig_"+aWaveName)
      if (V_FitError != 0)
         // Mark the results as being bad
         WAVE w = $("cf_"+aWaveName)
         w = NaN
         WAVE w = $("sig_"+aWaveName)
         w = NaN
         WAVE w = $("fit_"+aWaveName)
         w = NaN
         WAVE w = $("Res_"+aWaveName)
         w = NaN
         V_FitError = 0
      endif
      i += 1
   while(1)
End
```

This function is very limited. It simply does fits to a number of waves in a list. Igor includes a package that adds a great deal of sophistication to batch fitting, and provides a user interface. For a demonstration, choose File→Example Experiments→Curve Fitting→Batch Curve Fitting Demo.

Curve Fitting Examples

The Igor Pro Folder includes a number of example experiments that demonstrate the capabilities of curve fitting. These examples cover fitting with constraints, multivariate fitting, multipeak fitting, global fitting, fitting a line between cursors, and fitting to a user-defined function. All of these experiments can be found in Igor Pro Folder/Examples/Curve Fitting.

Singularities in Curve Fitting

You may occasionally run across a situation where you see a “singular matrix” error. This means that the system of equations being solved to perform the fit has no unique solution. This generally happens when the fitted curve contains degeneracies, such as if all Y values are equal.

In a fit to a user-defined function, a singular matrix results if one or more of the coefficients has no effect on the function’s return value. Your coefficients wave must have the exact same number of points as the number of coefficients that you actually use in your function or else you must hold constant unused coefficients.

Certain functions may have combinations of coefficients that result in one or more of the coefficients having no effect on the fit. Consider the Gaussian function:

$$K_0 + K_1 \exp((x - K_2)/K_3)^2$$

If K_1 is set to zero, then the following exponential has no effect on the function value. The fit will report which coefficients have no effect. In this example, it will report that K_2 and K_3 have no effect on the fit. However, as this example shows, it is often not the reported coefficients that are at fault.

Special Considerations for Polynomial Fits

Polynomial fits use the singular value decomposition technique. If you encounter singular values and some of the coefficients of the fit have been zeroed, you are probably asking for more terms than can be supported by your data. You should use the smallest number of terms that gives a “reasonable” fit. If your data does not support higher-order terms then you can actually get a poorer fit by including them.

If you really think your data should fit with the number of terms you specified, you can try adjusting the singular value threshold. You do this by creating a special variable called V_tol and setting it to a value smaller than the default value of 1e-10. You might try 1e-15.

Another way to run into trouble during polynomial fitting is to use a range of X values that are very much offset from zero. The solution is to temporarily offset your X values, do the fit and then restore the original X values. This is done for you by the poly_XOffset fit function; see **Built-in Curve Fitting Functions** on page III-206 for details.

Errors Due to X Values with Large Offsets

The single and double exponential fits can be thrown off if you try to fit to a range of X values that are very much offset from zero. In general, any function, which, when extrapolated to zero, returns huge or infinite values, can create problems. The solution is to temporarily offset your x values, perform the fit and then restore the original x values. You may need to perform a bit of algebra to fix up the coefficients.

As an example consider a fit to an exponential where the x values range from 100 to 101. We temporarily offset the x values by 100, perform the fit and then restore the x values by adding 100. When we did the fit, rather than fitting to k0+k1*exp(-k2*x) we really did the fit to c0+c1*exp(-c2*(x-100)). A little rearrangement and we have c0+c1*exp(-c2*x)*exp(c2*100). Comparing these expressions, we see that k0= c0, k1= c1*exp(c2*100) and k2= c2.

A better solution to the problem of fitting exponentials with large X offsets is to use the built-in exp_XOffset and dblexp_XOffset fit functions. These fit functions automatically incorporate the X shifting; see **Built-in Curve Fitting Functions** on page III-206 for details.

The same problem can occur when fitting to high-degree polynomials. In this case, the algebra required to transform the solution coefficients back to unoffset X values is nontrivial. It would be better to simply redefine your problem in terms of offset X value.

Curve Fitting Troubleshooting

If you are getting unsatisfactory results from curve fitting you should try the following before giving up.

Make sure your data is valid. It should not be all one value. It should bear some resemblance to the function that you're trying to fit it to.

If the fit is iterative try different initial guesses. Some fit functions require initial guesses that are very close to the correct solution.

If you are fitting to a user-defined function, check the following:

- Your coefficients wave must have exactly the same number of points as the number of coefficients that you actually use in your function unless you hold constant the unused coefficients.
- Your initial guesses should not be zero unless the expected range is near 1.0 or you have specified an epsilon wave. See **The Epsilon Wave** on page III-267 for details.
- Ensure that your function is working properly. Try plotting it over a representative domain.
- Examine your function to ensure all your coefficients are distinguishable. For example in the fragment (k0+k1)*x, k0 and k1 are indistinguishable. If this situation is detected, the history will contain the message: "Warning: These parameters may be linearly dependent:" followed by a line listing the two parameters that were detected as being indistinguishable.
- Because the derivatives for a user-defined fit function are calculated numerically, if the function depends only weakly on a coefficient, the derivatives may appear to be zero. The solution is to create an epsilon wave and set its values large enough to give a nonzero difference in the function output. See **The Epsilon Wave** on page III-267 for details.
- A variation the previous problem is a function that changes in a step-wise fashion, or is "noisy" because an approximation is used that is good to only a limited precision. Again, create an epsilon wave and set the values large enough to give nonzero differences that are of consistent sign.
- Verify that each of your coefficients has an effect on the function. In some cases, a coefficient may have an effect over a limited range of X values. If your data do not sample that range adequately the fit may not work well, or may give a singular matrix error.
- Make sure that the optimal value of your coefficients is not infinity (it takes a long time to increment to infinity).
- Check to see if your function could possibly return NaN or INF for any value of the coefficients. You might be able to add constraints to prevent this from happening. You will see warnings if a singular matrix error resulted from NaN or INF values returned by the fitting function.
- Use a double-precision coefficient wave. Curve fitting is numerically demanding and usually works best if all computations are done using double precision numbers. Using a single-precision coefficient wave often causes failures due to numeric truncation. Because the **Make** operation defaults to single precision, you must use the /D flag when creating your coefficient wave.
- If you use intermediate waves in your user-defined fit function, make sure they are all double precision. Because the **Make** operation defaults to single precision, you must use the /D flag when creating your coefficient wave.

The Epsilon Wave

Curve fitting uses partial derivatives of your fit function with respect to the fit coefficients in order to find the gradient of the chi-square surface. It then solves a linearized estimate of the chi-square surface to find the next estimate of the solution (the minimum in the chi-square surface).

Since Igor doesn't know the mathematical equation for your fit function, it must approximate derivatives numerically using finite differences. That is, a model value is calculated at the present estimate of the fit coefficients, then each coefficient is perturbed by a small amount, and the derivative is calculated from the difference. This small perturbation, which is different for each coefficient, is called "epsilon".

You can specify the epsilon value for each coefficient by providing an epsilon wave to the **CurveFit** or **FuncFit** operations using the /E flag. If you do not provide an epsilon wave, Igor determines epsilon for each coefficient using these rules:

If your coefficient wave is single precision (which is not recommended),

```
if coef[i] == 0
   eps[i] = 1e-4
else
   eps[i] = 1e-4*coef[i]
```

If your coefficient wave is double precision,

```
if coef[i] == 0
   eps[i] = 1e-10
else
   eps[i] = 1e-10*coef[i]
```

In the Curve Fitting dialog, when you are using a user-defined fitting function, you can select an epsilon wave from the Epsilon Wave menu. When you select _New Wave_, or if you select an existing wave from the menu, Igor adds a column to the Coefficients list where you can edit the epsilon values.

There are a couple of reasons for explicitly setting epsilon. One is if your fit function is insensitive to a coefficient. That is, perturbing the coefficient makes a very small change in the model value. Sometimes the dependence of the model is so small that floating-point truncation results in no change in the model value. You have to set epsilon to a sufficiently large value that the model actually changes when the perturbation is applied.

You also need to set epsilon is when the model is discrete or noisy. This can happen if the model involves a table look-up or a series solution of some sort. In the case of table look-up, epsilon needs to be large enough to make sure that you get two distinct values out of the table.

In the case of a series solution, you have to stop summing terms in the series at some point. If the truncation of the series results in less than full floating-point resolution of the series, you need to make sure epsilon is large enough that the change in the model is larger than the resolution of the series. A series might include something like a numerical solution of an ODE, using the **IntegrateODE** operation. It could also involve **FindRoots** or **Optimize**, each of which gives you an approximate result. Since these operations run faster if you don't demand high precision, there may be a strong incentive to decrease the accuracy of the computation, and that may in turn lead to a need for an epsilon wave.

Curve Fitting References

An explanation of the Levenberg-Marquardt nonlinear least squares optimization can be found in Chapter 14.4 of:

Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

The algorithm used for applying constraints is given in:

Shrager, Richard, Quadratic Programming for Nonlinear Regression, *Communications of the ACM*, *15*, 41-45, 1972.

The method is described in gory mathematical detail in:

Shrager, Richard, Nonlinear Regression With Linear Constraints: An Extension of the Magnified Diagonal Method, *Journal of the Association for Computing Machinery*, *17*, 446-452, 1970.

References for the ODRPACK95 package used for orthogonal distance regression:

Boggs, P.T., R.H. Byrd, and R.B. Schnabel, A Stable and Efficient Algorithm for Nonlinear Orthogonal Distance Regression, *SIAM Journal of Scientific and Statistical Computing*, *8*, 052-1078, 1987.

Boggs, P.T., R.H. Byrd, J.R. Donaldson, and R.B. Schnabel, Algorithm 676 - ODRPACK: Software for Weighted Orthogonal Distance Regression, *ACM Transactions on Mathematical Software*, *15*, 348-364, 1989

Boggs, P.T., J.R. Donaldson, R.B. Schnabel and C.H. Spiegelman, A Computational Examination of Orthogonal Distance Regression, *Journal of Econometrics*, *38*, 69-201, 1988.

An exhaustive, but difficult to read source for nonlinear curve fitting:

Seber, G.A.F, and C.J. Wild, *Nonlinear Regression*, John Wiley & Sons, 1989.

A discussion of the assumptions and approximations involved in calculating confidence bands for nonlinear functions can be found in the beginning sections of Chapter 5.

General books on curve fitting and statistics:

Draper, N., and H. Smith, *Applied Regression Analysis*, John Wiley & Sons, 1966.

Box, G.E.P., W.G. Hunter, and J.S. Hunter, *Statistics for Experimenters*, John Wiley & Sons, 1978.

Chapter

III-9

Signal Processing

Overview

Analysis tasks in Igor range from simple experiments using no programming to extensive systems tailored for specific fields. Chapter I-2, **Guided Tour of Igor Pro**, shows examples of the former. WaveMetrics' "Peak Measurement" technical note is an example of the latter.

The Signal Processing chapter covers basic analysis operations with emphasis on signal transformations.

Fourier Transforms

Igor uses the Fast Fourier Transform (FFT) algorithm to compute a Discrete Fourier Transform (DFT). The FFT is usually called from an Igor procedure as one step in a larger process, such as finding the magnitude and phase of a signal. Igor's FFT uses a prime factor decomposition multidimensional algorithm. Prime factor decomposition allows the algorithm to work on nearly any number of data points. Previous versions of Igor were restricted to a power-of-two number of data points.

This section concentrates on the one-dimensional FFT. See **Multidimensional Fourier Transform** on page II-98 for information on multidimensional aspects of the FFT.

You can perform a Fourier transform on a wave by choosing Analysis→Fourier Transforms. This displays the Fourier Transforms dialog.

Select the type of transform by clicking the Forward or Reverse radio button. Select the wave that you want to transform from the Wave list. If you enable the From Target box under the Wave list, only appropriate waves in the target window will appear in the list.

Why Some Waves Aren't Listed

What do we mean by "appropriate" waves?

The data can be either real or complex. If the data are real, the number of data points must be even. This is an artificial limitation that was introduced in order to guarantee that the inverse transform of a forward-transformed wave is equal to the original wave. For multidimensional data, only the number of rows must be even. You can work around some of the restrictions of the inverse FFT with the command line.

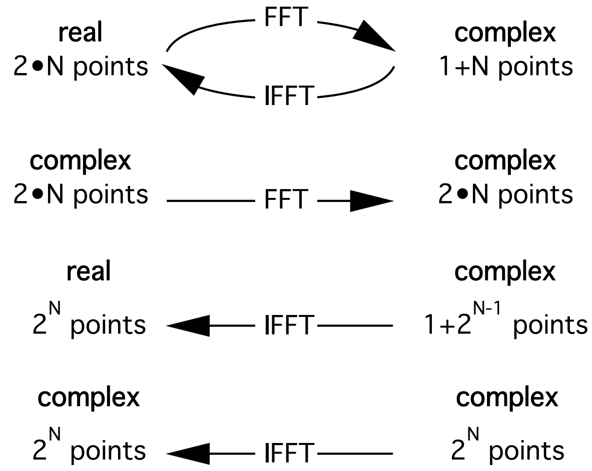
The inverse FFT requires complex data. There are no restrictions on the number of data points. However, for historic and compatibility reasons, certain values for the number of points are treated differently as described in the next sections.

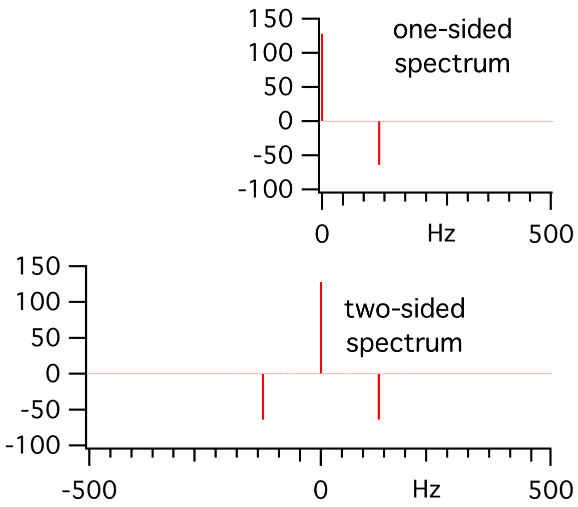
Changes in Wave Type and Number of Points

If the wave is a 1D real wave of N points (N must be even), the FFT operation results in a complex wave consisting of N/2+1 points containing the "one-sided spectrum". The negative spectrum is not computed, since it is identical for real input waves.

If the wave is complex (even if the imaginary part is zero), its data type and number of points are unchanged by the forward FFT. The FFT result is a "two-sided spectrum", which contains both the positive *and* the negative frequency spectra, which are different if the imaginary part of the complex input data is nonzero.

The diagram below shows the two-sided spectrum of 128-point data containing a zero imaginary component.





Magic Number of Points and the IFFT

When performing the inverse FFT, the input is always complex, but the result may be either real or complex.

Because versions of Igor prior to 3.0 only allowed an integral power of two (2^n) to be forward-transformed, Igor could tell from the number of points in the forward-transformed wave what kind of result to create for the inverse transform. To ensure compatibility, Igor versions 3.0 and after continue to treat certain numbers of points as "magical."

If the number of points in the wave is an integral power of two (2^n), then the wave resulting from the IFFT is complex. If the number of points in the wave is one greater than an integral power of two ($1+2^n$), then the wave resulting from the IFFT is real and of length (2^{n+1}).

If the number of points is not one of the two magic values, then the result from the inverse transform is real unless the complex result is selected in the Fourier Transforms dialog.

Changes in X Scaling and Units

The FFT operation changes the X scaling of the transformed wave. If the X-units of the transformed wave are time (s), frequency (Hz), length (m), or reciprocal length (m^{-1}), then the resulting wave units are set to the respective conjugate units. Other units are currently ignored. The X scaling's X_0 value is altered depending on whether the wave is real or complex, but dx is always set the same:

$$\Delta x_{FFT} = \frac{1}{N \cdot \Delta x_{original}} \qquad where,\ N \equiv original\ length\ of\ wave$$

If the original wave is real, then after the FFT its minimum X value (X_0) is zero and its maximum X value is:

$$\begin{aligned} x_{N/2} &= \frac{N}{2} \cdot \Delta x_{FFT} = \frac{N}{2} \cdot \frac{1}{N \cdot \Delta x_{original}} \\ &= \frac{1}{2 \cdot \Delta x_{original}} \\ &= Nyquist\ Frequency \end{aligned}$$

If the original wave is complex, then after the FFT its maximum X value is $X_{N/2}$ - dX_{FFT}, its minimum X value is -$X_{N/2}$, and the X value at point N/2 is zero.

The IFFT operation reverses the change in X scaling caused by the FFT operation except that the X value of point 0 will always be zero.

FFT Amplitude Scaling

Various programs take different approaches to scaling the amplitude of an FFTed waveform. Different scaling methods are appropriate for different analyses and there is no general agreement on how this is done. Igor uses the method described in *Numerical Recipes in C* (see **References** on page III-316) which differs from many other references in this regard.

The DFT equation computed by the FFT for a complex $wave_{orig}$ with N points is:

$$wave_{FFT}[n] = \sum_{k=0}^{N-1} wave_{orig}[k] \cdot e^{2\pi i \cdot kn/N}, \text{ where } i = \sqrt{-1}$$

$wave_{orig}$ and $wave_{FFT}$ refer to the same wave before and after the FFT operation.

The IDFT equation computed by the IFFT for a complex $wave_{FFT}$ with N points is:

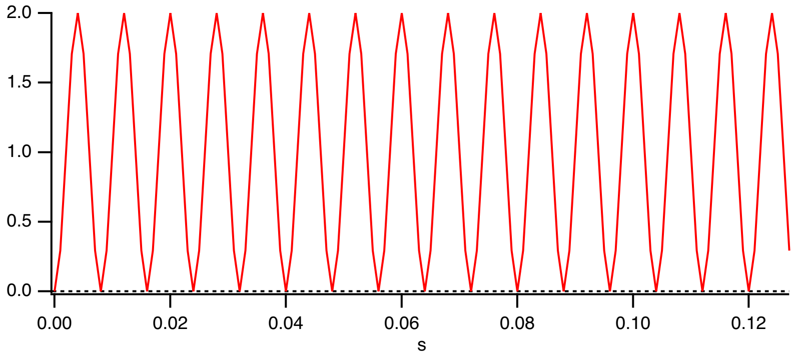
$$wave_{IFT}[n] = \frac{1}{N} \cdot \sum_{k=0}^{N-1} wave_{FFT}[k] \cdot e^{-2\pi i \cdot kn/N}, \text{ where } i = \sqrt{-1}$$

To scale wave_{FFT} to give the same results you would expect from the continuous Fourier Transform, you must divide the spectral values by N, the number of points in $\text{wave}_{\text{orig}}$.

However, for the FFT of a real wave, only the positive spectrum (containing spectra for positive frequencies) is computed. This means that to compare the Fourier and FFT amplitudes, you must account for the identical negative spectra (spectra for negative frequencies) by doubling the positive spectra (but not $\text{wave}_{\text{FFT}}[0]$, which has no negative spectral value).

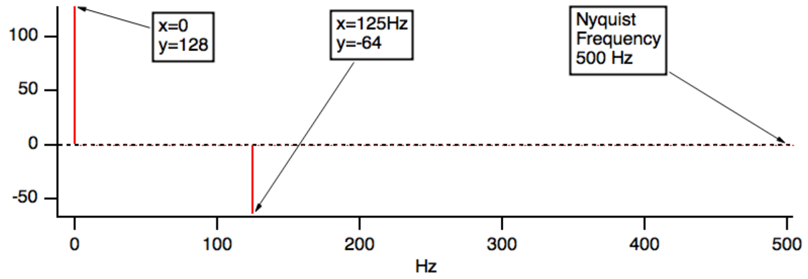
For example, here we compute the one-sided spectrum of a real wave, and compare it to the expected Fourier Transform values:

```
Make/N=128 wave0
SetScale/P x 0,1e-3,"s",wave0 // dx=1ms,Nyquist frequency is 500Hz
wave0= 1 - cos(2*Pi*125*x)    // signal frequency is 125Hz, amp. is -1
Display wave0;ModifyGraph zero(left)=3
```



```
FFT wave0
```

Igor computes the “one-sided” spectrum and updates the graph:



The Fourier Transform would predict a zero-frequency (“DC”) result of 1, which is what we get when we divide the FFT value of 128 by the number of input values which is also 128. In general, the Fourier Transform value at zero frequency is:

$$\textit{Fourier Transform Amplitude}(0) = \frac{1}{N} \cdot real(r2polar(wave_{FFT}(0)))$$

The Fourier Transform would predict a spectral peak at -125Hz of amplitude (-0.5 + i0), and an identical peak in the positive spectrum at +125Hz. The sum of those expected peaks would be (-1+0·i).

(This example is contrived to keep the imaginary part 0; the real part is negative because the input signal contains -cos(…) instead of + cos(…).)

Igor computed only the positive spectrum peak, so we double it to account for the negative frequency peak twin. Dividing the doubled peak of -128 by the number of input values results in (-1+i0), which agrees with the Fourier Transform prediction. In general, the Fourier Transform value at a nonzero frequency f is:

$$\textit{Fourier Transform Amplitude}(f) = \frac{2}{N} \cdot real(r2polar(wave_{FFT}(f)))$$

The only exception to this is the Nyquist frequency value (the last value in the one-sided FFT result), whose value in the one-sided transform is the same as in the two-sided transform (because, unlike all the other frequency values, the two-sided transform computes only one Nyquist frequency value). Therefore:

$$\textit{Fourier Transform Amplitude}(f_{Nyquist}) = \frac{1}{N} \cdot real(r2polar(wave_{FFT}(f_{Nyquist})))$$

The frequency resolution $dX_{FFT} = 1/(N_{original} \cdot dx_{original})$, or 1/(128*1e-3) = 7.8125 Hz. This can be verified by executing:

```
Print deltax(wave0)
```

Which prints into the history area:

```
  7.8125
```

You should be aware that if the input signal *is not* a multiple of the frequency resolution (our example *was* a multiple of 7.8125 Hz) that the energy in the signal will be divided among the two closest frequencies in the FFT result; this is different behavior than the continuous Fourier Transform exhibits.

Phase Polarity

There are two different definitions of the Fourier transform regarding the phase of the result. Igor uses a method that differs in sign from many other references. This is mainly of interest if you are comparing the result of an FFT in Igor to an FFT in another program. You can convert from one method to the other as follows:

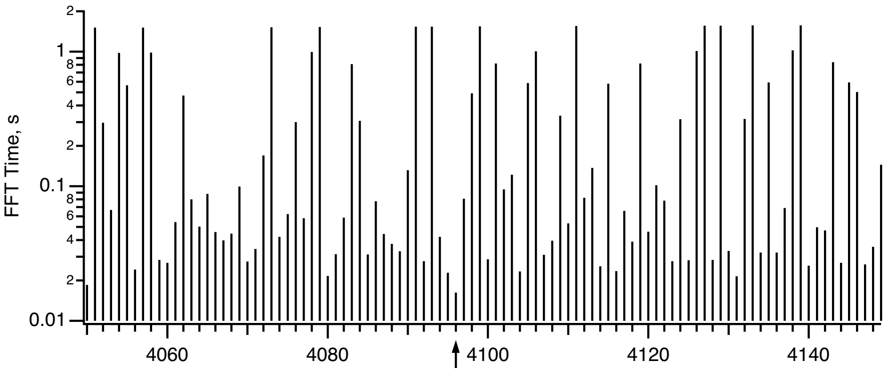
```
FFT wave0; wave0=conj(wave0)  // negate the phase angle by changing
                              // the sign of the imaginary component.
```

Effect of FFT and IFFT on Graphs

Igor displays complex waves in Lines between points mode by default. But, as demonstrated above, if you perform an FFT on a wave that is displayed in a graph and the display mode for that wave is lines between points, then Igor changes its display mode to Sticks to zero. Also, if you perform an IFFT on a wave that is displayed in a graph and the display mode for that wave is Sticks to zero then Igor changes its display mode to Lines between points.

Effect of the Number of Points on the Speed of the FFT

Although the prime factor FFT algorithm does not require that the number of points be a power of two, the speed of the FFT can degrade dramatically when the number of points can not be factored into small prime numbers. The following graph shows the speed of the FFT on a complex vector of varying number of points. Note that the time (speed) axis is log. The results are from a Power Mac 9500/120.



The arrow is at N=4096, a power of two. For that number of points, the FFT time was less than 0.02 seconds while other nearby values exceed one second. The moral of the story is that you should avoid numbers of points that have large prime factors (4078 takes a long time- it has prime factors 2039 and 2). You should endeavor to use a number with small prime factors (4080 is reasonably fast — it has prime factors 2*2*2*2*3*5*17). For best performance, the number of points should be a power of 2, like 4096.

Finding Magnitude and Phase

The FFT operation can create a complex, real, magnitude, magnitude squared, or phase result directly when you choose the desired output type in the Fourier Transforms dialog

If you choose to use the complex wave result of the FFT operation you can compute the magnitude and phase using the **WaveTransform** operation (see page V-1090) (with keywords magnitude, magsqr, and phase), or with various procedures from the WaveMetrics Procedures folder (described in the next section).

If you want to unwrap the phase wave (to eliminate the phase jumps that occur between ±180 degrees), use the Unwrap operation or the Unwrap Waves dialog in the Data menu. See **Unwrap** on page V-1050. In two dimensions you can use **ImageUnwrapPhase** operation (see page V-433).

Magnitude and Phase Using WaveMetrics Procedures

For backward compatibility you can compute FFT magnitude and phase using the WaveMetrics-provided procedures in the “WaveMetrics Procedures:Analysis:DSP (Fourier Etc)” folder.

You can access them using Igor’s “#include” mechanism. See **The Include Statement** on page IV-166 for instructions on including a procedure file.

The WM Procedures Index help file, which you can access from the Help→Help Windows menu, is a good way to find out what routines are available and how to access them.

FTMagPhase Functions

The FTMagPhase functions provide an easy interface to the FFT operation. FTMagPhase has the following features:

- Automatic display of the results.

- Original data is untouched.
- Can display magnitude in decibels.
- Optional phase display in degrees or radians.
- Optional 1D phase unwrapping.
- Resolution enhancement.
- Supports non-power-of-two data with optional windowing.

Use `#include <FTMagPhase>` in your procedure file to access these functions.

FTMagPhaseThreshold Functions

The FTMagPhaseThreshold functions are the same as the FTMagPhase procedures, but with an extra feature:

- Phase values for low-amplitude signals may be ignored.

Use `#include <FTMagPhaseThreshold>` in your procedure file to access these functions.

DFTMagPhase Functions

The DFTMagPhase functions are similar to the FTMagPhase procedures, except that the slower Discrete Fourier Transform is used to perform the calculations:

- User-selectable frequency start and end.
- User-selectable number of frequency bands.

The procedures also include the DFTAtOneFrequency procedure, which computes the amplitude and phase at a single user-selectable frequency.

Use `#include <DFTMagPhase>` in your procedure file to access these functions.

CmplxToMagPhase Functions

The CmplxToMagPhase functions convert a complex wave, presumably the result of an FFT, into separate magnitude and phase waves. It has many of the features of FTMagPhase, but doesn't do the FFT.

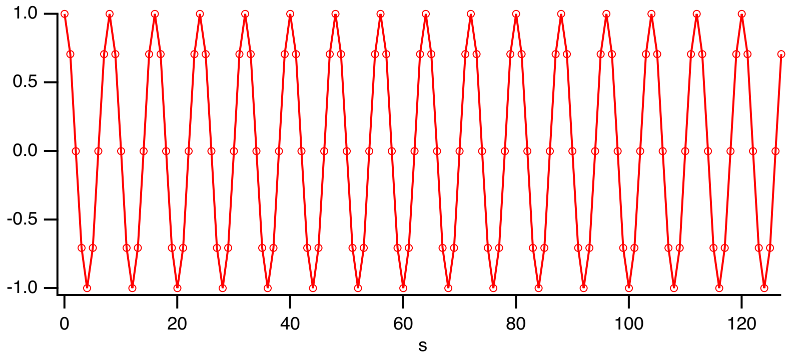
Use `#include <CmplxToMagPhase>` in your procedure file to access these functions.

Spectral Windowing

The FFT computation makes an assumption that the input data repeats over and over. This is important if the initial value and final value of your data are not the same. A simple example of the consequences of this repeating data assumption follows.

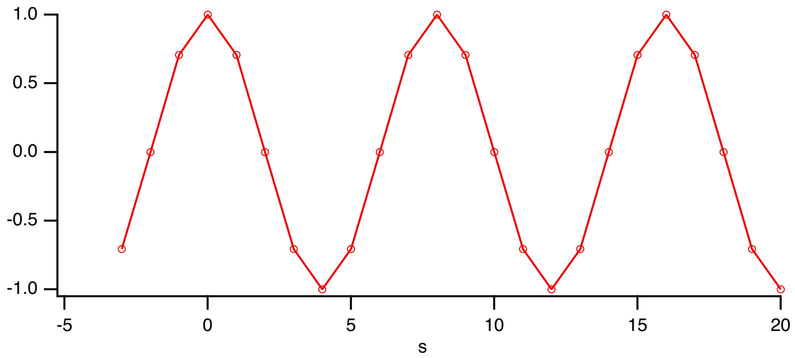
Suppose that your data is a sampled cosine wave containing 16 complete cycles:

```
Make/O/N=128 cosWave=cos(2*pi*p*16/128)
SetScale/P x, 0, 1, "s", cosWave
Display cosWave
ModifyGraph mode=4,marker=8
```



Notice that if you copied the last several points of cosWave to the front, they would match up perfectly with the first several points of cosWave. In fact, let's do that with the **Rotate** operation (see page V-810):

```
Rotate 3,cosWave        // wrap last three values to front of wave
SetAxis bottom,-5,20    // look more closely there
```

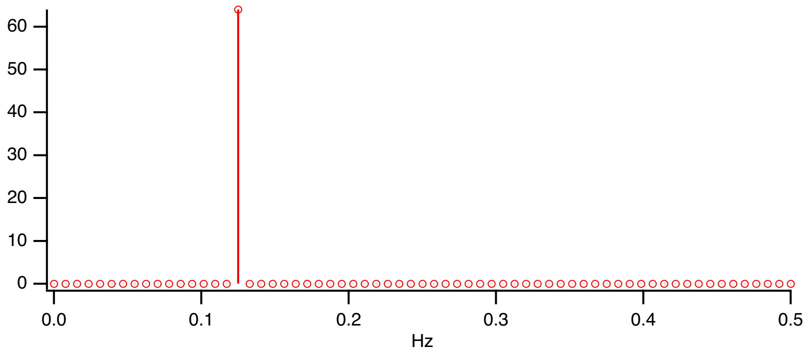


The rotated points appear at x=-3, -2, and -1. This indicates that there is no discontinuity as far as the FFT is concerned.

Because of the absence of discontinuity, the FFT magnitude result matches the ideal expectation:

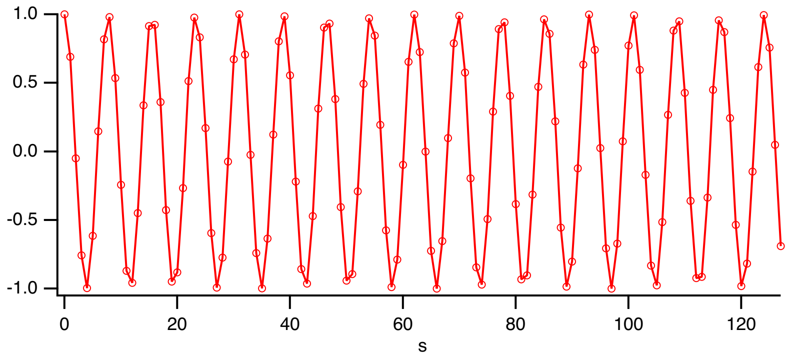
```
Ideal FFT amplitude = cosine amplitude * number of points/2 = 1 * 128 / 2 = 64
```

```
FFT /OUT=3 /DEST=cosWaveF cosWave
Display cosWaveF
ModifyGraph mode=8, marker=8
```



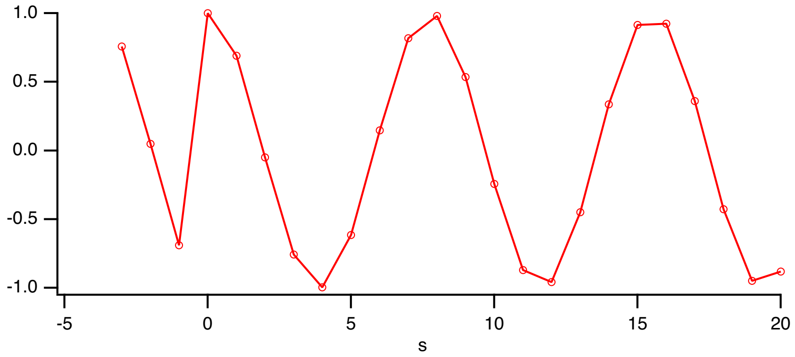
Notice that all other FFT magnitudes are zero. Now let us change the data so that there are 16.5 cosine cycles:

```
Make/O/N=128 cosWave = cos(2*pi*p*16.5/128)
SetScale/P x, 0, 1, "s", cosWave
SetAxis/A
```



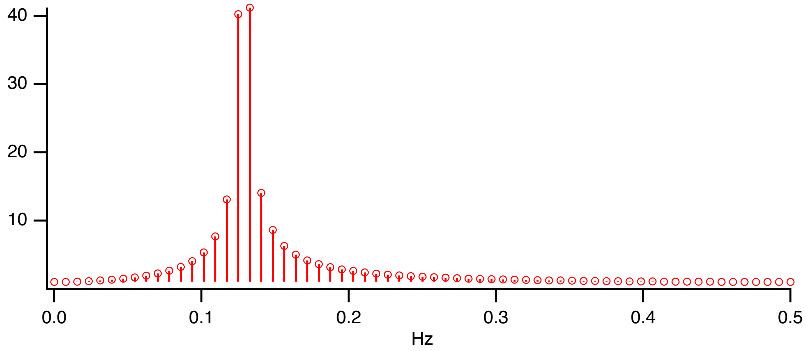
When we rotate this data as before, you can see what the FFT will perceive to be a discontinuity between the point 127 and point 0 of the unrotated data. In this next graph, the original point 127 has been rotated to x= -1 and point 0 is still at x=0.

```
Rotate 3, cosWave
SetAxis bottom, -5, 20
```



When the FFT of this data is computed, the discontinuity causes “leakage” of the main cosine peak into surrounding magnitude values.

```
FFT /OUT=3 /DEST=cosWaveF cosWave
SetAxis/A
```

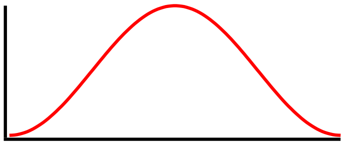


How does all this relate to spectral windowing? Spectral windowing reduces this leakage and gives more accurate FFT results. Specifically, windowing reduces the number of adjacent FFT values affected by leakage. A typical window accomplishes this by smoothly attenuating both ends of the data towards zero.

Hanning Window

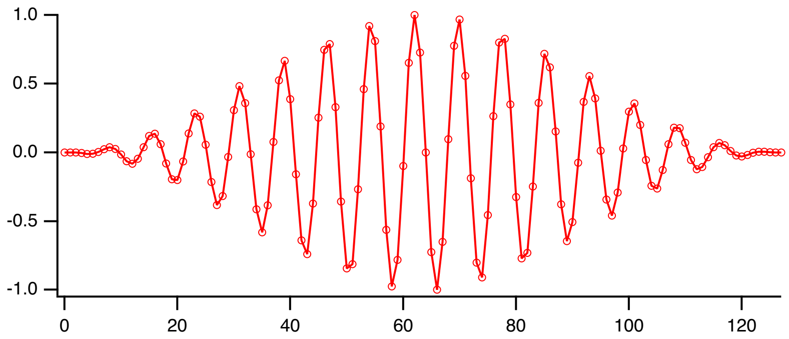
Windowing the data *before* the FFT is computed can reduce the leakage demonstrated above. The Hanning window is a simple raised cosine function defined by the equation:

$$hanning[p] = \frac{1 - \cos\left(\frac{2\pi p}{N-1}\right)}{2}$$



Let us apply the Hanning window to the 16.5 cycle cosine wave data:

```
Make/O/N=128 cosWave=cos(2*pi*p*16.5/128)
Hanning cosWave
Display cosWave
ModifyGraph mode=4, marker=8
```



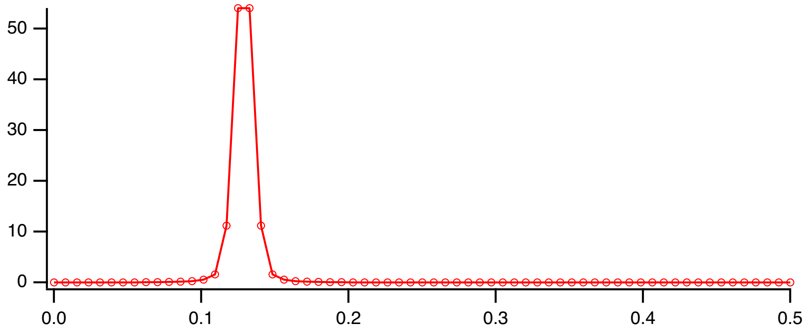
By smoothing the ends of the wave to zero, there is no discontinuity when wrapping around the ends.

In applying a window to the data, energy is lost. Depending on your application you may want to scale the output to account for coherent or incoherent gain. The coherent gain is sometimes expressed in terms of amplitude factor and it is equal to the sum of the coefficients of the window function over the interval. The incoherent gain is a power factor defined as the sum of the squares of the same coefficients. In the case that we are considering the correction factor is just the reciprocal of the coherent gain of the Hanning window

$$coherent\ gain \equiv \int_0^1 \frac{1-\cos(2\pi x / N)}{2} dx = 0.5$$

so we can multiply the FFT amplitudes by 2 to correct for them:

```
cosWave *= 2                              // Account for coherent gain
FFT /OUT=3 /DEST=cosWaveH cosWave
Display cosWaveH
ModifyGraph mode=4, marker=8
```



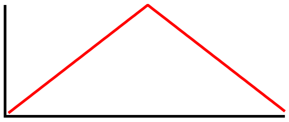
Note that frequency values in the neighborhood of the peak are less affected by the leakage, and that the amplitude is closer to the ideal of 64.

Other Windows

The Hanning window is not the ultimate window. Other windows that suppress more leakage tend to broaden the peaks. The FFT and WindowFunction operations have the following built-in windows: Hanning, Hamming, Bartlett, Blackman, Cosa(x), KaiserBessel, Parzen, Riemann, and Poisson.

You can create other windows by writing a user-defined function or by executing a simple wave assignment statement such as this one which applies a triangle window:

```
data *= 1-abs(2*p/numpnts(data)-1)
```



Use point indexing to avoid X scaling complications. You can determine the effect a window has by applying it to a perfect cosine wave, preferably a cosine wave at 1/4 of the sampling frequency (half the Nyquist frequency).

Other windows are provided in the WaveMetrics-supplied “DSP Window Functions” procedure file.

Multidimensional Windowing

When performing FFTs on images, artifacts are often produced because of the sharp boundaries of the image. As is the case for 1D waves, windowing of the image can help yield better results from the FFT.

To window images, you will need to use the ImageWindow operation, which implements the Hanning, Hamming, Bartlett, Blackman, and Kaiser windowing filters. See the **ImageWindow** operation on page V-435 for further details. For a windowing example, see **Correlations** on page III-362.

Power Spectra

Periodogram

The periodogram of a signal s(t) is an estimate of the power spectrum given by

$$P(f) = \frac{|F(f)|^2}{N},$$

where F(f) is the Fourier transform of s(t) computed by a Discrete Fourier Transform (DFT) and N is the normalization (usually the number of data points).

You can compute the periodogram using the FFT but it is easier to use the DSPPeriodogram operation, which has the same built-in window functions but you can also select your own normalization to suppress the DC term or to have the results expressed in dB as:

20log10(F/F0)

or

10log10(P/P0)

where P0 is either the maximum value of P or a user-specified reference value.

DSPPeriodogram can also compute the cross-power spectrum, which is the product of the Fourier transform of the first signal with the complex conjugate of the Fourier transform of the second signal:

$$P(f) = \frac{F(f)G^*(f)}{N}$$

where F(f) and G(f) are the DFTs of the two waves.

Power Spectral Density Functions

The PowerSpectralDensity routine supplied in the “Power Spectral Density” procedure file computes Power Spectral Density by averaging power spectra of segments of the input data. This is an early procedure file that does not take advantage of the new built-in features of the FFT or DSPPeriodogram operations. The procedure is still supported for backwards compatibility.

The PowerSpectralDensity functions take a long data wave on input and calculate the power spectral density function. These procedures have the following features:

- Automatic display of the results.
- Original data is untouched.
- Pop-up list of windowing functions.
- User setable segment length.

Use `#include <Power Spectral Density>` in your procedure file to access these functions. See **The Include Statement** on page IV-166 for instructions on including a procedure file.

PSD Demo Experiment

The PSD Demo experiment (in the Examples:Analysis: folder) uses the PowerSpectralDensity procedure and explains how it works in great detail, including justification for the scaling applied to the result.

Hilbert Transform

The Hilbert transform of a function f(x) is defined by

$$F_H(x) = \frac{1}{\pi}\int_{-\infty}^{\infty}\frac{f(t)}{t-x}dt .$$

The integral is evaluated as a Cauchy principal value. For numerical computation it is customary to express the integral as the convolution

$$F_H(x) = \left(\frac{-1}{\pi x}\right) \otimes f(x) .$$

Noting that the Fourier transform of (-1/πx) is i*sgn(x), we can evaluate the Hilbert transform using the convolution theorem of Fourier transforms. The **HilbertTransform** operation (see page V-348) is a convenient shortcut. In the next example we compute the Hilbert transform of a cosine function that gives us a sine function:

```
Make/N=512 cosWave=cos(2*pi*x*20/512)
HilbertTransform/Dest=hCosWave cosWave
Display cosWave,hCosWave
ModifyGraph rgb(hCosWave)=(0,0,65535)
```

Time Frequency Analysis

When you compute the Fourier spectrum of a signal you dispose of all the phase information contained in the Fourier transform. You can find out which frequencies a signal contains but you do not know when these frequencies appear in the signal. For example, consider the signal

$$f(t) = \begin{cases} \sin(2\pi f_1 t) & 0 \le t < t_1 \\ \sin(2\pi f_2 t) & t_1 \le t < t_2 \end{cases} .$$

The spectral representation of f(t) remains essentially unchanged if we interchange the two frequencies f_1 and f_2. In other words, the Fourier spectrum is not the best analysis tool for signals whose spectra fluctuate in time. One solution to this problem is the so-called “short time Fourier Transform”, in which you can compute the Fourier spectra using a sliding temporal window. By adjusting the width of the window you can determine the time resolution of the resulting spectra.

Two alternative tools are the Wigner transform and the Continuous Wavelet Transform (CWT).

Wigner Transform

The Wigner transform (also known as the Wigner Distribution Function or WDF) maps a 1D time signal U(*t*) into a 2D time-frequency representation. Conceptually, the WDF is analogous to a musical score where the time axis is horizontal and the frequencies (notes) are plotted on a vertical axis. The WDF is defined by the equation

$$W(t, \nu) = \int_{-\infty}^{\infty} dx U(t+x/2)U^*(t-x/2)e^{-i2\pi x\nu}$$

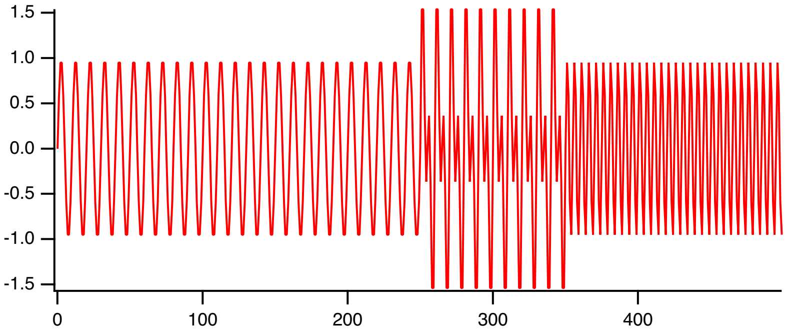
Note that the WDF W(*t*,ν) is real (this can be seen from the fact that it is a Fourier transform of an Hermitian quantity). The WDF is also a 2D Fourier transform of the Ambiguity function.

The localized spectrum can be derived from the WDF by integrating it over a finite area dtdn. Using Gaussian weight functions in both *t* and *n*, and choosing the minimum uncertainty condition dtdn=1, we obtain an estimate for the local spectrum

$$\hat{W}(t, \nu;\delta t) \propto \left|\int U(t')\exp\left[-2\pi\left(\frac{t-t'}{\delta t}\right)^2\right]\exp(-i2\pi\nu t')dt'\right|^2$$

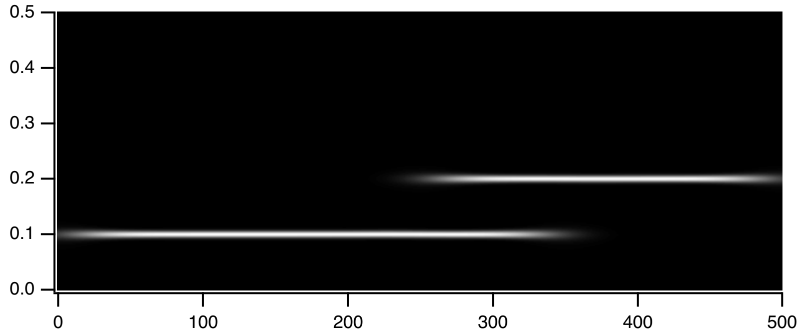
For an application of the **WignerTransform** operation (see page V-1095), consider the two-frequency signal:

```
Make/N=500 signal
signal[0,350]=sin(2*pi*x*50/500)
signal[250,]+=sin(2*pi*x*100/500)
WignerTransform /GAUS=100 signal
DSPPeriodogram signal          // Spectrum for comparison
Display signal
```

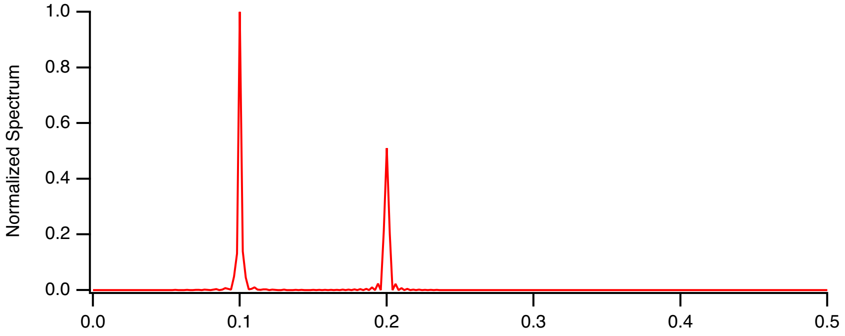


The signal used in this example consists of two “pure” frequencies that have small amount of temporal overlap:

```
Display; AppendImage M_Wigner
```



The temporal dependence is clearly seen in the Wigner transform. Note that the horizontal (time) transitions are not sharp. This is mostly due to the application of the minimum uncertainty relation dtdn=1 but it is also due to computational edge effects. By comparison, the spectrum of the signal while clearly showing the presence of two frequencies it provides no indication of the temporal variation of the signal's frequency content. Furthermore, the different power in the two frequencies may be attributed to either a different duration or a different amplitude.



Continuous Wavelet Transform

The Continuous Wavelet Transform (CWT) is a time-frequency representation of signals that graphically has a superficial similarity to the Wigner transform.

A wavelet transform is a convolution of a signal s(t) with a set of functions which are generated by translations and dilations of a main function. The main function is known as the mother wavelet and the translated or dilated functions are called wavelets. Mathematically, the CWT is given by

$$W(a, b) = \frac{1}{\sqrt{a}}\int s(t)\psi\left(\frac{t-b}{a}\right)dt .$$

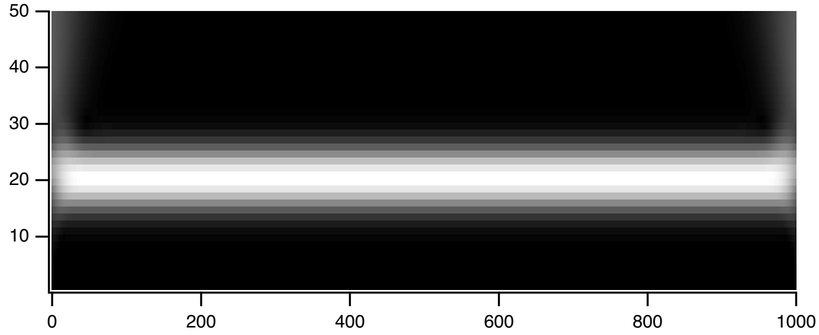
Here b is the time translation and a is the dilation of the wavelet.

From a computational point of view it is natural to use the FFT to compute the convolution which suggests that the results are dependent on proper sampling of s(t).

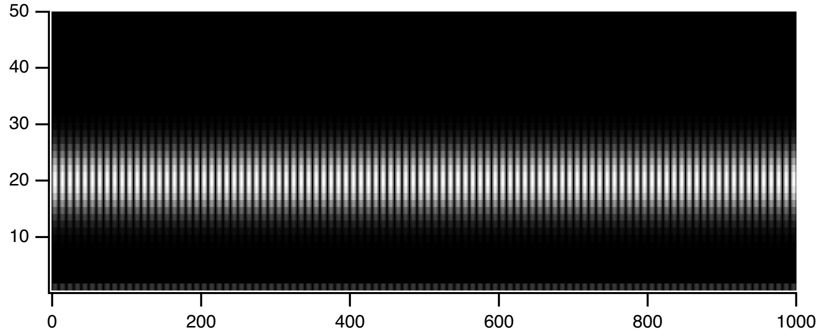
When the mother wavelet is complex, the CWT is also a complex valued function. Otherwise the CWT is real. The squared magnitude of the CWT $|W(a, b)|^2$ is equivalent to the power spectrum so that a typical display (image) of the CWT is a representation of the power spectrum as a function of time offset b. One should note however that the precise form of the CWT depends on the choice of mother wavelet y and therefore the extent of the equivalency between the squared magnitude of the CWT and the power spectrum is application dependent.

The **CWT** operation (see page V-137) is implemented using both the FFT and the direct sum approach. You can use either one to get a representation of the effective wavelet using a delta function as an input. When comparing two CWT results you should always check that both use exactly the same definition of the wavelet function, same normalization and same computation method. For example,

```
Make/N=1000 signal=sin(2*pi*x*50/1000)
CWT/OUT=4/SMP2=1/R2={1,1,40}/WBI1=Morlet/WPR1=5/FSCL signal
Rename M_CWT, M_CWT1
Display as "Morlet FFT"; AppendImage M_CWT1
```

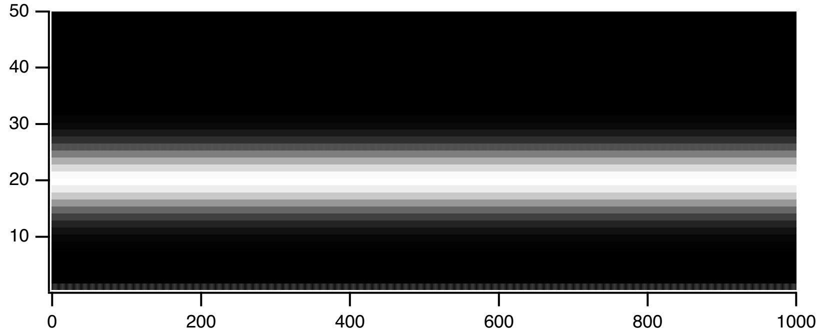


```
CWT /M=1/OUT=4/SMP2=1/R2={1,1,40}/WBI1=Morlet/FSCL /ENDM=2 signal
Rename M_CWT, M_CWT2
Display as "Morlet Direct Sum"; AppendImage M_CWT2
```



Using the complex Morlet wavelet in the direct sum method (/M=1) and displaying the squared magnitude we get:

```
CWT /M=1/OUT=4/SMP2=1/R2={1,1,40}/WBI1=MorletC/FSCL /ENDM=2 signal
Rename M_CWT, M_CWT3
Display as "Complex Morlet Direct Sum"; AppendImage M_CWT3
```



It is apparent that the last image has essentially the same results as the one generated using the FFT approach but in this case the edge effects are completely absent.

Discrete Wavelet Transform

The DWT is similar to the Fourier transform in that it is a decomposition of a signal in terms of a basis set of functions. In Fourier transforms the basis set consists of sines and cosines and the expansion has a single parameter. In wavelet transform the expansion has two parameters and the functions (wavelets) are generated from a single “mother” wavelet using dilation and offsets corresponding to the two parameters.

$$f(t) = \sum_a \sum_b c_{ab} \psi_{ab}(t),$$

where the two-parameter expansion coefficients are given by

$$c_{ab} = \int f(t) \psi_{ab}(t) dt$$

and the wavelets obey the condition

$$\psi_{ab}(t) = 2^{\frac{a}{2}} \Psi(2^a t - b).$$

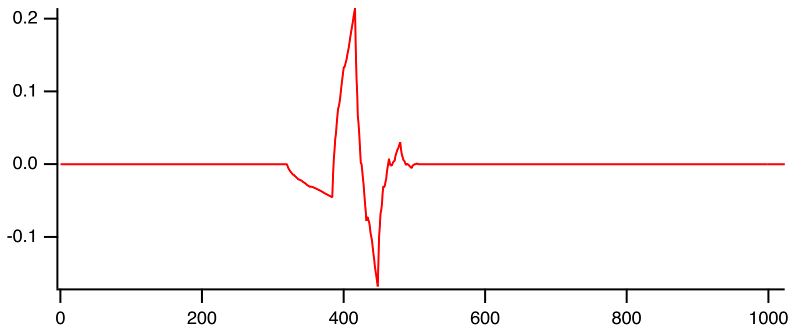
Here Ψ is the mother wavelet, *a* is the dilation parameter and *b* is the offset parameter.

The two parameter representation can complicate things quickly as one goes from 1D signal to higher dimensions. In addition, because the number of coefficients in each scale varies as a power of 2, the DWT of a 1D signal is not conveniently represented as a 2D image (as is the case with the CWT). It is therefore customary to “pack” the results of the transform so that they have the same dimensionality of the input. For example, if the input is a 1D wave of 128 (=27) points, there are 7-1=6 significant scales arranged as follows:

Scale	Storage Location
1	64-127
2	32-63
3	16-31
4	8-15
5	4-7
6	2-3

An interesting consequence of the definition of the DWT is that you can find out the shape of the wavelet by transforming a suitable form of a delta function. For example:

```
Make/N=1024 delta=0
delta[22]=1
DWT/I delta
Display W_DWT     // Daubechies 4 coefficient wavelet
```



Convolution

You can use convolution to compute the response of a linear system to an input signal. The linear system is defined by its impulse response. The convolution of the input signal and the impulse response is the output signal response. Convolution is also the time-domain equivalent of filtering in the frequency domain.

Smoothing is also a form of convolution – see **Smoothing** on page III-292.

The **FilterFIR** implements convolution in the time domain – see **Digital Filtering** on page III-299.

Igor implements general convolution with the **Convolve** operation. To use the Convolve operation, choose Analysis→Convolve.

The built-in Convolve operation computes the convolution of two waves named "source" and "destination" and overwrites the destination wave with the results. The operation can also convolve a single source wave with multiple destination waves (overwriting the corresponding destination wave with the results in each case). The Convolve dialog allows for more flexibility by preduplicating the second waves into new destination waves.

If the source wave is real-valued, each destination wave must be real-valued and if source wave is complex, each destination wave must be complex, too. Double and single precision waves may be freely intermixed; the calculations are performed in the higher precision.

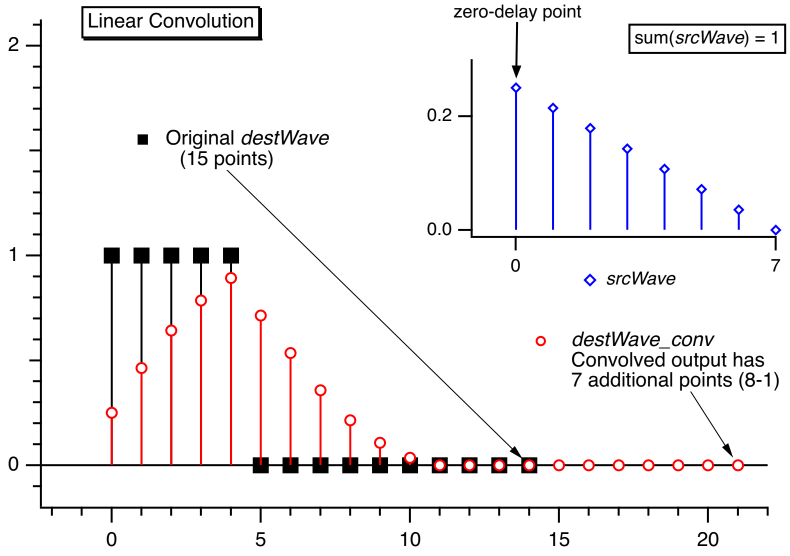
Convolve combines neighboring points before and after the point being convolved, and at the ends of the waves not enough neighboring points exist. This is a general problem in any convolution operation; the smoothing operations use the End Effect pop-up to determine what to do. The Convolve dialog presents three algorithms in the Algorithm group to deal with these missing points.

The Linear algorithm is similar to the Smooth operation's Zero end effect method; zeros are substituted for the values of missing neighboring points.

The Circular algorithm is similar to the Wrap end effect method; this algorithm is appropriate for data which is assumed to endlessly repeat.

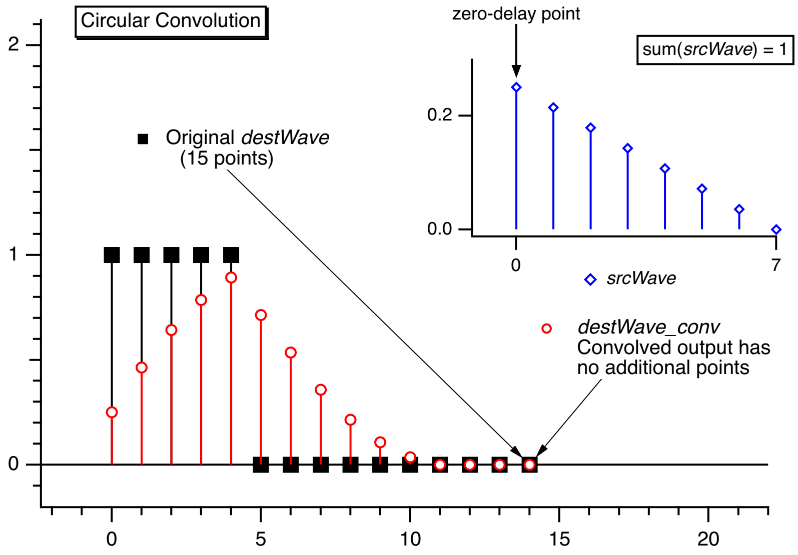
The acausal algorithm is a special case of Linear which eliminates the time delay that Linear introduces.

Depending on the algorithm chosen, the number of points in the destination waves may increase by the number of points in the source wave, less one. For linear and acausal convolution, the destination wave is first zero-padded by one less than the number of points in the source wave. This prevents the "wrap-around" effect that occurs in circular convolution. The zero-padded points are removed after acausal convolution, and retained after linear convolution.

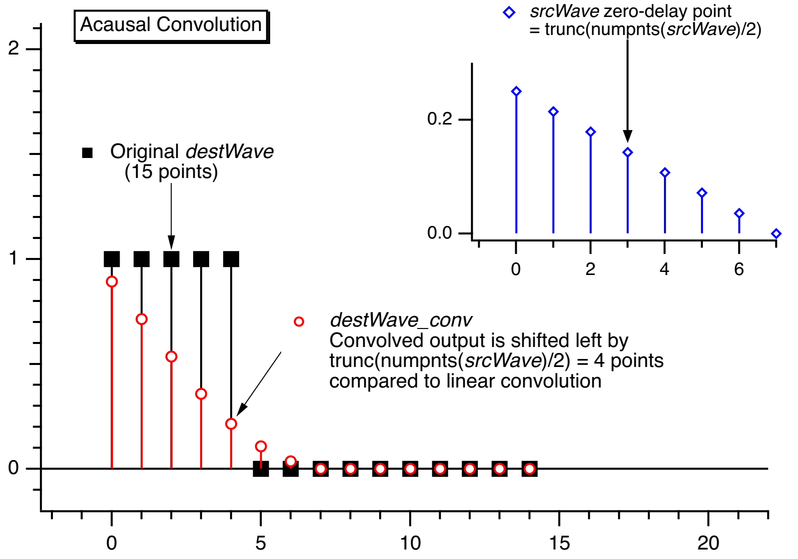


Use linear convolution when the source wave contains an impulse response (or filter coefficients) where the first point of *srcWave* corresponds to no delay ($t = 0$).

Use Circular convolution for the case where the data in the source wave and the destination waves are considered to endlessly repeat (or "wrap around" from the end back to the start), which means no zero padding is needed.

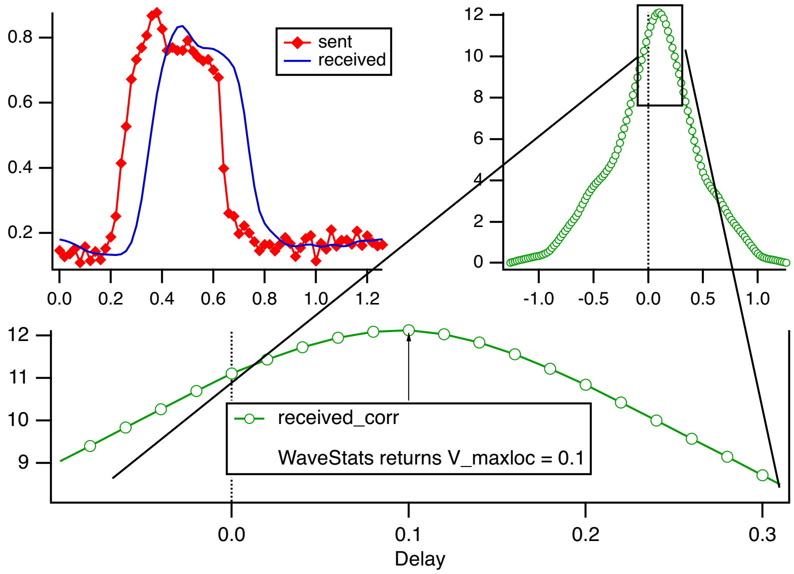


Use acausal convolution when the source wave contains an impulse response where the middle point of the source wave corresponds to no delay (*t* = 0).



Correlation

You can use correlation to compare the similarity of two sets of data. Correlation computes a measure of similarity of two input signals as they are shifted by one another. The correlation result reaches a maximum at the time when the two signals match best. If the two signals are identical, this maximum is reached at *t* = 0 (no delay). If the two signals have similar shapes but one is delayed in time and possibly has noise added to it then correlation is a good method to measure that delay.



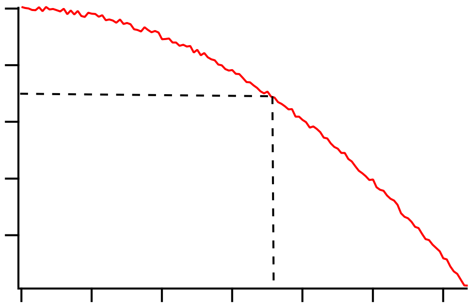
Igor implements correlation with the **Correlate** operation (see page V-107). The Correlate dialog in the Analysis menu works similarly to the Convolve dialog. The source wave may also be a destination wave, in which case afterward it will contain the “auto-correlation” of the wave. If the source and destination are different, this is called “cross-correlation”.

The same considerations about combining differing types of source and destination waves applies to correlation as to convolution. Correlation must also deal with end effects, and these are dealt with by the circular and linear correlation algorithm selections. See **Convolution** on page III-284.

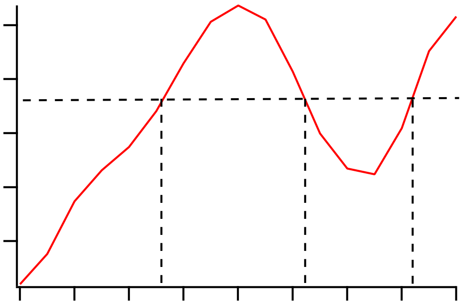
Level Detection

Level detection is the process of locating the X coordinate at which your data passes through or reaches a given Y value. This is sometimes called “inverse interpolation”. Stated another way, level detection answers the question: “given a Y level, what is the corresponding X value?” Igor provides two kinds of answers to that question.

One answer assumes your Y data is a list of unique Y values that increases or decreases monotonically. In this case there is only one X value that corresponds to a Y value. Since search position and direction don’t matter, a binary search is most appropriate. For this kind of data, use the BinarySearch or BinarySearchInterp functions.



The other answer assumes that your Y data varies irregularly, as it would with acquired data. In this case, there may be multiple X values that cross the Y level; the X value returned depends on where the search starts and the search direction through the data. The FindLevel, FindLevels, EdgeStats, and PulseStats operations deal with this kind of data.



A related, but different question is “given a function y = f(x), find x where y is zero (or some other value)”. This question is answered by the FindRoots operation. See **Finding Function Roots** on page III-338, and the **FindRoots** operation on page V-248.

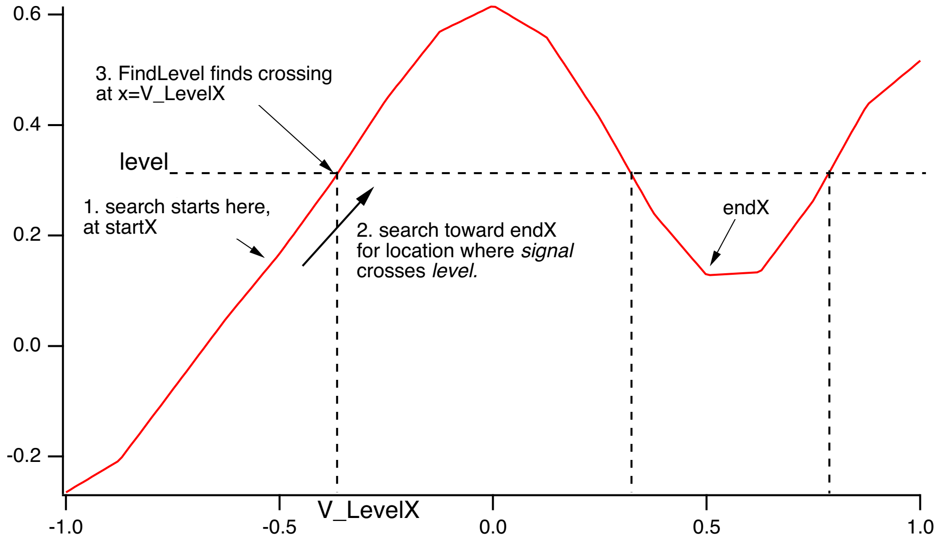
The following sections pertain to detecting level crossings in data that varies irregularly. The operations discussed are not designed to detect peaks; see **Peak Measurement** on page III-290.

Finding a Level in Waveform Data

You can use the **FindLevel** operation (see page V-242) to find a single level crossing, or the **FindLevels** operation (see page V-244) to find multiple level crossings in waveform data. Both of these operations can optionally smooth the waves they search to reduce the effects of noise. A subrange of the data can be searched, by either ascending or descending X values, depending on the *startX* and *endX* values you supply to the operation’s /R flag.

FindLevel locates the first level crossing encountered in the search range, starting at *startX* and proceeding toward *endX* until a level crossing is found. The search is performed sequentially. The outputs of FindLevel are two special numeric variables: V_Flag and V_LevelX. V_Flag indicates the success or failure of the search (0 is success), and V_LevelX contains the X coordinate of the level crossing.

For example, given the following data:



the command:

```
FindLevel/R=(-0.5,0.5) signal,0.30
```

prints this level crossing information into the history area:

```
V_Flag=0; V_LevelX=-0.37497; V_rising=1;
```

Finding a Level in XY Data

You can find a level crossing in XY data by searching the Y wave and then figuring out where in the X wave that X value can be found. This requires that the values in the X wave be sorted in ascending or descending order. To ensure this, the command:

```
Sort xWave,xWave,yWave
```

sorts the waves so that the values in xWave are ascending, and the XY correspondence is preserved.

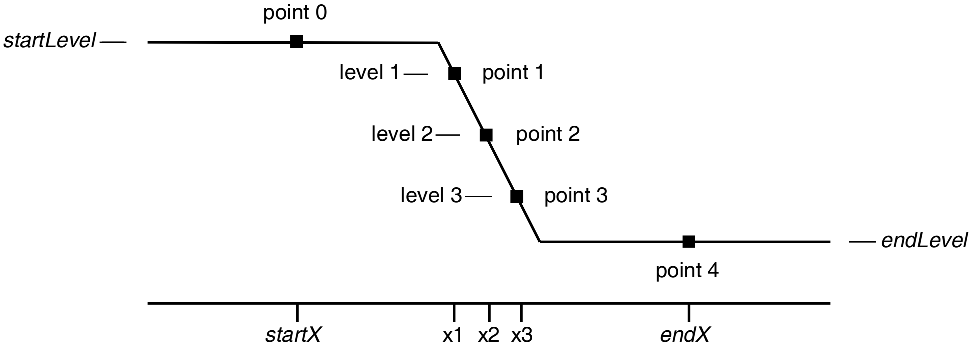
The following procedure finds the X location where a Y level is crossed within an X range, and stores the result in the output variable V_LevelX:

```
Function FindLevelXY()

   String swy,swx                // strings contain the NAMES of waves
   Variable startX=-inf,endX=inf // startX,endX correspond to VALUEs in wx, not any X
scaling
   Variable level
   // Put up a dialog to get info from user
   Prompt swy,"Y Wave",popup WaveList("*",";","")
   Prompt swx,"X Wave",popup WaveList("*",";","")
   Prompt startX, "starting X value"
   Prompt endX, "ending X value"
   Prompt level, "level to find"
   DoPrompt "Find Level XY", swy,swx,startX, endX, level

   WAVE wx = $swx
   WAVE wy = $swy

   // Here's where the interesting stuff begins
   Variable startP,endP                  //compute point range covering startX,endX
   startP=BinarySearch(wx,startX)
   endP=BinarySearch(wx,endX)
   FindLevel/Q/R=[startP,endP] wy,level          // search Y wave, assume success
   Variable p1,m
   p1=x2pnt(wy,V_LevelX-deltaX(wy)/2)            //x2pnt rounds; circumvent it
   // Linearly interpolate between two points in wx
   // that bracket V_levelX in wy
   m=(V_LevelX-pnt2x(wy,p1))/(pnt2x(wy,p1+1)-pnt2x(wy,p1))  // slope
   V_LevelX=wx[p1] + m * (wx[p1+1] -wx[p1] )      //point-slope equation
End
```

This function does not handle a level crossing that isn’t found; all that is missing is a test of V_Flag after searching the Y wave with FindLevel.

Edge Statistics

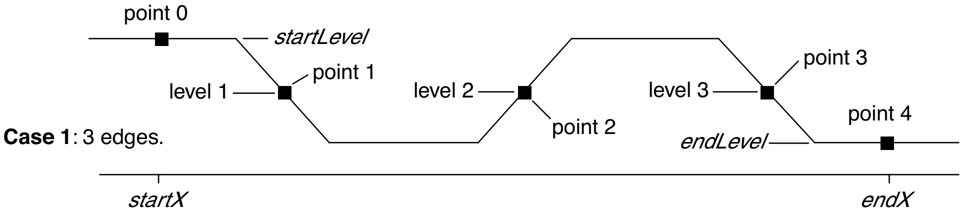
The **EdgeStats** operation (see page V-190) produces simple statistics (measurements, really) on a region of a wave that is expected to contain a single edge as shown below. If more than one edge exists, EdgeStats works on the first edge it finds. The edge statistics are stored in special variables which are described in the EdgeStats reference. The statistics are edge levels, X or point positions of various found “points”, and the distances between them. These found points are actually the locations of level crossings, and are usually located between actual waveform points (they are interpolation locations).



EdgeStats is based on the same principles as FindLevel. EdgeStats does not work on an XY pair. See **Converting XY Data to a Waveform** on page III-109.

Pulse Statistics

The **PulseStats** operation (see page V-783) produces simple statistics (measurements) on a region of a wave that is expected to contain three edges as shown below. If more than three edges exist, PulseStats works on the first three edges it finds. PulseStats handles two other cases in which there are only one or two edges. The pulse statistics are stored in special variables which are described in the PulseStats reference.



PulseStats is based on the same principles as EdgeStats. PulseStats does not work on an XY pair. See **Converting XY Data to a Waveform** on page III-109.

Peak Measurement

The building block for peak measurement is the FindPeak operation. You can use it to build your own peak measurement procedures or you can use procedures provided by WaveMetrics.

Our Multipeak Fitting package provides a powerful GUI and programming interface for curve fitting to peak data. It can fit a number of peak shapes and baseline functions. A demo experiment provides an introduction - choose File→Example Experiments→Curve Fitting→Multipeak Fit Demo.

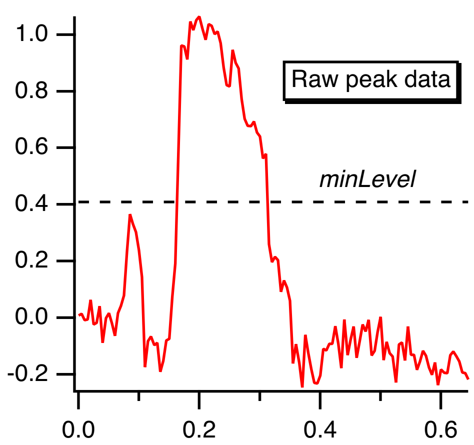
We have created several peak finding and peak fitting Technical Notes. They are described in a summary Igor Technical Note, TN020s-Choosing a Right One.ifn in the Technical Notes folder. There is also an example experiment, called Multi-peak Fit, that does fitting to multiple Gaussian, Lorentzian and Voigt peaks. Multi-peak Fit is less comprehensive but easier to use than Tech Note 20.

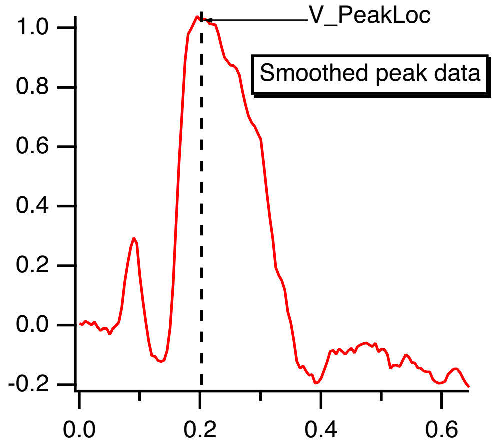
The **FindPeak** operation (see page V-247) searches a wave for a minimum or maximum by analyzing the smoothed first and second derivatives of the wave. The smoothing and differentiation is done on a copy of the input wave (so that the input wave is not modified). The peak maximum is detected at the smoothed first derivative zero-crossing, where the smoothed second derivative is negative. The position of the minimum or maximum is returned in the special variable V_PeakLoc. This and other special variables set by FindPeak are described in the operation reference.

The following describes the process that FindPeak goes through when it executes a command like this:

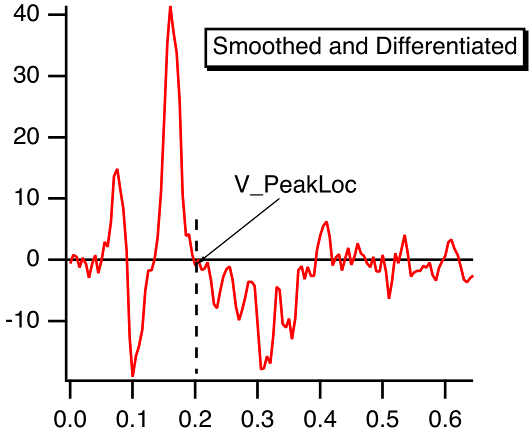
```
FindPeak/M=0.5/B=5 peakData   // 5 point smoothing, min level = 0.5
```

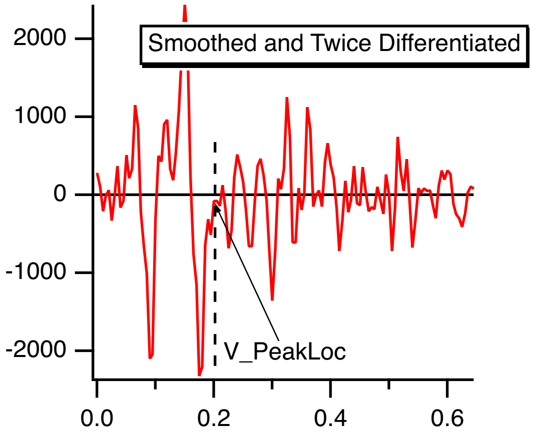
The box smoothing is performed first:





Then two central-difference differentiations are performed to find the first and second derivatives:

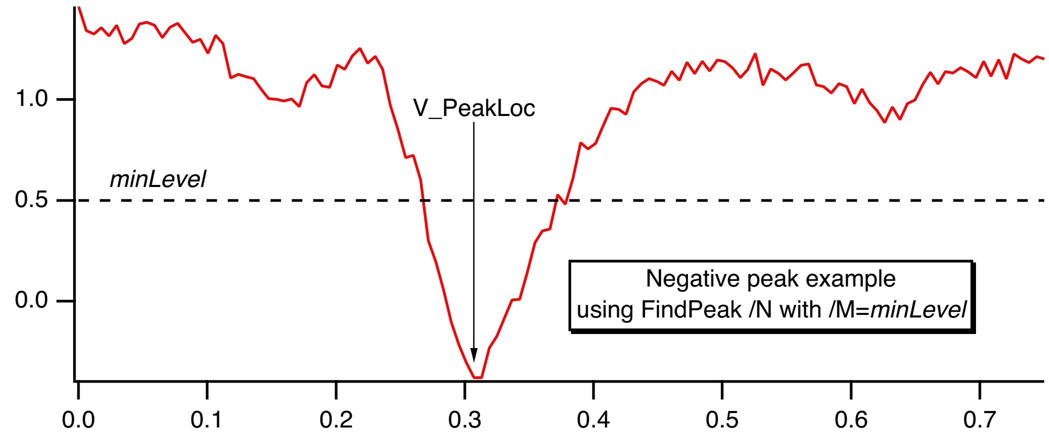




If you use the /M=*minLevel* flag, FindPeak ignores peaks that are lower than *minLevel* (i.e., the Y value of a found peak must exceed *minLevel*). The *minLevel* value is compared to the *smoothed* data, so peaks that appear to be large enough in the raw data may not be found if they are very near *minLevel*. If /N is also specified (search for minimum or "negative peak"), FindPeak ignores peaks whose amplitude is greater than *minLevel* (i.e., the Y value of a found peak will be *less* than *minLevel*). For negative peaks, the peak minimum is at the smoothed first derivative zero-crossing, where the smoothed second derivative is positive.

This command shows an example of finding a negative peak:

```
FindPeak/N/M=0.5/B=5 negPeakData    // 5 point smoothing, max level=0.5
```

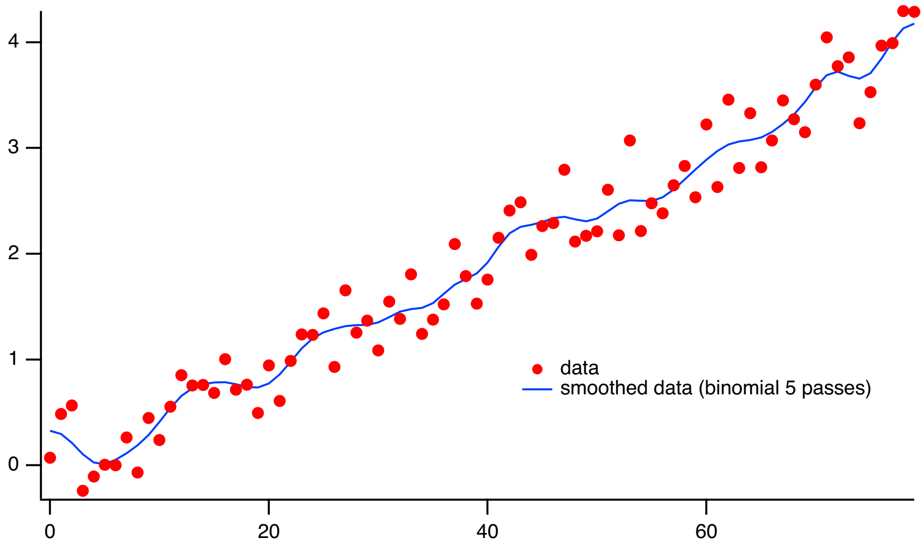


To find multiple peaks, write a procedure that calls FindPeak from within a loop. After a peak is found, restrict the range of the search with /R so that the just-found peak is excluded, and search again. Exit the loop when V_Flag indicates a peak wasn't found.

The FindPeak operation does not work on an XY pair. See **Converting XY Data to a Waveform** on page III-109.

Smoothing

Smoothing is a specialized filtering operation used to reduce the variability of data. It is sometimes used to reduce noise.



This section discusses smoothing 1-dimensional waveform data with the **Smooth**, **FilterFIR**, and **Loess** operations. Also see the **FilterIIR** and **Resample** operations.

Smoothing XY data can also be handled by the **Loess** operation and the Median.ipf procedure file (see **Median Smoothing** on page III-296).

The **MatrixFilter**, **MatrixConvolve**, and **ImageFilter** operations smooth image and 3D data.

Igor has several built-in 1D smoothing algorithms. In addition, you can supply your own smoothing coefficients.

Choose Analysis→Smooth to see the Smoothing dialog.

Depending on the smoothing algorithm chosen, there may be additional parameters to specify in the dialog.

Built-in Smoothing Algorithms

Igor has numerous built-in smoothing algorithms for 1-dimensional waveforms, and one that works with the **XY Model of Data** on page II-63:

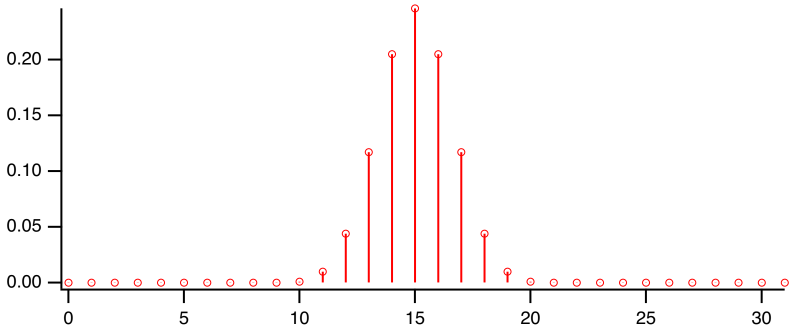
Algorithm	Operation	Data
Binomial	**Smooth**	1D waveform
Savitzky-Golay	**Smooth**/S	1D waveform
Box (Average)	**Smooth**/B	1D waveform
Custom Smoothing	**FilterFIR**	1D waveform
Median	**Smooth**/M	1D waveform
Percentile, Min, Max	**Smooth**/M/MPCT	1D waveform
Loess	**Loess**	1D waveform, XY 1D waves, false-color images*, matrix surfaces*, and multivariate data*.

* The Loess operation supports these data formats, but the Smooth dialog does not provide an interface to select them.

The first four algorithms precompute or apply one set of smoothing coefficients according to the smoothing parameters, and then replaces each data wave with the convolution of the wave with the coefficients.

You can determine what coefficients have been computed by smoothing a wave containing an impulse. For instance:

```
Make/O/N=32 wave0=0; wave0[15]=1; Smooth 5,wave0    // Smooth an impulse
Display wave0; ModifyGraph mode=8,marker=8          // Observe coefficients
```



Compute the FFT of the coefficients with magnitude output to view the frequency response. See **Finding Magnitude and Phase** on page III-274.

The last two algorithms (the **Smooth**/M and **Loess** operations) are not based on creating a fixed set of smoothing coefficients and convolution, so this technique is not applicable.

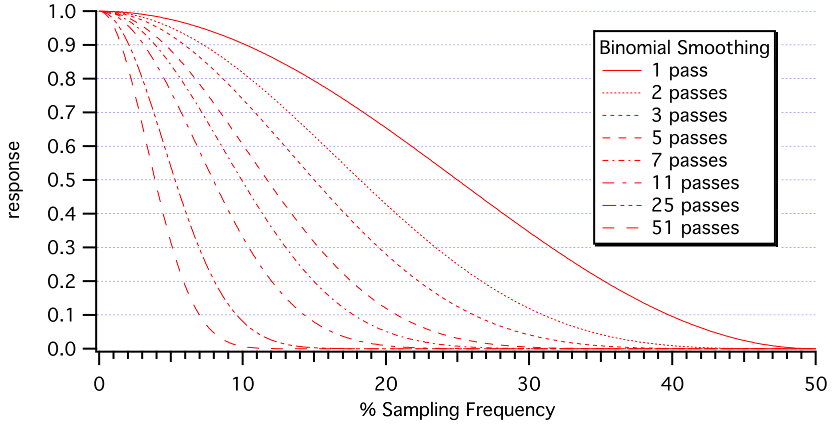
Smoothing Demo

For a demo of various smoothing techniques, choose Files→Example Experiments→Feature Demos→Smooth Curve Through Noise.

Binomial Smoothing

The Binomial smoothing operation is a Gaussian filter. It convolves your data with normalized coefficients derived from Pascal's triangle at a level equal to the Smoothing parameter. The algorithm is derived from an article by Marchand and Marmet (1983).

This graph shows the frequency response of the binomial smoothing algorithm expressed as a percentage of the sampling frequency. For example, if your data is sampled at 1000 Hz and you use 5 passes, the signal at 200 Hz (20% of the sampling frequency) will be approximately 0.1.



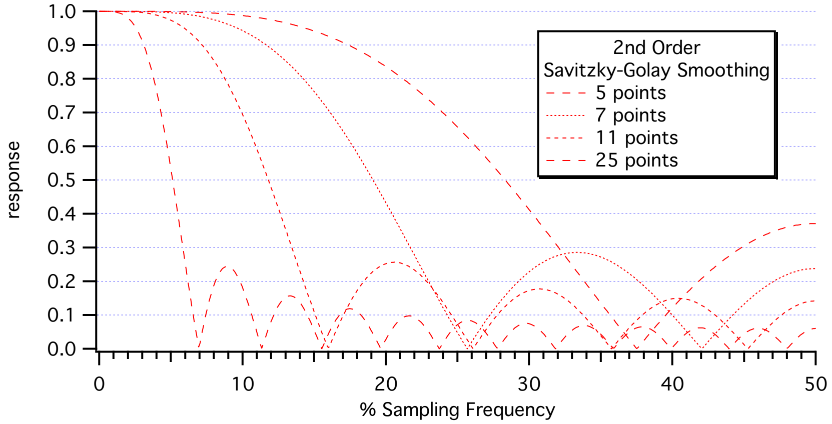
Savitzky-Golay Smoothing

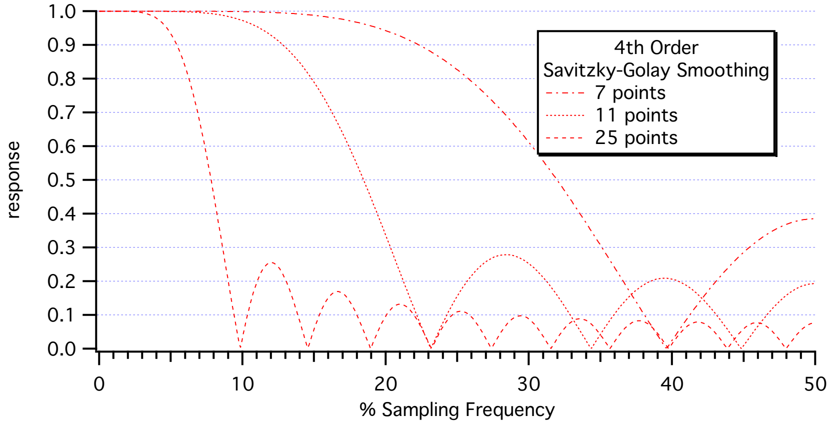
Savitzky-Golay smoothing uses a different set of precomputed coefficients popular in the field of chemistry. It is a type of Least Squares Polynomial smoothing. The amount of smoothing is controlled by two parameters: the polynomial order and the number of points used to compute each smoothed output value. This algorithm was first proposed by A. Savitzky and M.J.E. Golay in 1964. The coefficients were subsequently corrected by others in 1972 and 1978; Igor uses the corrected coefficients.

The maximum Points value is 32767; the minimum is either 5 (2nd order) or 7 (4th order). Note that 2nd and 3rd order coefficients are the same, so we list only the 2nd order choice. Similarly, 4th and 5th order coefficients are identical.

Even though Savitzky-Golay smoothing has been widely used, there are advantages to the binomial smoothing as described by Marchand and Marmet in their article.

The following graphs show the frequency response of the Savitzky-Golay algorithm for 2nd order and 4th order smoothing. The large responses in the higher frequencies show why binomial smoothing is often a better choice.

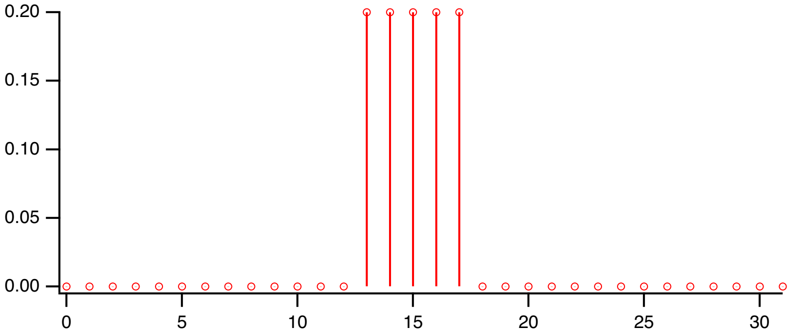




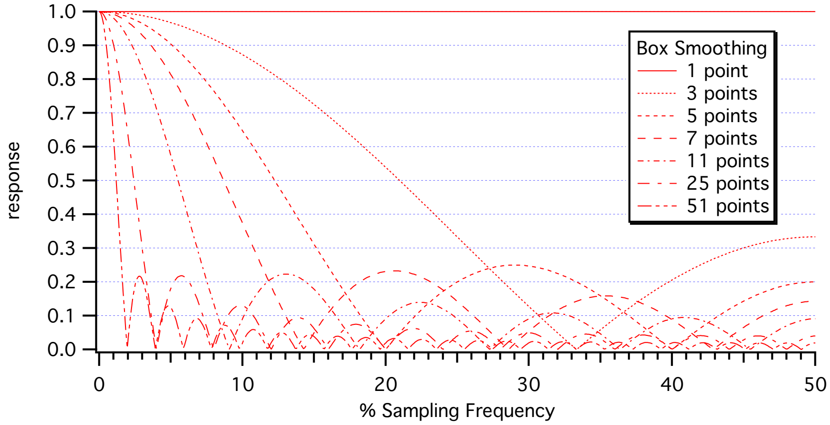
Box Smoothing

Box smoothing is similar to a moving average, except that an equal number of points before *and after* the smoothed value are averaged together with the smoothed value. The Points parameter is the total number of values averaged together. It must be an odd value, since it includes the points before, the center point, and the points after. For instance, a value of 5 averages two points before and after the center point, and the center point itself:

```
Make/O/N=32 wave0=0; wave0[15]=1; Smooth/B 5,wave0     //Smooth impulse
Display wave0; ModifyGraph mode=8,marker=8             // Observe coefficients
```



The following graph shows the frequency response of the box smoothing algorithm.



Median Smoothing

Median smoothing does not use convolution with a set of coefficients. Instead, for each point it computes the median of the values over the specified range of neighboring values centered about the point. NaN values in the waveform data are allowed and are excluded from the median calculations.

For simple XY data median smoothing, include the Median.ipf procedure file:

```
#include <Median>
```

and use the Analysis→Packages→Median XY Smoothing menu item. Currently this procedure file does not handle NaNs in the data and only implements method 1 as described below.

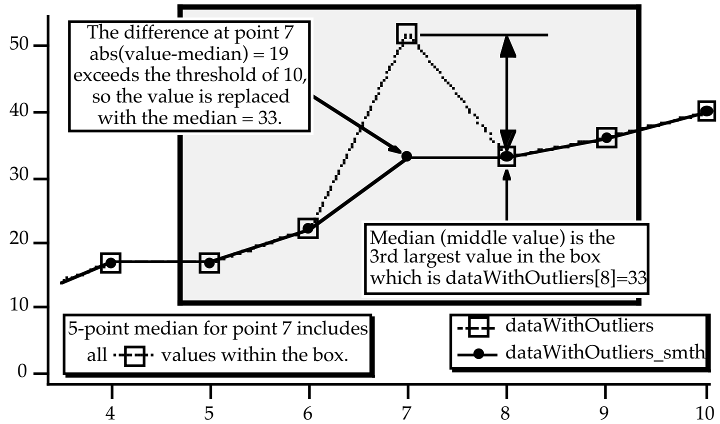
For image (2D matrix) median smoothing, use the **MatrixFilter** or **ImageFilter** operation with the median method. ImageFilter can smooth 3D matrix data.

There are several ways to use median smoothing (Smooth/M) on 1D waveform data:

1. Replace all values with the median of neighboring values.
2. Replace each value with the median if the value itself is NaN. See **Replace Missing Data Using Median Smoothing** on page III-114.
3. Replace each value with the median if the value differs from the median by a the specified threshold amount.
4. Instead of replacing the value with the computed median, replace it with a specified number, including 0, NaN, +inf, or -inf.

Median smoothing can be used to replace “outliers” in data. Outliers are data that seem “out of line” from the other data. One measure of this “out of line” is excessive deviation from the median of neighboring values. The Threshold parameter defines what is considered “excessive deviation”.

```
// Example uses integer wave to simplify checking the results
Make/O/N=20/I dataWithOutliers= 4*p+gnoise(1.5)      // simple line with noise
dataWithOutliers[7] *=2                              // make an outlier at point 7
Display dataWithOutliers
Duplicate/O dataWithOutliers,dataWithOutliers_smth
Smooth/M=10 5, dataWithOutliers_smth                 // threshold=10, 5 point median
AppendToGraph  dataWithOutliers_smth
```



The difference at point 7 abs(value-median) = 19 exceeds the threshold of 10, so the value is replaced with the median = 33.

Median (middle value) is the 3rd largest value in the box which is dataWithOutliers[8]=33

5-point median for point 7 includes all ·-□-· values within the box.

Percentile, Min, and Max Smoothing

Median smoothing is actually a specialization of Percentile smoothing, as are Min and Max.

Percentile smoothing returns the smallest value in the smoothing window that is greater than the smallest percentile % of the values:

Percentile	Type	Description
0	Min	The smoothed value is the minimum value in the smoothing window. 0 is the minimum value for percentile.
50	Median	The smoothed value is the median of the values in the smoothing window.
100	Max	The smoothed value is the maximum value in the smoothing window. 100 is the maximum value for percentile.

As an example, assume that percentile = 25, the number of points in the smoothing window is 7, and for one input point the values in the window after sorting are:

```
{0, 1, 8, 9, 10, 11, 30}
```

The 25th percentile is found by computing the rank R:

```
R = (percentile /100)*(num +1)
```

In this example, R evaluates to 2 so the second item in the sorted list, 1 in this example, is the percentile value for the input point.

The percentile algorithm uses an interpolated rank to compute the value of percentiles other than 0 and 100. See the **Smooth** operation for details.

Loess Smoothing

The Loess operation smooths data using locally-weighted regression smoothing. This algorithm is sometimes classified as a “nonparametric regression” procedure.

The regression can be constant, linear, or quadratic. A robust option that ignores outliers is available. In addition, for small data sets Loess can generate confidence intervals.

See the **Loess** operation on page V-515 help for a discussion of the basic and robust algorithms, examples, and references.

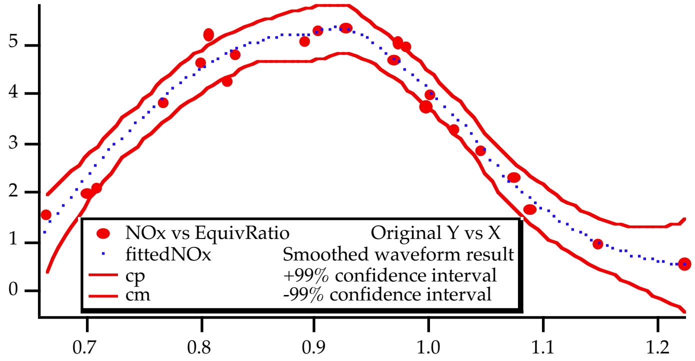
This implementation works with waveforms, XY pairs of waves, false-color images, matrix surfaces, and multivariate data (one dependent data wave with multiple independent variable data waves). Loess discards NaN input values.

The Smooth Dialog, however, provides an interface for only waveforms and XY pairs of waves (see **XY Model of Data** on page II-63), and does not provide an interface for confidence intervals or other less common options.

Here’s an example from the Loess operation help of interpolating (smoothing) an XY pair and creating an interpolated 1D waveform (Y vs. X scaling). **Note**: the Make commands below are wrapped to fit the page:

```
// 3.  1-D Y vs X wave data interpolated to waveform (Y vs X scaling)
//     with 99% confidence interval outputs (cp and cm)
// NOx = f(EquivRatio)
// Y wave
Make/O/D NOx = {4.818, 2.849, 3.275, 4.691, 4.255, 5.064, 2.118, 4.602, 2.286, 0.97,
3.965, 5.344, 3.834, 1.99, 5.199, 5.283, 3.752, 0.537, 1.64, 5.055, 4.937, 1.561};

// X wave (Note that the X wave is not sorted)
Make/O/D EquivRatio = {0.831, 1.045, 1.021, 0.97, 0.825, 0.891, 0.71, 0.801,  1.074,
1.148, 1, 0.928, 0.767, 0.701, 0.807, 0.902,   0.997, 1.224, 1.089, 0.973, 0.98, 0.665};

// Graph the input data
Display NOx vs EquivRatio; ModifyGraph mode=3,marker=19

// Interpolate to dense waveform over X range
Make/O/D/N=100 fittedNOx
WaveStats/Q EquivRatio
```

```
SetScale/I x, V_Min, V_max, "", fittedNOx
Loess/CONF={0.99,cp,cm}/DEST=fittedNOx/DFCT/SMTH=(2/3) srcWave=NOx,factors={EquivRatio}

// Display the fit (smoothed results) and confidence intervals
AppendtoGraph fittedNOx, cp,cm
ModifyGraph rgb(fittedNOx)=(0,0,65535)
ModifyGraph mode(fittedNOx)=2,lsize(fittedNOx)=2
Legend
```



Loess is memory intensive, especially when generating confidence intervals. Read the **Memory Details** section of the **Loess** operation (see page V-515) if you use confidence intervals.

Custom Smoothing Coefficients

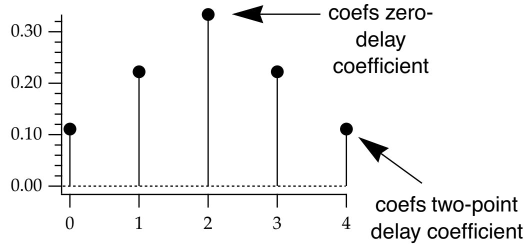
You can smooth data with your own set of smoothing coefficients by selecting the Custom Coefs algorithm. Use this option when you have low-pass filter (smoothing) coefficients created by another program or by the Igor Filter Design Laboratory.

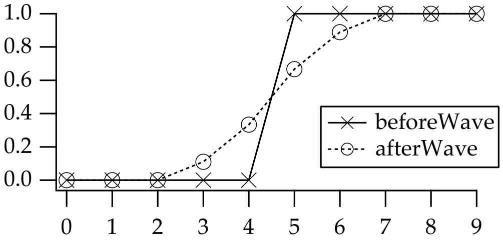
Choose the wave that contains your coefficients from the pop-up menu that appears. Igor will convolve these coefficients with the input wave using the **FilterFIR** operation (see page V-230). You should use FilterFIR when convolving a short wave with a much longer one. Use the **Convolve** operation (see page V-101) when convolving two waves with similar number of points; it's faster.

All the values in the coefficients wave are used. FilterFIR presumes that the middle point of the coefficient wave corresponds to the delay = 0 point. This is usually the case when the coefficient wave contains the two-sided impulse response of a filter, which has an odd number of points. (For a coefficient wave with an even number of points, the "middle" point is `numpnts(coefs)/2-1`, but this introduces a usually unwanted delay in the smoothed data).

In the following example, the coefs wave smooths the data by a simple 7 point Bartlett (triangle) window (omitting the first and last Bartlett window values which are 0):

```
// This example shows a unit step signal smoothed
//   by a 7-point Bartlett window
Make/O/N=10 beforeWave = (p>=5)         // unit step at p == 5
Make/O coefs={1/3,2/3,1,2/3,1/3}        // 7 point Bartlett window
WaveStats/Q coefs
coefs/= V_Sum
Duplicate/O beforeWave,afterWave
FilterFIR/E=3/COEF=coefs afterWave
Display beforeWave,afterWave
```





End Effects

The first four smoothing algorithms compute the output value for a given point using the point's neighbors. Each algorithm combines an equal number of neighboring points before and after the point being smoothed. At the start or end of a wave some points will not have enough neighbors, so a method for fabricating neighbor values must be implemented.

You choose how to fabricate those values with the End Effect pop-up menu in the Smoothing dialog. In the descriptions that follow, i is a small positive integer, and wave[n] is the last value in the wave to be smoothed.

The Bounce method uses wave[i] in place of the missing wave[-i] values and wave[n-i] in place of the missing wave[n+i] values. This works best if the data is assumed to be symmetrical about both the start and the end of the wave. If you don't specify the end effect method, Bounce is used.

The Wrap method uses wave[n-i] in place of the missing wave[-i] values and vice-versa. This works best if the wave is assumed to endlessly repeat.

The Zero method uses 0 for any missing value. This works best if the wave starts and ends with zero.

The Repeat method uses wave[0] in place of the missing wave[-i] values and wave[n] in place of the missing wave[n+i] values. This works best for data representing a single event.

When in doubt, use Repeat.

Digital Filtering

Digital filters are used to emphasize or de-emphasize frequencies present in waveforms. For example, low-pass filters preserve low frequencies and reject high frequencies.

Applying a filter to an input waveform results in a "response" output waveform.

Igor can design and apply Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) digital filters.

Other forms of digital filtering exist in Igor, signficantly the various Smoothing operations (see **Smoothing** on page III-292), which includes Savitzky-Golay, Loess, median, and moving-average smoothing.

Using the **Convolve** operation directly is another way to perform digital filtering, but that requires more knowledge than using the Filter Design and Application Dialog discussed below.

The Igor Filter Design Laboratory (IFDL) package can also be used to design and apply digital filters. IFDL is documented in the "Igor Filter Design Laboratory" help file.

Sampling Frequency and Design Frequency Bands

The XY Model of Data is not used in digital filtering. Use the Waveform Model of Data, and set the sampling frequency using **SetScale** or the Change Wave Scaling dialog.

For example, a waveform sampled at 44.1KHz (the sample rate of music on a compact disc) should have its X scaling set by a command such as:

```
SetScale/P x, 0, 1/44100, "s", musicWave
```

Typically filters are designed by specifying frequency "bands" that define a range of frequencies and the desired response amplitude (gain) and phase in that band.

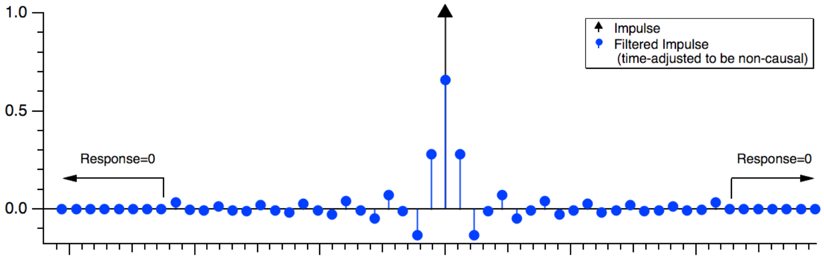
The range of frequencies that are possible range from 0 to one-half the sampling frequency of the signal, which is called the "Nyquist frequency". In the example of musicWave, the Nyquist frequency is 22,050 Hz, so filter designs for that waveform define frequency bands that end no higher than 22,050 Hz.

Filter Design Output

The result of a filter design is a set of filter "coefficients" that are used to implement the filtering. The coefficient values and format depend on the filter design type, number of bands, band frequencies, and other parameters that define the filter response. The formats for FIR and IIR designs are quite different.

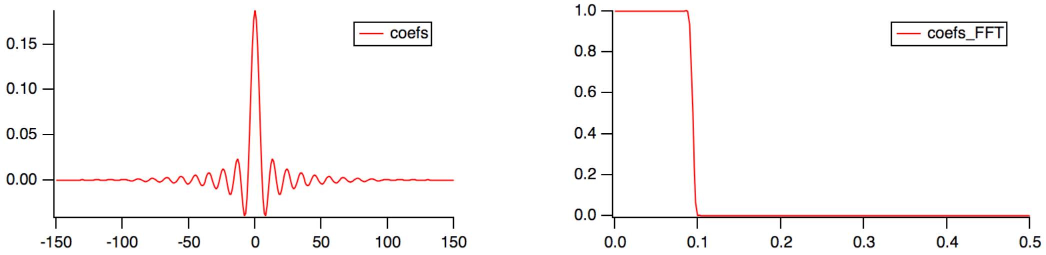
FIR Filters

Finite Impulse Response (FIR) means that the filter's time-domain response to an impulse ("spike") is zero after a finite amount of time:



An FIR filter is a finite length, evenly-spaced time series of impulses with varying amplitudes that is convolved with an input signal to produce a filtered output signal.

The impulse response amplitudes are termed "weighting factors" or "coefficients". They are identical to the filter's response to a unit impulse. You can observe an FIR filter's frequency response by simply computing the FFT of the coefficients. If you set the X scaling of the coefficients to match the sampling frequency of the data it will be applied to, the FFT result's frequency range will be scaled to the data's Nyquist frequency. For default X scaling, the frequency range will be 0 to 0.5 Hz:

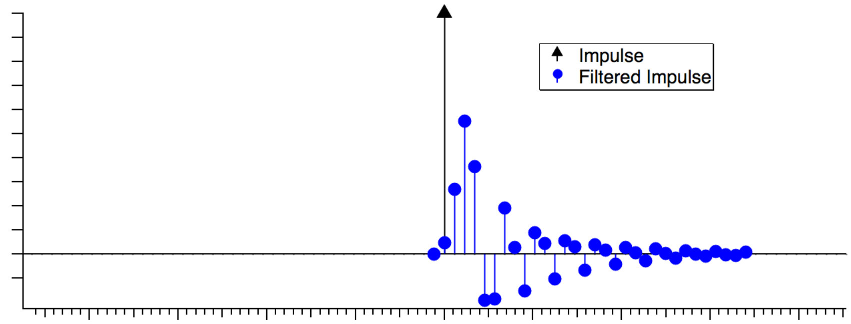


FIR filters are valued for their completely linear phase (constant delay for all frequencies), but they generally need many more coefficients than IIR filters do to achieve similar frequency responses. Consequently, electronic digital realizations of FIR filters are usually more expensive than the corresponding IIR filter.

You supply FIR coefficients to the **FilterFIR** operation along with the input waveform to compute the filtered output waveform.

IIR Filters

The response of an Infinite Impulse Response (IIR) filter continues indefinitely, as it does for analog electronic filters that employ inductors and capacitors:



An IIR filter is a set of coefficients or weights a0, a1, a2,… and b0, b1, b2… whose values and use depend on the digital implementation topology. Unlike the FIR filter, these coefficients are not the same as the filter's response to a unit impulse. See the “IIR Filter Design” topic in the “Igor Filter Design Laboratory” help file for further explanation.

IIR filters can realize quite sophisticated frequency responses with very few coefficients. The drawbacks are non-linear phase, potential for numerical instability (oscillation) when realized using limited-precision arithmetic, and the indirect design methodology (frequency transformations of conventional analog filter methods).

Igor uses two IIR implementations:

- Direct Form I (DF I)
- Cascaded Bi-Quad Direct Form II (DF II)

The IIR coefficients are represented in three forms:

- DF I
- DF II
- "zeros and poles" form

The zeros and poles form is discussed under “IIR Analog Prototype Design Graph” in the “Igor Filter Design Laboratory” help file.

You supply IIR coefficients to the FilterIIR operation along with the input waveform to compute the filtered output waveform. The format of IIR design coefficients depends on the implementation, as you can see in tables showing coefficients for Direct Form 1, Cascaded Bi-Quad Direct Form II, and pole-zero implementations of the same filter design.

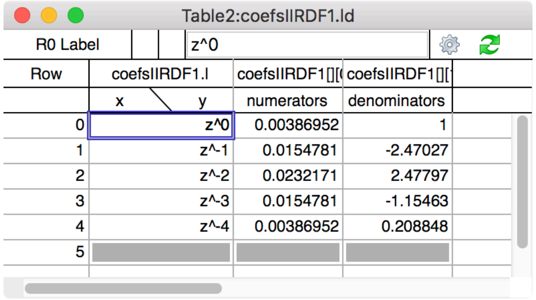


Table2:coefsIIRDF1.ld

R0 Label | z^0

Row	coefsIIRDF1.l x y	coefsIIRDF1[][(numerators	coefsIIRDF1[][denominators
0	z^0	0.00386952	1
1	z^-1	0.0154781	-2.47027
2	z^-2	0.0232171	2.47797
3	z^-3	0.0154781	-1.15463
4	z^-4	0.00386952	0.208848
5			

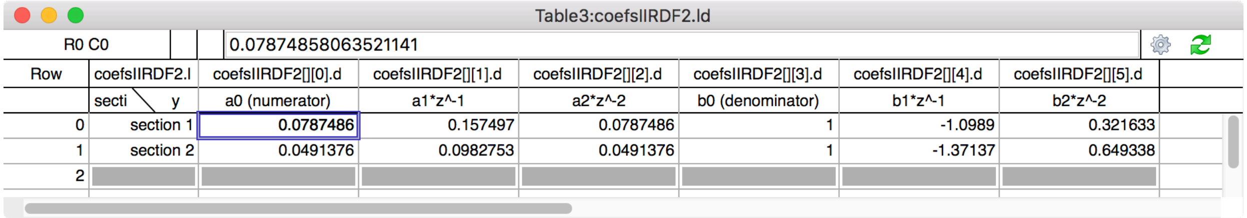


Table3:coefsIIRDF2.ld

R0 C0 | 0.07874858063521141

Row	coefsIIRDF2.l secti y	coefsIIRDF2[][0].d a0 (numerator)	coefsIIRDF2[][1].d a1*z^-1	coefsIIRDF2[][2].d a2*z^-2	coefsIIRDF2[][3].d b0 (denominator)	coefsIIRDF2[][4].d b1*z^-1	coefsIIRDF2[][5].d b2*z^-2
0	section 1	0.0787486	0.157497	0.0787486	1	-1.0989	0.321633
1	section 2	0.0491376	0.0982753	0.0491376	1	-1.37137	0.649338
2							

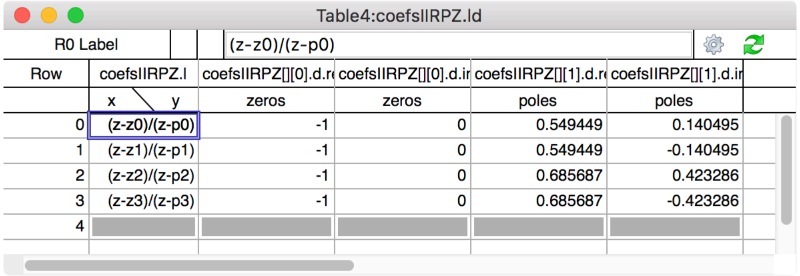


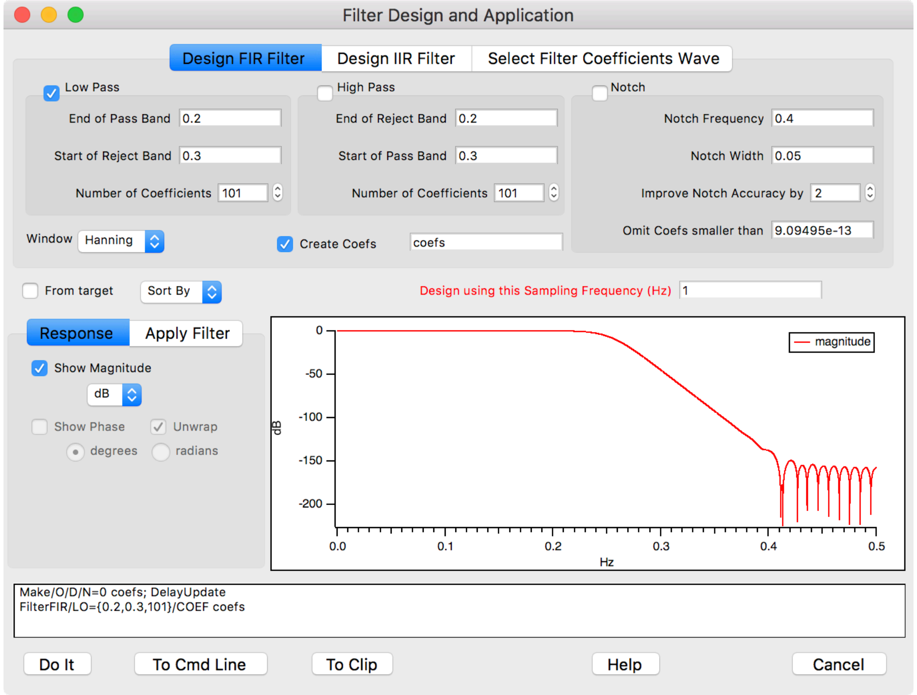
Table4:coefsIIRPZ.ld

R0 Label (z-z0)/(z-p0)

Row	coefsIIRPZ.l	coefsIIRPZ[][0].d.r	coefsIIRPZ[][0].d.i	coefsIIRPZ[][1].d.r	coefsIIRPZ[][1].d.i
	x y	zeros	zeros	poles	poles
0	(z-z0)/(z-p0)	-1	0	0.549449	0.140495
1	(z-z1)/(z-p1)	-1	0	0.549449	-0.140495
2	(z-z2)/(z-p2)	-1	0	0.685687	0.423286
3	(z-z3)/(z-p3)	-1	0	0.685687	-0.423286
4					

Filter Design and Application Dialog

The Filter Design and Application dialog provides a simple user-interface for designing and applying a digital filter. Choose Analysis→Filter to display it:



Filter Design and Application

☑ Low Pass
End of Pass Band 0.2
Start of Reject Band 0.3
Number of Coefficients 101

□ High Pass
End of Reject Band 0.2
Start of Pass Band 0.3
Number of Coefficients 101

□ Notch
Notch Frequency 0.4
Notch Width 0.05
Improve Notch Accuracy by 2
Omit Coefs smaller than 9.09495e-13

Window Hanning

☑ Create Coefs coefs

□ From target Sort By

Design using this Sampling Frequency (Hz) 1

☑ Show Magnitude dB
□ Show Phase ☑ Unwrap
● degrees ○ radians

```
Make/O/D/N=0 coefs; DelayUpdate
FilterFIR/LO={0.2,0.3,101}/COEF coefs
```

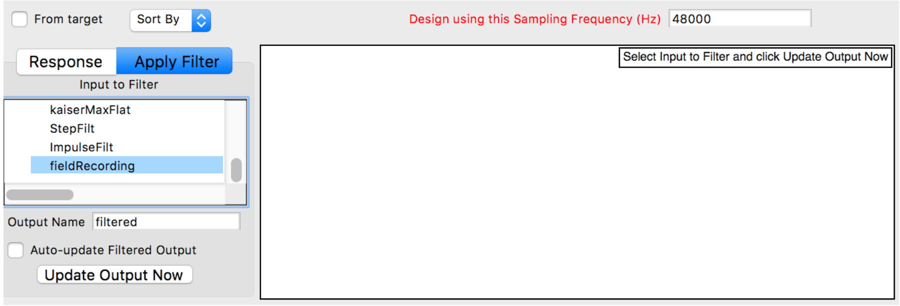
This dialog allows you to design a subset of the Igor Filter Design Laboratory's filters. It is simpler and the filters are sufficient for most purposes.

Initially the Design FIR Filter tab is shown with a simple Low Pass filter pre-selected. "Design using this Sampling Frequency (Hz)" is set to 1 and the frequencies shown are in the default range of 0 to 0.5 Hz because the default design sampling frequency is 1 Hz.

To start the filter design, either:

- Manually enter the sampling frequency or
- Click Apply Filter and select a wave to be filtered whose sampling frequency is properly set as described above

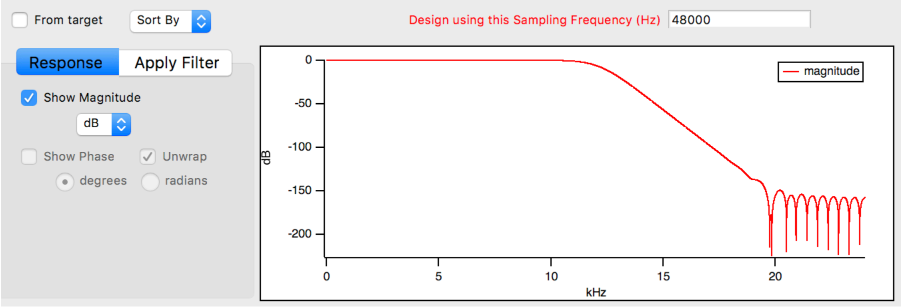
This fieldRecording wave was sampled at 48000 Hz:



Design using this Sampling Frequency (Hz) 48000

Output Name filtered

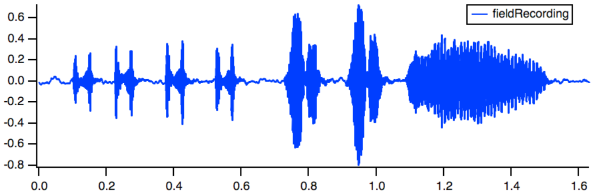
Switch back to the Response tab to show the default Low Pass filter using the entered sampling frequency. The frequency range is now 0-24000 Hz:



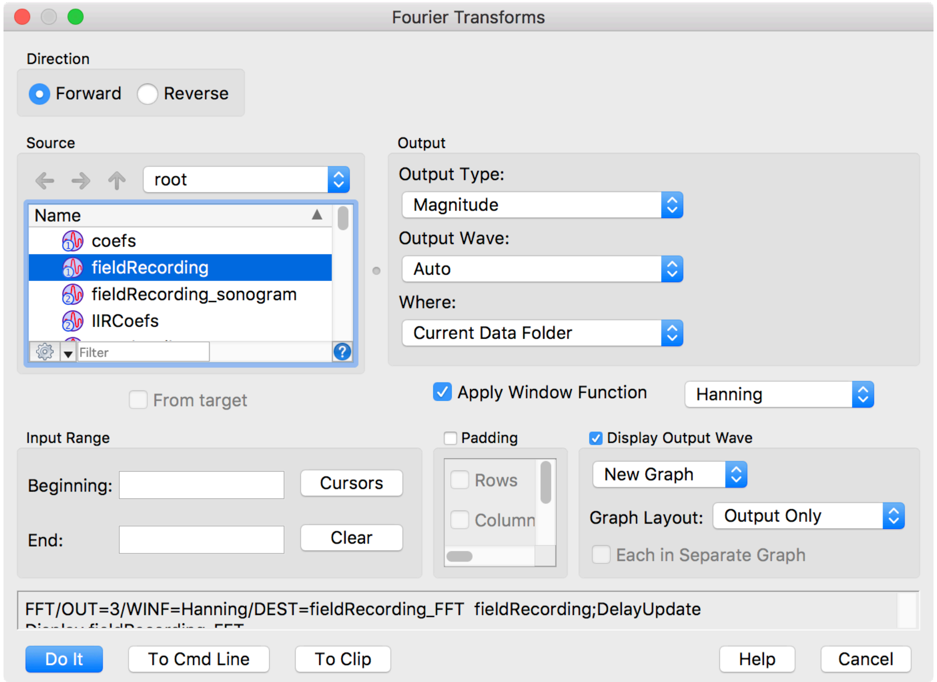
Design using this Sampling Frequency (Hz) 48000

You can use any combination of one low pass band, one high pass band, and one notch to pass or reject frequency components of the sampled data wave. By using both low pass and high pass bands you can create **Band Pass and Band Stop Filters**.

Before we apply a filter to fieldRecording, let's graph the original waveform:



It is also helpful if you know the frequency content of the input wave before filtering. Use the Analysis→Transforms→Fourier dialog:



Direction: ● Forward ○ Reverse

Source: root

Output Type: Magnitude

Output Wave: Auto

Where: Current Data Folder

□ From target

☑ Apply Window Function: Hanning

Beginning:

End:

□ Padding

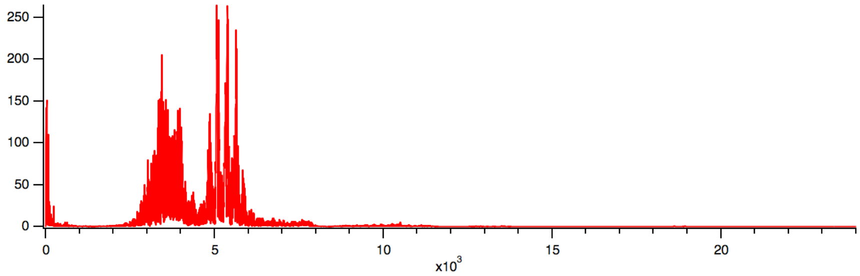
☑ Display Output Wave: New Graph

Graph Layout: Output Only

□ Each in Separate Graph

```
FFT/OUT=3/WINF=Hanning/DEST=fieldRecording_FFT  fieldRecording;DelayUpdate
```

This signal has two interesting bands of frequencies: 0 to 4.5KHz and 4.5 to about 10KHz.



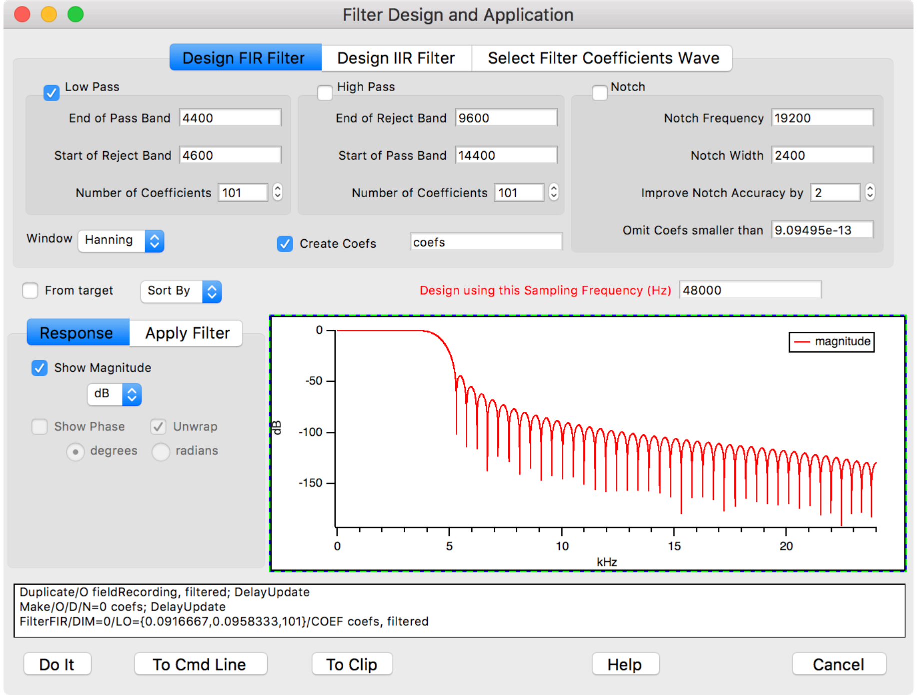
For illustrative purposes, let's walk through designing two filters that isolate each band, a low pass filter and a high pass filter.

Example: Low Pass FIR Filter

A low pass filter can be designed to keep the signal frequencies below 4.5KHz and reject higher frequencies.

An infinitely sharp cutoff between those frequencies isn't practical – it takes an infinite number of coefficients – so we specify two frequencies over which the transition from pass band to reject band happens. The smaller this transition band is, the more coefficients are needed to get a useful rejection of the higher frequencies.

Let's choose a 200 Hz transition width (4400 to 4600 Hz) and look at the frequency response with a reasonable number of coefficients (the default of 101):



☑ Low Pass
End of Pass Band 4400
Start of Reject Band 4600
Number of Coefficients 101

□ High Pass
End of Reject Band 9600
Start of Pass Band 14400
Number of Coefficients 101

□ Notch
Notch Frequency 19200
Notch Width 2400
Improve Notch Accuracy by 2
Omit Coefs smaller than 9.09495e-13

Window Hanning
☑ Create Coefs coefs
□ From target
Design using this Sampling Frequency (Hz) 48000

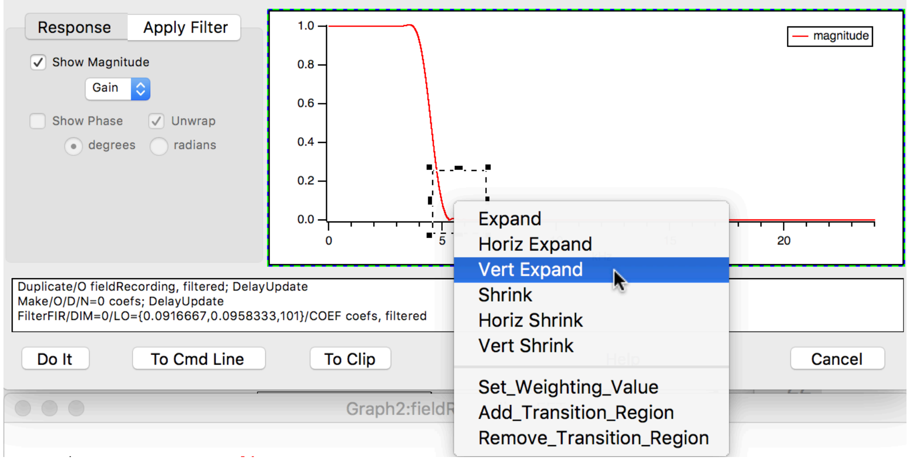
☑ Show Magnitude dB
□ Show Phase ☑ Unwrap ● degrees ○ radians

```
Duplicate/O fieldRecording, filtered; DelayUpdate
Make/O/D/N=0 coefs; DelayUpdate
FilterFIR/DIM=0/LO={0.0916667,0.0958333,101}/COEF coefs, filtered
```

What does dB mean?

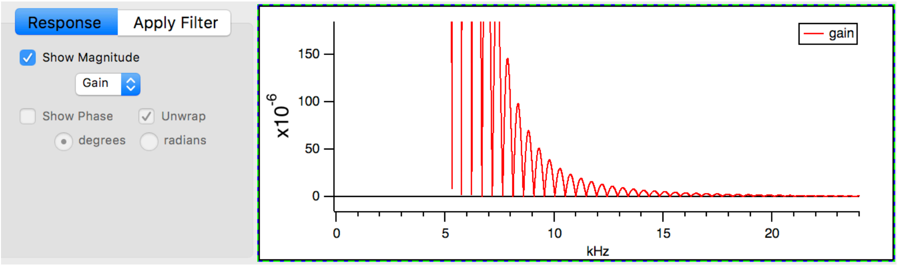
dB, an abbreviation for "decibel", is a logarithmic unit used to express the ratio of one value to another. In filtering it is the ratio of the output amplitude to the input amplitude at a given frequency. 0 dB means no change in ratio and is the response of a filter in the pass band.

dB is computed as 20 * log(ratio) where ratio is output amplitude divided by input amplitude. So -100dB means a ratio of 10^{-5} or 0.00001. You can see this by switching the response from dB to Gain and expanding the range vertically quite a lot:



☑ Show Magnitude Gain
□ Show Phase ☑ Unwrap ● degrees ○ radians

```
Duplicate/O fieldRecording, filtered; DelayUpdate
Make/O/D/N=0 coefs; DelayUpdate
FilterFIR/DIM=0/LO={0.0916667,0.0958333,101}/COEF coefs, filtered
```



☑ Show Magnitude

Gain

□ Show Phase ☑ Unwrap

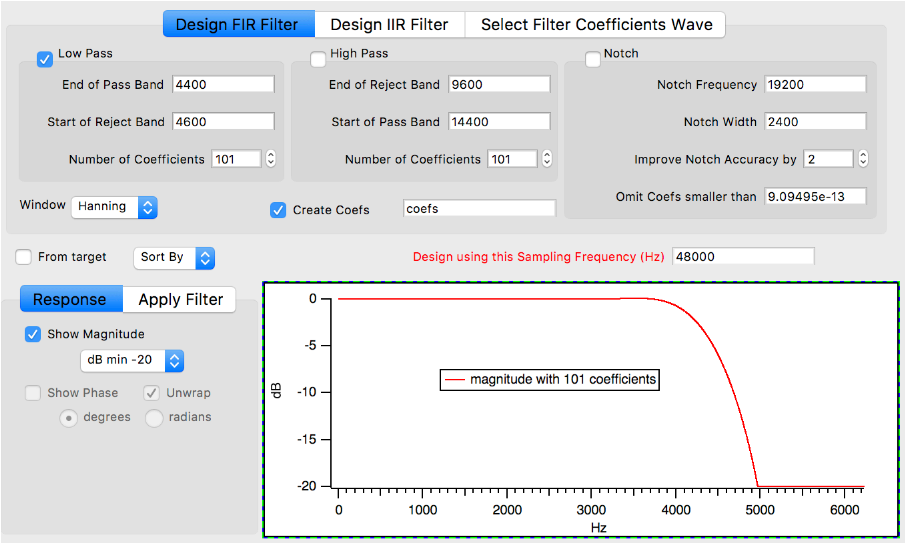
● degrees ○ radians

Choosing FIR Band Frequencies

You can avoid reducing the amplitude of frequencies near the end of the pass band by increasing the End of Pass Band frequency. A larger number of cofficients may be needed.

Choosing Number of Coefficients for FIR Designs

You can obtain a steeper transition from the pass band to the reject band by increasing the number of coefficients.



☑ Low Pass

End of Pass Band 4400

Start of Reject Band 4600

Number of Coefficients 101

□ High Pass

End of Reject Band 9600

Start of Pass Band 14400

Number of Coefficients 101

□ Notch

Notch Frequency 19200

Notch Width 2400

Improve Notch Accuracy by 2

Omit Coefs smaller than 9.09495e-13

Window Hanning

☑ Create Coefs coefs

□ From target Sort By

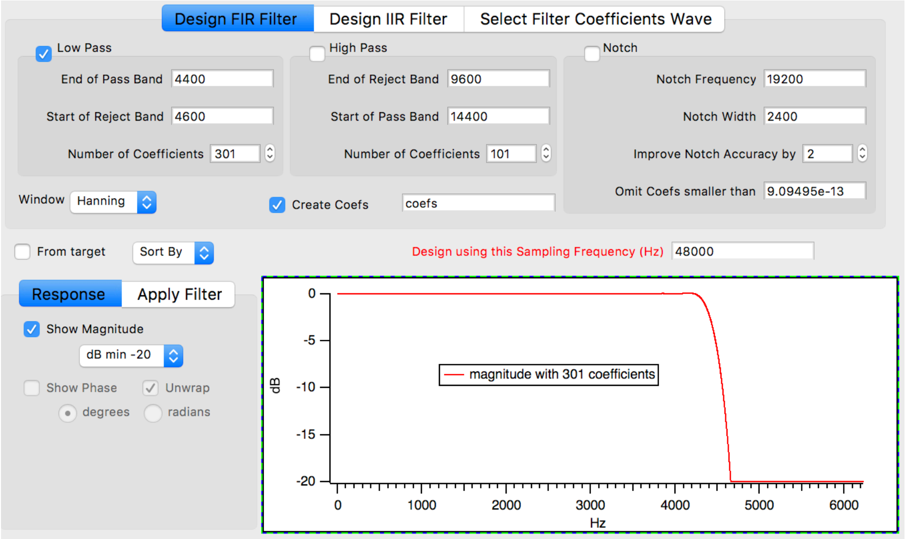
Design using this Sampling Frequency (Hz) 48000

☑ Show Magnitude

dB min -20

□ Show Phase ☑ Unwrap

● degrees ○ radians



☑ Low Pass

End of Pass Band 4400

Start of Reject Band 4600

Number of Coefficients 301

□ High Pass

End of Reject Band 9600

Start of Pass Band 14400

Number of Coefficients 101

□ Notch

Notch Frequency 19200

Notch Width 2400

Improve Notch Accuracy by 2

Omit Coefs smaller than 9.09495e-13

Window Hanning

☑ Create Coefs coefs

□ From target Sort By

Design using this Sampling Frequency (Hz) 48000

☑ Show Magnitude

dB min -20

□ Show Phase ☑ Unwrap

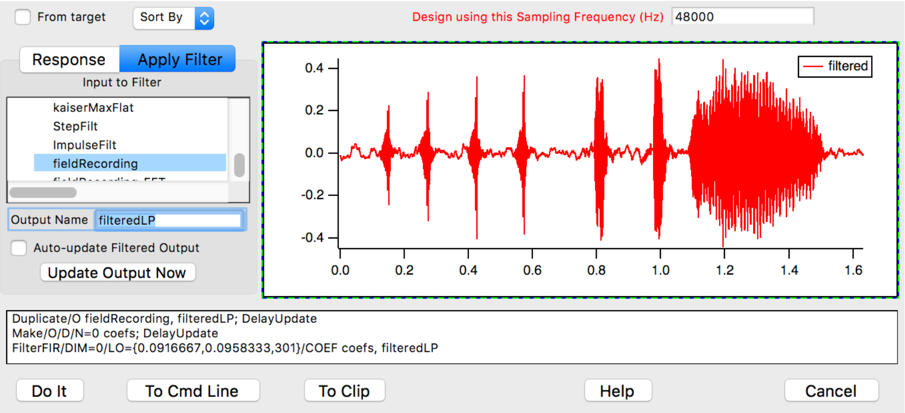
● degrees ○ radians

For an FIR filter, use an odd number of coefficients so that the the filtered output waveform is not delayed by a one-half sample.

Applying an FIR Filter to Data

Click the Apply Filter tab to see the result of applying the designed filter to a waveform.

Select the input wave in the Input to Filter listbox and click either Auto-update Filtered Output checkbox or the Update Output Now button. This updates a preview of the filtered result:



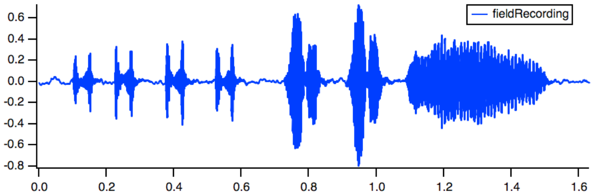
Design using this Sampling Frequency (Hz) 48000

Output Name filteredLP

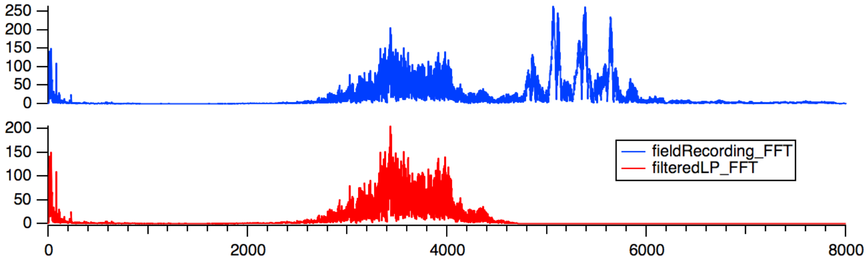
```
Duplicate/O fieldRecording, filteredLP; DelayUpdate
Make/O/D/N=0 coefs; DelayUpdate
FilterFIR/DIM=0/LO={0.0916667,0.0958333,301}/COEF coefs, filteredLP
```

Click Do It to create a final output wave in the current data folder. You can set the name of the final output wave in the Output Name field. Here we used "filteredLP".

For comparision, here is the unfiltered fieldRecording:



The preview in the dialog shows that the higher-frequency elements are removed in the filtered output. An FFT of the filteredLP result verifies the change:



Applying an FIR Filter to Other Data

You can reuse a filter if you keep a copy of the design's output coefficients. For example:

```
Duplicate/O coefs, savedFIRfilter // Keep a copy of the filter design
```

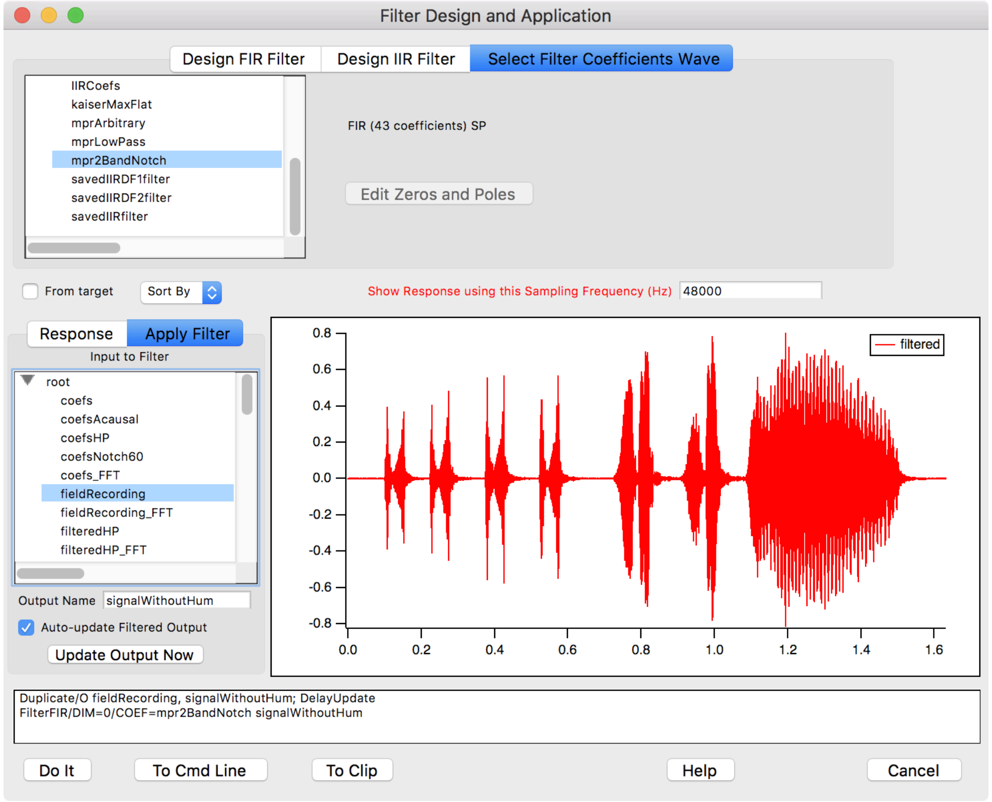
You can apply the saved FIR filter to other data using the FilterFIR operation directly or using the **Select Filter Coefficients Wave** tab of the Filter dialog. Using the **FilterFIR** operation:

```
Duplicate/O otherData, otherDataFiltered
FilterFIR /DIM=0 /COEF=savedFIRfilter otherDataFiltered
```

Select Filter Coefficients Wave

This section shows how to apply a saved FIR or IIR filter to other data using the Filter Design and Application Dialog dialog.

1. Select the saved filter design wave in the Select Filter Coefficients Wave tab of the dialog.
2. Select the wave to be filtered from the Apply Filter tab below.



FIR (43 coefficients) SP

Show Response using this Sampling Frequency (Hz) 48000

Output Name signalWithoutHum

☑ Auto-update Filtered Output

```
Duplicate/O fieldRecording, signalWithoutHum; DelayUpdate
FilterFIR/DIM=0/COEF=mpr2BandNotch signalWithoutHum
```

Example: High Pass FIR Filter

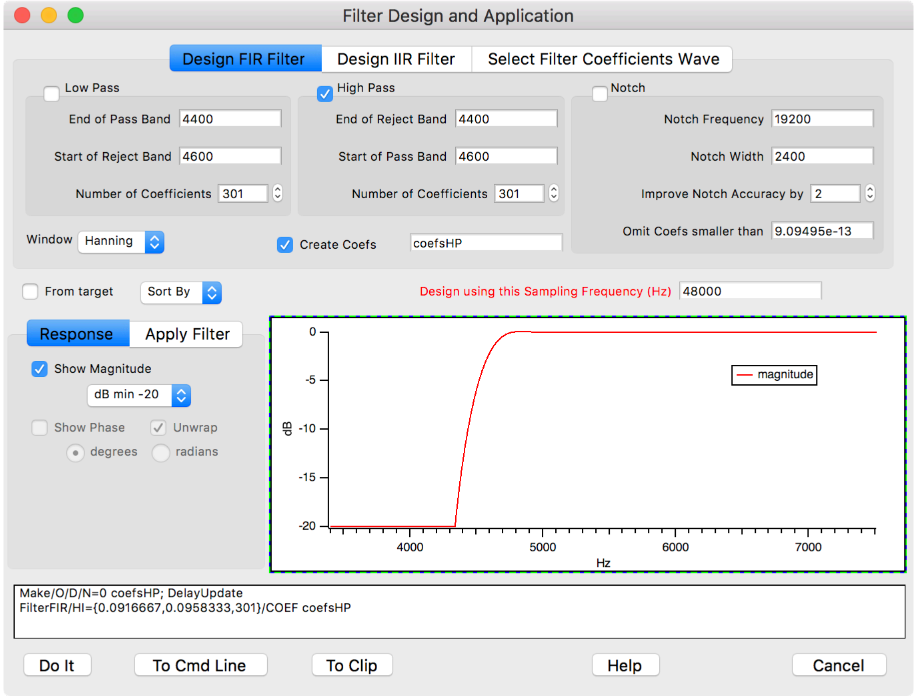
Next we design a high pass filter that preserves only the signal components that were removed by the low pass filter.

Choose Analysis→Filter and set "Design using this Sampling Frequency (Hz)" to 48000.

Uncheck Low Pass, check High Pass, and leave uncheck Notch.

Choosing High Pass FIR Band Frequencies

Set the End of Reject Band to 4400 and set Start of Pass Band to 4600 to define the same transition band as the low pass filter that we created above. Use 301 terms to get a steep transition between rejecting low frequencies and passing high frequencies:



Filter Design and Application

Design FIR Filter | Design IIR Filter | Select Filter Coefficients Wave

□ Low Pass
End of Pass Band 4400
Start of Reject Band 4600
Number of Coefficients 301

☑ High Pass
End of Reject Band 4400
Start of Pass Band 4600
Number of Coefficients 301

□ Notch
Notch Frequency 19200
Notch Width 2400
Improve Notch Accuracy by 2
Omit Coefs smaller than 9.09495e-13

Window Hanning

☑ Create Coefs coefsHP

□ From target Sort By

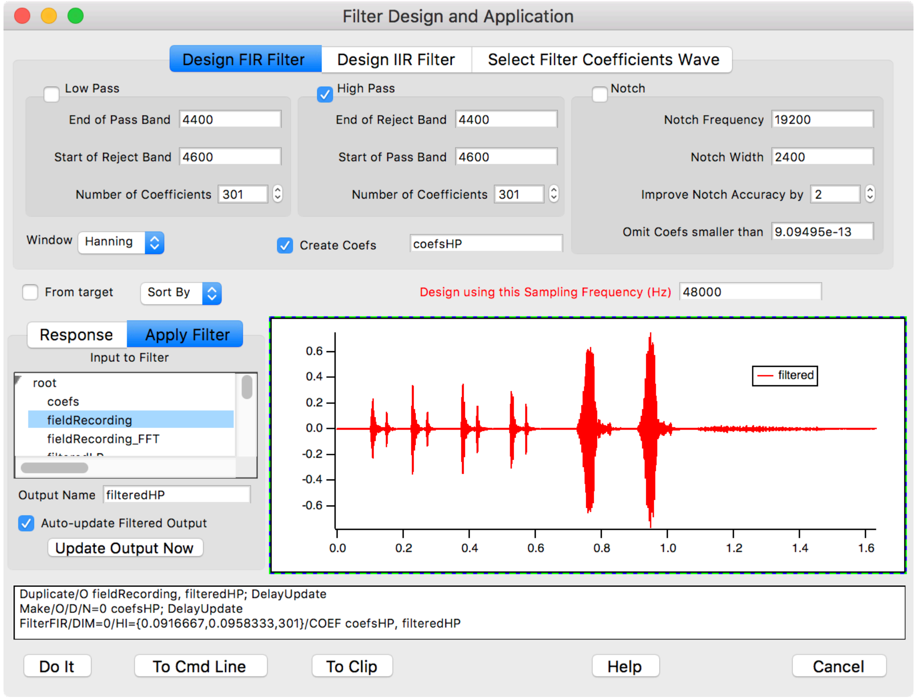
Design using this Sampling Frequency (Hz) 48000

Response | Apply Filter

☑ Show Magnitude
dB min -20
□ Show Phase ☑ Unwrap
● degrees ○ radians

```
Make/O/D/N=0 coefsHP; DelayUpdate
FilterFIR/HI={0.0916667,0.0958333,301}/COEF coefsHP
```

Click the Apply Filter tab to see the result of applying the designed filter to a waveform. Select the fieldRecording wave and click Update Output Now:



Filter Design and Application

Design FIR Filter | Design IIR Filter | Select Filter Coefficients Wave

□ Low Pass
End of Pass Band 4400
Start of Reject Band 4600
Number of Coefficients 301

☑ High Pass
End of Reject Band 4400
Start of Pass Band 4600
Number of Coefficients 301

□ Notch
Notch Frequency 19200
Notch Width 2400
Improve Notch Accuracy by 2
Omit Coefs smaller than 9.09495e-13

Window Hanning

☑ Create Coefs coefsHP

□ From target Sort By

Design using this Sampling Frequency (Hz) 48000

Response | Apply Filter

Input to Filter
root
coefs
fieldRecording
fieldRecording_FFT

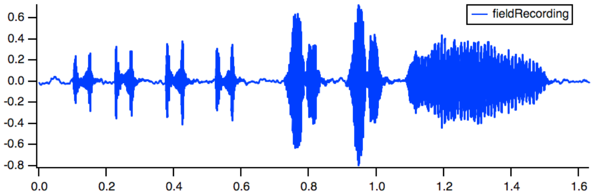
Output Name filteredHP

☑ Auto-update Filtered Output

Update Output Now

```
Duplicate/O fieldRecording, filteredHP; DelayUpdate
Make/O/D/N=0 coefsHP; DelayUpdate
FilterFIR/DIM=0/HI={0.0916667,0.0958333,301}/COEF coefsHP, filteredHP
```

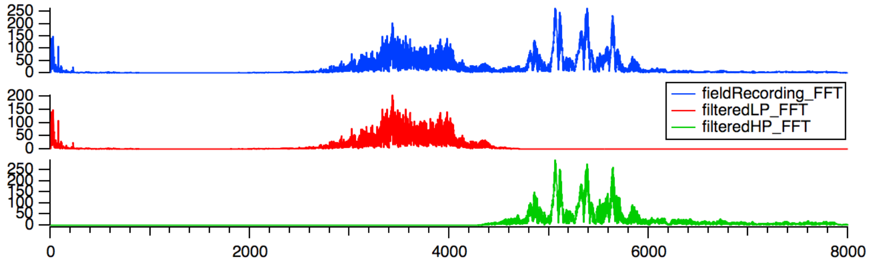
For comparision, here is the unfiltered fieldRecording:



The preview in the dialog shows that the lower-frequency elements are removed in the filtered output.

Click Do It to create the filteredHP result of high pass filtering the fieldRecording waveform.

An FFT of the filteredHP result verifies the change:



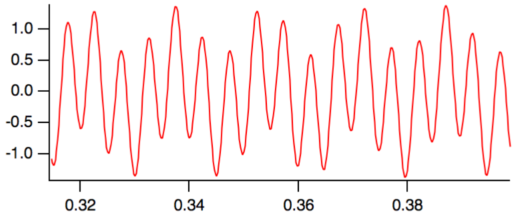
Another way to evaluate the filtering result is to use the PlaySound operation on the original and filtered waveforms:

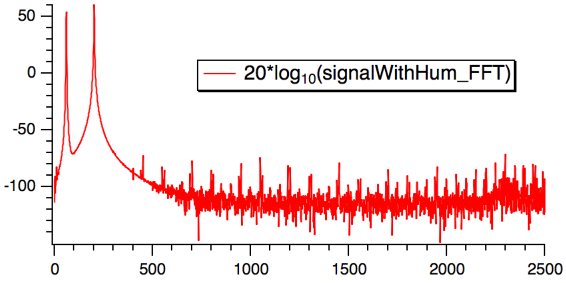
```
PlaySound fieldRecording
PlaySound filteredLP
PlaySound filteredHP
```

Example: Notch FIR Filter

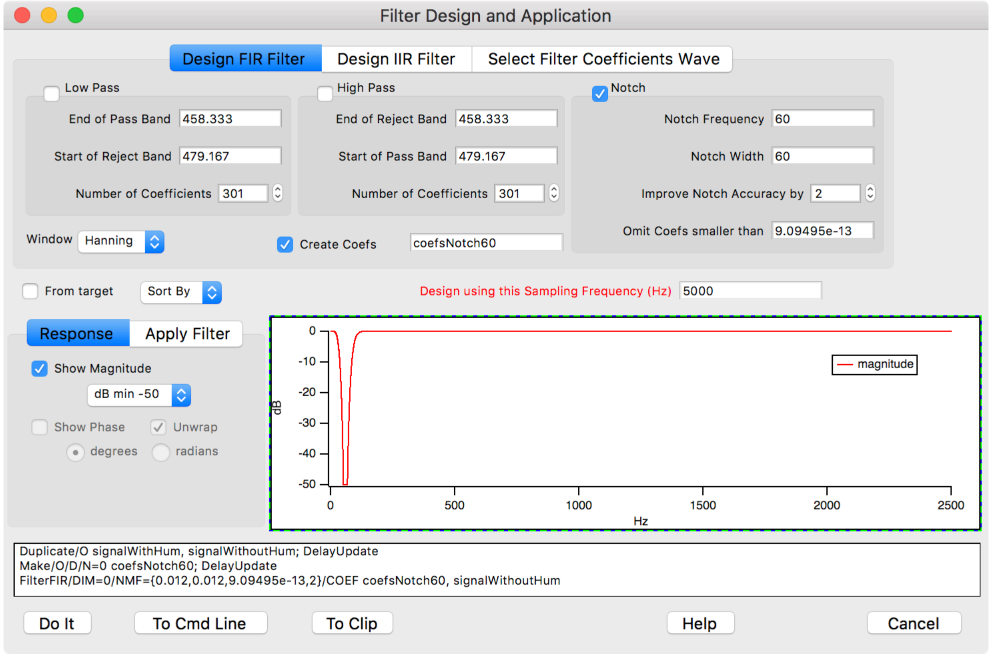
A notch filter is usually employed to reject a very narrow range of frequencies that interfere with the desired signal. Removing the interference of 50 or 60 Hz power signals from phsyiological waveforms is one such use case.

Here is a synthesized waveform with a 5000 Hz sampling frequency, a 200 Hz signal, and 60 Hz interference. The graph on the right shows it's spectral content:





Check the Notch checkbox and uncheck the Low Pass and High Pass checkboxes to create a notch-only filter.



□ Low Pass
End of Pass Band 458.333
Start of Reject Band 479.167
Number of Coefficients 301

□ High Pass
End of Reject Band 458.333
Start of Pass Band 479.167
Number of Coefficients 301

☑ Notch
Notch Frequency 60
Notch Width 60
Improve Notch Accuracy by 2
Omit Coefs smaller than 9.09495e-13

Window Hanning
☑ Create Coefs coefsNotch60

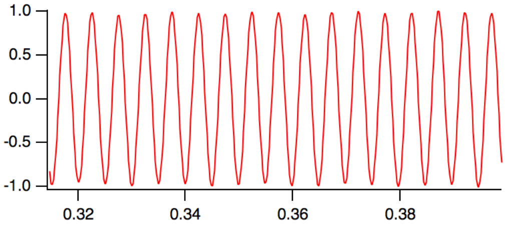
□ From target Sort By
Design using this Sampling Frequency (Hz) 5000

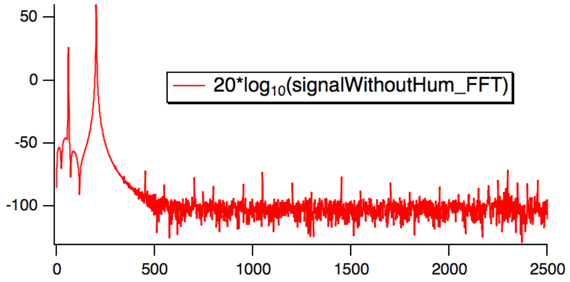
☑ Show Magnitude
dB min -50
□ Show Phase ☑ Unwrap
● degrees ○ radians

```
Duplicate/O signalWithHum, signalWithoutHum; DelayUpdate
Make/O/D/N=0 coefsNotch60; DelayUpdate
FilterFIR/DIM=0/NMF={0.012,0.012,9.09495e-13,2}/COEF coefsNotch60, signalWithoutHum
```

The FilterFIR operation uses high-precision calculations to get a deep notch at the selected frequency, preferring to adjust the frequency to get deeper notches. The Improve Notch Accuracy value (nMult in the **FilterFIR** documentation) indirectly sets the number of coefficients used to implement the notch.

Here is the result of this filter design:





The frequency response on the right does not look all that different but the filtered signal on the left has much less of the interfering 60 Hz signal.

Other FIR Designs using IFDL

The FIR filters created using the Filter Design and Application dialog are simple filters created by applying a "window" shape - such as the Hanning **WindowFunction** - to truncated sin(x)/x kernels.

The filters are functional but require a lot of coefficients to get high performance (steep filter transition bands, good rejection of unwanted frequencies). Often these aren't important shortcomings, but if the designed filter is intended for actual electronic implementation, those extra coefficients get expensive.

High-performance FIR filters using far fewer coefficients can be computed by using the Igor Filter Design Laboratory package. It optimizes both filter response and the number of filter coefficients using the Remez Exchange algorithm as described in the seminal paper by [McClellan], Parks, and Rabiner. See the **Remez** operation for additional references. See the “Igor Filter Design Laboratory” help file for details.

IIR Designs

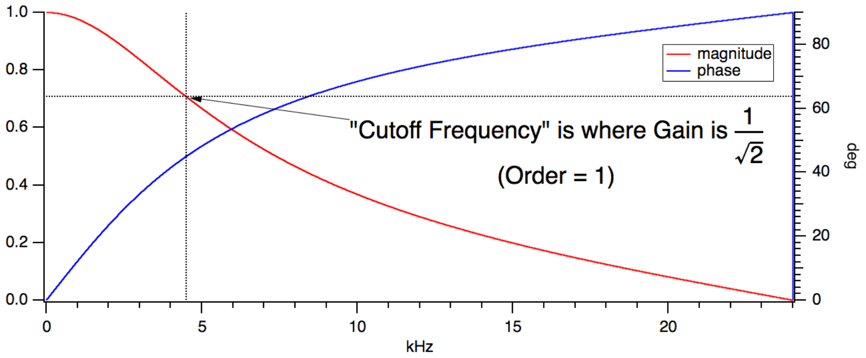
The IIR filters created using the Filter Design and Application dialog are based on tranforms of analog Butterworth filters, a standard smooth-response filter of the electrical and mechanical engineering worlds. See **IIR Filters** on page III-300 for details.

Use the Igor Filter Design Laboratory to design IIR filters based on transforms of analog Bessel and Chebyshev filters. See the “Igor Filter Design Laboratory”help file for details.

Like the **IIR Filters**, the easily-designed IIR filters are specified in terms of filter type (low pass, high pass, notch) and design frequencies.

Choosing IIR Band Frequencies

Unlike FIR Design, the IIR Design uses a single cutoff frequency to define pass and reject bands. You can think of the cutoff frequency as the frequency where the response begins to "cut off" (reject) frequency components of the signal. "Begins" is chosen to be at the -3 dB point of the response, the so-called "half-power" amplitude, where the gain is 1/sqrt(2) = 0.707107:

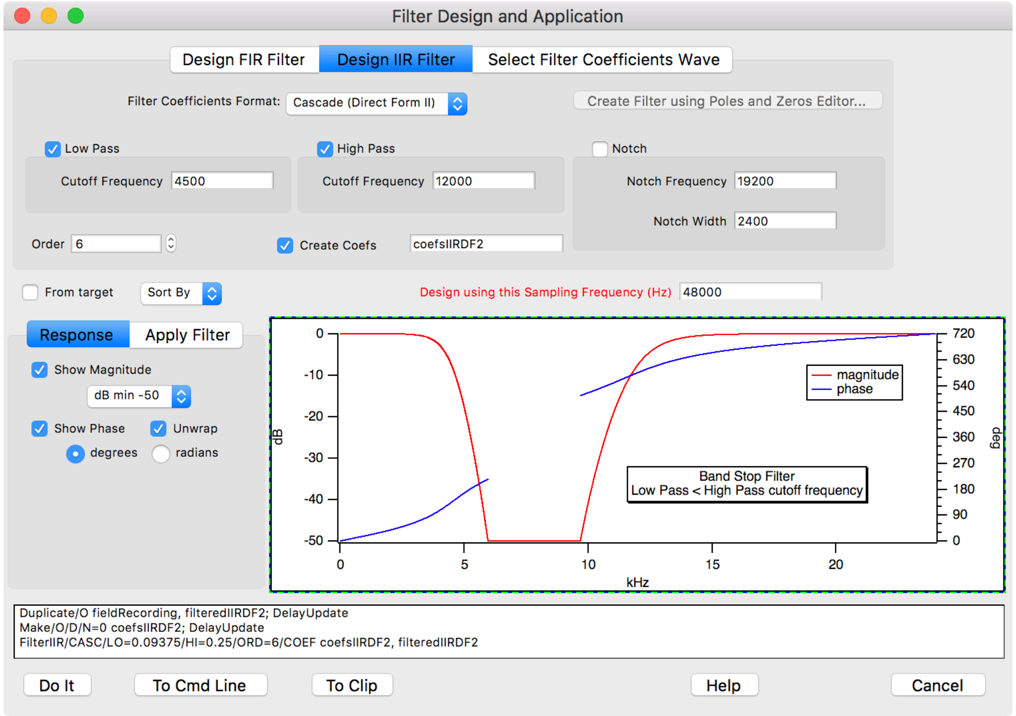


Band Pass and Band Stop Filters

A filter that passes or rejects a range of frequencies that do not include 0 or the Nyquist frequency is called a band pass or band stop filter. Such filters are useful only for preserving or rejecting a narrow range of frequencies. A notch filter is a kind of band stop filter that has its own special implementation.

Band pass and band stop filters both use the Low Pass and High Pass settings of the dialog. The difference is which cutoff frequency is lower than the other.

If the low pass cutoff frequency is less than high pass cutoff frequency, the result is a band stop filter:



Filter Coefficients Format: Cascade (Direct Form II)

☑ Low Pass Cutoff Frequency 4500

☑ High Pass Cutoff Frequency 12000

□ Notch Notch Frequency 19200 Notch Width 2400

Order 6

☑ Create Coefs coefsIIRDF2

□ From target

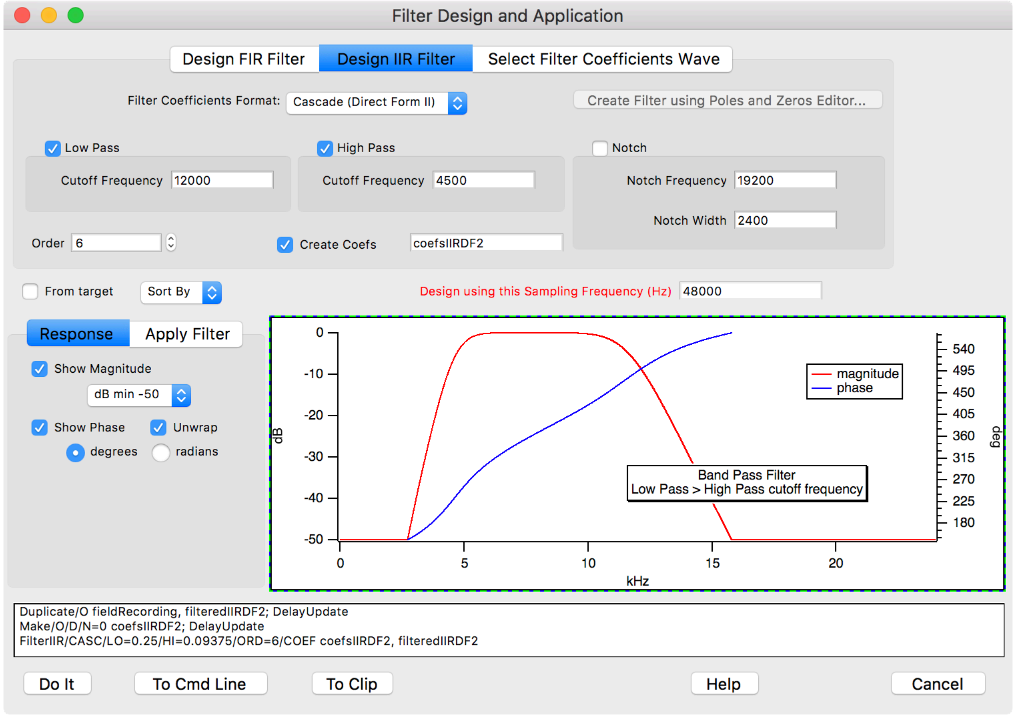
Design using this Sampling Frequency (Hz) 48000

☑ Show Magnitude dB min -50

☑ Show Phase ☑ Unwrap ● degrees ○ radians

```
Duplicate/O fieldRecording, filteredIIRDF2; DelayUpdate
Make/O/D/N=0 coefsIIRDF2; DelayUpdate
FilterIIR/CASC/LO=0.09375/HI=0.25/ORD=6/COEF coefsIIRDF2, filteredIIRDF2
```

If the high pass cutoff frequency is less than low pass cutoff frequency, the result is a band pass filter:



Filter Coefficients Format: Cascade (Direct Form II)

☑ Low Pass Cutoff Frequency 12000

☑ High Pass Cutoff Frequency 4500

□ Notch Notch Frequency 19200 Notch Width 2400

Order 6

☑ Create Coefs coefsIIRDF2

□ From target

Design using this Sampling Frequency (Hz) 48000

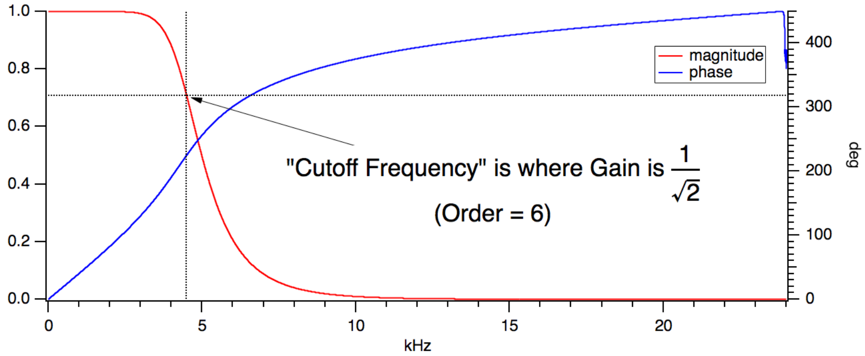
☑ Show Magnitude dB min -50

☑ Show Phase ☑ Unwrap ● degrees ○ radians

```
Duplicate/O fieldRecording, filteredIIRDF2; DelayUpdate
Make/O/D/N=0 coefsIIRDF2; DelayUpdate
FilterIIR/CASC/LO=0.25/HI=0.09375/ORD=6/COEF coefsIIRDF2, filteredIIRDF2
```

Choosing Order for IIR Designs

Instead of adjusting the number of coefficients to alter the performance of the filtering, IIR designs use a filter "order". Essentially, each order represents another layer of recursive filtering. A higher-order filter has steeper band transitions, more phase shift, and can be numerically less stable. Increasing the order from 1 to 6 creates a much steeper transition:



"Cutoff Frequency" is where Gain is $\frac{1}{\sqrt{2}}$

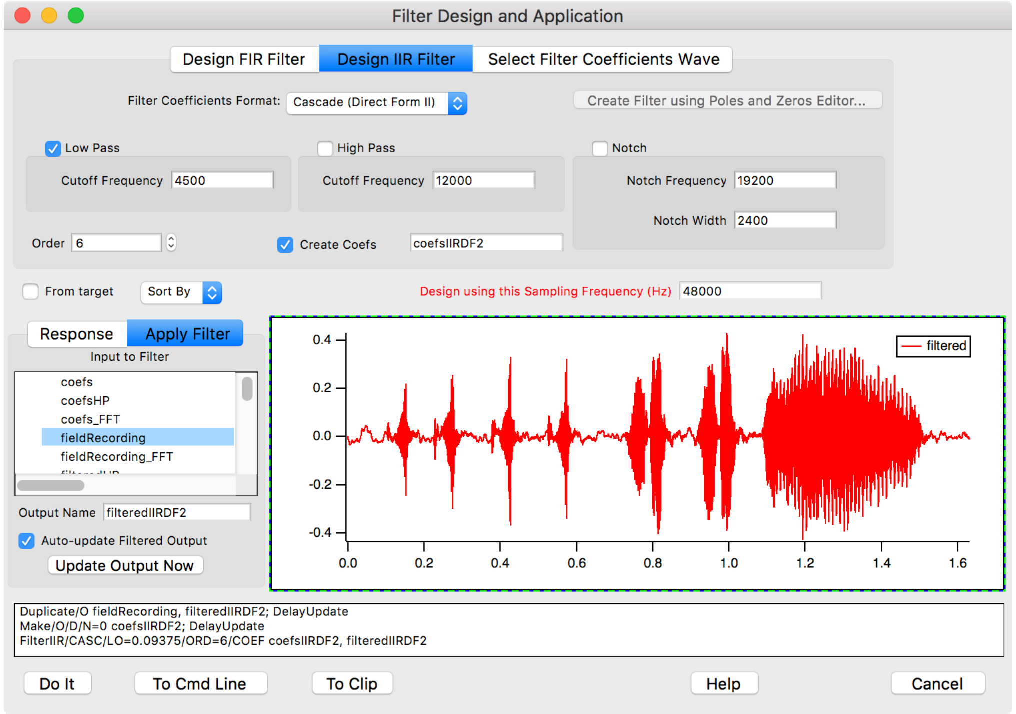
(Order = 6)

Applying an IIR Filter to Data

Click the Apply Filter tab to see the result of applying the designed filter to a waveform.

Select the input wave in the Input to Filter listbox and click either Auto-update Filtered Output checkbox or the Update Output Now button. This updates a preview of the filtered result.

Click Do It to create a final output wave in the current data folder. You can set the name of the final output wave in the Output Name field. Here we used filteredIIRDF2:



Filter Coefficients Format: Cascade (Direct Form II)

☑ Low Pass
Cutoff Frequency 4500

□ High Pass
Cutoff Frequency 12000

□ Notch
Notch Frequency 19200
Notch Width 2400

Order 6

☑ Create Coefs coefsIIRDF2

□ From target

Design using this Sampling Frequency (Hz) 48000

Output Name filteredIIRDF2

☑ Auto-update Filtered Output

```
Duplicate/O fieldRecording, filteredIIRDF2; DelayUpdate
Make/O/D/N=0 coefsIIRDF2; DelayUpdate
FilterIIR/CASC/LO=0.09375/ORD=6/COEF coefsIIRDF2, filteredIIRDF2
```

Evaluating a Digital Filter

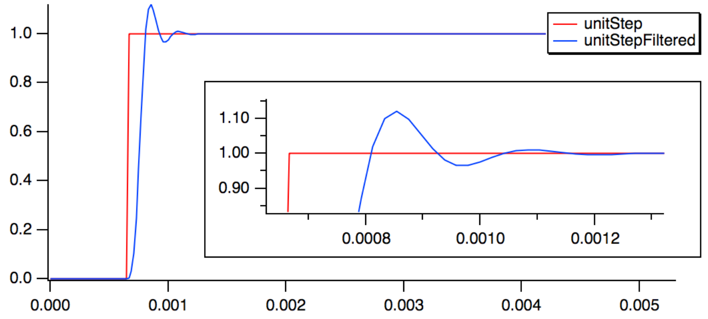
The graph in the Response tab of the Filter Design and Application dialog is the most direct way to evaluate what the filter will do to an input waveform.

You can also graph the original data and filtered data for a detailed visual comparision.

A useful technique, borrowed from electrical engineering, is to see how the filter responds to an ideal "unit step" waveform:

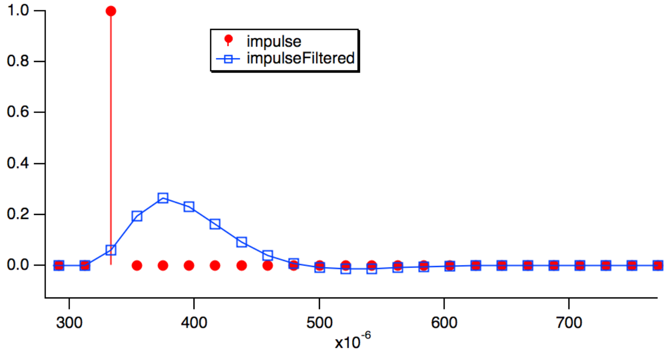
```
Make/O/N=256 unitStep = p >= 32      // Unit step wave for causal IIR filters
CopyScales/P yourData, unitStep      // Same sampling frequency as your data
```

```
Duplicate/O unitStep, unitStepFiltered
FilterIIR/DIM=0/COEF=savedIIRDF1filter unitStepFiltered
Display unitStep, unitStepFiltered
```

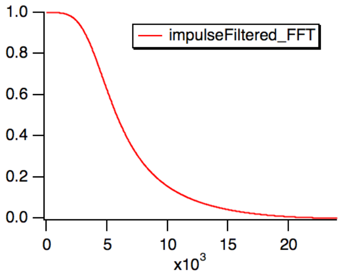


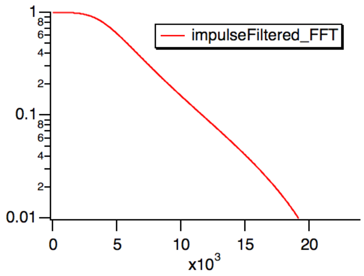
The next example shows how the filter responds to an ideal "unit impulse" waveform and display it's FFT magnitude, as the Filter Design and Application dialog does:

```
Make/O/N=2048 impulse = p == 16     // Unit step wave for causal IIR filters
CopyScales/P yourData, impulse
Duplicate/O impulse, impulseFiltered
FilterIIR/CASC/DIM=0/COEF=savedIIRDF2filter impulseFiltered // DF II 
implementation needs /CASC
Display impulse, impulseFiltered
```



```
FFT/MAG/DEST=impulseFiltered_FFT impulseFiltered   // Magnitude of response
Display impulseFiltered_FFT
Display impulseFiltered_FFT      // A logarithmic axis has the same shape
ModifyGraph log(left)=1          // as computing 20*log(response)
```





Applying an IIR Filter to Other Data

You can reuse a filter if you keep a copy of the design's output coefficients. For example:

```
Duplicate/O coefs, savedIIRfilter // Keep a copy of the filter design.
```

You can apply the saved IIR filter to other data using the **FilterIIR** operation:

```
Duplicate/O otherData, otherDataFiltered
FilterIIR/DIM=0/COEF=savedIIRfilter otherDataFiltered
```

You can also apply the saved IIR filter to other data using the **Select Filter Coefficients Wave** tab of the Filter dialog.

Rotate Operation

The **Rotate** operation (see page V-810) rotates the data values of the selected waves by a specified number of points. Choose Data→Rotate Waves to display the Rotate dialog.

Think of the data values of a wave as a column of numbers. If the specified number of points is positive the points in the wave are rotated downward. If the specified number of points is negative the points in the wave are rotated upward. Values that are rotated off one end of the column wrap to the other end.

The rotate operation shifts the X scaling of the rotated wave so that, except for the points which wrap around, the X value of a given point is not changed by the rotation. To observe this, display the X scaling and data values of the wave in a table and notice the effect of Rotate on the X values.

This change in X scaling may or may not be what you want. It is usually not what you want if you are rotating an XY pair. In this case, you should undo the X scaling change using the SetScale operation:

```
SetScale/P x,0,1,"",waveName     // replace waveName with name of your wave
```

Also see the example of rotation in **Spectral Windowing** on page III-275.

For multi-dimensional wave rotation, see the **MatrixOp** rotateRows, rotateCols, rotateLayers, and rotateChunks functions.

Unwrap Operation

The **Unwrap** operation (see page V-1050) scans through each specified wave trying to undo the effect of a modulus operation. For example, if you perform an FFT on a wave, the result is a complex wave in rectangular coordinates. You can create a real wave which contains the phase of the result of the FFT with the command:

```
wave2 = imag(r2polar(wave1))
```

However the rectangular-to-polar conversion leaves the phase information modulo 2π. You can restore the continuous phase information with the command:

```
Unwrap 2*Pi, wave2
```

The Unwrap operation is designed for 1D waves only. Unwrapping 2D data is considerably more difficult. See the **ImageUnwrapPhase** operation on page V-433 for more information

Choose Analysis→Unwrap to display the Unwrap Waves dialog.

References

Cleveland, W.S., Robust locally weighted regression and smoothing scatterplots, *J. Am. Stat. Assoc.*, *74*, 829-836, 1977.

Marchand, P., and L. Marmet, Binomial smoothing filter: A way to avoid some pitfalls of least square polynomial smoothing, *Rev. Sci. Instrum.*, *54*, 1034-41, 1983.

Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

Savitzky, A., and M.J.E. Golay, Smoothing and differentiation of data by simplified least squares procedures, *Analytical Chemistry*, *36*, 1627–1639, 1964.

Wigner, E. P., On the quantum correction for thermo-dynamic equilibrium, *Physics Review*, *40*, 749-759, 1932.

Chapter

III-10

Analysis of Functions

Operations that Work on Functions

Some Igor operations work on functions rather than data in waves. These operations take as input one or more functions that you define in the Procedure window. The result is some calculation based on function values produced when Igor evaluates your function.

Because the operations evaluate a function, they work on continuous data. That is, the functions are not restricted to data values that you provide from measurements. They can be evaluated at any input values. Of course, a computer works with discrete digital numbers, so even a “continuous” function is broken into discrete values. Usually these discrete values are so close together that they are continuous for practical purposes. Occasionally, however, the discrete nature of computer computations causes problems.

The following operations use functions as inputs:

- **IntegrateODE** computes numerical solutions to ordinary differential equations. The differential equations are defined as user functions. The IntegrateODE operation is described under **Solving Differential Equations** on page III-322.
- **FindRoots** computes solutions to f(x)=a, where a is a constant (often zero). The input x may represent a vector of x values. A special form of FindRoots computes roots of polynomials. The FindRoots operation is described in the section **Finding Function Roots** on page III-338.
- **Optimize** finds minima or maxima of a function, which may have one or more input variables. The Optimize operation is described in the section **Finding Minima and Maxima of Functions** on page III-343.
- **Integrate1D** integrates a function between two specified limits. Despite its name, it can also be used for integrating in two or more dimensions. See **Integrating a User Function** on page III-336.

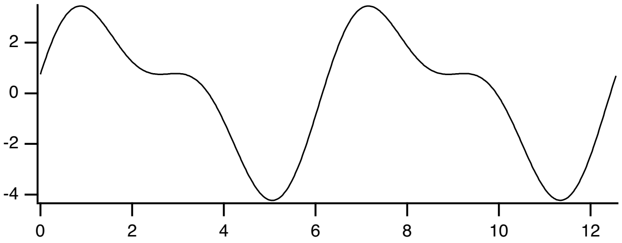
Function Plotting

Function plotting is very easy in Igor, assuming that you understand what a waveform is (see **Waveform Model of Data** on page II-62) and how X scaling works. Here are the steps to plot a function.

1. Decide how many data points you want to plot.
2. Make a wave with that many points.
3. Use the SetScale operation to set the wave’s X scaling. This defines the domain over which you are going to plot the function.
4. Display the wave in a graph.
5. Execute a waveform assignment statement to set the data values of the wave.

Here is an example.

```
Make/O/N=500 wave0
SetScale/I x, 0, 4*PI, wave0      // plot function from x=0 to x=4π
Display wave0
wave0 = 3*sin(x) + 1.5*sin(2*x + PI/6)
```



To evaluate the function over a different domain, you need to reexecute the SetScale command with different parameters. This redefines “x” for the wave. Then you need to reexecute the waveform assignment statement. For example,

```
SetScale/I x, 0, 2*PI, wave0      // plot function from x=0 to x=2π
wave0 = 3*sin(x) + 1.5*sin(2*x + PI/6)
```

Reexecuting commands is easy, using the shortcuts shown in **History Area** on page II-9.

Function Plotting Using Dependencies

If you get tired of reexecuting the waveform assignment statement each time you change the domain, you can use a dependency to cause Igor to automatically reexecute it. To do this, use := instead of =.

```
wave0 := 3*sin(x) + 1.5*sin(2*x + PI/6)
```

See Chapter IV-9, **Dependencies**, for details.

You have made `wave0` depend on “X”. The SetScale operation changes the meaning of “X” for the wave. Now when you do a SetScale on `wave0`, Igor will automatically reexecute the assignment.

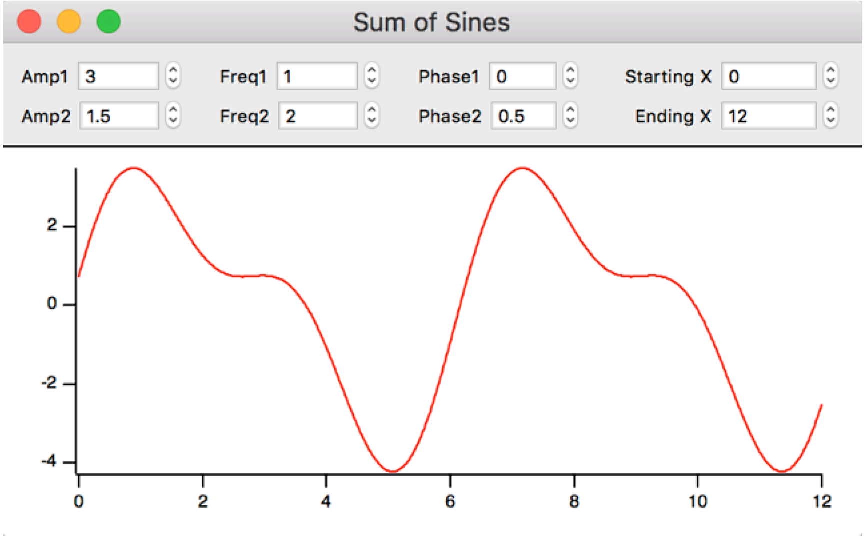
You can take this further by using global variables instead of literal numbers in the right-hand expression. For example:

```
Variable/G amp1=3, amp2=1.5, freq1=1, freq2=2, phase1=0, phase2=PI/6
wave0 := amp1*sin(freq1*x + phase1) + amp2*sin(freq2*x + phase2)
```

Now, wave0 depends on these global variables. If you change them, Igor will automatically reexecute the assignment.

Function Plotting Using Controls

For a slick presentation of function plotting, you can put controls in the graph to set the values of the global variables. When you change the value in the control, the global variable changes, which reexecutes the assignment. This changes the wave, which updates the graph. Here is what the graph would look like.



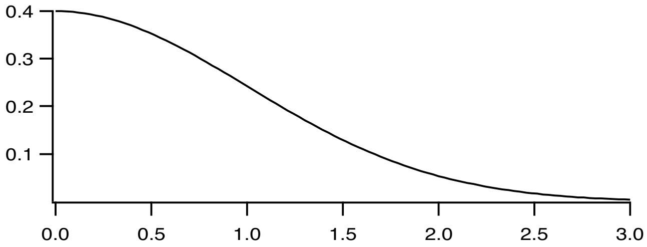
We’ve added two additional global variables and connected them to the Starting X and Ending X controls. This allows us to set the domain. These controls are both linked to an action procedure that does a SetScale on the wave.

Controls are explained in detail in Chapter III-14, **Controls and Control Panels**.

Plotting a User-Defined Function

In the preceding example we used the built-in sin function in the right-hand expression. We can also use a user-defined function. Here is an example using a very simple function — the normal probability distribution function.

```
Function NormalProb(x)
   Variable x

   // the constant is 1/sqrt(2*pi) evaluated in double-precision
   return     0.398942280401433*exp(-(0.5*x^2))
End

Make/N=100 wave0; SetScale/I x, 0, 3, wave0; wave0 = NormalProb(x)
Display wave0
```



Note that, although we are using the NormalProb function to fill a wave, the NormalProb function itself has nothing to do with waves. It merely takes an input and returns a single output. We could also test the NormalProb function at a single point by executing

```
Print NormalProb(0)
```

This would print the output of the function in the history area.

It is the act of using the `NormalProb` function in a wave assignment statement that fills the wave with data values. As it executes the wave assignment, Igor calls the NormalProb function over and over again, 100 times in this case, passing it a different parameter each time and storing the output from the `NormalProb` function in successive points of the destination wave.

For more information on Wave Assignments, see **Waveform Arithmetic and Assignments** on page II-74. You may also find it helpful to read Chapter IV-1, **Working with Commands**.

WaveMetrics provides a procedure package that provides a convenient user interface to graph mathematical expressions. To use it, choose Analysis→Packages→Function Grapher. This displays a graph with controls to create and display a function. Click the Help button in the graph to learn how to use it.

Solving Differential Equations

Numerical solutions to initial-value problems involving ordinary differential equations can be calculated using the **IntegrateODE** operation (see page V-452). You provide a user-defined function that implements a system of differential equations. The solution to your differential equations are calculated by marching the solution forward or backward from the initial conditions in a series of steps or increments in the independent variable.

Terminology

Referring to the independent variable and the dependent variables is very cumbersome, so we refer to these as X and Y[*i*]. Of course, X may represent distance or time or anything else.

A system of differential equations will be written in terms of derivatives of the Y[*i*]s, or dy[*i*]/dx.

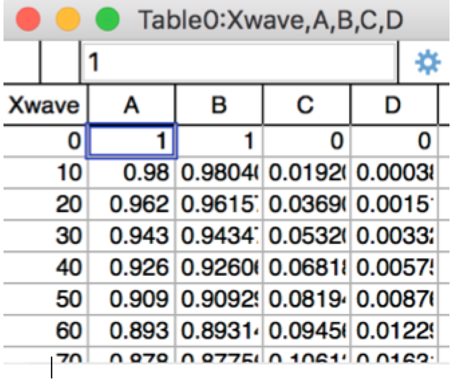
ODE Inputs

You provide to IntegrateODE a function to calculate the derivatives or right-hand-sides of your system of differential equations.

You also provide one output wave for each equation in the system to receive the solution. The solution waves will have a row for each output point you want.

You specify the independent variable either by setting the X scaling of the output waves, by specifying x0 and deltax using the /X={x0,deltax} flag, or by providing an explicit X wave using the /X=*xWave* flag.

For a system of four equations (fourth-order system), if you provide an X wave to specify where you want values, you might have this situation:



Xwave	A	B	C	D
0	1	1	0	0
10	0.98	0.9804	0.0192	0.0003
20	0.962	0.9615	0.0369	0.0015
30	0.943	0.9434	0.0532	0.0033
40	0.926	0.9260	0.0681	0.0057
50	0.909	0.9092	0.0819	0.0087
60	0.893	0.8931	0.0945	0.0122
70	0.878	0.8775	0.1061	0.0163

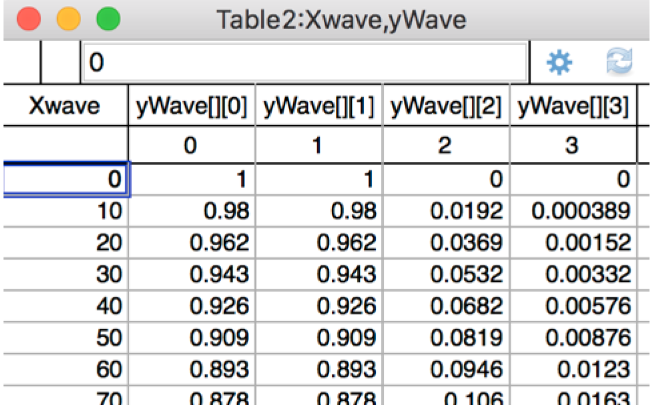
X wave, four Y waves (A, B, C, and D).

First row contains initial conditions.

Subsequent rows receive solution values.

X wave specifies where to report solutions.
In free-run mode, X wave receives X values for solution rows.

Instead of multiple 1D output waves, you can provide a single 2D output wave:



Xwave	yWave[][0]	yWave[][1]	yWave[][2]	yWave[][3]
	0	1	2	3
0	1	1	0	0
10	0.98	0.98	0.0192	0.000389
20	0.962	0.962	0.0369	0.00152
30	0.943	0.943	0.0532	0.00332
40	0.926	0.926	0.0682	0.00576
50	0.909	0.909	0.0819	0.00876
60	0.893	0.893	0.0946	0.0123
70	0.878	0.878	0.106	0.0163

Igor calculates a solution value for each element of the Y (output) waves.

Before executing IntegrateODE, you must load the initial conditions (the initial Y[*i*] values) into the first row of the Y waves. Igor then calculates the solution starting from those values. The first solution value is stored in the second element of the Y waves.

If you use the /R flag with IntegrateODE to start the integration at a point other than the beginning of the Y wave, the initial conditions must be in the first row specified by the /R flag. See **Stopping and Restarting IntegrateODE** on page III-334.

ODE Outputs

The algorithms Igor uses to integrate your ODE systems use adaptive step-size control. That is, the algorithms advance the solution by the largest increment in X that result in errors at least as small as you require. If the solution is changing rapidly, or the solution has some other difficulty, the step sizes may get very small.

IntegrateODE has two schemes for returning solution results to you: you can specify X values where you need solution values, or you can let the solution “free run”.

In the first mode, results are returned to you at values of x corresponding to the X scaling of your Y waves, or at X values that you provide via an X wave or by providing X0 and deltaX and letting Igor calculate the X values. The actual calculation may require X increments smaller than those you ask for. Igor returns results only at the X values you ask for.

In free-run mode, IntegrateODE returns solution values for every step taken by the integration algorithm. In some cases, this may give you extremely small steps. Free-run mode returns to you not only the Y[*i*] values from the solution, but also values of x[*i*]. Free-run mode can be useful in that, to some degree, it will return results closely spaced when the solution is changing rapidly and with larger spacing when the solution is changing slowly.

The Derivative Function

You must provide a user-defined function to calculate derivatives corresponding to the equations you want to solve. All equations are solved as systems of first-order equations. Higher-order equations must be transformed to multiple first-order equations (an example is shown later).

The derivative function has this form:

```
Function D(pw, xx, yw, dydx)
   Wave pw           // parameter wave (input)
   Variable xx       // x value at which to calculate derivatives
   Wave yw           // wave containing y[i] (input)
   Wave dydx         // wave to receive dy[i]/dx (output)

   dydx[0] = <expression for one derivative>
   dydx[1] = <expression for next derivative>
   <etc.>

   return 0
End
```

Note the return statement at the end of the function. The function result should normally be 0. If it is 1, IntegrateODE will stop. If the return statement is omitted, the function returns NaN which IntegrateODE treats the same as 0. But it is best to explicitly return 0.

Because the function may produce a large number of outputs, the outputs are returned via a wave in the parameter list.

The parameter wave is simply a wave containing possible adjustable constants. Using a wave for these makes it convenient to change the constants and try a new integration. It also will make it more convenient to do a curve fit to a differential equation. You must create the parameter wave before invoking IntegrateODE. The contents of the parameter wave are of no concern to IntegrateODE and are not touched. In fact, you can change the contents of the parameter wave inside your function and those changes will be permanent.

Other inputs are the value of x at which the derivatives are to be evaluated, and a wave (yw in this example) containing current values of the y[*i*]’s. The value of X is determined when it calls your function, and the waves yw and dydx are both created and passed to your function when Igor needs new values for the derivatives. Both the input Y wave and the output dydx wave have as many elements as the number of derivative equations in your system of ODEs.

The values in the yw wave correspond to a row of the table in the example above. That is:

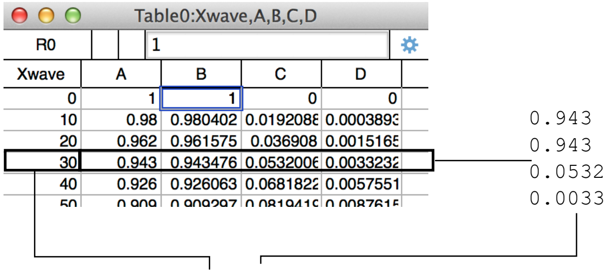


Table0:Xwave,A,B,C,D

R0 | 1

Xwave	A	B	C	D
0	1	1	0	0
10	0.98	0.980402	0.0192088	0.0003893
20	0.962	0.961575	0.036908	0.0015165
30	0.943	0.943476	0.0532006	0.0033232
40	0.926	0.926063	0.0681822	0.0057551
50	0.909	0.909297	0.0819419	0.0087615

```
Function D(pw, xx, yw, dydx)
    Wave pw     // parameter wave (input)
    Variable xx // x value at which to calculate derivatives
    Wave yw     // wave containing y[i] (input)
    Wave dydx   // wave to receive dy[i]/dx (output)

    dydx[0] = <expression for one derivative>
    dydx[1] = <expression for next derivative>
    <etc.>

    return 0
End
```

The wave yw contains the present value, or estimated value, of Y[*i*] at X =xx. You may need this value to calculate the derivatives.

Your derivative function is called many times during the course of a solution, and it will be called at values of X that do not correspond to X values in the final solution. The reason for this is two-fold: First, the solution method steps from one value of X to another using estimates of the derivatives at several intermediate X values. Second, the spacing between X values that you want may be larger than can be calculated accurately, and Igor may need to find the solution at intermediate values. These intermediate values are not reported to you unless you call the **IntegrateODE** operation (see page V-452) in free-run mode.

Because the derivative function is called at intermediate X values, the yw wave is not the same wave as the Y wave you create and pass to IntegrateODE. Note that one row of your Y wave, or one value from each Y wave, corresponds to the elements of the one-dimensional yw wave that is passed in to your derivative function. While the illustration implies that values from your Y wave are passed to the derivative function, in fact the values in the yw wave passed into the derivative function correspond to whatever Y values the integrator needs at the moment. The correspondence to your Y wave or waves is only conceptual.

You should be aware that, with the exception of the parameter wave (`pw` above) the waves are not waves that exist in your Igor experiment. Do not try to resize them with InsertPoints/DeletePoints and don't do anything to them with the Redimension operation. The yw wave is input-only; altering it will not change anything. The dydx wave is output-only; the only thing you should do with it is to assign appropriate derivative (right-hand-side) values.

Some examples are presented in the following sections.

A First-Order Equation

Let's say you want a numerical solution to a simple first-order differential equation:

$$\frac{dy}{dx} = -ay$$

First you need to create a function that calculates the derivative. Enter the following in the procedure window:

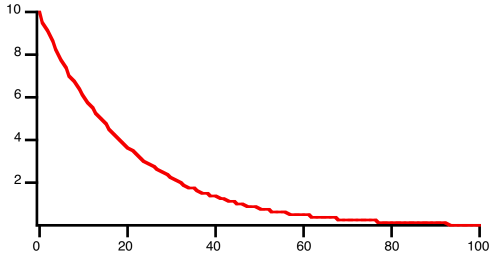
```
Function FirstOrder(pw, xx, yw, dydx)
   Wave pw        // pw[0] contains the value of the constant a
   Variable xx    // not actually used in this example
   Wave yw        // has just one element- there is just one equation
   Wave dydx      // has just one element- there is just one equation
```

```
    // There's only one equation, so only one expression here.
    // The constant a in the equation is passed in pw[0]
    dydx[0] = -pw[0]*yw[0]

    return 0
End
```

Paste the function into the procedure window and then execute these commands:

```
Make/D/O/N=101 YY     // wave to receive results
YY[0] = 10            // initial condition- y0=10
Display YY            // make a graph
Make/D/O PP={0.05}    // set constant a to 0.05
IntegrateODE FirstOrder, PP, YY
```

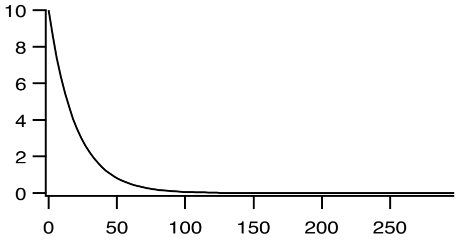
This results in the following graph with the expected exponential decay:



The IntegrateODE command shown in the example is the simplest you can use. It names the derivative function, `FirstOrder`, a parameter wave, `PP`, and a results wave, `YY`.

Because the IntegrateODE command does not explicitly set the X values, the output results are calculated according to the X scaling of the results wave `YY`. You can change the spacing of the X values by changing the X scaling of `YY`:

```
SetScale/P x 0,3,YY         // now the results will be at an x interval of 3
IntegrateODE FirstOrder, PP, YY
```



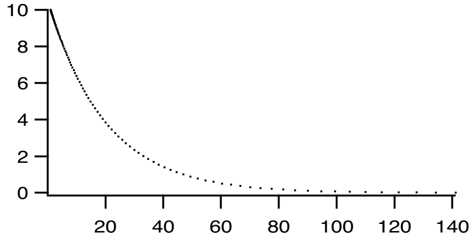
The same thing can be achieved by using your specified x0 and deltax with the /X flag:

```
IntegrateODE/X={0,3} FirstOrder, PP, YY
```

We presume that you have your own reasons for using the /X={x0, deltax} form. Note that when you do this, it doesn't use the X scaling of your Y wave. If you graph the Y wave the values on the X axis may not match the X values used during the calculation.

Finally, you don't have to use a constant spacing in X if you provide an X wave. You might want to do this to get closely-spaced values only where the solution changes rapidly. For instance:

```
Make/D/O/N=101 XX     // same length as YY
XX = exp(p/20)        // X values get farther apart as X increases
Display YY vs XX      // make an XY graph
ModifyGraph mode=2    // plot with dots so you can see the points
IntegrateODE/X=XX FirstOrder, PP, YY
```



Note that throughout these examples the initial value of YY has remained at 10.

A System of Coupled First-Order Equations

While many interesting systems are described by simple (possibly nonlinear) first-order equations, more interesting behavior results from systems of coupled equations.

The next example comes from chemical kinetics. Suppose you mix two substances A and B together in a solution and they react to form intermediate phase C. Over time C transforms into final product D:

$$A+B \underset{k_2}{\overset{k_1}{\rightleftarrows}} C \qquad\qquad C \underset{k_3}{\rightarrow} D$$

Here, k_1, k_2, and k_3 are rate constants for the reactions. The concentrations of the substances might be given by the following coupled differential equations:

$$\frac{dA}{dt} = \frac{dB}{dt} = -k_1 \cdot A \cdot B + k_2 \cdot C \qquad \frac{dC}{dt} = k_1 \cdot A \cdot B - k_2 \cdot C - k_3 \cdot C \qquad \frac{dD}{dt} = k_3 \cdot C$$

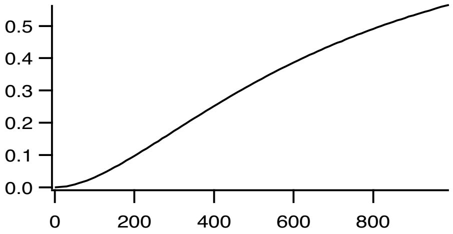
To solve these equations, first we need a derivative function:

```
Function ChemKinetic(pw, tt, yw, dydt)
   Wave pw           // pw[0] = k1, pw[1] = k2, pw[2] = k3
   Variable tt       // time value at which to calculate derivatives
   Wave yw           // yw[0]-yw[3] containing concentrations of A,B,C,D
   Wave dydt         // wave to receive dA/dt, dB/dt etc. (output)
   dydt[0] = -pw[0]*yw[0]*yw[1] + pw[1]*yw[2]
   dydt[1] = dydt[0]  // first two equations are the same
   dydt[2] = pw[0]*yw[0]*yw[1] - pw[1]*yw[2] - pw[2]*yw[2]
   dydt[3] = pw[2]*yw[2]

   return 0
End
```

We think that it is easiest to keep track of the results using a single multicolumn Y wave. These commands make a four-column Y wave and use dimension labels to keep track of which column corresponds to which substance:

```
Make/D/O/N=(100,4) ChemKin
SetScale/P x 0,10,ChemKin  // calculate concentrations every 10 s
SetDimLabel 1,0,A,ChemKin  // set dimension labels to substance names
SetDimLabel 1,1,B,ChemKin  // this can be done in a table if you make
SetDimLabel 1,2,C,ChemKin  // the table using edit ChemKin.ld
SetDimLabel 1,3,D,ChemKin
ChemKin[0][%A] = 1         // initial conditions: concentration of A
ChemKin[0][%B] = 1         // and B is 1, C and D is 0
ChemKin[0][%C] = 0         // note indexing using dimension labels
ChemKin[0][%D] = 0
Make/D/O KK={0.002,0.0001,0.004}    // rate constants
```

```
Display ChemKin[][%D]    // graph concentration of the product
                         // Note graph made with subrange of wave
IntegrateODE/M=1 ChemKinetic, KK, ChemKin
```



Note that the waves yw and dydt in the derivative function have four elements corresponding to the four equations in the system of ODEs. At a given value of X (or t) yw[0] and dydt[0] correspond to the first equation, yw[1] and dydt[1] to the second, etc.

Note also that we have used the /M=1 flag to request the Bulirsch-Stoer integration method. For well-behaved systems, it is likely to be the fastest method, taking the largest steps in the solution.

Optimizing the Derivative Function

The Igor compiler does no optimization of your code. Because IntegrateODE may call your function thousands (or millions!) of times, efficient code can significantly reduce the time it takes to calculate the solution. For instance, the `ChemKinetic` example function above was written to parallel the chemical equations to make the example clearer. There are three terms that appear multiple times. As written, these terms are calculated again from scratch each time they are encountered. You can save some computation time by precalculating these terms as in the following example:

```
Function ChemKinetic2(pw, tt, yw, dydt)
   Wave pw        // pw[0] = k1, pw[1] = k2, pw[2] = k3
   Variable tt    // time value at which to calculate derivatives
   Wave yw        // yw[0]-yw[3] containing concentrations of A,B,C,D
   Wave dydt      // wave to receive dA/dt, dB/dt etc. (output)

   // Calculate common subexpressions
   Variable t1mt2 = pw[0]*yw[0]*yw[1] - pw[1]*yw[2]
   Variable t3 = pw[2]*yw[2]

   dydt[0] =  -t1mt2
   dydt[1] = dydt[0]          // first two equations are the same
   dydt[2] = t1mt2 - t3
   dydt[3] = t3

   return 0
End
```

These changes reduced the time to compute the solution by about 13 per cent. Your mileage may vary. Larger functions with subexpression repeated many times are prime candidates for this kind of optimization.

Note also that IntegrateODE updates the display every time 10 result values are calculated. Screen updates can be very time-consuming, so IntegrateODE provides the /U flag to control how often the screen is updated. For timing this example we used /U=1000000 which effectively turned off screen updating.

Higher Order Equations

Not all differential equations (in fact, not many) are expressed as systems of coupled first-order equations, but IntegrateODE can only handle such systems. Fortunately, it is always possible to make substitutions to transform an Nth-order differential equation into N first-order coupled equations.

You need one equation for each order. Here is the equation for a forced, damped harmonic oscillator (using y for the displacement rather than x to avoid confusion with the independent variable, which is t in this case):

$$\frac{d^2y}{dt^2} + 2\lambda\frac{dy}{dt} + \omega^2 y = F(t)$$

If we define a new variable *v* (which happens to be velocity in this case):

$$v = \frac{dy}{dt}$$

Then

$$\frac{d^2y}{dt^2} = \frac{dv}{dt}$$

Substituting into the original equation gives us two coupled first-order equations:

$$\frac{dv}{dt} = -2\lambda v - \omega^2 y + F(t)$$

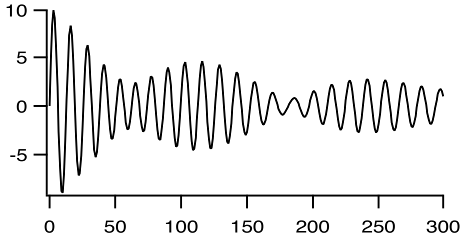
$$\frac{dy}{dt} = v$$

Of course, a real implementation of these equations will have to provide something for F(t). A derivative function to implement these equations might look like this:

```
Function Harmonic(pw, tt, yy, dydt)
   Wave pw      // pw[0]=damping, pw[1]=undamped frequency
                // pw[2]=Forcing amplitude, pw[3]=Forcing frequency
   Variable tt
   Wave yy      // yy[0] = velocity, yy[1] = displacement
   Wave dydt

   // simple sinusoidal forcing
   Variable Force = pw[2]*sin(pw[3]*tt)

   dydt[0] = -2*pw[0]*yy[0] - pw[1]*pw[1]*yy[1]+Force
   dydt[1] = yy[0]

   return 0
End
```

And the commands to integrate the equations:

```
Make/D/O/N=(300,2) HarmonicOsc
SetDimLabel 1,0,Velocity,HarmonicOsc
SetDimLabel 1,1,Displacement,HarmonicOsc
HarmonicOsc[0][%Velocity] = 5          // initial velocity
HarmonicOsc[0][%Displacement] = 0      // initial displacement
Make/D/O HarmPW={.01,.5,.1,.45}        // damping, freq, forcing amp and freq
Display HarmonicOsc[][%Displacement]
IntegrateODE Harmonic, HarmPW, HarmonicOsc
```



Free-Run Mode

Most of the examples shown so far use the Y wave's X scaling to set the X values where a solution is desired. In the section **A First-Order Equation** on page III-325, examples are also shown in which the /X flag is used to specify the sequence of X values, either by setting X0 and deltaX or by supplying a wave filled with X values.

These methods have the advantage that you have complete control over the X values where the solution is reported to you. They also are completely deterministic — you know before running IntegrateODE exactly how many points will be calculated and how big your waves need to be.

They also have the potential drawback that you may force IntegrateODE to use smaller X increments than required. If your ODE system is expensive to calculate, this may exact a considerable cost in computation time.

IntegrateODE also offers a "free-run" mode in which the solution is allowed to proceed using whatever X increments are required to achieve the requested accuracy limit. This mode has two possible advantages — it will use the minimum number of solution steps required and it may also produce a higher density of points in areas where the solution changes rapidly (but watch out for stiff systems, see page III-331).

Free-run mode has the disadvantage that in certain cases the solution may require miniscule steps to tip toe through difficult terrain, inundating you with huge numbers of points that you don't really need. You also don't know ahead of time how many points will be required to cover a certain range in X.

To illustrate the use of free-run mode, we will return to the example used in the section **A First-Order Equation** on page III-325. (Make sure the FirstOrder function is compiled in the procedure window.) Because we don't know how many points will be produced, we will make the waves large:

```
Make/D/O/N=1000 FreeRunY      // wave to receive results
FreeRunY = NaN
FreeRunY[0] = 10              // initial condition- y0=10
```

Free-run mode requires that you supply an X wave. Unlike the previous use of an X wave, in free-run mode the X wave is filled by IntegrateODE with the X values at which solution values have been calculated. Like the Y waves, you must provide an initial value in the first row of the X wave. As before, it must have the same number of rows as the Y waves:

```
Make/O/D/N=1000 FreeRunX      // same length as YY
FreeRunX = NaN                // prevent display of extra points
FreeRunX[0] = 0               // initial value of X
```

In free-run mode, only the points that are required are altered. Thus, if you have some preexisting wave contents, they will be seen on a graph. We prevent the resulting confusion by filling the X wave with NaN's (Not a Number, or blanks). Igor graphs do not display points that have NaN values.

Make a graph:

```
Display FreeRunY vs FreeRunX      // make an XY graph
ModifyGraph mode=3, marker=19     // plot with dots to show the points
```

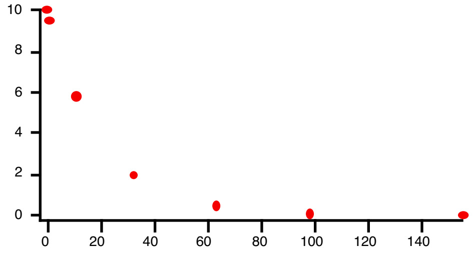
Make the parameter wave and set the value of the equation's lone coefficient:

```
Make/D/O PP={0.05}      // set constant a to 0.05
```

And finally do the integration in free-run mode. The /XRUN flag specifies a suggested first step size and the maximum X value. When the solution passes the maximum X value (100 in this case) or when your waves are filled, IntegrateODE will stop.

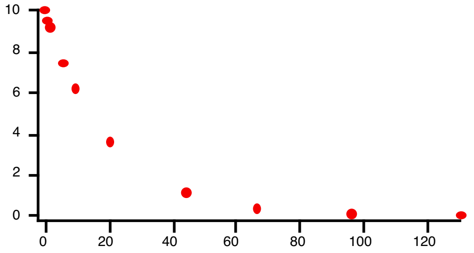
```
FreeRunX = NaN;FreeRunX[0] = 0
IntegrateODE/M=1/X=FreeRunX/XRUN={1,100} FirstOrder, PP, FreeRunY
```

In the earlier example, we (rather arbitrarily) chose 100 steps to make a reasonably smooth plot. In this case, it took 6 steps to cover the same X range, and the steps are closest together at the beginning where the exponential decay is most rapid:



Asking for more accuracy will cause smaller steps to be taken (9 when we executed the following command):

```
FreeRunX = NaN;FreeRunX[0] = 0
IntegrateODE/M=1/X=FreeRunX/XRUN={1,100}/E=1e-14 FirstOrder, PP, FreeRunY
```



After IntegrateODE has finished, you can use Redimension and the V_ODETotalSteps variable to adjust the size of the waves to just the points actually calculated:

```
Redimension/N=(V_ODETotalSteps+1) FreeRunY, FreeRunX
```

Note that we added 1 to V_ODETotalSteps to account for the initial value in row zero.

Stiff Systems

Some systems of differential equations involve components having very different time (or decay) constants. This can create what is called a “stiff” system; even though the short time constant decays rapidly and contributes negligibly to the solution after a very short time, ordinary solution methods (/M = 0, 1, and 2) are unstable because of the presence of the short time-constant component. IntegrateODE offers the Backward Differentiation Formula method (BDF, flag /M=3) to handle stiff systems.

A rather artificial example is the system (see “Numerical Recipes in C”, edition 2, page 734; see **References** on page III-349)

```
du/dt = 998u + 1998v
dv/dt = -999u - 1999v
```

Here is the derivative function that implements this system:

```
Function StiffODE(pw, tt, yy, dydt)
   Wave pw      // not actually used because the coefficients
                // are hard-coded to give a stiff system
   Variable tt
   Wave yy
   Wave dydt

   dydt[0] = 998*yy[0] + 1998*yy[1]
   dydt[1] = -999*yy[0] - 1999*yy[1]

   return 0
End
```

Commands to set up the wave required and to make a suitable graph:

```
Make/D/O/N=(3000,2) StiffSystemY
Make/O/N=0 dummy              // dummy coefficient wave
StiffSystemY = 0
StiffSystemY[0][0] = 1        // initial condition for u component
make/D/O/N=3000 StiffSystemX
Display StiffSystemY[][0] vs StiffSystemX
AppendToGraph/L=vComponentAxis StiffSystemY[][1] vs StiffSystemX

// make a nice-looking graph with dots to show where the solution points are
ModifyGraph axisEnab(left)={0,0.48},axisEnab(vComponentAxis)={0.52,1}
DelayUpdate
ModifyGraph freePos(vComponentAxis)={0,kwFraction}
ModifyGraph mode=2,lsize=2,rgb=(0,0,65535)
```

These commands solve this system using the Bulirsch-Stoer method using free run mode to minimize the number of solution steps computed:

```
StiffSystemX = nan      // hide unused solution points
StiffSystemX[0] = 0     // initial X value
IntegrateODE/M=1/X=StiffSystemX/XRUN={1e-6, 2} StiffODE, dummy, StiffSystemY
Print "Required ",V_ODETotalSteps, " steps to solve using Bulirsch-Stoer"
```

which results in this message in the history area:

```
  Required   401 steps to solve using Bulirsch-Stoer
```

These commands solve this system using the BDF method:

```
StiffSystemX = nan   // hide unused solution points
StiffSystemX[0] = 0  // initial X value
IntegrateODE/M=3/X=StiffSystemX/XRUN={1e-6, 2} StiffODE, dummy, StiffSystemY
Print "Required ",V_ODETotalSteps, " steps to solve using BDF"
```

This results in this message in the history area:

```
  Required   133 steps to solve using BDF
```

The difference between 401 steps and 133 is significant! Be aware, however, that the BDF method is not the most efficient for nonstiff problems.

Error Monitoring

To achieve the fastest possible solution to your differential equations, Igor uses algorithms with adaptive step sizing. As each step is calculated, an estimate of the truncation error is also calculated and compared to a criterion that you specify. If the error is too large, a smaller step size is used. If the error is small compared to what you asked for, a larger step size is used for the next step.

Igor monitors the errors by scaling the error by some (hopefully meaningful) number and comparing to an error level.

The Runge-Kutta and Bulirsch-Stoers methods (IntegrateODE flag /M=0 or /M=1) estimates the errors for each of your differential equations and the largest is used for the adjustments:

$$Max\left(\frac{Error_i}{Scale_i}\right) < eps$$

The Adams-Moulton and BDF methods (IntegrateODE flag /M=2 or /M=3) estimate the errors and use the root mean square of the error vector:

$$\left[\frac{1}{N}\sum\left(\frac{Error_i}{Scale_i}\right)^2\right]^{1/2} < eps\,.$$

Igor sets *eps* to 10^{-6} by default. If you want a different error level, use the /E=*eps* flag to set a different value of *eps*. Using the harmonic oscillator example, we now set a more relaxed error criterion than the default:

```
IntegrateODE/E=1e-3 Harmonic, HarmPW, HarmonicOsc
```

The error scaling can be composed of several parts, each optional:

$$Scale_i = h\cdot(C_i + y_i + dy_i/dx)$$

By default Igor uses constant scaling, setting *h*=1 and C_i=1, and does not use the y_i and dy_i/dx terms making $Scale_i$ = 1. In that case, *eps* represents an absolute error level: the error in the calculated values should be less than *eps*. An absolute error specification is often acceptable, but it may not be appropriate if the output values are of very different magnitudes.

You can provide your own customized values for C_i using the /S=*scaleWave* flag. You must first create a wave having one point for each differential equation. Fill it with your desired scaling values, and add the /S flag to the IntegrateODE operation:

```
Make/O/D errScale={1,5}
IntegrateODE/S=errScale Harmonic, HarmPW, HarmonicOsc
```

Typically, the constant values should be selected to be near the maximum values for each component of your system of equations.

Finally, you can control what Igor includes in $Scale_i$ using the /F=*errMethod* flag. The argument to /F is a bitwise value with a bit for each component of the equation above:

errMethod	**What It Does**
1	Add a constant C_i from *scaleWave* (or 1's if no *scaleWave*).
2	Add the current value of y_i's, the calculated result.
4	Add the current value of the derivatives, dy_i/dx.
8	Multiply by *h*, the current step size

Use *errMethod* = 2 if you want the errors to be a fraction of the current value of Y. That might be appropriate for solutions that asymptotically approach zero when you need smaller errors as the solution approaches zero.

The Scale numbers can never equal zero, and usually it isn't appropriate for $Scale_i$ to get very small. Thus, it isn't usually a good idea to use *errMethod* = 2 with solutions that pass through zero. A good way to avoid this problem can be to add the values of the derivatives (*errMethod* = (2+4)), or to add a small constant:

```
Make/D errScale=1e-6
IntegrateODE/S=errScale/F=(2+1) …
```

Finally, in some cases you need the much more stringent requirement that the errors be less than some global value. Since the solutions are the result of adding up myriad sequential solutions, any truncation error has the potential to add up catastrophically if the errors happen to be all of the same sign. If you are using Runge-Kutta and Bulirsch-Stoers methods (IntegrateODE flag /M=0 or /M=1), you can achieve global error limits by setting bit 3 of *errMethod* (/F=8) to multiply the error by the current step size (h in the equation above). If you are using Adams-Moulton and BDF methods (IntegrateODE flag /M=2 or /M=3) bit 3 does nothing; in that case, a conservative value of *eps* would be needed.

Higher accuracy will make the solvers use smaller steps, requiring more computation time. The trade-off for smaller step size is computation time. If you get too greedy, the step size can get so small that the X increments are smaller than the computer's digital resolution. If this happens Igor will stop the calculation and complain.

Solution Methods

Igor makes four solution methods available, Runge-Kutta-Fehlberg, Bulirsch-Stoers with Richardson extrapolation, Adams-Moulton and Backward Differentiation Formula.

Runge-Kutta-Fehlberg is a robust method capable of surviving solutions or derivatives that aren't smooth, or even have discontinuous derivatives. Bulirsch-Stoers, for well-behaved systems, will take larger steps than Runge-Kutta-Fehlberg, so it may be considerably faster. Step size for a given problem is larger so you get greater accuracy. Of course, if you ask for values closer together than the achievable step size, you get no advantage from this.

Details of these methods can be found in the second edition of *Numerical Recipes*(see **References** on page III-349).

The Adams-Moulton and Backward Differentiation Formula (BDF) methods are adapted from the CVODE package developed at Lawrence Livermore National Laboratory. In our very limited experience, for well-behaved nonstiff systems the Bulirsch-Stoers method is much more efficient than either the Runge-Kutta-Fehlberg method or the Adams-Moulton method, in that it requires significantly fewer steps for a given problem.

As shown above, stiff systems benefit greatly by the use of the BDF method. However, for nonstiff methods, it is not as efficient as the other methods.

See **IntegrateODE** on page V-452 for references on these methods.

Interrupting IntegrateODE

Numerical solutions to differential equations can require considerable computation and, therefore, time. If you find that a solution is taking too long you can abort the operation by clicking the Abort button in the status bar. You may need to press and hold to make sure IntegrateODE notices.

When you abort an integration, IntegrateODE returns whatever results have been calculated. If those results are useful, you can restart the calculation from that point, using the last calculated result row as the initial conditions. Use the /R=(*startX*) flag to specify where you want to start.

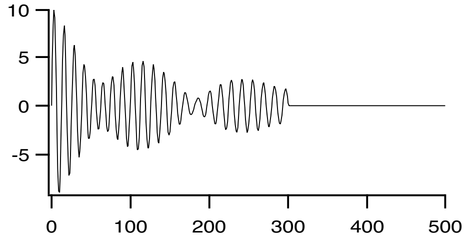
For Igor programmers, the V_ODEStepCompleted variable will be set to the last result. It is probably a good idea to restart a step or two before that:

```
IntegrateODE/R=(V_ODEStepCompleted-1) …
```

Stopping and Restarting IntegrateODE

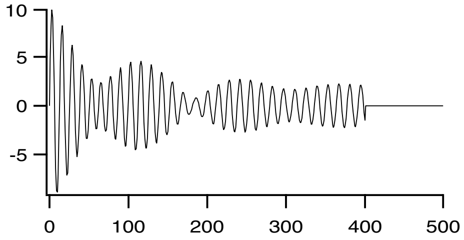
Any result can be used as initial conditions for a new solution. Thus, you can use the /R flag to calculate just a part of the solution, then finish later using the /R flag to pick up where you left off. For instance, using the harmonic oscillator example:

```
Make/D/O/N=(500,2) HarmonicOsc = 0
SetDimLabel 1,0,Velocity,HarmonicOsc
SetDimLabel 1,1,Displacement,HarmonicOsc
HarmonicOsc[0][%Velocity] = 5           // initial velocity
HarmonicOsc[0][%Displacement] = 0       // initial displacement
Make/D/O HarmPW={.01,.5,.1,.45}      // damping, freq, forcing amp and freq
Display HarmonicOsc[][%Displacement]

IntegrateODE/M=1/R=[,300] Harmonic, HarmPW, HarmonicOsc
```



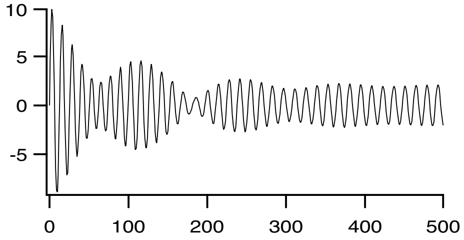
The calculation has been done for points 0-300. Note the comma in /R=[,300], which sets 300 as the end point, not the start point. Now you can restart at 300 and continue to the 400th point:

```
IntegrateODE/M=1/R=[300,400] Harmonic, HarmPW, HarmonicOsc
```



or finish the entire 500 points. Perhaps you need to start from an earlier point:

```
IntegrateODE/M=1/R=[350] Harmonic, HarmPW, HarmonicOsc
```



Stopping IntegrateODE on a Condition

Sometimes it is useful to be able to stop the calculation based on output values from the integration, rather than stopping when a certain value of the independent variable is reached. For instance, a common way to simulate a neuron firing is to solve the relevant system of equations until the output reaches a certain value. At that point, the solution should be stopped and the initial conditions reset to values appropriate to the triggered condition. Then the calculation can be re-started from that point.

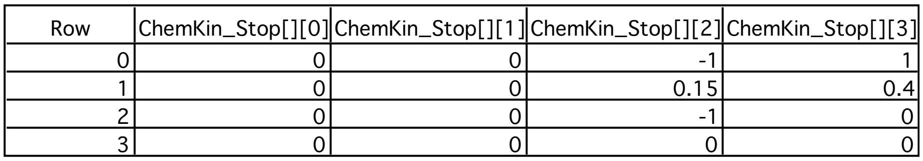
The ability to stop and re-start the calculation is a general solution to the problem of discontinuities in the system you are solving. Integrate the system up to the point of the discontinuity, stop and re-start using a derivative function that reflects the system after the discontinuity.

There are two ways to stop the integration depending on the solution values.

The first way is to use the /STOP={*stopWave*, *mode*} flag, supplying a *stopWave* containing stopping conditions. *StopWave* must have one column for each equation in your system. Each column can specify stopping on a value of the solution for the equation corresponding to the column, or stopping on a value of the derivative corresponding to that equation, or both. Each row has different significance:

Row	Meaning
0	Stop flag for solution value 0: Ignore condition on solution for this equation 1: Stop when solution value is greater than the value in row 1 -1: Stop when solution value is less than the value in row 1
1	Value of solution at which to stop
2	Stop flag for derivative 0: Ignore condition on derivative for this equation 1: Stop when derivative value is greater than the value in row 3 -1: Stop when derivative value is less than the value in row 3
3	Value of derivative at which to stop

In the chemical kinetics example above (see **A System of Coupled First-Order Equations** on page III-327) the system has four equations so you need a stop wave with four columns. This wave:



Row	ChemKin_Stop[][0]	ChemKin_Stop[][1]	ChemKin_Stop[][2]	ChemKin_Stop[][3]
0	0	0	-1	1
1	0	0	0.15	0.4
2	0	0	-1	0
3	0	0	0	0

will stop the integration when the concentration of species C (column 2) is less than 0.15, or when the concentration of species D (column 3) is greater than 0.4, or when the derivative of the concentration of species C is less than zero.

When you have multiple stopping criteria, as in this example, you can specify either OR stopping mode or AND stopping mode using the mode parameter of the /STOP flag. If *mode* is 0, OR mode is applied — any of the conditions with a non-zero flag will stop the integration. If *mode* is 1, AND mode is applied — all conditions with a non-zero flag must be satisfied in order for the integration to be stopped.

The second way to stop the integration is by returning a value of 1 from the derivative function. You can apply any condition you like in the function so it is possible to make much more complex stopping conditions this way than using the /STOP flag. However, the derivative function is called for a many intermediate points during a single step, some of which aren't necessarily even on the eventual solution trajectory. That means that you could be applying your stopping criterion to values that are not meaningful to the final solution. That may be particularly true at a time when the internal step size is contracting — the derivative function may be called for points beyond the eventual solution point as the solver tries a step size that doesn't succeed.

Integrating a User Function

You can use the Integrate1D function to numerically integrate a user function. For example, if you want to evaluate the integral

$$I(a, b) = \int_a^b \exp[-x^3 \sin(2\pi/x^2)]dx,$$

you need to start by defining the user function

```
Function userFunc(v)
   Variable v
   return exp(-v^3*sin(2*pi/v^2))
End
```

The Integrate1D function supports three integration methods: Trapezoidal, Romberg and Gaussian Quadrature.

```
Printf "%.10f\r" Integrate1D(userfunc,0.1,0.5,0)      // default trapezoidal
0.3990547412
Printf "%.10f\r" Integrate1D(userfunc,0.1,0.5,1)      // Romberg
0.3996269165
Printf "%.10f\r" Integrate1D(userfunc,0.1,0.5,2,100)  // Gaussian Q.
0.3990546953
```

For comparison, you can also execute:

```
Make/O/N=1000 tmp
Setscale/I x,.1,.5,"" tmp
tmp=userfunc(x)
Printf "%.10f\r" area(tmp,-inf,inf)
0.3990545084
```

Integrate1D can also handle complex valued functions. For example, if you want to evaluate the integral

$$I(a,b) = \int_a^b \exp\{ix\sin x\}dx'$$

a possible user function could be

```
Function/C cUserFunc(v)
   Variable v
   Variable/C arg=cmplx(0,v*sin(v))
   return exp(arg)
End
```

Note that the user function is declared with a /C flag and that Integrate1D must be assigned to a complex number in order for it to accept a complex user function.

```
Variable/C complexResult=Integrate1D(cUserFunc,0.1,0.2,1)
print complexResult
(0.0999693,0.00232272)
```

Also note that if you just try to print the result without using a complex variable as shown above, you need to use the /C flag with print:

```
Print/C Integrate1D(cUserFunc,0.1,0.2,1)
```

in order to force the function to integrate a complex valued expression.

You can also evaluate multidimensional integrals with the help of Integrate1D. The trick is in recognizing the fact that the user function can itself return a 1D integral of another function which in turn can return a 1D integral of a third function and so on. Here is an example of integrating a 2D function: $f(x,y) = 2x + 3y + xy$.

```
Function do2dIntegration(xmin,xmax,ymin,ymax)
   Variable xmin,xmax,ymin,ymax
   Variable/G globalXmin=xmin
   Variable/G globalXmax=xmax
   Variable/G globalY
   return Integrate1D(userFunction2,ymin,ymax,1)
End

Function userFunction1(inX)
   Variable inX
   NVAR globalY=globalY
   return (3*inX+2*globalY+inX*globalY)
End
```

```
Function userFunction2(inY)
   Variable inY
   NVAR globalY=globalY
   globalY=inY
   NVAR globalXmin=globalXmin
   NVAR globalXmax=globalXmax
   return Integrate1D(userFunction1,globalXmin,globalXmax,1)
End
```

Finding Function Roots

The **FindRoots** operation finds roots or zeros of a nonlinear function, a system of nonlinear functions, or of a polynomial with real coefficients.

Here we discuss how the operation works, and give some examples. The discussion falls naturally into three sections:

- Polynomial roots
- Roots of 1D nonlinear functions
- Roots of systems of multidimensional nonlinear functions

Igor's FindRoots operation finds function zeroes. Naturally, you can find other solutions as well. If you have a function f(x) and you want to find the X values that result in f(x) = 1, you would find roots of the function g(x) = f(x)-1. The FindRoots operation provides the /Z flag to make this more convenient.

A related problem is to find places in a curve defined by data points where the data pass through zero or another value. In this case, you don't have an analytical expression of the function. For this, use either the **FindLevel** operation (see page V-242) or the **FindLevels** operation (see page V-244); applications of these operations are discussed under **Level Detection** on page III-287.

Roots of Polynomials with Real Coefficients

The FindRoots operation can find all the complex roots of a polynomial with real coefficients. As an example, we will find the roots of

$x^4 -3.75x^2 - 1.25x + 1.5$

We just happen to know that this polynomial can be factored as (x+1)(x-2)(x+1.5)(x-0.5) so we already know what the roots are. But let's use Igor to do the work.

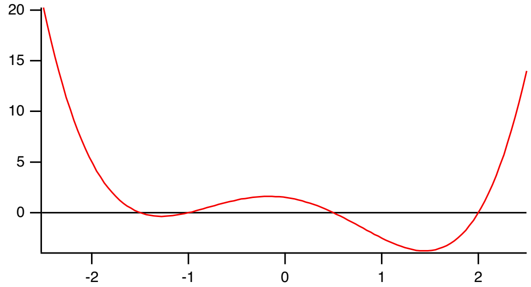
First, we need to make a wave with the polynomial coefficients. The wave must have N+1 points, where N is the degree of the polynomial. Point zero is the coefficient for the constant term, the last point is the coefficient for the highest-order term:

```
Make/D/O PolyCoefs = {1.5, -1.25, -3.75, 0, 1}
```

This wave can be used with the **poly** function to generate polynomial values. For instance:

```
Make/D/O PWave               // a wave with 128 points
SetScale/I x -2.5,2.5,PWave  // give it an X range of (-2.5, 2.5)
PWave = Poly(PolyCoefs, x)   // fill it with polynomial values

Display PWave                // and make a graph of it
ModifyGraph zero(left)=1     // add a zero line to show the roots
```

These commands make the following graph:



Now use FindRoots to find the roots:

```
FindRoots/P=PolyCoefs    // roots are now in W_polyRoots
Print W_polyRoots
```

This prints:

```
W_polyRoots[0] = {cmplx(0.5,0), cmplx(-1,0), cmplx(-1.5,0), cmplx(2,0)}
```

Note that the imaginary part of the roots are zero, because this polynomial was constructed from real factors. In general, this won't be the case.

The FindRoots operation uses the Jenkins-Traub algorithm for finding roots of polynomials:

Jenkins, M.A., "Algorithm 493, Zeros of a Real Polynomial", *ACM Transactions on Mathematical Software, 1,* 178-189, 1975, used by permission of ACM (1998).

Roots of a 1D Nonlinear Function

Unlike the case with polynomials, there is no general method for finding all the roots of a nonlinear function. Igor searches for a root of the function using Brent's method, and, depending on circumstances will find one or two roots in one shot.

You must write a user-defined function to define the function whose roots you want to find. Igor calls your function with values of X in the process of searching for a root. The format of the function is as follows:

```
Function myFunc(w,x)
   Wave w
   Variable x

   return <an arithmetic expression>
End
```

The wave *w* is a coefficient wave — it specifies constant coefficients that you may need to include in the function. It provides a convenient way to alter the coefficients so that you can find roots of members of a function family without having to edit your function code every time. Igor does not alter the values in w.

As an example we will find roots of the function y = a+b*sinc(c*(x-x0)). Here is a user-defined function to implement this:

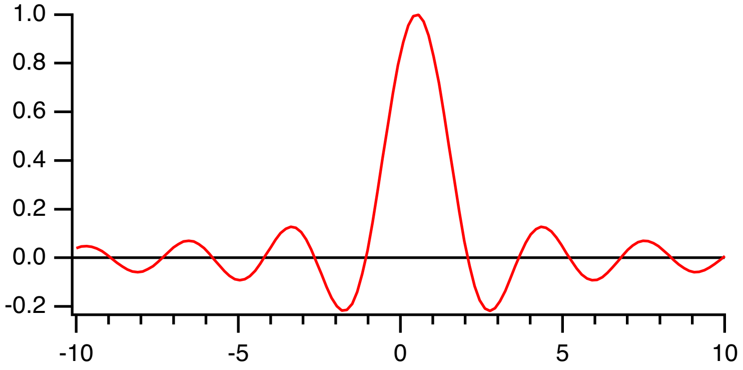
```
Function mySinc(w, x)
   Wave w
   Variable x

   return w[0]+w[1]*sinc(w[2]*(x-w[3]))
End
```

Enter this code into the Procedure window and then close the window.

Make a graph of the function:

```
Make/D/O SincCoefs={0, 1, 2, .5}     // a sinc function offset by 0.5
Make/D/O SincWave                    // a wave with 128 points
```

```
SetScale/I x -10,10,SincWave          // give it an X range of (-2, 2)
SincWave = mySinc(SincCoefs, x)       // fill it with function values

Display SincWave                      // and make a graph of it
ModifyGraph zero(left)=1              // add a zero line to show the roots
ModifyGraph minor(bottom)=1           // add minor ticks to the graph
```



Now we're ready to find roots.

The algorithm for finding roots requires that the roots first be bracketed, that is, you need to know two X values that are on either side of the root (that is, that give Y values of opposite sign). Making a graph of the function as we did here is a good way to figure out bracketing values. For instance, inspection of the graph above shows that there is a root between x=1 and x=3. The FindRoots command line to find a root in this range is

```
FindRoots/L=1/H=3 mySinc, SincCoefs
```

The /L flag sets the lower bracket and the /H flag sets the upper bracket. Igor then finds the root between these values, and prints the results in the history:

```
Possible root found at 2.0708
Y value there is -1.46076e-09
```

Igor reports to you both the root (in this case 2.0708) and the Y value at the root, which should be very close to zero (in this case it is -1.46x10^{-9}). Some pathological functions can fool Igor into thinking it has found a root when it hasn't. The Y value at the supposed root should identify if this has happened.

The bracketing values don't actually have to be at points of opposite sign. If the bracket encloses an extreme point, Igor will find it and then find two roots. Thus, you might use this command:

```
FindRoots/L=0/H=5 mySinc, SincCoefs
```

The Y values at X=0 and X=5 are both positive. Igor finds the minimum point at about X=2.7 and then uses that with the original bracketing values as the starting points for finding two roots. The result, printed in the history:

```
Looking for two roots...

Results for first root:
Possible root found at 2.0708
Y value there is -2.99484e-11

Results for second root:
Possible root found at 3.64159
Y value there is 3.43031e-11
```

Finally, it isn't always necessary to provide bracketing values. If the /L and /H flags are absent, Igor assigns them the values 0 and 1. In this case, there is no root between X=0 and X=1, and there is no extreme point. So Igor searches outward in a series of expanding jumps looking for values that will bracket a root (that is, for X values having Y values of opposite sign). Thus, in this case, the following simple command works:

```
FindRoots mySinc, SincCoefs
```

Igor finds the first of the roots we found previously:

```
Possible root found at 2.0708
Y value there is 2.748e-13
```

You may have noticed by now that FindRoots is reporting Y values that are merely small instead of zero. It isn't usually possible to find exact roots with a computer. And asking for very high accuracy requires more iterations of the search algorithm. If function evaluation is time-consuming and you don't need much accuracy, you may not want to find the root with high accuracy. Consequently, you can use the /T flag to alter the acceptable accuracy:

```
FindRoots/T=1e-3 mySinc, SincCoefs     // low accuracy
Possible root found at 2.07088
   Y value there is -5.5659e-05
Print/D V_root, V_YatRoot
   2.07088376061435  -5.56589998578327e-05

FindRoots/T=1e-15 mySinc, SincCoefs    // high accuracy
Possible root found at 2.0708
   Y value there is 0
Print/D V_root, V_YatRoot
   2.0707963267949  0
```

This also illustrates another point: the results of FindRoots are stored in variables. We used these variables in this case to print the results to higher accuracy than the six digits used by the report printed by FindRoots.

Roots of a System of Multidimensional Nonlinear Functions

Finding roots of a system of multidimensional nonlinear functions works very similarly to finding roots of a 1D nonlinear function. You provide user-defined functions that define your functions. These functions have nearly the same form as a 1D function, but they have a parameter for each independent variable. For instance, if you are going to find roots of a pair of 2D functions, the functions will look like this:

```
Function myFunc1(w, x1, x2)
   Wave w
   Variable x1, x2

   return <an arithmetic expression>
End

Function myFunc2(w, x1, x2)
   Wave w
   Variable x1, x2

   return <an arithmetic expression>
End
```

These function look just like the 1D function mySinc we wrote in the previous section, but they have two input X variables, one for each dimension. The number of functions must match the number of dimensions.

We will use the functions

```
f1= w[0]*sin((x-3)/w[1])*cos(y/w[2])
```

and

```
f2 = w[0]*cos(x/w[1])*tan((y+5)/w[2])
```

Enter this code into your procedure window:

```
Function myf1(w, xx, yy)
   Wave w
   Variable xx,yy
   return w[0]*sin(xx/w[1])*cos(yy/w[2])
End

Function myf2(w, xx, yy)
   Wave w
   Variable xx,yy

   return w[0]*cos(xx/w[1])*tan(yy/w[2])
End
```

Before starting, let’s make a contour plot to see what we’re up against. Here are some commands to make a convenient one:

```
Make/D/O params2D={1,5,4}           // nice set of parameters for both f1 and f2
Make/D/O/N=(50,50) f1Wave, f2Wave   // matrix waves for contouring
SetScale/I x -20,20,f1Wave, f2Wave  // nice range of X values
SetScale/I y -20,20,f1Wave, f2Wave  // and Y values
f1Wave = myf1(params2D, x, y)       // fill f1Wave with values from f1(x,y)
f2Wave = myf2(params2D, x, y)       // fill f2Wave with values from f2(x,y)
Display /W=(5,42,399,396)           // graph window for contour plot
AppendMatrixContour f1Wave
AppendMatrixContour f2Wave
ModifyContour f2Wave labels=0       // suppress contour labels
ModifyContour f1Wave labels=0
ModifyContour f1Wave rgbLines=(65535,0,0)    // make f1 red
ModifyContour f2Wave rgbLines=(0,0,65535)    // make f2 blue
ModifyGraph lsize('f2Wave=0')=2              // make zero contours heavy
ModifyGraph lsize('f1Wave=0')=2
```

Places where the zero contours for the two functions cross are the roots we are looking for. In the contour plot you can see several, for instance the points (0,0) and (7.8, 6.4) are approximate roots.

The algorithm that searches for roots needs a starting point. You can specify this in the FindRoots command with the /X flag, or if you don’t use /X, Igor starts by default at the origin, $X_n = 0$. You must also specify both functions and a coefficient wave for each function. In this case we will use the same coefficient wave for each. The functions and coefficient waves are specified in pairs. Since we are looking for roots of two 2D functions, we have two function-wave pairs:

```
FindRoots myf1,params2D, myf2,params2D
```

Igor finds a root at the origin, and prints the results. The X,Y coordinates of the root are stored in the wave W_Root:

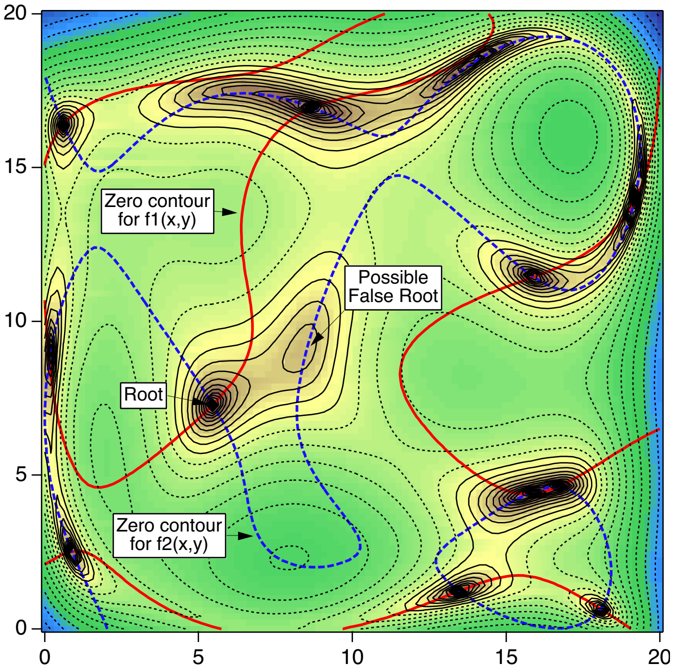
```
   Root found after 4 function evaluations.
W_Root={0,0}
   Function values at root:
W_YatRoot={0,0}
```

The wave W_YatRoot holds the values of each of the functions evaluated at the root.

If that’s not the root you want to find, use /X to specify a different starting point:

```
   FindRoots/X={7.7,6.3} myf1,params2D, myf2,params2D
   Root found after 47 function evaluations.
W_Root={-1.10261e-14,12.5664}
   Function values at root:
W_YatRoot={2.20522e-15,-1.89882e-15}
```

Caveats for Multidimensional Root Finding



Finding roots of multidimensional nonlinear functions is not straightforward. There is no general, foolproof way to do it. The method Igor uses is to search for minima in the sum of the squares of the functions. Since the squared values must be positive, the only places where this sum can be zero is at points where all the functions are zero at the same time. That point is a root, and it is also a minimum in the summed squares of the functions.

To find the zero points, Igor searches for local minima by travelling downhill from the starting point. Unfortunately, a local minimum doesn't have to be a root, it just has to be someplace where the sum of squares of the functions is less than surrounding points.

The adjacent graph shows how this can happen.

The two heavy lines are the zero contours for two functions (they happen to be fifth-order 2D polynomials). Where these zero contours cross are the roots for the system of the two functions.

The thin lines are contours of $f1(x,y)^2+f2(x,y)^2$, with dotted lines for high values; minima are surrounded by thin, solid contours. You can see that every intersection between the heavy zero contours is surrounded by thin contours showing that these are minima in the sum of the squared functions. One such point is labeled “Root”.

There is at least one point, labelled “False Root”, where there is a minimum but the zero contours don’t cross. That is not a root, but FindRoots may find it anyway. For instance, a real root:

```
FindRoots /x={3,6} MyPoly2d, nn1coefs, MyPoly2d, nn2coefs
   Root found after 11 function evaluations.
   W_Root={5.4623,7.28975}
   Function values at root:
   W_YatRoot={-4.15845e-13,1.08297e-12}
```

This point is the point marked “Root”. However:

```
FindRoots/x={9,10} MyPoly2d, nn1coefs, MyPoly2d, nn2coefs
   Root found after 52 function evaluations.
   W_Root={8.38701,9.10129}
   Function values at root:
   W_YatRoot={0.0686792,0.0129881}
```

You can see from the values in W_YatRoot that this is not a root. This point is marked “False root” on the figure above.

The polynomials used in this example have too many coefficients to be conveniently shown here. To see this example and others in action, try out the demo experiment. It is called “MD Root Finder Demo” and you will find it in your Igor Pro 7 folder, in the Examples:Analysis: folder.

Finding Minima and Maxima of Functions

The **Optimize** operation finds extreme values (maxima or minima) of a nonlinear function.

Here we discuss how the operation works, and give some examples. The discussion falls naturally into two sections:

- Extrema of 1D nonlinear functions
- Extrema of multidimensional nonlinear functions

A related problem is to find peaks or troughs in a curve defined by data points. In this case, you don't have an analytical expression of the function. To do this with one dimensional data, use the **FindPeak** operation (see page V-247).

Extreme Points of a 1D Nonlinear Function

The Optimize operation finds local maxima or minima of functions. That is, if a function has some X value where the nearby Y values are all higher than at that X value, it is deemed to be a minimum. Finding the point where a functions value is lower or higher than any other point anywhere is a much more difficult problem that is not addressed by the Optimize operation.

You must write a user-defined function to define the function for which the extreme points are calculated. Igor will call your function with values of X in the process of searching for a root. The format of the function is as follows:

```
Function myFunc(w,x)
   Wave w
   Variable x

   return f(x)          // an expression...
End
```

The wave w is a coefficient wave — it specifies constant coefficients that you may need to include in the function. It provides a convenient way to alter the coefficients so that you can find extreme points of members of a function family without having to edit your function code every time. Igor does not alter the values in *w*.

Although the coefficient wave must be present in the Function declaration, it does not have to be referenced in the function body. This may save computation time, arriving at the solution faster. You will have to create a dummy wave to list in the FindRoots command.

As an example we will find extreme points of the equation

```
y = a+b*sinc(c*(x-x0))
```

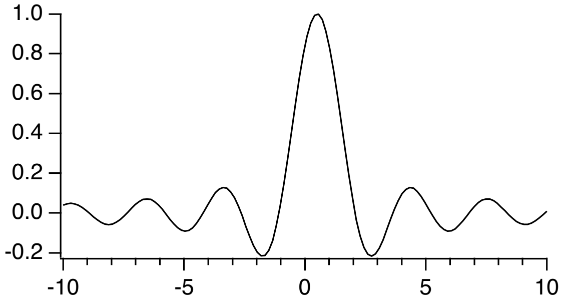
A suitable user-defined function might look like this:

```
Function mySinc(w, x)
   Wave w
   Variable x

   return w[0]+w[1]*sinc(w[2]*(x-w[3]))
End
```

Enter this code into the Procedure window and then close the window.

Make a graph of the function:

```
Make/D/O SincCoefs={0, 1, 2, .5} // a sinc function offset by 0.5
Make/D/O SincWave                // a wave with 128 points
SetScale/I x -10,10,SincWave     // give it an X range of [-10, 10]
SincWave = mySinc(SincCoefs, x)  // fill it with function values

Display SincWave                 // and make a graph of it
ModifyGraph minor(bottom)=1      // add minor ticks to the graph
```



Now we're ready to find extreme points.

The algorithm for finding extreme points requires that the extreme points first be bracketed, that is, you need to know two X values that are on either side of the extreme point (that is, two points that have a lower or higher point between). Making a graph of the function as we did here is a good way to figure out bracketing values. For instance, inspection of the graph above shows that there is a minimum between x=1 and x=4. The Optimize command line to find a minimum in this range is

```
Optimize/L=1/H=4 mySinc, SincCoefs
```

The /L flag sets the lower bracket and the /H flag sets the upper bracket. Igor then finds the minimum between these values, and prints the results in the history:

```
Optimize probably found a minimum. Optimize stopped because
the Optimize operation found a minimum within the specified tolerance.
Current best solution: 2.7467
Function value at solution: -0.217234
 13 iterations, 14 function calls
V_minloc = 2.7467, V_min = -0.217234, V_OptNumIters = 13, V_OptNumFunctionCalls = 14
```

Igor reports to you both the X value that minimizes the function (in this case 2.7467) and the Y value at the minimum.

The bracketing values don't necessarily have to bracket the solution. Igor first tries to find the desired extremum between the bracketing values. If it fails, the bracketing interval is expanded searching for a suitable bracketing interval. If you don't use /L and /H, Igor sets the bracketing interval to [0,1]. In the case of the mySinc function, that doesn't include a minimum. Here is what happens in that case:

```
Optimize mySinc, SincCoefs
```

```
Optimize probably found a minimum. Optimize stopped because
the Optimize operation found a minimum within the specified tolerance.
Current best solution: 2.74671
Function value at solution: -0.217234
 16 iterations, 47 function calls
V_minloc = 2.74671, V_min = -0.217234, V_OptNumIters = 16, V_OptNumFunctionCalls = 47
```

Note that Igor found the same minimum that it found before.

The mySinc function makes it easy to find bracketing values because of the oscillatory nature of the function. Other functions may be more difficult if they contain just one extreme point, or if they have local extreme points but are unbounded elsewhere. Even in an easy case like mySinc, you can't be sure which extreme point Igor will find, so it is always better to supply a good bracket if you possibly can.

You may wish to find maximum points instead of minima. Use the /A flag to specify this:

```
Optimize/A/L=0/H=2 mySinc, SincCoefs
```

```
Optimize probably found a maximum. Optimize stopped because
the Optimize operation found a maximum within the specified tolerance.
Current best solution: 0.499999
Function value at solution: 1
 16 iterations, 17 function calls
V_maxloc = 0.499999, V_max = 1, V_OptNumIters = 16, V_OptNumFunctionCalls = 17
```

The results of the Optimize operation are stored in variables. Note that the report that Optimize prints in the history includes only six digits of the values. You can print the results to greater precision using the printf operation and the variables:

```
Printf "The max is at %.15g. The Y value there is %.15g\r", V_maxloc, V_max
```

yields this in the history:

```
The max is at 0.499999480759779. The Y value there is 0.99999999999982
```

Extrema of Multidimensional Nonlinear Functions

Finding extreme points of multidimensional nonlinear functions works very similarly to finding extreme points of a 1D nonlinear function. You provide a user-defined function having almost the same format as for 1D functions. A 2D function will look like this:

```
Function myFunc1(w, x1, x2)
   Wave w
   Variable x1, x2

   return f1(x1, x2)        // an expression...
End
```

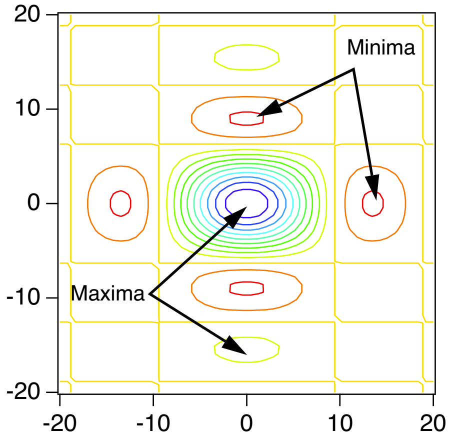
This function looks just like the 1D function mySinc we wrote in the previous section, but it has two input X variables, one for each dimension.

We will make a 2D function based on the sinc function. Copy this code into your procedure window:

```
Function Sinc2D(w, xx, yy)
   Wave w
   Variable xx,yy

   return w[0]*sinc(xx/w[1])*sinc(yy/w[2])
End
```

Before starting, let’s make a contour plot to see what we’re up against. Here are some commands to make a convenient one:

```
Make/D/O params2D={1,3,2}         // nice set of parameters
Make/D/O/N=(50,50) Sinc2DWave     // matrix wave for contouring
SetScale/I x -20,20,Sinc2DWave    // nice range of X values for these functions
SetScale/I y -20,20,Sinc2DWave    // and Y values
Sinc2DWave = Sinc2D(params2D, x, y) // fill f1Wave with values from f1(x,y)
Display /W=(5,42,399,396)         // graph window for contour plot
AppendMatrixContour Sinc2DWave
ModifyContour Sinc2DWave labels=0// suppress contour labels to reduce clutter
```



The algorithm that searches for extreme points needs a starting guess which you provide using the /X flag. Here is a command to find a minimum point:

```
Optimize/X={1,5} Sinc2D,params2D
```

This command results in the following report in the history:

```
Optimize probably found a minimum. Optimize stopped because
the gradient is nearly zero.
Current best solution:
   W_Extremum={-0.000147116,8.98677}
Function gradient:
   W_OptGradient={-4.65661e-08,1.11923e-08}
Function value at solution: -0.217234
11 iterations, 36 function calls
V_min = -0.217234, V_OptTermCode = 1, V_OptNumIters = 11, V_OptNumFunctionCalls = 36
```

Note that the minimum is reported via a wave called W_extremum. Had you specified a wave using the /X flag, that wave would have been used instead. Another wave, W_OptGradient, receives the function gradient at the solution point.

There are quite a few options available to modify the workings of the Optimize operation. See **Optimize** operation on page V-725 for details, including references to reading material.

Stopping Tolerances

The Optimize operation stops refining the solution when certain criteria are met. The most desirable result is that it stops because the function gradient is very close to zero, since that is (almost) diagnostic of an extreme point. The algorithm also will take very small steps near an extreme point, so this is also a stopping criterion. You can set the values for the stopping criteria using the /T={*gradTol*, *stepTol*} flag.

Optimize stops when these conditions are met:

$$\max_{1 \le i \le n} \left\{ |g_i| \cdot \frac{max(|x_i|, typX_i)}{max(|f|, funcSize)} \right\} \le gradTol$$

or

$$\max_{1 \le i \le n} \left\{ \frac{|\Delta x_i|}{max(x_i, typX_i)} \right\} \le stepTol$$

Note that these conditions use values of the gradient (g_i) and step size (Δx_i) that are scaled by a measure of the magnitude of values encountered in the problem. In these equations, x_i is the value of a component of the solution and *f* is the value of the function at the solution; $typX_i$ is a "typical" value of the X component that is set by you using the /R flag and *f* is a typical function value magnitude which you set using the /Y flag. The values of $typX_i$ and *f* are one if the /R and /F flags are not present.

The default values for *gradTol* and *stepTol* are {8.53618x10^{-6}, 7.28664x10^{-11}}. These are the values recommended by Dennis and Schnabel (see the references in **Optimize** operation on page V-725) for well-behaved functions when the function values have full double precision resolution. These values are (6.022x10^{-16})$^{1/3}$ and (6.022x10^{-16})$^{2/3}$ as suggested by Dennis and Schnabel (see the references in **Optimize** operation on page V-725), where 6.022x10^{-16} is the smallest double precision floating point number that, when added to 1, is different from 1. Usually the default is pretty good.

Due to floating point truncation errors, it is possible to set *gradTol* and *stepTol* to values that can never be achieved. In that case you may get a message about "no solution was found that is better than the last iteration".

Problems with Multidimensional Optimization

Finding minima of multidimensional functions is by no means foolproof. The methods used by the Optimize operation are "globally convergent" which means that under suitable circumstances Optimize will be able to find some extreme point from just about any starting guess.

If the gradient of your function is zero, or very nearly so at the starting guess, Optimize has no information on which way to go to find an extreme point. Note that the Sinc2D function has a maximum exactly at (0,0). Here is what happens if you try to find a minimum starting at the origin:

```
Optimize/X={0,0} Sinc2D,params2D
```

```
==== The Optimize operation failed to find a minimum. ====
Optimize stopped because
The function gradient at your starting guess is too near zero, suggesting that
it is a critical point.
A different starting guess usually solves this problem.
Current best solution:
   W_Extremum={0,0}
Function gradient:
   W_OptGradient={0,0}
Function value at solution: 1
0 iterations, 3 function calls
V_min = 1, V_OptTermCode = 6, V_OptNumIters = 0, V_OptNumFunctionCalls = 3
```

In this example the function gradient is zero at the origin because there is a function maximum there. A gradient of zero could also be a minimum or a saddle point.

The algorithms used by the Optimize operation assume that your function is smooth, that is, that the first and second derivatives are continuous. Optimize may work with functions that violate this assumption, but it is not guaranteed.

Although Optimize tends to look downhill to the nearest minimum (or uphill to the nearest maximum), it is not guaranteed to find any particular minimum, especially if your starting guess is near a point where the gradient is small. Sometimes using a different method (/M=(*stepMethod*, *hessianMethod*) will result in a different answer. You can limit the maximum step size to keep progress more or less local (/S=*maxStep*). If you set the maximum step size too small, however, Optimize may stop early because the maximum step size is exceeded too many times. Here is an example using the Sinc2D function. If the starting guess is near the origin, the gradient is small and the solution shoots off into the hinterlands (only a portion of the history report is shown):

```
Optimize/X={1,1} Sinc2D,params2D
   W_Extremum={-61.1192,298.438}
Function gradient:
   W_OptGradient={-1.04095e-08,-1.19703e-08}
```

Use /S to limit the step size, and find a minimum nearer to the starting guess:

```
Optimize/X={1,1}/S=10 Sinc2D,params2D
   W_Extremum={-0.00014911,8.9868}
Function gradient:
   W_OptGradient={9.31323e-09,5.92775e-08}
```

But if the maximum step is too small, it doesn't work:

```
Optimize/X={1,1}/S=1 Sinc2D,params2D
```

```
==== The Optimize operation failed to find a minimum. ====
Optimize stopped because
the maximum step size was exceeded in five consecutive iterations.
This can happen if the function is unbounded (there is no minimum),
or the function approaches the minimum asymptotically. It may also be that the
maximum step size (1) is too small.
```

Another way to get a solution near by is to set the initial trust region to a small value. This works if you select double dogleg or More Hebdon as the step selection method. It does not apply to the default line search method. Here is an example (note that the double dogleg method is selected using /M={1,0}):

```
Optimize/X={1,1}/M={1,0}/F=1 Sinc2D,params2D
   W_Extremum={-0.000619834,8.9872}
Function gradient:
   W_OptGradient={1.09896e-07,2.9017e-08}
```

References

The IntegrateODE operation is based on routines in *Numerical Recipes*, and are used by permission:

Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

The Adams-Moulton and BDF methods are based on the CVODE package developed at Lawrence Livermore National Laboratory:

Cohen, Scott D., and Alan C. Hindmarsh, *CVODE User Guide*, LLNL Report UCRL-MA-118618, September 1994.

The CVODE package was derived in part from the VODE package. The parts used in Igor are described in this paper:

Brown, P.N., G. D. Byrne, and A. C. Hindmarsh, VODE, a Variable-Coefficient ODE Solver, *SIAM J. Sci. Stat. Comput.*, *10*, 1038-1051, 1989.

The Optimize operation uses Brent's method for univariate functions. *Numerical Recipes* has an excellent discussion in section 10.2 of this method (but we didn't use their code).

For multivariate functions Optimize uses code based on Dennis and Schnabel. To truly understand what Optimize does, read their book:

Dennis, J. E., Jr., and Robert B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Methods*, 378 pp., Society for Industrial and Applied Mathematics, Philadelphia, 1996.

The FindRoots operation uses the Jenkins-Traub algorithm for finding roots of polynomials:

Jenkins, M.A., "Algorithm 493, Zeros of a Real Polynomial", *ACM Transactions on Mathematical Software*, *1*, 178-189, 1975..

Chapter

III-11

Image Processing

Overview

Image processing is a broad term describing most operations that you can apply to image data which may be in the form of a 2D, 3D or 4D waves. Image processing may sometimes provide the appropriate analysis tools even if the data have nothing to do with imaging. In Chapter II-16, **Image Plots**, we described operations relating to the display of images. Here we concentrate on transformations, analysis operations and special utility tools that are available for working with images.

You can use the IP Tutorial experiment (inside the Learning Aids folder in your Igor Pro 7 folder) in parallel with this chapter. The experiment contains, in addition to some introductory material, the sample images and most of the commands that appear in this chapter. To execute the commands you can select them in the Image Processing help file and press Control-Enter.

For a listing of all image analysis operations, see **Image Analysis** on page V-4.

Image Transforms

The two basic classes of image transforms are color transforms and grayscale/value transforms. Color transforms involve conversion of color information from one color space to another, conversions from color images to grayscale, and representing grayscale images with false color. Grayscale value transforms include, for example, pixel level mapping, mathematical and morphological operations.

Color Transforms

There are many standard file formats for color images. When a color image is stored as a 2D wave, it has an associated or implied colormap and the RGB value of every pixel is obtained by mapping values in the 2D wave into the colormap.

When the image is a 3D wave, each image plane corresponds to an individual red, green, or blue color component. If the image wave is of type unsigned byte (/B/U), values in each plane are in the range [0,255]. Otherwise, the range of values is [0,65535].

There are two other types of 3D image waves. The first consists of 4 layers corresponding to RGBA where the 'A' represents the alpha (transparency) channel. The second contains more than three planes in which case the planes are grayscale images that can be displayed using the command:

```
ModifyImage imageName plane=n
```

Multiple color images can be stored in a single 4D wave where each chunk corresponds to a separate RGB image.

You can find most of the tools for converting between different types of images in the **ImageTransform** operation. For example, you can convert a 2D image wave that has a colormap to a 3D RGB image wave. Here we create a 3-layer 3D wave named M_RGBOut from the 2D image named 'Red Rock' using RGB values from the colormap wave named 'Red RockCMap':

```
ImageTransform /C='Red RockCMap' cmap2rgb 'Red Rock'
NewImage M_RGBOut          // Resulting 3D wave is M_RGBOut
```

Note: The images in the IP Tutorial experiment are not stored in the root data folder, so many of the commands in the tutorial experiment include data folder paths. Here the data folder paths have been removed for easier reading. If you want to execute the commands you see here, use the commands in the “IP Tutorial Help” window. See Chapter II-8, **Data Folders**, for more information about data folders.

In many situations it is necessary to dispose of color information and convert the image into grayscale. This usually happens when the original color image is to be processed or analyzed using grayscale operations. Here is an example using the RGB image which we have just generated:

```
ImageTransform rgb2gray M_RGBOut
NewImage M_RGB2Gray           // Display the grayscale image
```

The conversion to gray is based on the YIQ standard where the gray output wave corresponds to the Y channel:

```
gray=0.299*red+0.587*green+0.114*blue.
```

If you wish to use a different set of transformation constants say {c_i}, you can perform the conversion on the command line:

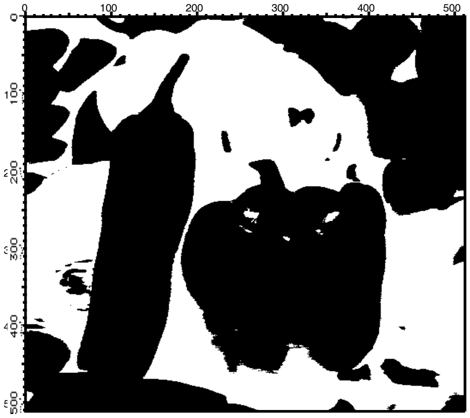
```
gray2DWave=c1*image[p][q][0]+c2*image[p][q][1]+c3*image[p][q][2]
```

For large images this operation may be slow. A more efficient approach is:

```
Make/O/N=3 scaleWave={c1,c2,c3}
ImageTransform/D=scaleWave scalePlanes image // Creates M_ScaledPlanes
ImageTransform sumPlanes M_ScaledPlanes
```

In some applications it is desirable to extract information from the color of regions in the image. We therefore convert the image from RGB to the HSL color space and then perform operations on the first plane (hue) of the resulting 3D wave. In the following example we convert the RGB image wave peppers into HSL, extract the hue plane and produce a binary image in which the red hues are nonzero.

```
ImageTransform /U rgb2hsl peppers// Note the /U for unsigned short result
MatrixOp/O RedPlane=greater(5000,M_RGB2HSL[][][0])+greater(M_RGB2HSL[][][0],60000)
NewImage RedPlane         // Here white corresponds to red hues in the source
```



As you can see, the resulting image is binary, with white pixels corresponding to regions where the original image was predominantly red. The binary image can be used to discriminate between red and other hue regions. The second command line above converts hue values that range from 0 to 65535 to 1 if the color is in the "reddish" range, or zero if it is outside that range. The selection of values below 5000 is due to the fact that red hues appear on both sides of 0° (or 360°) of the hue circle.

Hue based image segmentation is also supported through the **ImageTransform** operation (see page V-417) using the **hslSegment**, **matchPlanes** or **selectColor** keywords. The same operation also supports color space conversions from RGB to CIE XYZ (D65 based) and from XYZ to RGB. See also **Handling Color** on page III-379 and **HSL Segmentation** on page III-374.

Grayscale or Value Transforms

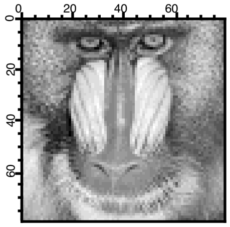
This class of transforms applies only to 2D waves or to individual layers of higher dimensional waves. They are called "grayscale" because 2D waves by themselves do not contain color information. We divide grayscale transforms into level mappings and mathematical operations.

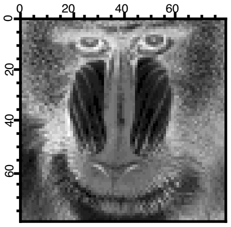
Explicit Lookup Tables

Here is an example of using an explicit lookup table (LUT) to create the negative of an image the hard way:

```
Make/B/U/O/N=256 negativeLookup=255-x     // Create the lookup table
Duplicate/O baboon negativeBaboon
negativeBaboon=negativeLookup[negativeBaboon]// The lookup transformation
```

```
NewImage baboon
NewImage negativeBaboon
```





In this example the negativeBaboon image is a derived wave displayed with standard linear LUT. You can also obtain the same result using the original baboon image but displaying it with a negative LUT:

```
NewImage baboon
Make/N=256 negativeLookup=1-x/255   // Negative slope LUT from 1 to 0
ModifyImage baboon lookup=negativeLookup
```

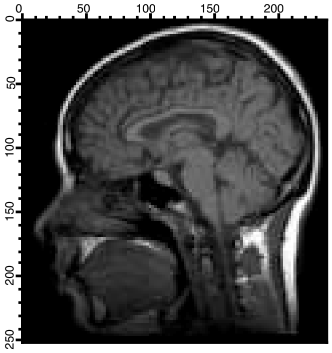
If you are willing to modify the original data you can execute:

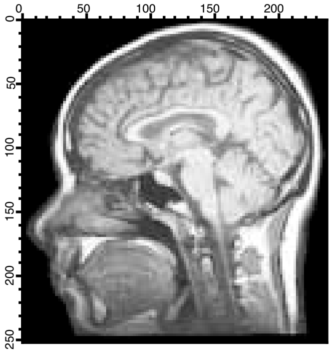
```
ImageTransform invert baboon
```

Histogram Equalization

Histogram equalization maps the values of a grayscale image so that the resulting values utilize the entire available range of intensities:

```
NewImage MRI
ImageHistModification MRI
NewImage M_ImageHistEq
```





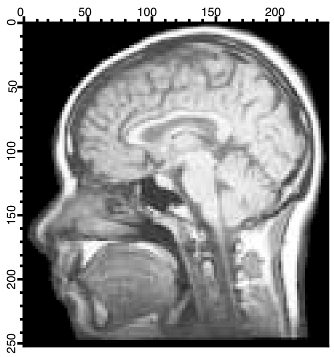
Adaptive Histogram Equalization

The **ImageHistModification** operation calculates a lookup table based on the cumulative histogram of the whole source image. The lookup table is then applied the output image. In cases where there are significant spatial variations in the histogram, a more local approach may be needed, i.e., perform the histogram equalization independently for different parts of the image and then combine the regional results by matching them across region boundaries. This is commonly referred to as "Adaptive Histogram Equalization".

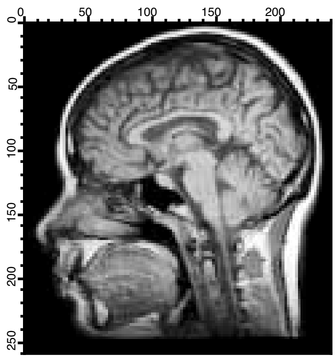
```
ImageHistModification MRI
Duplicate/O M_ImageHistEq, globalHist
NewImage globalHist
ImageTransform/N={2,7} padImage MRI // To make the image divisible
ImageHistModification/A/C=10/H=2/V=2 M_paddedImage
NewImage M_ImageHistEq
```

The original image is 238 by 253 pixels. Because the number of rows and columns must be divisible by the number of equalization intervals, we first padded the image using the ImageTransform **padImage**. The result is an image that is 240 by 260. If you do not find the resulting adaptive histogram sufficiently different from the global histogram equalization, you can increase the number of vertical and horizontal regions that are processed:

```
ImageHistModification/A/C=100/H=20/V=20 M_paddedImage
```



Global



Adaptive

You can now compare the global and adaptive histogram results. Note that the adaptive histogram performed better (increased contrast) over most of the image. The increase in the clipping value (/C flag) gave rise to a minor artifact around the boundary of the head.

Threshold

The threshold operation is an important member of the level mapping class. It converts a grayscale image into a binary image. A binary image in Igor is usually stored as a wave of type unsigned byte. While this may appear to be wasteful, it has advantages in terms of both speed and in allowing you to use some bits of each byte for other purposes (e.g., bits can be turned on or off for binary masking). The threshold operation, in addition to producing the binary thresholded image, can also provide a correlation value which is a measure of the threshold quality.

You can use the **ImageThreshold** operation (see page V-415) either by providing a specific threshold value or by allowing the operation to determine the threshold value for you. There are various methods for automatic threshold determination:

Iterated: Iteration over threshold levels to maximize correlation with the original image.

Bimodal: Attempts to fit a bimodal distribution to the image histogram. The threshold level is chosen between the two modal peaks.

Adaptive: Calculates a threshold for every pixel based on the last 8 pixels on the same scan line. It usually gives rise to drag lines in the direction of the scan lines. You can compensate for this artifact as we show in an example below.

Fuzzy Entropy: Considers the image as a fuzzy set of background and object pixels where every pixel may belong to a set with some probability. The algorithm obtains a threshold value by minimizing the fuzziness which is calculated using Shannon’s Entropy function.

Fuzzy Means: Minimizes a fuzziness measure that is based on the product of the probability that the pixel belongs in the object and the probability that the pixel belongs to the background.

Histogram Clusters: Determines an ideal threshold by histograming the data and representing the image as a set of clusters that is iteratively reduced until there are two clusters left. The threshold value is then set to the highest level of the lower cluster. This method is based on a paper by A.Z. Arifin and A. Asano (see reference below) but modified for handling images with relatively flat histograms. If the image histogram results in less than two clusters, it is impossible to determine a threshold using this method and the threshold value is set to NaN.

Variance: Determines the ideal threshold value by maximizing the total variance between the "object" and "background". See http://en.wikipedia.org/wiki/Otsu’s_method.

Each of the thresholding methods has its advantages and disadvantages. It is sometimes useful to try all the methods before you decide which method applies best to a particular class of images. The following example illustrates the different thresholding methods for an image of light gray blobs on a dark gray background (the “blobs” image in the IP Tutorial).

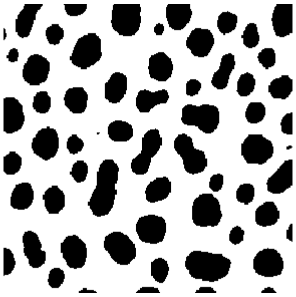
Threshold Examples

This section shows results using various methods for automatic threshold determination.

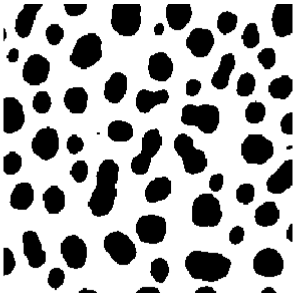
The commands shown were executed in the demo experiment that you can open by choosing File→Example Experiments→Tutorials→Image Processing Tutorial.

```
// User defined method
ImageThreshold/Q/T=128 root:images:blobs
Duplicate/O M_ImageThresh UserDefined
NewImage/S=0 UserDefined; DoWindow/T kwTopWin, "User-Defined Thresholding"

// Iterated method
ImageThreshold/Q/M=1 root:images:blobs
Duplicate/O M_ImageThresh iterated
NewImage/S=0 iterated; DoWindow/T kwTopWin, "Iterated Thresholding"
```

User Defined



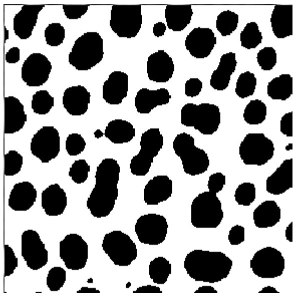
Iterated



```
// Bimodal method
ImageThreshold/Q/M=2 root:images:blobs
Duplicate/O M_ImageThresh bimodal
NewImage/S=0 bimodal; DoWindow/T kwTopWin, "Bimodal Thresholding"

// Adaptive method
ImageThreshold/Q/I/M=3 root:images:blobs
Duplicate/O M_ImageThresh adaptive
NewImage/S=0 adaptive; DoWindow/T kwTopWin, "Adaptive Thresholding"
```

Bimodal

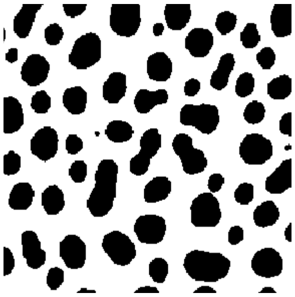
Adaptive



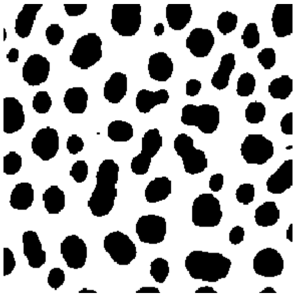
```
// Fuzzy-entropy method
ImageThreshold/Q/M=4 root:images:blobs
Duplicate/O M_ImageThresh fuzzyE
NewImage/S=0 fuzzyE; DoWindow/T kwTopWin, "Fuzzy Entropy Thresholding"

// Fuzzy-M method
ImageThreshold/Q/M=5 root:images:blobs
Duplicate/O M_ImageThresh fuzzyM
```

```
NewImage/S=0 fuzzyM; DoWindow/T kwTopWin, "Fuzzy Means Thresholding"
```

Fuzzy Entropy

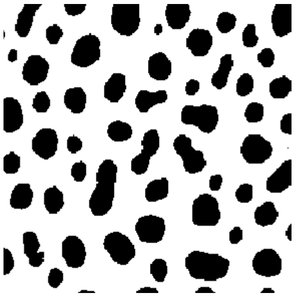


Fuzzy Means

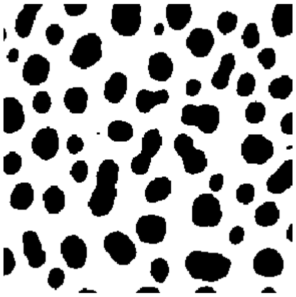


```
// Arifin and Asano method
ImageThreshold/Q/M=6 root:images:blobs
Duplicate/O M_ImageThresh A_and_A
NewImage/S=0 A_and_A; DoWindow/T kwTopWin, "Arifin and Asano Thresholding"

// Otsu method
ImageThreshold/Q/M=7 root:images:blobs
Duplicate/O M_ImageThresh otsu
NewImage/S=0 otsu ; DoWindow/T kwTopWin, "Otsu Thresholding"
```

Arifin and Asano



Otsu



In the these examples, you can add the /C flag to each ImageThreshold command and remove the /Q flag to get some feedback about the quality of the threshold; the correlation coefficient will be printed to the history.

It is easy to determine visually that, for this data, the adaptive and the bimodal algorithms performed rather poorly.

You can improve the results of the adaptive algorithm by running the adaptive threshold also on the transpose of the image, so that the operation becomes column based, and then combining the two outputs with a binary AND.

Spatial Transforms

Spatial transforms describe a class of operations that change the position of the data within the wave. These include the operations **ImageTransform** with multiple keywords, **MatrixTranspose**, **ImageRotate**, **ImageInterpolate**, and **ImageRegistration**.

Rotating Images

You can rotate images using the **ImageRotate** operation (see page V-405). There are two issues that are worth noting in connection with image rotations where the rotation angle is not a multiple of 90 degrees. First, the image size is always increased to accommodate all source pixels in the rotated image (no clipping is done). The second issue is that rotated pixels are calculated using bilinear interpolation so the result of N consecutive rotations by 360/N degrees will not, in general, equal the original image. In cases of multiple rotations you should consider keeping a copy of the original image as the same source for all rotations.

Image Registration

In many situations one has two or more images of the same object where the differences between the images have to do with acquisition times, dissimilar acquisition hardware or changes in the shape of the object between exposures. To facilitate comparison between such images it is necessary to register them, i.e., to adjust them so that they match each other. The **ImageRegistration** operation (see page V-398) modifies a test image to match a reference image when the key features are not too different. The algorithm is capable of subpixel resolution but it does not handle very large offsets or large rotations. The algorithm is based on an iterative processing that proceeds from coarse to fine detail. The optimization is performed using a modified Levenberg-Marquardt algorithm and results in an affine transformation for the relative rotation and translation with optional isometric scaling and contrast adjustment. The algorithm is most effective with square images where the center of rotation is not far from the center of the image.

ImageRegistration is based on an algorithm described by Thévenaz and Unser.

Mathematical Transforms

This class of transforms includes standard wave assignments, interpolation and sampling, Fourier, Wavelet, Hough, and Hartley transforms, convolution filters, edge detectors and morphological operators.

Standard Wave Operations

Grayscale image waves are regular 2D Igor waves that can be processed using normal Igor wave assignments (**Waveform Arithmetic and Assignments** on page II-74 and **Multidimensional Wave Assignment** on page II-96). For example, you can perform simple linear operations:

```
Duplicate/O root:images:blobs sample
Redimension/S sample       // create a single precision sample
sample=10+5*sample         // linear operation
NewImage sample            // keep this image displayed
```

Note that the display of the image is invariant for this linear operation.

Nonlinear operations are just as easy:

```
sample=sample^3-sample     // not particularly useful
```

You can add noise and change the background using simple wave assignment:

```
sample=root:images:blobs   // rest to original
sample+=gnoise(20)+x+2*y   // add Gaussian noise and background plane
```

As we have shown in several examples above, it is frequently necessary to create a binary image from your data. For instance, if you want an image that is set to 255 for all pixels in the image wave sample that are between the values of 50 and 250, and set to 0 otherwise, you can use the following one line wave assignment:

```
MatrixOp/O sample=255*greater(sample,50)*greater(250,sample)
```

More Efficient Wave Operations

There are several operations in this category that are designed to improve performance of certain image calculations. For example, you can obtain one plane (layer) from a multiplane image using a wave assignment like:

```
Make/N=(512,512) newImagePlane
newImagePlane[][]=root:Images:peppers[p][q][0]
```

Alternatively, you can execute:

```
ImageTransform/P=0 getPlane root:Images:peppers
```

or

```
MatrixOp/O outWave=root:Images:peppers[][][0]
```

ImageTransform and MatrixOp are much faster for this size of image than the simple wave assignment. See **General Utilities: ImageTransform Operation** on page III-381 and **Using MatrixOp** on page III-140.

Interpolation and Sampling

You can use the **ImageInterpolate** operation (see page V-382) as both an interpolation and sampling tool. In the following example we create an interpolated image from a portion of the MRI image. The resulting image is sampled at four times the original resolution horizontally and twice vertically.

```
NewImage root:images:MRI
ImageInterpolate /S={70,0.25,170,70,0.5,120} bilinear root:images:MRI
NewImage M_InterpolatedImage
```

As the keyword suggests, the interpolation is bilinear. You can use the same operation to sample the image. In the following example we reduce the image size by a factor of 4:

```
NewImage root:images:MRI              // display for comparison
ImageInterpolate /f={0.5,0.5} bilinear root:images:MRI
NewImage M_InterpolatedImage          // the sampled image
```

Note that in reducing the size of certain images, it may be useful to apply a blurring operation first (e.g., MatrixFilter gauss). This becomes important when the image contains thin (smaller than sample size) horizontal or vertical lines.

If the bilinear interpolation does not satisfy your requirements you can use spline interpolations of degrees 2-5. Here is a comparison between the bilinear and spline interpolation of degree 5 used to scale an image:

```
ImageInterpolate /f={1.5,1.5} bilinear MRI
Rename M_InterpolatedImage Bilinear
NewImage Bilinear
ImageInterpolate /f={1.5,1.5}/D=5 spline MRI
NewImage M_InterpolatedImage
```

Another form of sampling is creating a pixelated image. A pixelated image is computed by subdividing the original image into non-overlapping rectangles of nx by ny pixels and computing the average pixel value for each rectangle:

```
ImageInterpolate/PXSZ={5,5}/DEST=pixelatedImage pixelate, MRI
NewImage pixelatedImage
```

Fast Fourier Transform

There are many books on the application of Fourier transforms in imaging so we will only discuss some of the technical aspects of using the **FFT** operation (see page V-222) in Igor.

It is important to keep in mind is that, for historical reasons, the default FFT operation *overwrites* and *modifies* the image wave. You can also specify a destination wave in the FFT operation and your source wave will be preserved. The second issue that you need to remember is that the transformed wave is converted into a complex data type and the number of points in the wave is also changed to accommodate this conversion. The third issue is that when performing the FFT operation on a real wave the result is a one-sided spectrum, i.e., you have to obtain the rest of the spectrum by reflecting and complex-conjugating the result.

A typical application of the FFT in image processing involves transforming a real wave of 2N rows by M columns. The complex result of the FFT is (N+1) rows by M columns. If the original image wave has wave scaling of dx and dy, the new wave scaling is set to 1/(N*dx) and 1/(M*dy) respectively.

The following examples illustrate a number of typical applications of the FFT in imaging.

Calculating Convolutions

To calculate convolutions using the FFT it is necessary that the source wave and the convolution kernel wave have the same dimensions (see **MatrixOp** convolve for an alternative). Consider, for example, smoothing noise via convolution with a Gaussian:

```
// Create and display a noisy image.
Duplicate /O root:images:MRI mri    // an unsigned byte image.
Redimension/s mri                   // convert to single precision.
mri+=gnoise(10)                     // add noise.
NewImage mri
ModifyImage mri ctab= {*,*,Rainbow,0}// show the noise using false color.

// Create the filter wave.
Duplicate/O mri gResponse     // just so that we have the same size wave.
SetScale/I x -1,1,"" gResponse
SetScale/I y -1,1,"" gResponse

// Change the width of the Gaussian below to set the amount of smoothing.
gResponse=exp(-(x^2+y^2)/0.001)

// Calculate the convolution.
Duplicate/O mri processedMri
FFT processedMri                    // Transform the source
FFT gResponse                       // Transform the kernel
processedMri*=gResponse    // (complex) multiplication in frequency space
IFFT processedMri

// Swap the IFFT to properly center the result.
ImageTransform swap processedMri
Newimage processedM
ModifyImage processedMri ctab= {*,*,Rainbow,0}
```

In practice one can perform the convolution with fewer commands. The example above has a number of commands that are designed to make it clearer. Also note that we used the **SetScale** operation (see page V-853) to create the Gaussian filter. This was done to make sure that the Gaussian was created at the center of the filter image, a choice that is compatible with the ImageTransform **swap** operation. This example is also not ideal because one can take advantage of the properties of the Gaussian (the Fourier transform of a Gaussian is also Gaussian) and perform the convolution as follows:

```
// Calculate the convolution.
Duplicate/O mri shortWay
FFT shortWay
shortWay*=cmplx(exp(-(x^2+y^2)/0.01),0)
IFFT shortWay
Newimage shortWay
ModifyImage shortWay ctab={*,*,Rainbow,0}
```

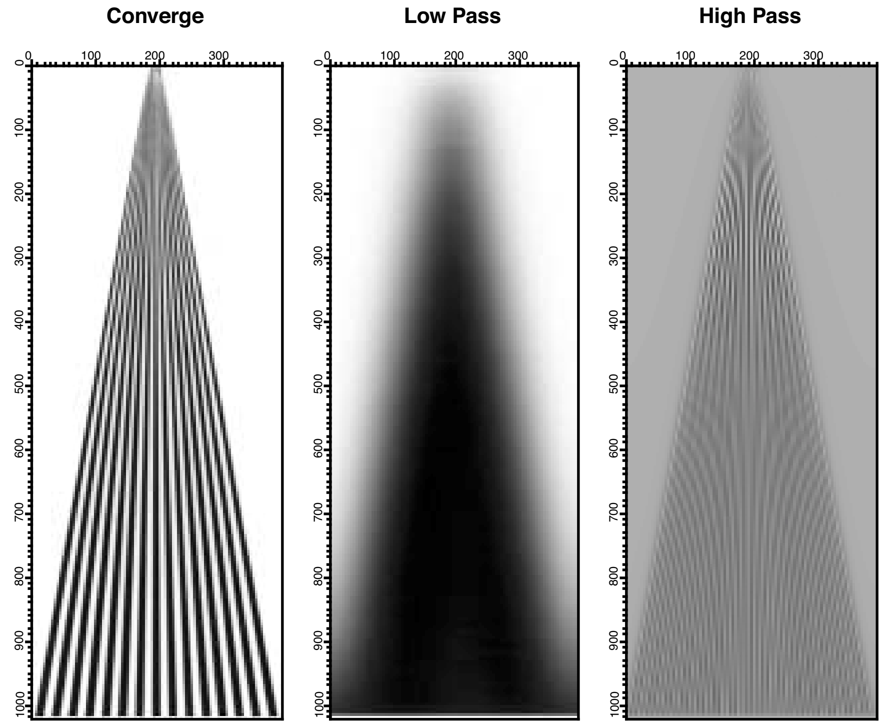
Spatial Frequency Filtering

The concept behind spatial frequency filtering is to transform the data into spatial frequency space. Once in frequency domain we can modify the spatial frequency distribution of the image and then inverse-transform to obtain the modified image.

Here is an example of low and high pass filtering. The converge image consists of wide black lines converging to a single point. If you draw a horizontal line profile anywhere below the middle of the image you will get a series of 15 rectangles which will give rise to a broad range of spatial frequencies in the horizontal direction.

```
// Prepare for FFT; we need SP or DP wave.
Duplicate/O root:images:converge converge
Redimension /s converge
FFT converge
Duplicate/O converge lowPass     // new complex wave in freq. domain
lowPass=lowPass*cmplx(exp(-(p)^2/5),0)
```

```
IFFT lowPass
NewImage lowPass                  // nonoptimal lowpass

Duplicate/O converge hiPass
hiPass=hiPass*cmplx(1-1/(1+(p-20)^2/2000),0)
IFFT hiPass
NewImage hiPass                   // nonoptimal highpass
```



Converge

Low Pass

High Pass

We arbitrarily chose the Gaussian form for the low-pass filter. In practical applications it is usually important to select an exact “cutoff” frequency and at the same time choose a filter that is sufficiently smooth so that it does not give rise to undesirable filtering artifacts such as ringing, etc. The high-pass filter that we used above is almost a notch filter that rejects low frequencies. Both filters are essentially one-dimensional filters.

Calculating Derivatives

Using the derivative property of Fourier transform, you can calculate, for example, the x-derivative of an image in the following way:

```
Duplicate/O root:images:mri xDerivative   // retain the original.
Redimension/S xDerivative
FFT xDerivative
xDerivative*=cmplx(0,p)    // neglecting 2pi factor & wave scaling.
IFFT xDerivative
NewImage xDerivative
```

Although this approach may not be appealing in all applications, its advantages are apparent when you need to calculate higher order derivatives. Also note that this approach does not take into account any wave scaling that may be associated with the rows or the columns.

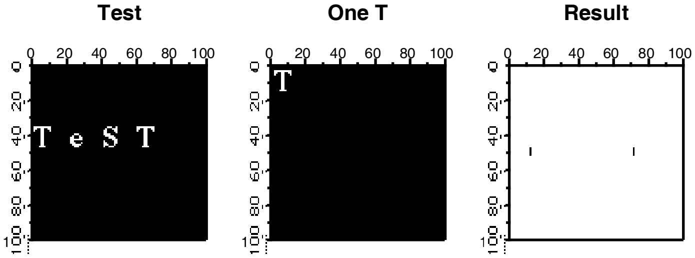
Calculating Integrals or Sums

Another useful property of the Fourier transform is that the transform values along the axes correspond to integrals of the image. There is usually no advantage in using the FFT for this purpose. However, if the FFT is calculated anyway for some other purpose, one can make use of this property. A typical situation where this is useful is in calculating correlation coefficient (normalized cross-correlation).

Correlations

The FFT can be used to locate objects of a particular size and shape in a given image. The following example is rather simple in that the test object has the same scale and rotation angle as the ones found in the image.

```
// Test image contains the word Test.
NewImage test                   // We will be looking for the two T's
Duplicate/O root:images:oneT oneT// the object we are looking for
NewImage oneT

Duplicate/O test testf          // because the FFT overwrites
FFT testf
Duplicate/O oneT oneTf
FFT oneTf
testf*=oneTf                    // not a "proper" correlation
IFFT testf
ImageThreshold/O/T=1.25e6 testf  // remove noise (due to overlap with other
characters
NewImage testf          // the results are the correlation spots for the T's
```



When using the FFT it is sometimes necessary to operate on the source image with one of the built-in window functions so that pixel values go smoothly to zero as you approach image boundaries. The **ImageWindow** operation (see page V-435) supports the Hanning, Hamming, Bartlett, Blackman, and Kaiser windows. Normally the **ImageWindow** operation (see page V-435) works directly on an image as in the following example:

```
// The redimension is required for the FFT operation anyway, so you
// might as well perform it here and reduce the quantization of the
// results in the ImageWindow operation.
Redimension/s blobs
ImageWindow /p=0.03 kaiser blobs
NewImage M_WindowedImage
```

To see what the window function looks like:

```
Redimension/S blobs                 // SP or DP waves are necessary
ImageWindow/i/p=0.01 kaiser blobs   // just creates the window data
NewImage M_WindowedImage      // you can also make a surface plot from this.
```

Wavelet Transform

The wavelet transform is used primarily for smoothing, noise reduction and lossy compression. In all cases the procedure we follow is first to transform the image, then perform some operation on the transformed wave and finally calculate the inverse transform.

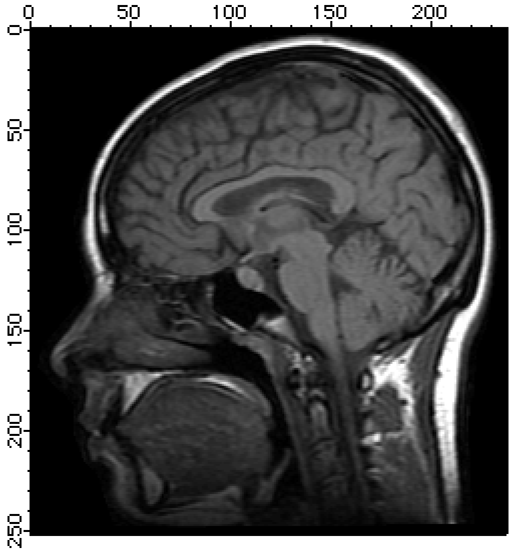
The next example illustrates a wavelet compression procedure. Start by calculating the wavelet transform of the image. Your choice of wavelet and coefficients can significantly affect compression quality. The compressed image is the part of the wave that corresponds to the low order coefficients in the transform (similar to low pass filtering in 2D Fourier transform). In this example we use the **ImageInterpolate** operation (see page V-382) to create a wave from a 64x64 portion of the transform.

```
DWT /N=4/P=1/T=1 root:images:MRI,wvl_MRI     // Wavelet transform
// reduce size by a factor of 16
ImageInterpolate/s={0,1,63,0,1,63} bilinear wvl_MRI
```

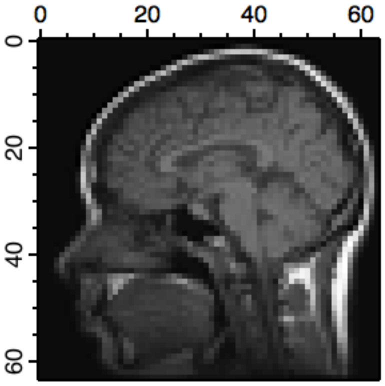
To reconstruct the image and evaluate compression quality, inverse-transform the compressed image and display the result:

```
DWT /I/N=4/P=0/T=1 M_InterpolatedImage,iwvl_compressed
NewImage iwvl_compressed
```

Original



Compressed

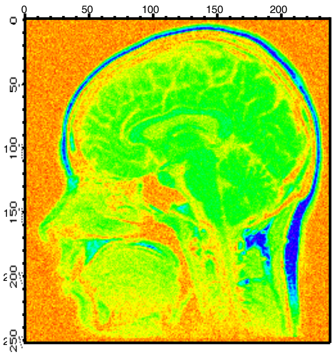


The reconstructed image exhibits a number of compression-related artifacts, but it is worth noting that unlike an FFT based low-pass filter, the advantage of the wavelet transform is that the image contains a fair amount of high spatial frequency content. The factor of 16 mentioned above is not entirely accurate because the original image was stored as a one byte per pixel while the compressed image consists of floating point values (so the true compression ratio is only 4).

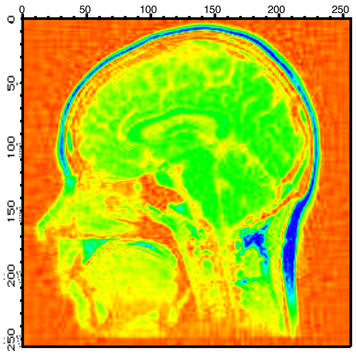
To illustrate the application of the wavelet transform to denoising, we start by adding Gaussian distributed noise with standard deviation 10 to the MRI image:

```
Redimension/S Mri                        // SP so we can add bipolar noise
Mri+=gnoise(10)                          // Gaussian noise added
NewImage Mri
ModifyImage Mri ctab={*,*,Rainbow,0}// false color for better discrimination.
DWT/D/N=20/P=1/T=1/V=0.5 Mri,dMri        // increase /V for more denoising
NewImage dMri                            // display denoised image
ModifyImage dMri ctab={*,*,Rainbow,0}
```

MRI



dMRI



Hough Transform

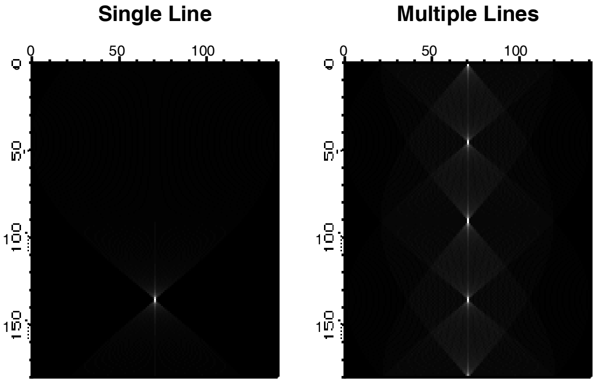
The Hough Transform is a mapping algorithm in which lines in image space map to single points in the transform space. It is most often used for line detection. Specifically, each point in the image space maps to a sinusoidal curve in the transform space. If pixels in the image lie along a line, the sinusoidal curves associated with these pixels all intersect at a single point in the transform space. By counting the number of sinusoids intersecting at each point in the transform space, lines can be detected. Here is an example of an image that consists of one line.

```
Make/O/B/U/N=(100,100) lineImage
lineImage=(p==q ? 255:0)            // single line at 45 degrees
Newimage lineimage
ImageTransform hough lineImage
Newimage M_Hough
```

The Hough transform of a family of lines:

```
lineImage=((p==100-q)|(p==q)|(p==50)|(q==50)) ? 255:0
ImageTransform Hough lineImage
```

The last image shows a series of bright pixels in the center. The first and last points correspond to lines at 0 and 180 degrees. The second point from the top corresponds to the line at 45 degrees and so on.



Fast Hartley Transform

The Hartley transform is similar to the Fourier transform except that it uses only real values. The transform is based on the *cas* kernel defined by:

$$cas(vx) = \cos(vx) + \sin(vx) .$$

The discrete Hartley transform is given by

$$H(u,v) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)\left\{\cos\left(\left[2\pi\left(\frac{ux}{M}-\frac{vy}{N}\right)\right] + \sin\left[2\pi\left(\frac{ux}{M}-\frac{vy}{N}\right)\right]\right)\right\}$$

The Hartley transform has two interesting mathematical properties. First, the inverse transform is identical to the forward transform, and second, the power spectrum is given by the expression:

$$P(f) = \frac{[H(f)]^2 + [H(-f)]^2}{2}$$

The implementation of the Fast Hartley Transform is part of the **ImageTransform** operation (see page V-417). It requires that the source wave is an image whose dimensions are a power of 2.

```
ImageTransform /N={18,3}/O padImage Mri      // make the image 256^2
ImageTransform fht mri
NewImage M_Hartley
```

Convolution Filters

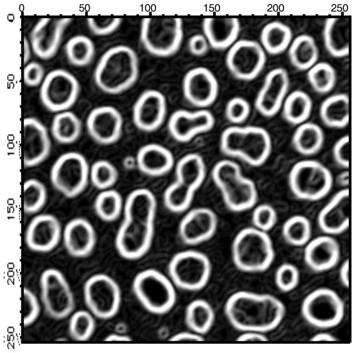
Convolution operators usually refer to a class of 2D kernels that are convolved with an image to produce a desirable effect (simple linear filtering). In some cases it is more efficient to perform convolutions using the FFT (similar to the convolution example above), i.e., transform both the image and the filter waves, multiply the transforms in the frequency domain and then compute the inverse transformation using the IFFT. The FFT approach is more efficient for convolution with kernels that are greater than 13x13 pixels. However, there is a very large number of useful kernels which play an important role in image processing that are 3x3 or 5x5 in size. Because these kernels are so small, it is fairly efficient to implement the corresponding linear filter as direct convolution without using the FFT.

In the following example we implement a low-pass filter with equal spatial frequency response along both axes.

```
Make /O/N=(9,9) sKernel       // first create the convolution kernel
SetScale/I x -4.5,4.5,"", sKernel
SetScale/I y -4.5,4.5,"", sKernel

// Equivalent to rect(2*fx)*rect(2*fy) in the spatial frequency domain.
sKernel=sinc(x/2)*sinc(y/2)

// Remember: MatrixConvolve executes in place; save your image first!
Duplicate/O root:images:MRI mri
Redimension/S mri                          // to avoid integer truncation
MatrixConvolve sKernel mri
NewImage mri
ModifyImage mri ctab= {*,*,Rainbow,0}    // just to see it better
```

The next example illustrates how to perform edge detection using a built-in convolution filter in the **ImageFilter** operation (see page V-372):

```
Duplicate/O root:images:blobs blobs
ImageFilter findEdges blobs
NewImage blobs
```



Other notable examples of image filters are Gauss, Median and Sharpen. You can also apply the same operation to 3D waves. The filters Gauss3D, avg3D, point3D, min3D, max3D and median3Dare the extensions of their 2D counterparts to 3x3x3 voxel neighborhoods. Note that the last three filters are not true *convolution* filters.

Edge Detectors

In many applications it is necessary to detect edges or boundaries of objects that appear in images. The edge detection consists of creating a binary image from a grayscale image where the pixels in the binary image are turned off or on depending on whether they belong to region boundaries or not. In other words, the detected edges are described by an image, not a vector (1D wave). If you need to obtain a wave describing boundaries of regions, you might want to use the **ImageAnalyzeParticles** operation (see page V-363).

Igor supports eight built-in edge detectors (methods) that vary in performance depending on the source image. Some methods require that you provide several parameters which tend to have a significant effect on the quality of the result. In the following examples we illustrate the importance of these choices.

```
// Create and display a simple artificial edge image.
Make/B/U/N=(100,100) edgeImage
edgeImage=(p<50? 50:5)
Newimage edgeImage

// Try a simple Sobel detector using iterated threshold detection.
ImageEdgeDetection/M=1/N Sobel, edgeImage
NewImage M_ImageEdges
ModifyImage M_ImageEdges explicit=0 // to see binary image in color
ModifyImage M_ImageEdges ctab= {*,*,Rainbow,0}
```

This result (the red line) is pretty much what we would expect. Here are other examples that work similarly well:

```
ImageEdgeDetection/M=1/N Kirsch, edgeImage   // same output wave
```

or

```
ImageEdgeDetection/M=1/N Roberts, edgeImage  // same output wave
```

The innocent looking /M=1 flag implies that the operation uses an iterative automatic thresholding. This appears to work well in the examples above, but it fails completely when using the Frei filter:

```
ImageEdgeDetection/M=1/N Frei, edgeImage
```

On the other hand, the bimodal fit thresholding works much better here:

```
ImageEdgeDetection/M=2/N Frei, root:edgeImage
```

The performance of this filter improves dramatically if you add a little noise to the image:

```
edgeImage+=gnoise(1)
ImageEdgeDetection/M=1/N/S=1 Canny, edgeImage
```

Using More Exotic Edge Detectors

The more exotic edge detectors consist of multistep operations that usually involve smoothing and differentiation. Here is an example that illustrates the effect of smoothing:

```
Duplicate/O root:images:blobs blobs
ImageEdgeDetection/M=1/N/S=1 Canny,blobs
Duplicate/O M_ImageEdges smooth1
ImageEdgeDetection/M=1/N/S=2 Canny,blobs
Duplicate/O M_ImageEdges smooth2
ImageEdgeDetection/M=1/N/S=3 Canny,blobs
Duplicate/O M_ImageEdges smooth3
NewImage smooth1
ModifyImage smooth1 explicit=0
NewImage smooth2
ModifyImage smooth2 explicit=0
NewImage smooth3
ModifyImage smooth3 explicit=0
```

As you can see, the third image (smooth3) is indeed much cleaner than the first or the second, however, that result is obtained at the cost of loosing some of the small blobs. The following commands will draw a circle around one of the blobs that is missing in the third image:

```
DoWindow/F Graph0
SetDrawLayer UserFront
SetDrawEnv linefgc= (65280,0,0),fillpat= 0
DrawOval 0.29,0.41,0.35,0.48
DoWindow/F Graph1
SetDrawLayer UserFront
SetDrawEnv linefgc= (65280,0,0),fillpat= 0
```

```
DrawOval 0.29,0.41,0.35,0.48
DoWindow/F Graph2
SetDrawLayer UserFront
SetDrawEnv linefgc= (65280,0,0),fillpat= 0
DrawOval 0.29,0.41,0.35,0.48
```

It is instructive to make a similar set of images using the Marr and Shen detectors.

```
// Note: This will take considerably longer time to execute!
Duplicate/O root:images:blobs blobs
ImageEdgeDetection/M=1/N/S=1 Marr,blobs
Duplicate/O M_ImageEdges smooth1
ImageEdgeDetection/M=1/N/S=2 Marr,blobs
Duplicate/O M_ImageEdges smooth2
ImageEdgeDetection/M=1/N/S=3 Marr,blobs
Duplicate/O M_ImageEdges smooth3
NewImage smooth1
ModifyImage smooth1 explicit=0
NewImage smooth2
ModifyImage smooth2 explicit=0
NewImage smooth3
ModifyImage smooth3 explicit=0
SetDrawLayer UserFront
SetDrawEnv linefgc= (65280,0,0),fillpat= 0
DrawOval 0.29,0.41,0.35,0.48
```

The three images of the calculated edges demonstrate the reduction of noise with the increase in the size of the convolution kernel. It's also worth noting that the blob that disappeared when we used the Canny detector is clearly visible using the Marr detector.

In the following example we use the Shen-Castan detector with various smoothing factors. Note that this edge detection algorithm does not use the standard thresholding (you have to specify the threshold using the /F flag).

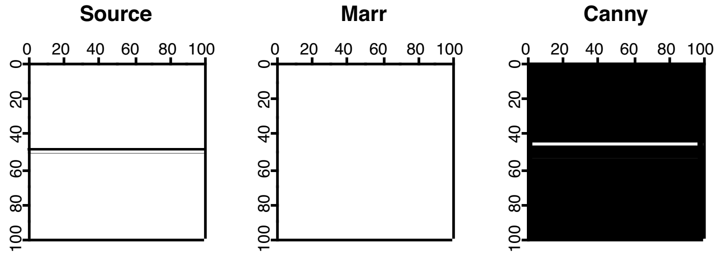
```
Duplicate/O root:images:blobs blobs
ImageEdgeDetection/N/S=0.5 shen,blobs
Duplicate/O M_ImageEdges smooth1
ImageEdgeDetection/N/S=0.75 shen,blobs
Duplicate/O M_ImageEdges smooth2
ImageEdgeDetection/N/S=0.95 shen,blobs
Duplicate/O M_ImageEdges smooth3
NewImage smooth1
ModifyImage smooth1 explicit=0
NewImage smooth2
ModifyImage smooth2 explicit=0
NewImage smooth3
ModifyImage smooth3 explicit=0
SetDrawLayer UserFront
SetDrawEnv linefgc=(65280,0,0),fillpat=0
DrawOval 0.29,0.41,0.35,0.48
```

As you can see in this example, the Shen detector produces a thin, though sometimes broken, boundary. The noise reduction is a trade-off with edge quality.

One of the problems of edge detectors that employ smoothing is that they usually introduce errors when there are two edges that are relatively close to each other. In the following example we construct an artificial image that illustrates this point:

```
Make/B/U/O/N=(100,100) sampleEdge=0
sampleEdge[][49]=255
sampleEdge[][51]=255
NewImage sampleEdge
ImageEdgeDetection/N/S=1 Marr, sampleEdge
Duplicate/O M_ImageEdges s2
```

```
NewImage s2
ImageEdgeDetection/M=1/S=3 Canny, sampleEdge
Duplicate/O M_ImageEdges s3
NewImage s3
```

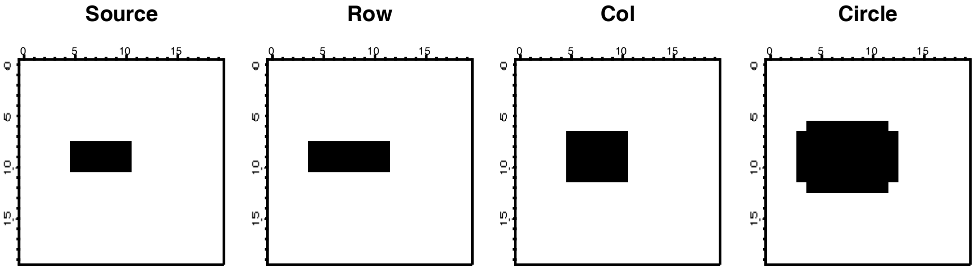


Note that the Marr detector completely misses the edge with the smoothing setting set to 1. Also, the position of the edge moves away from the true edge with increased smoothing in the Canny detector.

Morphological Operations

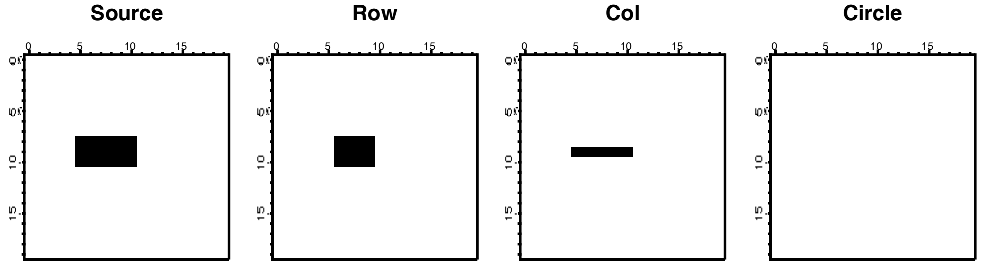
Morphological operators are tools that affect the shape and boundaries of regions in the image. Starting with dilation and erosion, the typical morphological operation involves an image and a structure element. The structure element is normally much smaller in size than the image. Dilation consists of reflecting the structure element about its origin and using it in a manner similar to a convolution mask. This can be seen in the next example:

```
Make/B/U/N=(20,20) source=0
source[5,10][8,10]=255                 // source is a filled rectangle
NewImage source
Imagemorphology /E=2 BinaryDilation source// dilation with 1x3 element
Duplicate M_ImageMorph row
NewImage row                     // display the result of dialation
Imagemorphology /E=3 BinaryDilation source// dilation by 3x1 column
Duplicate M_ImageMorph col
NewImage col                              // display column dilation
Imagemorphology /E=5 BinaryDilation source// dilation by a circle
NewImage M_ImageMorph                     // display circle dilation
```



The result of erosion is the set of pixels x, y such that when the structure element is translated by that amount it is still contained within the set.

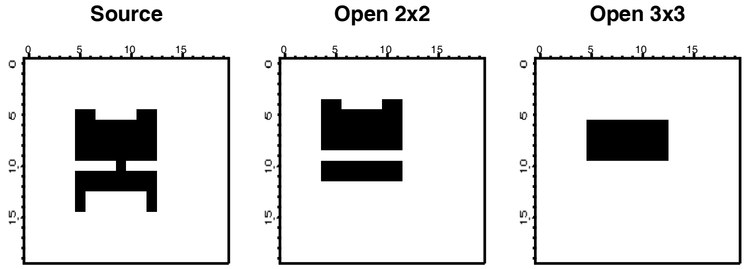
```
Make/B/U/N=(20,20) source=0
source[5,10][8,10]=255                 // source is a filled rectangle
NewImage source
Imagemorphology /E=2 BinaryErosion source// erosion with 1x3 element
Duplicate M_ImageMorph row
NewImage row                           // display the result of erosion
Imagemorphology /E=3 BinaryErosion source// erosion by 3x1 column
Duplicate M_ImageMorph col
NewImage col                           // display column erosion
Imagemorphology /E=5 BinaryErosion source// erosion by a circle
NewImage M_ImageMorph                  // display circle erosion
```



We note first that erosion by a circle erased all source pixels. We get this result because the circle structure element is a 5x5 “circle” and there is no x, y offset such that the circle is completely inside the source. The row and the col images show erosion predominantly in one direction. Again, try to imagine the 1x3 structure element (in the case of the row) sliding over the source pixels to produce the erosion.

The next pair of morphological operations are the opening and closing. Functionally, opening corresponds to an erosion of the source image by some structure element (say E), and then dilating the result using the same structure element E again. In general opening has a smoothing effect that eliminates small (narrow) protrusions as we show in the next example:

```
Make/B/U/N=(20,20) source=0
source[5,12][5,14] = 255
source[6,11][13,14] = 0
source[5,8][10,10] = 0
source[10,12][10,10] = 0
source[7,10][5,5] = 0
NewImage source
ImageMorphology /E=1 opening source // open using 2x2 structure element
Duplicate M_ImageMorph OpenE1
NewImage OpenE1
ImageMorphology /E=4 opening source // open using a 3x3 structure element
NewImage M_ImageMorph
```

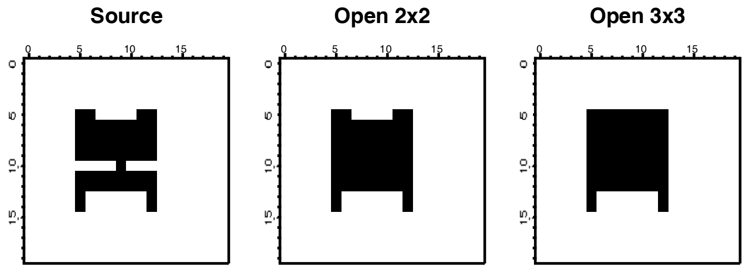


As you can see, the 2x2 structure element removed the thin connection between the top and the bottom regions as well as the two protrusions at the bottom. On the other hand, the two protrusions at the top were large enough to survive the 2x2 structure element. The third image shows the result of the 3x3 structure element which was large enough to eliminate all the protrusions but also the bottom region as well.

The closing operation corresponds to a dilation of the source image followed by an erosion using the same structure element.

```
Make/B/U/N=(20,20) source=0
source[5,12][5,14] = 255
source[6,11][13,14] = 0
source[5,8][10,10] = 0
source[10,12][10,10] = 0
source[7,10][5,5] = 0
NewImage source
ImageMorphology /E=4 closing source // close using 3x3 structure element
Duplicate M_ImageMorph CloseE4
NewImage CloseE4
```

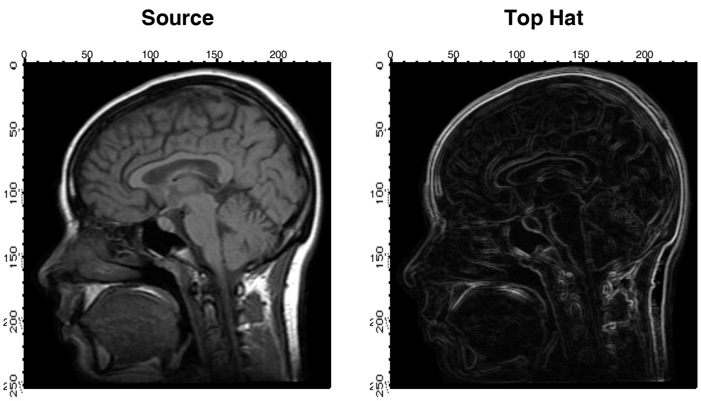
```
ImageMorphology /E=5 closing source // close using 5x5 structure element
NewImage M_ImageMorph
```

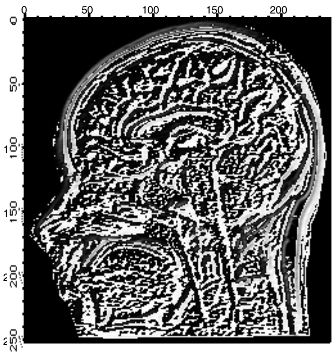


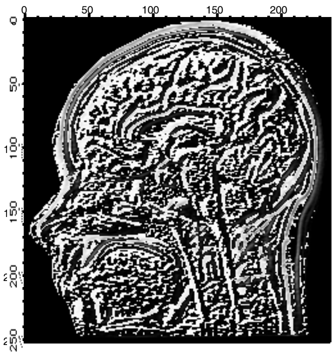
The center image above corresponds to a closing using a 3x3 structure element which appears to be large enough to close the gap between the top and bottom regions but not sufficiently large to fill the gaps between the top and bottom protrusions. The image on the right was created with a 5x5 “circle” structure element, which was evidently large enough to close the gap between the protrusions at the top but not at the bottom.

There are various definitions for the Top Hat morphological operation. Igor’s Top Hat calculates the difference between an eroded image and a dilated image. Other interpretations include calculating the difference between the image itself and its closing or opening. In the following example we illustrate some of these variations.

```
Duplicate root:images:mri source
ImageMorphology /E=1 tophat source  // close using 2x2 structure element
Duplicate M_ImageMorph tophat
NewImage tophat
ImageMorphology /E=1 closing source // close using 3x3 structure element
Duplicate M_ImageMorph closing
closing-=source
NewImage closing
ImageMorphology /E=1 opening source // close using 3x3 structure element
Duplicate M_ImageMorph opening
opening=source-opening
NewImage opening
```



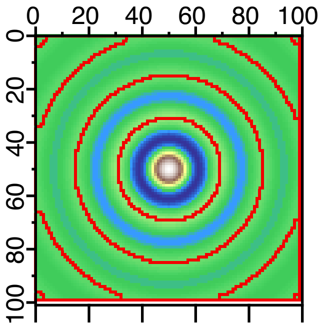

Closing


Opening

As you can see from the four images, the built-in Top Hat implementation enhances the boundaries (contours) of regions in the image whereas the opening or closing tophats enhance small grayscale variations.

The watershed operation locates the boundaries of watershed regions as we show below:

```
Make/O/N=(100,100) sample
sample=sinc(sqrt((x-50)^2+(y-50)^2)/2.5)// looks like concentric circles.
ImageTransform/O convert2Gray sample
NewImage sample
ModifyImage sample ctab= {*,*,Terrain,0}// color for better discrimination
ImageMorphology /N/L watershed sample
AppendImage M_ImageMorph
ModifyImage M_ImageMorph explicit=1, eval={0,65000,0,0}
```



Note that omitting the /L flag in the watershed operation may result in spurious watershed lines as the algorithm follows 4-connectivity instead of 8.

Image Analysis

The distinction between image processing and image analysis is rather fine. The pure analysis operations are: ImageStats, line profile, histogram, hsl segmentation and particle analysis.

ImageStats Operation

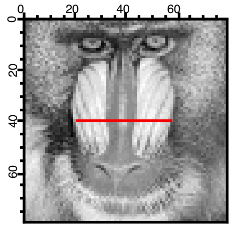
You can obtain global statistics on a wave using the standard **WaveStats** operation (see page V-1082). The **ImageStats** operation (see page V-414) works specifically with 2D and 3D waves. The operation can define a completely arbitrary ROI using a standard ROI wave (see **Working with ROI** on page III-378). A special flag /M=1, speeds up the operation when you only want to know the minimum, maximum and average values in the ROI region, skipping over the additional computation time required to evaluate higher moments. This operation was designed to work in user defined adaptive algorithms.

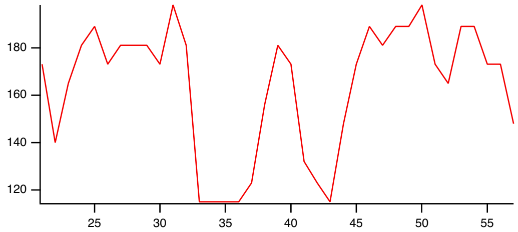
ImageStats can also operate on a specific plane of a 3D wave using the /P flag.

ImageLineProfile Operation

The **ImageLineProfile** operation (see page V-389) is somewhat of a misnomer as it samples the image along a path consisting of an arbitrary number of line segments. To use the operation you first need to create the description of the path using a pair of waves. Here is a simple example:

```
NewImage root:images:baboon    // Display the image that we want to profile
// Create the pair of waves representing a straight line path.
Make/O/N=2 xPoints={21,57}, yPoints={40,40}
AppendToGraph/T yPoints vs xPoints  // display the path on the image
// Calculate the profile.
ImageLineProfile xwave=xPoints, ywave=yPoints, srcwave=root:images:baboon
Display W_ImageLineProfile vs W_LineProfileX // display the profile
```





You can create a more complex path consisting of an arbitrary number of points. In this case you may want to take advantage of the W_LineProfileX and W_LineProfileY waves that the operation creates and plot the profile as a 3D path plot (see “Path Plots” in the Visualization help file). See also the IP Tutorial experiment for more elaborate examples.

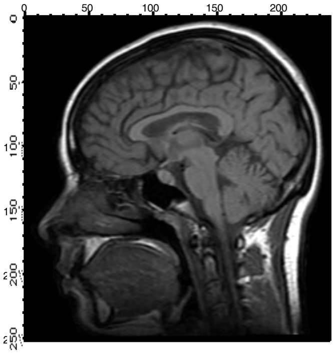
Note: If you are working with 3D waves with more than 3 layers, you can use `ImageLineProfile/P=`*`plane`* to specify the plane for which the profile is computed.

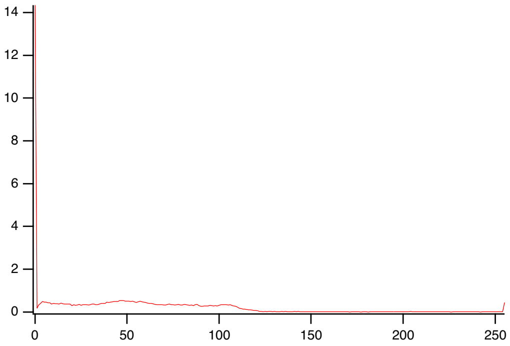
If you are using the line profile to extract a sequential array of data (a row or column) from the wave it is more efficient (about a factor of 3.5 in speed) to extract the data using ImageTransform **getRow** or **getCol**.

Histograms

The histograms is a very important tool in image analysis. For example, the simplest approach to automating the detection of the background in an image is to calculate the histogram and to choose the pixel value which occurs with the highest frequency. Histograms are also very important in determining threshold values and in enhancing image contrast. Here are some examples using image histograms:

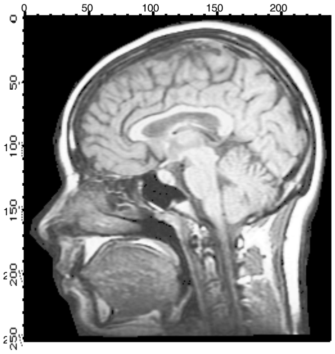
```
NewImage root:images:mri
ImageHistogram root:images:mri
Duplicate W_ImageHist origMriHist
Display /W=(201.6,45.2,411,223.4) origMriHist
```

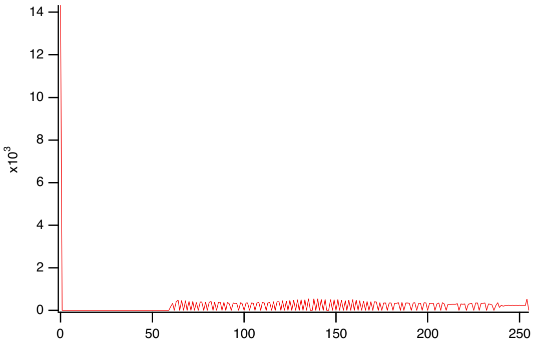




It is obvious from the histogram that the image is rather dark and that the background is most likely zero. The small counts for pixels above 125 suggests that the image is a good candidate for histogram equalization.

```
ImageHistModification root:images:mri
ImageHistogram M_ImageHistEq
NewImage M_ImageHistEq
Display /W=(201.6,45.2,411,223.4) W_ImageHist
```





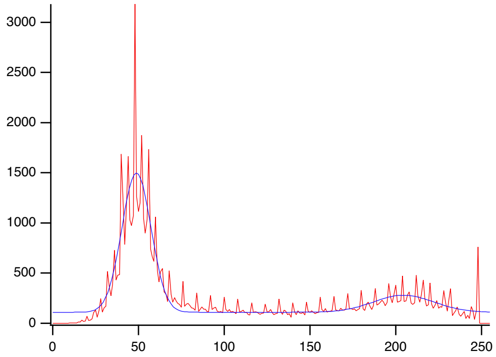
Comparing the two histograms two features stand out: first, there is no change in the dark background because it is only one level (0). Second, the rest of the image which was mostly between the values of 0 and 120 has now been stretched to the range 57-255.

The next example illustrates how you can use the histogram information to determine a threshold value.

```
NewImage root:images:blobs
ImageHistogram root:images:blobs
Display /W=(201.6,45.2,411,223.4) W_ImageHist
```

The resulting histogram is clearly bimodal. Let’s fit it to a pair of Gaussians:

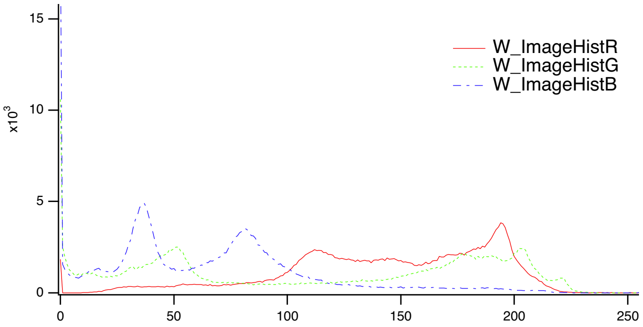
```
// Guess coefficient wave based on the histogram.
Make/O/N=6 coeff={0,3000,50,10,500,210,20}
Funcfit/Q twoGaussians,coeff,W_ImageHist /D
ModifyGraph rgb(fit_W_ImageHist)=(0,0,65000)
```



The curve shown in the graph is the functional fit of the sum of two Gaussians. You can now choose, by visual inspection, an x-value between the two Gaussians — probably somewhere in the range of 100-150. In fact, if you test the same image using the built-in thresholding operations that we have discussed above, you will see that the iterated algorithm chooses the value 125, fuzzy entropy chooses 109, etc.

Histograms of RGB or HSL images result in a separate histogram for each color channel:

```
ImageHistogram root:images:peppers
Display W_ImageHistR,W_ImageHistG,W_ImageHistB
ModifyGraph rgb(W_ImageHistG)=(0,65000,0),rgb(W_ImageHistB)=(0,0,65000)
```



Histograms of 3D waves containing more than 3 layers can be computed by specifying the layer with the /P flag. For example,

```
Make/N=(10,20,30) ddd=gnoise(5)
ImageHistogram/P=10 ddd
Display W_ImageHist
```

Unwrapping Phase

Unwrapping phase in two dimensions is more complicated than in one dimension because the operation's results must be independent of the unwrapping path. The path independence means that any path integral over a closed contour in the unwrapped domain must vanish. In many situations there are points in the domain around which closed contour path integrals do not vanish. Such points are called "residues". The residues are positive if a counter-clockwise path integral is positive. When unwrapping phase in two dimensions, the residues are typically ±1. This suggests that whenever two opposing residues are connected by a line (known as a "branch cut"), any contour integral whose path does not cross the branch cut will vanish. When a positive and negative residues are side by side they combine to a "dipole" which may be removed because a path integral around the dipole also vanishes. It follows that unwrapping can be performed using paths that either do not encircle unbalanced residues or paths that do not cross branch cuts.

The **ImageUnwrapPhase** operation (see page V-433) performs 2D phase unwrapping using either a fast method that ignores possible residues or a slower method which locates residues and attempts to find paths around them. The fast method uses direct integration of the differential phases. It can lead to incorrect results if there are residues in the domain. The slow method first identifies all residues, draws them into an internal bitmap adding branch cuts and then applying repeatedly the algorithm used in ImageSeedFill to obtain the paths around the residues and branch cuts until all pixels have been processed. Sometimes the distribution of residues and branch cuts is such that the domain of the data is covered by several regions, each of which is completely bounded by branch cuts or the data boundary. In this case, the phase is computed independently in each individual region with an offset that is based on the first processed pixel in that region. Note that when you use ImageUnwrapPhase using a method that computes the residues, the operation creates the variables V_numResidues and V_numRegions. You can also obtain a copy of the internal bitmap which could be useful for analyzing the results.

The ImageUnwrapPhase Demo in the Examples:Analysis folder provides a detailed example illustrating different types of residues, branch cuts and resulting unwrapped phase.

HSL Segmentation

When you work with color images you have two analogs to grayscale thresholding. The first is simple thresholding of the luminance of the image. To do this you need to convert the image from RGB to HSL and then perform the thresholding on the luminance plane. The second equivalent of thresholding is HSL segmentation, where the image is subdivided into regions of HSL values that fall within a certain range. In the following example we segment the peppers image to locate regions corresponding to red peppers:

```
NewImage root:images:peppers
ImageTransform/H={330,50}/L={0,255}/S={0,255} root:images:peppers
NewImage M_HueSegment
```

Note that we used /H={330,50}. The apparent flip of the limits is allowed in the case of hue values to cover the single range from hue angle 330 degrees to hue angle 50 degrees.

There are two additional approaches for color segmentation that should be mentioned here. You can use ImageTransform **matchPlanes** operation to segment an image for pixels that satisfy prescribed value ranges in all planes. This operation has the advantage that it can be applied to images in any color space. Another segmentation operation is ImageTransform **selectColor** which is based on RGB color space and a user provided tolerance value. The same concept can be applied with ImageSeedFill to get the effect of a "magic wand" selection.

Particle Analysis

Typical particle analysis consists of three steps. First you need to preprocess the image. This may include noise removal or reduction, possible background adjustments (see **ImageRemoveBackground** operation on page V-403) and thresholding. Once you obtain a binary image, your second step is to invoke the **ImageAnalyzeParticles** operation (see page V-363). The third and final step is making some sense of all the data produced by the ImageAnalyzeParticles operation or "post-processing".

Issues related to the preprocessing have been discussed elsewhere in this chapter. We will assume that we are starting with a preprocessed, clean, binary image which contains some particles.

```
NewImage root:images:blobs                // display the original image

// Step 1:create binary image.
// Note the /I flag to invert the output wave so that particles are marked by 
zero.
ImageThreshold/I/Q/M=1 root:images:blobs  // Note the /I flag!!

// Step 2:Here we are invoking the operation in quiet mode, specifying particles
// of size equal or greater than 2 pixels. We are also asking for particles 
moment
// Information, boundary waves and a particle masking wave.
ImageAnalyzeParticles /Q/A=2/E/W/M=2 stats M_ImageThresh

// Step 3:post processing choices
// Display the detected boundaries on top of the particles
AppendToGraph/T W_BoundaryY vs W_BoundaryX

// If you browse the numerical data:
Edit W_SpotX,W_SpotY,W_circularity,W_rectangularity,W_ImageObjPerimeter
AppendToTable W_xmin,W_xmax,W_ymin,W_ymax,M_Moments,M_RawMoments
```

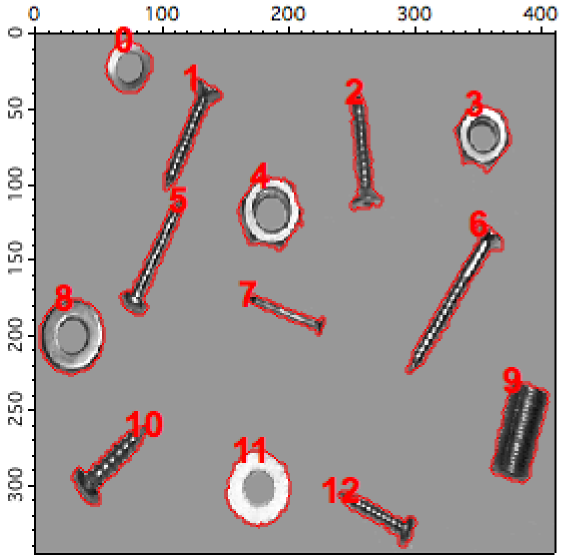
Note that particles that intersect the boundary of the image may give rise to inaccuracies in particle statistics. It is therefore useful sometimes to remove these particles before performing the analysis.

The raw values generated by ImageAnalyzeParticles operation can be used for further processing.

The following example illustrates slightly different pre and post-processing.

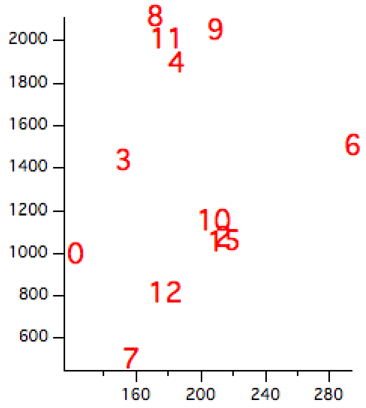
```
NewImage screws
// Here we have a synthetic background so the conversion to binary is easy.
screws=screws==163 ? 255:0
ImageMorphology /O/I=2/E=1 binarydilation screws
ImageMorphology /O/I=2/E=1 erosion screws

// Now the particle analysis operation with the option to fill the holes.
ImageAnalyzeParticles/E/W/Q/M=3/A=5/F stats, screws

NewImage root:images:screws
AutoPositionWindow/E $WinName(0,1)
// Show the detected boundaries.
AppendToGraph/T W_BoundaryY vs W_BoundaryX
AppendToGraph/T W_SpotY vs W_SpotX
Duplicate/O w_spotx w_index
```

```
w_index=p
ModifyGraph mode(W_SpotY)=3
ModifyGraph textMarker(W_SpotY)={w_index,"default",1,0,5,0.00,0.00}
ModifyGraph msize(W_SpotY)=6
```



Now for some shape classification in which we plot particle area versus perimeter:

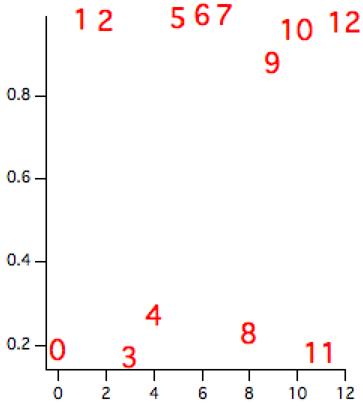
```
Display/W=(23.4,299.6,297,511.4) W_ImageObjArea vs W_ImageObjPerimeter
ModifyGraph
mode=3,textMarker(W_ImageObjArea)={w_index,"default",0,0,5,0.00,0.00}
ModifyGraph msize=6
```



The classification diagram we just created uses two parameters (area and perimeter) that are very sensitive to image noise. We can see that there are two basic classes that can be associated with the roundness of the boundaries but it is difficult to accept the classification of particle 9.

In the following we compute another classification based on the eccentricity of the objects:

```
Make/O/N=(DimSize(M_Moments,0)) ecc
ecc=sqrt(1-M_Moments[p][3]^2/M_Moments[p][2]^2)

Display /W=(23.4,299.6,297,511.4) ecc
ModifyGraph mode=3,textMarker(ecc)={w_index,"default",0,0,5,0.00,0.00}
ModifyGraph msize=6
```



The second classification produces a distinct separation of the screws from the washers and nuts. It also illustrates the importance of selecting the best classification parameters.

You can use the ImageAnalyzeParticles operation also for the purpose of creating masks for particular particles. For example, to create a mask for particle 9 in the example above:

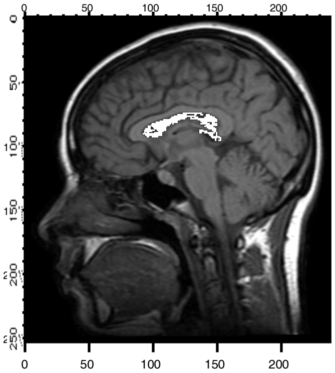
```
ImageAnalyzeParticles /L=(w_spotX[9],w_spotY[9]) mark screws
NewImage M_ParticleMarker
```

You can use this feature of the operation to color different classes of objects using an overlay.

Seed Fill

In some situations you may need to define segments of the image based on a contiguous region of pixels whose values fall within a certain range. The **ImageSeedFill** operation (see page V-409) helps you do just that.

```
NewImage mri
ImageSeedFill/B=64 seedX=132,seedY=77,min=52,max=65,target=255,srcWave=mri
AppendImage M_SeedFill
ModifyImage M_SeedFill explicit=1, eval={255,65535,65535,65535}
```



Here we have used the /B flag to create an overlay image but it can also be used to create an ROI wave for use in further processing. This example represents the simplest use of the operation. In some situations the criteria for a pixel's inclusion in the filled region are not so sharp and the operation may work better if you use the adaptive or fuzzy algorithms. For example (**Note**: the command is wrapped over two lines):

```
ImageSeedFill/B=64/c seedX=144,seedY=83,min=60,max=150,target=255,
srcWave=mri,adaptive=3
```

Note that the min and max values have been relaxed but the adaptive parameter provides alternative continuity criterion.

Other Tools

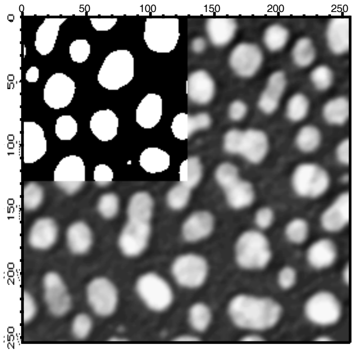
Igor provides a number of utility operations that help you manage and manipulate image data.

Working with ROI

Many of the image processing operations support a region of interest (ROI). The region of interest is the portion of the image that we want to affect by an operation. Igor supports a completely general ROI, specified as a binary wave (unsigned byte) that has the same dimensions as the image. Set the ROI wave to zero for all pixels within the region of interest and to any other value outside the region of interest. Note that in some situations it is useful to set the ROI pixels to a nonzero value (e.g., if you use the wave as a multiplicative factor in a mathematical operation). You can use ImageTransform with the invert keyword to quickly convert between the two options.

The easiest way to create an ROI wave is directly from the command line. For example, an ROI wave that covers a quarter of the blobs image may be generated as follows:

```
Duplicate root:images:blobs myROI
Redimension/B/U myROI
myROI=64                      // arbitrary nonzero value
myROI[0,128][0,127]=0         // the actual region of interest

// Example of usage:
ImageThreshold/Q/M=1/R=myROI root:images:blobs
NewImage M_ImageThresh
```



If you want to define the ROI using graphical drawing tools you need to open the tools and set the drawing layer to progFront. This can be done with the following instructions:

```
SetDrawLayer progFront
ShowTools/A rect        // selects the rectangle drawing tool first
```

Generating ROI Masks

You can now define the ROI as the area inside all the closed shapes that you draw. When you complete drawing the ROI you need to execute the commands:

```
HideTools/A             // Drawing tools are not needed any more.
// M_ImageThresh is the top image in this example.
ImageGenerateROIMask M_ImageThresh
// The ROI wave has been created; To see it,
NewImage M_ROIMask
```

The Image Processing procedures provide a utility for creating an ROI wave by drawing on a displayed image.

Converting Boundary to a Mask

A third way of generating an ROI mask is using the **ImageBoundaryToMask** operation (see page V-367). This operation takes a pair of waves (y versus x) that contain pixel values and scan-converts them into a mask. When you invoke the operation you also have to specify the rectangular width and height of the output mask.

```
// Create a circle.
Make/N=100 ddx,ddy
ddx=50*(1-sin(2*pi*x/100))
ddy=50*(1-cos(2*pi*x/100))
ImageBoundaryToMask width=100,height=100,xwave=ddx,ywave=ddy
// The result is an image not a curve!
NewImage M_ROIMask
```

Note that the resulting binary wave has the values 0 and 255, which you may need to invert before using them in certain operations.

In many situations the operation ImageBoundaryToMask is followed by ImageSeedFill in order to convert the mask to a filled region. You can obtain the desired mask in one step using the keywords seedX and seedY in ImageBoundaryToMask but you must make sure that the mask created by the boundary waves is a closed domain.

```
ImageBoundaryToMask width=100,height=100,xwave=ddx,ywave=ddy,
seedX=50,seedY=50
ModifyImage M_ROIMask explicit=0
```

Marquee Procedures

A fourth way to create an ROI mask is using the Marquee2Mask procedures. To use this in your own experiment you will have to add the following line to your procedure window:

```
#include <Marquee2Mask>
```

You can now create the ROI mask by selecting one or more rectangular marquees (drag the mouse) in the image. After you select each marquee click inside the marquee and choose MarqueeToMask or Append-MarqueeToMask.

Subimage Selection

You can use an ROI to apply various image processing operations to selected portions of an image. The ROI is a very useful tool especially when the region of interest is either not contiguous or not rectangular. When the region of interest is rectangular, you can usually improve performance by creating a new subimage which consists entirely of the ROI. If you know the coordinates and dimensions of the ROI it is simplest to use the `Duplicate/R` operation. If you want to make an interactive selection you can use the marquee together with CopyImageSubset marquee procedure (after making a marquee selection in the image, click inside the marquee and choose CopyImageSubset).

Handling Color

Most of the image operations are designed to work on grayscale images. If you need to perform an operation on a color image certain aspects become a bit more complicated. In the next example we illustrate how you might sharpen a color image.

```
NewImage root:images:rose
ImageTransform rgb2hsl root:images:rose   // first convert to hsl
ImageTransform/P=2 getPlane M_RGB2HSL
ImageFilter Sharpen M_ImagePlane    // you can also use sharpenmore
ImageTransform /D=M_ImagePlane /P=2 setPlane M_RGB2HSL
ImageTransform hsl2rgb M_RGB2HSL
NewImage M_HSL2RGB
```

Background Removal

There are many approaches to removing the effect of a nonuniform background from an image. If the non uniformity is additive, it is sometimes useful to fit a polynomial to various points which you associate with the background and then subtract the resulting polynomial surface from the whole image. If the nonuniformity is multiplicative, you need to generate an image corresponding to the polynomial surface and use it to scale the original image.

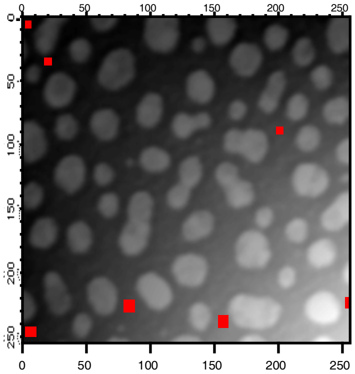
Additive Background

```
Duplicate/O root:images:blobs addBlobs
Redimension/S addBlobs              // convert to single precision
addBlobs+=0.01*x*y                  // add a tilted plane
NewImage addBlobs
```

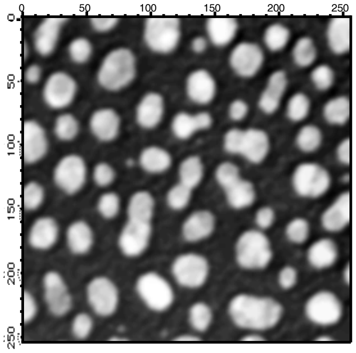
To use the **ImageRemoveBackground** operation (see page V-403), we need an ROI mask designating regions in the image that represent the background. You can create one using one of the ROI construction methods that we discussed above. For the purposes of this example, we choose the ROI that consists of the 7 rectangles shown in the Degraded Source image below.

```
// Show the ROI background selection.
AppendImage root:images:addMask
ModifyImage addMask explicit=1, eval={1,65000,0,0}

// Create a corrected image and display it.
ImageRemoveBackground /R=root:images:addMask /w/P=2 addBlobs
NewImage M_RemovedBackground
```

Degraded Source



Background Removed



If the source image contains relatively small particles on a nonuniform background, you may remove the background (for the purpose of particle analysis) by iterating grayscale erosion until the particles are all gone. You are then left with a fairly good representation of the background that can be subtracted from the original image.

Multiplicative Background

This case is much more complicated because the removal of the background requires division of the image by the calculated background (it is assumed here that the system producing the image has an overall gamma of 1). The first complication has to do with the possible presence of zeros in the calculated background. The second complication is that the calculations give us the additional freedom to choose one constant factor to scale the resulting image. There are many approaches for correcting a multiplicative background. The following example shows how an image can be corrected if we assume that the peak values (identified by the ROI mask) would all have the same value in the absence of a background.

```
Duplicate/O root:images:blobs mulBlobs
Redimension/S mulBlobs           // convert to single precision
mulBlobs*=(1+0.005*x*y)
NewImage mulBlobs

// Show us the ROI foreground selection; you can use histogram
// equalization to find fit regions in the dark area.
AppendImage root:images:multMask
ModifyImage multMask explicit=1, eval={1,65000,0,0}

ImageRemoveBackground /R=root:images:multMask/F/w/P=2 mulBlobs
// Normalize the fit.
WaveStats/Q/M=1 M_RemovedBackground

// Renormalize the fit--we can use that one free factor.
```

```
M_RemovedBackground=(M_RemovedBackground-V_min)/(V_max-V_min)
// Remove zeros by replacing with average value.
WaveStats/Q/M=1 M_RemovedBackground
MatrixOp/O M_RemovedBackground=M_RemovedBackground+V_avg*equal(M_RemovedBackground,0)
MatrixOp/O mulBlobs=mulBlobs/M_RemovedBackground   // scaled image.
```

In the example above we have manually created the ROI masks that were needed for the fit. You can automate this process (and actually improve performance) by subdividing the image into a number of smaller rectangles and selecting in each one the highest (or lowest) pixel values. An example of such procedure is provided in connection with the ImageStats operation above.

General Utilities: ImageTransform Operation

As we have seen above, the **ImageTransform** operation (see page V-417) provides a number of image utilities. As a rule, if you are unable to find an appropriate image operation check the options available under ImageTransform. Here are some examples:

When working with RGB or HSL images it is frequently necessary to access one plane at a time. For example, the green plane of the peppers image can be obtained as follows:

```
NewImage root:images:peppers        // display original
Duplicate/O root:images:peppers peppers
ImageTransform /P=1 getPlane peppers
NewImage M_ImagePlane               // display green plane in grayscale
```

The complementary operation can insert a plane into a 3D wave. For example, suppose you wanted to modify the green plane of the peppers image:

```
ImageHistModification/o M_ImagePlane
ImageTransform /p=1 /D=M_ImagePlane setPlane peppers
NewImage peppers                    // display the processed image
```

Some operations are restricted to waves of particular dimensions. For example, if you want to use the Adaptive histogram equalization, the number of horizontal and vertical partitions is restricted by the requirement that the image be an exact multiple of the dimensions of the subregion. The ImageTransform operation provides three image padding options: If you specify a negative number to the changed rows or columns, the corresponding rows and columns are removed from the image. If the numbers are positive, rows and columns are added. By default the added rows and columns contain exactly the same pixel values as the last row and column in the image. If you specify the /W flag the operation duplicates the relevant portion of the image into the new rows and columns. Here are some examples:

```
Duplicate/o root:images:baboon baboon
NewImage baboon
ImageTransform/N={-20,-10} padImage baboon
Rename M_PaddedImage, cropped
NewImage cropped
ImageTransform/N={40,40} padImage baboon
Rename M_PaddedImage, padLastVals
NewImage padLastVals
ImageTransform/W/N={100,100} padImage baboon
NewImage M_PaddedImage
```

Another utility operation is the conversion of any 2D wave into a normalized (0-255) 8-bit image wave. This is accomplished with the ImageTransform operation using the keyword convert2gray. Here is an example:

```
// Create some numerical data
Make/O/N=(50,80) numericalWave=x*sin(x/10)*y*exp(y/100)
ImageTransform convert2gray numericalWave
NewImage M_Image2Gray
```

The conversion to an 8-bit image is required for certain operation. It is also useful sometimes when you want to reduce the size of your image waves.

References

Ghiglia, Dennis C., and Mark D. Pritt, *Two Dimensional Phase Unwrapping — Theory, Algorithms and Software*, John Wiley & Sons, 1998.

Gonzalez, Rafael C., and Richard E. Woods, *Digital Image Processing*, 3rd ed., Addison-Wesley, 1992.

Pratt, William K., *Digital Image Processing*, 2nd ed., John Wiley & Sons, 1991.

Thévenaz, P., and M. Unser, A Pyramid Approach to Subpixel Registration Based on Intensity, *IEEE Transactions on Image Processing*, *7*, 27-41, 1998.

Chapter

III-12

Statistics

Overview

This chapter describes operations and functions for statistical analysis together with some general guidelines for their use. This is not a statistics tutorial; for that you can consult one of the references at the end of this chapter or the references listed in the documentation of a particular operation or function. The material below assumes that you are familiar with techniques and methods of statistical analysis.

Most statistics operations and functions are named with the prefix "Stats". Naming exceptions include the random noise functions that have traditionally been named based on the distribution they represent.

There are six natural groups of statistics operations and functions. They include:

- Test operations
- Noise functions
- Probability distribution functions (PDFs)
- Cumulative distribution functions (CDFs)
- Inverse cumulative distribution functions
- General purpose statistics operations and functions

Statistics Test Operations

Test operations analyze the input data to examine the validity of a specific hypothesis. The common test involves a computation of some numeric value (also known as "test statistic") which is usually compared with a critical value in order to determine if you should accept or reject the test hypothesis (H_0). Most tests compute a critical value for the given significance *alpha* which has the default value 0.05 or a user-provided value via the /ALPH flag. Some tests directly compute the P value which you can compare to the desired significance value.

Critical values have been traditionally published in tables for various significance levels and tails of distributions. They are by far the most difficult technical aspect in implementing statistical tests. The critical values are usually obtained from the inverse of the CDF for the particular distribution, i.e., from solving the equation

$$cdf(criticalValue) = 1 - alpha,$$

where *alpha* is the significance. In some distributions (e.g., Friedman's) the calculation of the CDF is so computationally intensive that it is impractical (using desktop computers in 2006) to compute for very large parameters. Fortunately, large parameters usually imply that the distributions can be approximated using simpler expressions. Igor's tests provide whenever possible exact critical values as well as the common relevant approximations.

Comparison of critical values with published table values can sometimes be interesting as there does not appear to be a standard for determining the published critical value when the CDF takes a finite number of discrete values (step-like). In this case the CDF attains the value (1-*alpha*) in a vertical transition so one could use the X value for the vertical transition as a critical value or the X value of the subsequent vertical transition. Some tables reflect a "conservative" approach and print the X value of subsequent transitions.

Statistical test operations can print their results to the history area of the command window and save them in a wave in the current data folder. Result waves have a fixed name associated with the operation. Elements in the wave are designated by dimension labels. You can use the /T flag to display the results of the operation in a table with dimension labels. The argument for this flag determines what happens when you kill the table. You can use/Q in all test operations to prevent printing information in the history area and you can use the /Z flag to make sure that the operations do not report errors except by setting the V_Flag variable to -1.

Statistical test operations tend to include several variations of the named test. You can usually choose to execute one or more variations by specifying the appropriate flags. The following table can be used as a guide for identifying the operation associated with a given test name.

Statistical Test Operations by Name

Test Name	Where to find it
Angular Distance	**StatsAngularDistanceTest**
Bartlett's test for variances	**StatsVariancesTest**
BootStrap	**StatsResample**
Brown and Forsythe	**StatsANOVA1Test**
Chi-squared test for means	**StatsChiTest**
Cochran's test	**StatsCochranTest**
Dunn-Holland-Wolfe	**StatsNPMCTest**
Dunnette multicomparison test	**StatsDunnettTest**, **StatsLinearRegression**
Fisher's Exact Test	**StatsContingencyTable**
Fixed Effect Model	**StatsANOVA1Test**
Friedman test on randomized block	**StatsFriedmanTest**
F-test on two distributions	**StatsFTest**
Hodges-Ajne (Batschelet)	**StatsHodgesAjneTest**
Hartigan test for unimodality	**StatsDIPTest**
Hotelling	**StatsCircularTwoSampleTest**, **StatsCircularMeans**
Jackknife	**StatsResample**
Jarque-Bera Test	**StatsJBTest**
Kolmogorov-Smirnov	**StatsKSTest**
Kruskal-Wallis	**StatsKWTest**
Kuiper Test	**StatsCircularMoments**
Levene's test for variances	**StatsVariancesTest**
Linear Correlation Test	**StatsLinearCorrelationTest**
Linear Order Statistic	**StatsCircularMoments**
Mann-Kendall	**StatsKendallTauTest**
Moore test	**StatsCircularTwoSampleTest**, **StatsCircularMeans**
Nonparametric multiple contrasts	**StatsNPMCTest**
Nonparametric angular-angular correlation	**StatsCircularCorrelationTest**
Nonparametric second order circular analysis	**StatsCircularMeans**
Nonparametric serial randomness (nominal)	**StatsNPNominalSRTest**
Parametric angular-angular correlation	**StatsCircularCorrelationTest**
Parametric angular-Linear correlation	**StatsCircularCorrelationTest**
Parametric second order circular analysis	**StatsCircularMeans**
Parametric serial randomness test	**StatsSRTest**
Rayleigh	**StatsCircularMoments**
Repeated Measures	**StatsANOVA2RMTest**

Test Name	Where to find it
Scheffe equality of means	**StatsScheffeTest**
Shapiro-Wilk test for normality	**StatsShapiroWilkTest**
Spearman	**StatsRankCorrelationTest**
Student-Newman-Keuls	**StatsNPMCTest**
Tukey Test	**StatsTukeyTest StatsLinearRegression**, **StatsMultiCorrelationTest**, **StatsNPMCTest**
Two-Factor ANOVA	**StatsANOVA2NRTest**
T-test	**StatsTTest**
Watson’s nonparametric two-sample U2	**StatsWatsonUSquaredTest**, **StatsCircularTwoSampleTest**
Watson-Williams	**StatsWatsonWilliamsTest**
Weighted-rank correlation test	**StatsWRCorrelationTest**
Wheeler-Watson nonparametric test	**StatsWheelerWatsonTest**
Wilcoxon-Mann-Whitney two-sample	**StatsWilcoxonRankTest**
Wilcoxon signed rank	**StatsWilcoxonRankTest**

Statistical Test Operations by Data Format

The following tables group statistical operations and functions according to the format of the input data.

Tests for Single Waves

Analysis Method	Comments
StatsChiTest	Compares with known binned values
StatsCircularMoments	WaveStats for circular data
StatsKendallTauTest	Similar to Spearman's correlation
StatsMedian	Returns the median
StatsNPNominalSRTest	Nonparametric serial randomness test
StatsQuantiles	Computes quantiles and more
StatsResample	Bootstrap analysis
StatsSRTest	Serial randomness test
StatsTrimmedMean	Returns the trimmed mean
StatsTTest	Compares with known mean
Sort	Reorders the data
WaveStats	Basic statistical description
StatsJBTest	Jarque-Bera test for normality
StatsKSTest	Limited scope test for normality
StatsDIPTest	Hartigan test for unimodality
StatsShapiroWilkTest	Shapiro-Wilk test for normality

Tests for Two Waves

Analysis Method	Comments
StatsChiTest	Chi-squared statistic for comparing two distributions
StatsCochranTest	Randomized block or repeated measures test
StatsCircularTwoSampleTest	Second order analysis of angles
StatsDunnettTest	Compares multiple groups to a control
StatsFTest	Computes ratio of variances
StatsFriedmanTest	Nonparametric ANOVA
StatsKendallTauTest	Similar to Spearman's correlation
StatsTTest	Compares the means of two distributions
StatsANOVA1Test	One-way analysis of variances
StatsLinearRegression	Linear regression analysis
StatsLinearCorrelationTest	Linear correlation coefficient and its error
StatsRankCorrelationTest	Computes Spearman's rank correlation
StatsVariancesTest	Compares variances of waves
StatsWilcoxonRankTest	Two-sample or signed rank test
StatsWatsonUSquaredTest	Compares two populations of circular data
StatsWatsonWilliamsTest	Compares mean values of angular distributions
StatsWheelerWatsonTest	Compares two angular distributions

Tests for Multiple or Multidimensional Waves

Analysis Method	Comments
StatsANOVA1Test	One-way analysis of variances
StatsANOVA2Test	Two-factor analysis of variances
StatsANOVA2RMTest	Two-factor repeated measure ANOVA
StatsCochranTest	Randomized block or repeated measures test
StatsContingencyTable	Contingency table analysis
StatsDunnettTest	Comparing multiple groups to a control
StatsFriedmanTest	Nonparametric ANOVA
StatsNPMCTest	Nonparametric multiple comparison tests
StatsScheffeTest	Tests equality of means
StatsTukeyTest	Multiple comparisons based on means
StatsWatsonWilliamsTest	Compares mean values of angular distributions
StatsWheelerWatsonTest	Compares two angular distributions

Statistical Test Operations for Angular/Circular Data

StatsAngularDistanceTest

StatsCircularMoments

StatsCircularMeans

StatsCircularTwoSampleTest

StatsCircularCorrelationTest

StatsHodgesAjneTest

StatsWatsonUSquaredTest

StatsWatsonWilliamsTest

StatsWheelerWatsonTest

Statistical Test Operations: Nonparametric Tests

Operation	Comments
StatsAngularDistanceTest	
StatsFriedmanTest	
StatsCircularTwoSampleTest	Parametric or nonparametric
StatsCircularCorrelationTest	Parametric or nonparameteric
StatsCircularMeans	Parametric or nonparameteric
StatsHodgesAjneTest	
StatsKendallTauTest	
StatsKWTest	
StatsNPMCTest	
StatsNPNominalSRTest	
StatsRankCorrelationTest	
StatsWatsonUSquaredTest	
StatsWheelerWatsonTest	
StatsWilcoxonRankTest	

Noise Functions

The following functions return numbers from a pseudo-random distribution of the specified shapes and parameters. Except for enoise and gnoise where you have an option to select a random number generator, the remaining noise functions use a Mersenne Twister algorithm for the initial uniform pseudo-random distribution. Note that whenever you need repeatable results you should use SetRandomSeed prior to executing any of the noise functions.

The following noise generation functions are available:

binomialNoise

enoise

expNoise

gammaNoise

gnoise

hyperGNoise

logNormalNoise

lorentzianNoise

poissonNoise

StatsPowerNoise

StatsVonMisesNoise

wnoise

Cumulative Distribution Functions

A cumulative distribution function (CDF) is the integral of its respective probability distribution function (PDF). CDFs are usually well behaved functions with values in the range [0,1]. CDFs are important in computing critical values, P values and power of statistical tests.

Many CDFs are computed directly from closed form expressions. Others can be difficult to compute because they involve evaluating a very large number of states, e.g., Friedman or USquared distributions. In these cases you have the following options:

1. Use a built-in table that consists of exact, precomputed values.
2. Compute an approximate CDF based on the prevailing approximation method or using a Monte-Carlo approach.
3. Compute the exact CDF.

Built-in tables are ideal if they cover the range of the parameters that you need. Monte-Carlo methods can be tricky in the sense that repeated application may return small variations in values. Computing the exact CDF may be desirable, but it is often impractical. In most situations the range of parameters that is practical to compute on a desktop machine is already covered in the built-in tables. Larger parameters have not been considered because they take days to compute or because they require 64 bit processors. In addition, most of the approximations tend to improve with increasing size of the parameters.

The functions to calculate values from CDFs are as follows:

StatsBetaCDF

StatsBinomialCDF

StatsCauchyCDF

StatsChiCDF

StatsCMSSDCDF

StatsDExpCDF

StatsErlangCDF

StatsEValueCDF

StatsExpCDF

StatsHyperGCDF

StatsKuiperCDF

StatsLogisticCDF

StatsLogNormalCDF

StatsMaxwellCDF

StatsInvMooreCDF

StatsNBinomialCDF

StatsNCFCDF

StatsNCTCDF

StatsQCDF

StatsRayleighCDF

StatsRectangularCDF

StatsRunsCDF

StatsSpearmanRhoCDF

StatsStudentCDF

StatsTopDownCDF

StatsTriangularCDF

StatsUSquaredCDF

StatsFCDF

StatsFriedmanCDF

StatsGammaCDF

StatsGeometricCDF

StatsNormalCDF

StatsParetoCDF

StatsPoissonCDF

StatsPowerCDF

StatsVonMisesCDF

StatsQCDF

StatsWaldCDF

StatsWeibullCDF

Probability Distribution Functions

Probability distribution functions (PDF) are sometimes known as probability densities. In the case of continuous distributions, the area under the curve of the PDF for each interval equals the probability for the random variable to fall within that interval. The PDFs are useful in calculating event probabilities, characteristic functions and moments of a distribution.

The functions to calculate values from PDFs are as follows:

StatsBetaPDF

StatsBinomialPDF

StatsCauchyPDF

StatsChiPDF

StatsDExpPDF

StatsErlangPDF

StatsErrorPDF

StatsEValuePDF

StatsExpPDF

StatsFPDF

StatsGammaPDF

StatsGeometricPDF

StatsHyperGPDF

StatsLogNormalPDF

StatsMaxwellPDF

StatsBinomialPDF

StatsNCChiPDF

StatsNCFPDF

StatsNCTPDF

StatsNormalPDF

StatsParetoPDF

StatsPoissonPDF

StatsPowerPDF

StatsRayleighPDF

StatsRectangularPDF

StatsStudentPDF

StatsTriangularPDF

StatsVonMisesPDF

StatsWaldPDF

StatsWeibullPDF

Inverse Cumulative Distribution Functions

The inverse cumulative distribution functions return the values at which their respective CDFs attain a given level. This value is typically used as a critical test value. There are very few functions for which the inverse CDF can be written in closed form. In most situations the inverse is computed iteratively from the CDF.

The functions to calculate values from inverse CDFs are as follows:

StatsInvBetaCDF

StatsInvBinomialCDF

StatsInvCauchyCDF

StatsInvChiCDF

StatsInvCMSSDCDF

StatsInvDExpCDF

StatsInvEValueCDF

StatsInvExpCDF

StatsInvFCDF

StatsInvKuiperCDF

StatsInvLogisticCDF

StatsInvLogNormalCDF

StatsInvMaxwellCDF

StatsInvMooreCDF

StatsInvNBinomialCDF

StatsInvNCFCDF

StatsInvNormalCDF

StatsInvParetoCDF

StatsInvQpCDF

StatsInvRayleighCDF

StatsInvRectangularCDF

StatsInvSpearmanCDF

StatsInvStudentCDF

StatsInvTopDownCDF

StatsInvTriangularCDF

StatsInvUSquaredCDF

StatsInvVonMisesCDF

StatsInvFriedmanCDF

StatsInvGammaCDF

StatsInvGeometricCDF

StatsInvPoissonCDF

StatsInvPowerCDF

StatsInvQCDF

StatsInvWeibullCDF

General Purpose Statistics Operations and Functions

This group includes operations and functions that existed before IGOR Pro 6.0 and some general purpose operations and functions that do not belong to the main groups listed.

binomial

binomialln

erf

erfc

inverseErf

inverseErfc

Sort

StatsCircularMoments

StatsCorrelation

StatsMedian

StatsQuantiles

StatsResample

StatsTrimmedMean

StudentA

StudentT

WaveStats

StatsPermute

Hazard and Survival Functions

Igor does not provide built-in functions to calculate the Survival or Hazard functions. They can be calculated easily from the **Probability Distribution Functions** on page III-391 and **Cumulative Distribution Functions** on page III-390.

In the following, the cumulative distribution functions are denoted by *F*(*x*) and the probability distribution functions are denoted by *p*(*x*).

The Survival Function *S*(*x*) is given by

$$S(x) = 1 - F(x).$$

The Hazard function *h*(*x*) is given by

$$h(x) = \frac{p(x)}{S(x)} = \frac{p(x)}{1 - F(x)}.$$

The cumulative hazard function *H*(*x*) is

$$H(x) = \int_{-\infty}^{x} h(u)\,du,$$

$$H(x) = -\ln\left[1 - F(x)\right].$$

Inverse Survival Function *Z*(*a*) is

$$Z(\alpha) = G(1 - \alpha),$$

where *G*() is the inverse CDF (see **Inverse Cumulative Distribution Functions** on page III-391).

Statistics Procedures

Several procedure files are provided to extend the built-in statistics capability described in this chapter. Some of these procedure files provide user interfaces to the built-in statistics functionality. Others extend the functionality.

In the Analysis menu you will find a Statistics item that brings up a submenu. Selecting any item in the submenu will cause all the statistics-related procedure files to be loaded, making them ready to use. Alternatively, you can load all the statistics procedures by adding the following include statement to the top of your procedure window:

```
#include <AllStatsProcedures>
```

Functionality provided by the statistics procedure files includes the 1D Statistics Report package for automatic analysis of single 1D waves, and the ANOVA Power Calculations Panel, as well as functions to create specialized graphs:

StatsAutoCorrPlot()

StatsBoxPlot()

StatsPlotLag()

StatsProbPlot()

StatsPlotHistogram()

Also included are these convenience functions:

WM_2MeanConfidenceIntervals()

WM_2MeanConfidenceIntervals2()

WM_BernoulliCdf()

WM_BinomialPdf()

WM_CIforPooledMean()

WM_CompareCorrelations()

WM_EstimateMinDetectableDiff()

WM_EstimateReqSampleSize()

WM_EstimateReqSampleSize2()

WM_EstimateSampleSizeForDif()

WM_GetANOVA1Power()

WM_GetGeometricAverage()

WM_GetHarmonicMean()

WM_GetPooledMean()

WM_GetPooledVariance()

WM_MCPointOnRegressionLines()

WM_MeanConfidenceInterval()

WM_OneTailStudentA()

WM_OneTailStudentT()

WM_PlotBiHistogram()

WM_RankForTies()

WM_RankLetterGradesWithTies()

WM_RegressionInversePrediction()

WM_SSEstimatorFunc()

WM_SSEstimatorFunc2()

WM_SSEstimatorFunc3()

WM_VarianceConfidenceInterval()

WM_WilcoxonPairedRanks()

WM_StatsKaplanMeier()

Statistics References

Ajne, B., A simple test for uniformity of a circular distribution, *Biometrica*, *55*, 343-354, 1968.

Bradley, J.V., *Distribution-Free Statistical Tests*, Prentice Hall, Englewood Cliffs, New Jersey, 1968.

Cheung, Y.K., and J.H. Klotz, The Mann Whitney Wilcoxon distribution using linked lists, *Statistica Sinica*, *7*, 805-813, 1997.

Copenhaver, M.D., and B.S. Holland, Multiple comparisons of simple effects in the two-way analysis of variance with fixed effects, *Journal of Statistical Computation and Simulation*, *30*, 1-15, 1988.

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

Fisher, N.I., *Statistical Analysis of Circular Data*, 295pp., Cambridge University Press, New York, 1995.

Iman, R.L., and W.J. Conover, A measure of top-down correlation, *Technometrics*, *29*, 351-357, 1987.

Kendall, M.G., *Rank Correlation Methods*, 3rd ed., Griffin, London, 1962.

Klotz, J.H., *Computational Approach to Statistics*.

Moore, B.R., A modification of the Rayleigh test for vector data, *Biometrica*, *67*, 175-180, 1980.

Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

van de Wiel, M.A., and A. Di Bucchianico, Fast computation of the exact null distribution of Spearman’s rho and Page’s L statistic for samples with and without ties, *J. of Stat. Plan. and Inference*, *92*, 133-145, 2001.

Wallace, D.L., Simplified Beta-Approximation to the Kruskal-Wallis H Test, *J. Am. Stat. Assoc*., *54*, 225-230, 1959.

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

Chapter

III-13

Procedure Windows

Overview

This chapter explains what procedure windows are, how they are created and organized, and how you work with them. It does not cover programming. See Chapter IV-2, **Programming Overview** for an introduction.

A procedure window is where Igor procedures are stored. Igor procedures are the macros, functions and menu definitions that you create or that Igor creates automatically for you.

The content of a procedure window is stored in a procedure file. In the case of a packed Igor experiment, the procedure file is packed into the experiment file.

Types of Procedure Files

There are four types of procedure files:

- The experiment procedure file, displayed in the built-in procedure window
- Global procedure files, displayed in auxiliary procedure windows
- Shared procedure files, displayed in auxiliary procedure windows
- Auxiliary experiment procedure files, displayed in auxiliary procedure windows

The built-in procedure window holds experiment-specific procedures of the currently open experiment. This is the only procedure window that beginning or casual Igor users may need.

All other procedure windows are called **auxiliary** to distinguish them from the built-in procedure window. You create an auxiliary procedure window using Windows→New→Procedure. You can then save it to a standalone file, using File→Save Procedure As, or allow Igor to save it as part of the current experiment.

A global procedure file contains procedures that you might want to use in any experiment. It must be saved as a standalone file in the "Igor Procedures" folder of your **Igor Pro User Files** folder. Procedure files in "Igor Procedures" are automatically opened by Igor at startup and left open until Igor quits. This is the easiest way to make procedures available to multiple experiments.

A shared procedure file contains procedures that you want to use in more than one experiment but that you don't want to be open all of the time. It must be saved as a standalone file. The recommended location is the "User Procedures" folder of your **Igor Pro User Files** folder.

An auxiliary experiment procedure file contains procedures that you want to use in a single experiment but want to keep separate from the built-in procedure window for organizational purposes. In a packed experiment it is saved as a packed file within the experiment file. In an unpacked experiment it is saved as a standalone file in the experiment folder.

Working with the Built-in Procedure Window

Procedures that are specific to the current experiment are usually stored in the built-in procedure window. Also, when Igor automatically generates procedures, it stores them in the built-in procedure window.

To show the built-in procedure window, choose Windows→Procedure Windows→Procedure Window or press Command-M (*Macintosh*) or Ctrl+M (*Windows*). To hide it, click the close button or press Command-W (*Macintosh*) or Ctrl+W (*Windows*).

To create a procedure, just type it into the procedure window.

The contents of the built-in procedure window are automatically stored when you save the current Igor experiment. For unpacked experiments, the contents are stored in a file called “procedure” in the experiment folder. For packed experiments, the contents are stored in the packed experiment file. When you open an experiment Igor loads its procedures back into the built-in procedure window.

Compiling the Procedures

When you modify the text in the procedure window, you will notice that a Compile button appears at the bottom of the window.

Clicking the Compile button scans the procedure window looking for macros, functions and menu definitions. Igor compiles user-defined functions, generating low-level instructions that can be executed quickly.

Igor also compiles the code in the procedure window if you choose Compile from the Macros menu or if you activate any window other than a procedure or help window.

Templates Pop-Up Menu

The Templates pop-up menu lists all of the built-in and external operations and functions in alphabetical order and also lists the common flow control structures.

If you choose an item from the menu, Igor inserts the corresponding template in the procedure window.

If you select, click in, or click after a recognized operation, function or flow-control keyword in the procedure window, two additional items are listed at the top of the menu. The first item inserts a template and the second takes you to help.

In addition to templates, procedure windows also support **Command Completion**.

Procedure Navigation Bar

The procedure navigation bar appears at the top of each procedure window. It consists of a menu and a settings button. The menu provides a quick way to find procedures in the active procedure window.

Certain syntax errors prevent Igor from parsing procedures. In that case, the menu shows no procedures.

You can hide the navigation bar by choosing Misc-Miscellaneous Settings, selecting the Text Editing section, and unchecking the Show Navigation Bar checkbox.

Write-Protect Icon

Procedure windows have a write-enable/write-protect icon which appears in the lower-left corner of the window and resembles a pencil. If you click this icon, Igor Pro displays an icon indicating that the procedure window is write-protected. The main purpose of this is to prevent accidental alteration of shared procedure files.

Igor opens included user procedure files for writing but turns on the write-protect icon so that you will get a warning if you attempt to change them. If you do want to change them, simply click the write-protect icon to turn protection off. You can turn this default write-protection off via the Text Encoding section of the Miscellaneous Settings dialog.

If a procedure file is opened for reading only, you will see a lock icon instead of the pencil icon. A file opened for read-only can not be modified.

WaveMetrics procedures, in the WaveMetrics Procedures folder, are assumed to be the same for all Igor users and should not be modified by users. Therefore, Igor opens included WaveMetrics procedures for reading only.

Magnifier Icon

You can magnify procedure text to make it more readable. See **Text Magnification** on page II-53 for details.

Creating Procedure Windows

There are three ways to create procedures:

- Automatically by Igor

- Manually, when you type in a procedure window
- Semiautomatically, when you use various dialogs

Igor offers to automatically create a **window recreation macro** when you close a target window. A window recreation macro is a procedure that can recreate a graph, table, page layout or control panel window. See **Saving a Window as a Recreation Macro** on page II-47 for details.

You can add procedures by merely typing them in a procedure window.

You can create user-defined controls in a graph or control panel. Each control has an optional **action procedure** that runs when the control is used. You can create a control and its corresponding action procedure using dialogs that you access through the Add Controls submenu in the Graph or Panel menus. These action procedures are initially stored in the built-in procedure window.

You can create user-defined curve fitting functions via the Curve Fitting dialog. These functions are initially stored in the built-in procedure window.

Creating New Procedure Files

You create a new procedure file if you want to write procedures to be used in more than one experiment.

Note: There is a risk in sharing procedure files among experiments. If you copy the experiment to another computer and forget to also copy the shared files, the experiment will not work on the other computer. See **References to Files and Folders** on page II-24 for further details.

If you do create a shared procedure file then you are responsible for copying the shared file when you copy an experiment that relies on it.

To create a new procedure file, choose Windows→New→Procedure. This creates a new procedure *window*. The procedure *file* is not created until you save the procedure window or save the experiment.

You can explicitly save the procedure window by choosing File→Save Procedure As or by closing it and choosing to save it in the resulting dialog. This saves the file as an auxiliary procedure file, separate from the experiment.

If you don’t save the procedure window explicitly, Igor saves it as part of the current experiment the next time you save the experiment.

Opening an Auxiliary Procedure File

You can open a procedure file using the File→Open File→Procedure menu item.

When you open a procedure file, Igor displays it in a new procedure window. The procedures in the window can be used in the current experiment. When you save the current experiment, Igor will save a *reference* to the shared procedure file in the experiment file. When you later open the experiment, Igor will reopen the procedure file.

For procedure files that you use from a large number of experiments, it is better to configure the files as global procedure files. See **Global Procedure Files** on page III-399.

For commonly used auxiliary files, it is better to use the include statement. See **Including a Procedure File** on page III-401.

Showing Procedure Windows

We usually show procedure windows when we are doing programming and hide them for normal use.

To show the built-in procedure window, choose Windows→Procedure Windows→Procedure Window or press Command-M (*Macintosh*) or Ctrl+M (*Windows*). To show auxiliary procedure windows, use the Windows→Procedure Windows submenu.

If you have more than one procedure window, you can cycle to the next procedure window by pressing Command-Option-M (*Macintosh*) or Ctrl+Alt+M (*Windows*). Pressing Command-Shift-Option-M (*Macintosh*) or Ctrl+Shift+Alt+M (*Windows*) hides the active procedure window and shows the next one.

You can also show a procedure window by choosing a menu item added by that window while pressing Option (*Macintosh*) or Alt (*Windows*). This feature works only if the top window is a procedure window.

You can show all procedure windows and hide all procedure windows using the Windows→Show and Windows→Hide submenus.

Hiding and Killing Procedure Windows

The built-in procedure window always exists as part of the current experiment. You can hide it by clicking the close button, pressing Command-W (*Macintosh*) or Ctrl+W (*Windows*) or by choosing Hide from the Windows menu. You can not kill it.

Auxiliary procedure files can be opened (added to the experiment), hidden and killed (removed from the experiment). This leads to a difference in behavior between auxiliary procedure windows and the built-in procedure window.

When you click the close button of an auxiliary procedure file, Igor presents the Close Procedure Window dialog to find out what you want to do.

If you just want to hide the window, you can press Shift while clicking the close button. This skips the dialog and just hides the window.

Killing a procedure window closes the window and removes it from the current experiment but does not delete or otherwise affect the procedure file with which the window was associated. If you have made changes or the procedure was not previously saved, you will be presented with the choice of saving the file before killing the procedure.

The Close item of the Windows menu and the equivalent, Command-W (*Macintosh*) or Ctrl+W (*Windows*), behave the same as the close button.

Saving All Standalone Procedure Files

When a procedure window is active, you can save all modified standalone procedure files at once by choosing File→Save All Standalone Procedure Files. This saves only standalone procedure files. It does not save the built-in procedure window, packed procedure files, or procedure windows that were just created and never saved to disk; these are saved when you save the experiment.

Autosaving Standalone Procedure Files

Igor can automatically save modified standalone procedure files. See **Autosave** on page II-36 for details.

Global Procedure Files

Global procedure files contain procedures that you want to be available in all experiments. They differ from other procedure files in that Igor opens them automatically and never closes them. Configuring a procedure file as a global procedure file is the easiest way to make it widely available.

When Igor starts running, it searches "Igor Pro Folder/Igor Procedures" and "Igor Pro User Files/Igor Procedures" (see **Igor Pro User Files** on page II-31 for details), as well as files and folders referenced by aliases or shortcuts. Igor opens any procedure file that it finds during this search as a global procedure file.

You should save your global procedure files in "Igor Pro User Files/Igor Procedures". You can locate this folder by choosing Help→Show Igor Pro User Files.

Igor opens global procedure files with write-protection on since they presumably contain procedures that you have already debugged and which you don’t want to inadvertently modify. If you *do* want to modify a global procedure file, click the write-protect icon (pencil in lower-left corner of the window). You can turn this default write-protection off via the Text Encoding section of the Miscellaneous Settings dialog.

When you create a new experiment or open an existing one, Igor normally closes any open procedure files, but it leaves global procedure files open. You can explicitly close a global procedure window at any time and then you can manually reopen it. Igor will not automatically reopen it until the next time Igor is launched.

Although its procedures can be used by the current experiment, a global procedure file is not part of the current experiment. Therefore, Igor does not save a global procedure file or a reference to a global procedure file inside an experiment file.

Note: There is a risk in using global procedure files. If you copy an experiment that relies on a global procedure file to another computer and forget to also copy the global procedure file, the experiment will not work on the other computer.

Saving Global Procedure Files

If you modify a global procedure file, Igor will save it when you save the current experiment even though the global procedure file is not part of the current experiment. However, you might want to save the procedure file without saving the experiment. For this, use the File→Save Procedure menu item.

Shared Procedure Files

You may develop procedures that you want to use in several but not all of your experiments. You can facilitate this by creating a shared procedure file. This is a procedure file that you save in its own file, separate from any experiment. Such a file can be opened from any experiment.

There are two ways to open a shared procedure file from an experiment:

- By explicitly opening it, using the File→Open File submenu or by double-clicking it or by drag-and-drop
- By adding an include statement to your experiment procedure window

The include method is preferred and is described in detail under **Including a Procedure File** on page III-401.

When Igor encounters an include statement, it searches for the included file in "Igor Pro Folder/User Procedures" and in "Igor Pro User Files/User Procedures" (see **Igor Pro User Files** on page II-31 for details). The Igor Pro User Files folder is the recommended place for storing user files. You can locate it by choosing Help→Show Igor Pro User Files.

You can store your shared procedure procedure file directly in "Igor Pro User Files/User Procedures" or you can store it elsewhere and put an alias (*Macintosh*) or shortcut (*Windows*) for it in "Igor Pro User Files/User Procedures". If you have many shared procedure files you can put them all in your own folder and put an alias/shortcut for the folder in "Igor Pro User Files/User Procedures".

When you explicitly open a procedure file using the Open File submenu, by double-clicking it, or by drag-and-drop, you are adding it to the current experiment. When you save the experiment, Igor saves a *reference* to the procedure file in the experiment file. When you close the experiment, Igor closes the procedure file. When you later reopen the experiment, Igor reopens the procedure file.

When you use an include statement, the included file is not considered part of the experiment but is still referenced by the experiment. Igor automatically opens the included file when it hits the include statement during procedure compilation.

Note: There is a risk in sharing procedure files among experiments. If you copy the experiment to another computer and forget to also copy the shared files, the experiment will not work on the other computer. See **References to Files and Folders** on page II-24 for more explanation.

Saving Shared Procedure Files

If you modify a shared procedure file, Igor saves it when you save the experiment that is sharing it. However, you might want to save the procedure file without saving the experiment. For this, choose File→Save Procedure.

Including a Procedure File

You can put an include statement in any procedure file. An include statement automatically opens another procedure file. This is the recommended way of accessing files that contain utility routines which you may want to use in several experiments. Using an include statement is preferable to opening a procedure file explicitly because it doesn't rely on the exact location of the file in the file system hierarchy.

Here is a typical include statement:

```
#include <MatrixToXYZ>
```

This automatically opens the MatrixToXYZ.ipf file supplied by WaveMetrics in "Igor ProFolder/WaveMetrics Procedures/Data Manipulation". The angle brackets tell Igor to search the "Igor Pro Folder/WaveMetrics Procedures" hierarchy.

To see what WaveMetrics procedure files are available, choose Help→Help Windows→WM Procedures Index.

You can include your own utility procedure files by using double-quotes instead of the angle-brackets shown above:

```
#include "Your Procedure File"
```

The double-quotes tell Igor to search the "Igor Pro Folder/User Procedures" and "Igor Pro User Files/User Procedures" hierarchies (see **Igor Pro User Files** on page II-31 for details) for the named file. Igor searches those folders and subfolders and files or folders referenced by aliases/shortcuts in those folders.

These are the two main variations on the include statement. For details on less frequently used variations, see **The Include Statement** on page IV-166.

Included procedure files are not considered part of the experiment but are automatically opened by Igor when it compiles the experiment's procedures.

To prevent accidental alteration of an included procedure file, Igor opens it either write-protected (User Procedures) or read-only (WaveMetrics Procedures). See **Write-Protect Icon** on page III-397.

A #include statement must omit the file's ".ipf" extension:

```
#include <Strings as Lists>      // RIGHT

#include <Strings as Lists.ipf>  // WRONG
```

Creating Packages

A package is a set of procedure files, help files and other support files that add significant functionality to Igor.

Igor comes pre-configured with numerous WaveMetrics packages accessed through the Data→Packages, Analysis→Packages, Misc→Packages, Windows→New→Packages and Graph→Packages submenus as well as others.

Intermediate to advanced programmers can create their own packages. See **Packages** on page IV-246 for details.

Invisible Procedure Files

If you create a package of Igor procedures to be used by regular Igor users (as opposed to programmers), you may want to hide the procedures to reduce clutter or to eliminate the possibility that they might inadvertently change them. You can do this by making the procedure files invisible.

Invisible procedure files are omitted from Igor's Procedure Windows submenu which appears in the Windows menu. This keeps them out of the way of regular users.

There are three ways to make a procedure file invisible. In order of difficulty they are:

- Using the #pragma hide compiler directive
- Using an independent module
- Using the operating-system-supplied file visibility property

Invisible Procedure Windows Using #pragma hide

You can make a procedure file invisible by inserting this compiler directive in the file:

#pragma hide=1

This prevents the procedure window from being listed in the Windows→Procedures submenu. Procedures windows that include this compiler directive become invisible on the next compile.

You can make these windows visible during development by executing:

```
SetIgorOption IndependentModuleDev=1
```

and return them to invisible by executing:

```
SetIgorOption IndependentModuleDev=0
```

You must force a compile for this to take effect.

Prior to Igor Pro 6.30 this feature worked for #included procedure files only, not for packed and standalone procedure files.

Invisible Procedure Windows Using Independent Modules

You can also make a set of procedure files invisible by making them an independent module. The independent module technique is more difficult to implement but has additional advantages. For details, see **The IndependentModule Pragma** on page IV-55.

Invisible Procedure Files Using The Files Visibility Property

This section discusses making procedure files invisible by setting the operating-system-supplied file "visible" property.

Note: This is an old technique that is no longer recommended. It may not be supported in future versions of Igor.

When Igor opens a procedure file, it asks the operating system if the file is invisible (*Macintosh*) or hidden (*Windows*). We will use the term "invisible" to mean invisible on Macintosh and hidden on Windows.

If the file is invisible, Igor makes the file inaccessible to the user. Igor checks the invisible property only when it opens the file. It does not pay attention to whether the property is changed while the file is open.

You create Igor procedures using normal visible procedure files, typically all in a folder or hierarchy of folders. When it comes time to ship to the end user, you set the files to be invisible. If you set a file to be invisible, you should also make it read-only.

You can use the **SetFileFolderInfo** operation (see page V-845) to set the visibility and read-only properties of a file:

```
SetFileFolderInfo /INV=1 /RO=1 "<path to file>"
```

The file will be invisible in Igor the next time you open it, typically by opening an experiment or using a #include statement.

On Macintosh, the file disappears from the Finder when you execute the command.

On Windows, merely setting the hidden property is not sufficient to actually hide the file. It is actually hidden only if the Hide Files of These Types radio button in the View Options dialog is turned on. You can access this dialog by opening a folder in the Windows desktop and choosing View→Options from the folder's menu bar. Although the hidden property in Windows does not guarantee that the file will be hidden in the Windows desktop, it does guarantee that it will be hidden from within Igor.

After the files are set to be invisible and read-only, if you want to edit them in Igor, you must close them (typically by closing the open experiment), set the files to be visible and read/write again, and then open them again.

Igor's behavior is changed in the following ways for a procedure file set to invisible:

1. The window will not appear in the Windows→Procedure Windows menu.
2. Procedures in the window will not appear in the contextual pop-up menu in other procedure windows (Control-click on *Macintosh*, right-click on *Windows*).
3. If the user presses Option (*Macintosh*) or Alt (*Windows*) while choosing the name of a procedure from a menu, Igor will do nothing rather than its normal behavior of displaying the procedure window.
4. When cycling through procedure windows using Command-Shift-Option-M (*Macintosh*) or Ctrl+Shift+Alt+M (*Windows*), Igor will omit the procedure window.
5. The Button Control dialog, Pop-Up Menu Control dialog, and other control dialogs will not allow you to edit procedures in the invisible file.
6. The Edit Procedure and Debug buttons will not appear in the Macro Execute Error dialog.
7. If an error occurs in a function in the invisible file and the Debug On Error flag (Procedure menu) is on, the debugger will act as if Debug On Error were off.
8. The debugger won't allow you to step into procedures in the invisible file.
9. The **ProcedureText** and **ProcedureVersion** functions and **DisplayProcedure** will act as if procedures in the invisible file don't exist. The **MacroList** and **FunctionList** functions will, however work as usual.
10. The GetIgorProcedure and SetIgorProcedure XOPSupport routines in the XOP Toolkit will act as if procedures in the invisible file don't exist. The GetIgorProcedureList function will, however work as usual.

Inserting Text Into a Procedure Window

On occasion, you may want to copy text from one procedure file to another. With the procedure window active, choose Edit→Insert File. This displays an Open File dialog in which you can choose a file and then inserts its contents into the procedure window.

Adopting a Procedure File

Adoption is a way for you to copy a procedure file into the current experiment and break the connection to its original file. The reason for doing this is to make the experiment self-contained so that, if you transfer it to another computer or send it to a colleague, all of the files needed to recreate the experiment are stored in the experiment itself.

To adopt a file, open its associated window and choose File→Adopt Window. This item is available only if the active window is a notebook or procedure file that is stored separate from the current experiment and the current experiment has been saved to disk.

If the current experiment is stored in packed form then, when you adopt a file, Igor does a save-as to a temporary file. When you subsequently save the experiment, the contents of the temporary file are stored in the packed experiment file.

If the current experiment is stored in unpacked form then, when you adopt a file, Igor does a save-as to the experiment's home folder. When you subsequently save the experiment, Igor updates the experiment's recreation procedures to open the new file in the home folder instead of the original file. If you adopt a file in an unpacked experiment and then you do not save the experiment, the new file will still exist in the home folder but the experiment's recreation procedures will still refer to the original file. Thus, you should normally save the experiment soon after adopting a file.

Adoption does not cause the original file to be deleted. You can delete it from the desktop if you want.

To "unadopt" a procedure file, choose Save Procedure File As from the File menu.

It is possible to do adopt multiple files at one time. For details see **Adopt All** on page II-25.

Auto-Compiling

If you modify a procedure window and then activate a non-procedure window other than a help window, Igor automatically compiles the procedures.

If you have a lot of procedures and compiling takes a long time, you may want to turn auto-compiling off. You can do this by deselecting the Auto-compile item in the Macros menu. This item appears only when the procedures need to be compiled (you have modified a procedure file or opened a new one). If you uncheck it item, Igor will not auto-compile and compilation will be done only when you click the Compile button or choose Compile from the Macros menu.

Debugging Procedures

Igor includes a symbolic debugger. This is described in **The Debugger** on page IV-212.

Finding Text

To find text in the active window, choose Edit→Find or press Command-F (*Macintosh*) or Ctrl+F (*Windows*). This displays the Find bar. See **Finding Text in the Active Window** on page II-52 for details.

To search multiple windows for text, choose Edit→Find in Multiple Windows. See **Finding Text in Multiple Windows** on page II-53 for details.

On Macintosh, you can search for the next occurrence of a string by selecting the string, pressing Command-E (Use Selection for Find in the Edit menu), and pressing Command-G (Find Same in the Edit menu).

On Windows, you can search for the next occurrence of a string by selecting the string and pressing Ctrl+H (Find Selection in the Edit menu).

After doing a find, you can search for the same text again by pressing Command-G (*Macintosh*) or Ctrl+G (*Windows*) (Find Same in the Edit menu). You can search for the same text but in the reverse direction by pressing Command-Shift-G (*Macintosh*) or Shift+Ctrl+G (*Windows*).

You can search back and forth through a procedure window by repeatedly pressing Command-G and Command-Shift-G (*Macintosh*) or Ctrl+G and Shift-Ctrl+G (*Windows*).

Replacing Text

To replace text in the active window, press Command-R (*Macintosh*) or Ctrl+R (*Windows*). This displays the Find bar in replace mode. See **Find and Replace** on page II-53 for details.

The Search Selected Text Only option is handy for limiting the replacement to a particular procedure.

While replacing text is undoable, the potential for unintended and wide-ranging consequences is such that we recommend saving the file before doing a mass replace so you can revert-to-saved if necessary.

Another method for searching and replacing consists of repeating Command-F or Ctrl-F (Find) followed by Command-V or Ctrl-V (Paste). This has the virtue of allowing you to inspect each occurrence of the target text before replacing it.

Printing Procedure Text

To print the active procedure window, first deselect all text and then choose File→Print Procedure Window.

To print part of the active procedure window, select the text you want to print and choose File→Print Procedure Selection.

Indentation

We use indentation to indicate the structure of a procedure. This is described in **Indentation Conventions** on page IV-26.

To make it easy to use the indentation conventions, Igor maintains indentation when you press Return or Enter in a procedure window. It automatically inserts enough tabs in the new line to have the same indentation as the previous line.

To indent more, as when going into the body of a loop, press Return or Enter and then Tab. To indent less, as when leaving the body of a loop, press Return or Enter and then Delete. When you don’t want to change the level of indentation, just press Return.

Included in the Edit menu for Procedure windows, is the Adjust Indentation item, which adjusts indentation of all selected lines of text to match Igor standards. The Edit menu also contains Indent Left and Indent Right commands that add or remove indentation for all selected lines.

Procedure Window Document Settings

The Document Settings dialog controls settings that affect the procedure window as a whole. You can summon it via the Procedure menu.

Igor does not store document settings for plain text files. When you open a procedure file, these settings are determined by preferences. You can capture preferences by choosing Procedure→Capture Procedure Prefs.

Procedure Window Default Tabs

The default tab width setting controls the location of tabs stops in procedure windows. You set this using the Document Settings dialog in the Procedure menu or using a DefaultTab pragma. In a procedure window the default tab width setting controls the location of all tab stops which affect indentation and alignment of comments.

Igor does not store document settings for procedure files, so your preferred default tab width settings apply to all procedure files subsequently opened unless they include a DefaultTab pragma.

Three modes are available: points, spaces, and mixed. In points mode, you specify the default tab width in units of points. In the Document Settings dialog, you can enter this setting in inches, centimeters, or points, but it is always stored as points. Prior to Igor Pro 9.00, this was the only mode available.

In spaces mode, you specify the default tab width in units of spaces.

In mixed mode, you specify the default tab width for procedure windows controlled by proportional fonts in units of points and for procedure windows controlled by monospace fonts in units of spaces.

Mixed mode is recommended for compatibility with other text editors which typically set default tab widths in units of spaces.

In Igor Pro 9.00 and later, when you create a new procedure window, Igor inserts a DefaultTab pragma statement, like this:

```
#pragma DefaultTab = {3,20,4} // Sets default tab width in Igor Pro 9 or later
```

The DefaultTab pragma is explained in the next section.

The Default Tab Pragma For Procedure Files

In Igor Pro 9.00 and later, you can specify default tab settings by entering a pragma statement in the procedure window, like this:

```
#pragma DefaultTab = {<mode>,<width in points>,<width in spaces>}
```

Older versions of Igor ignore this pragma.

The pragma is applied when the procedure file is compiled. It helps keep aligned comments aligned when you share the procedure file with another Igor user or if you change the font, text size, or magnification of the procedure window.

The DefaultTab pragma, if present, sets the default tab width parameters the same as if you set them in the Document Settings dialog. <mode> is 1 for points mode, 2 for spaces mode, and 3 for mixed mode. Specifying the default tab width in spaces keeps comments in procedure files that you share with others aligned so long as the other users use a proportional font for procedure files. The recommended pragma is:

```
#pragma DefaultTab = {3,20,4} // Sets default tab width in Igor Pro 9 or later
```

When you create a new procedure window, Igor inserts this DefaultTab statement in the new window. It overrides your default tab width preferences and specifies mixed mode (3) with the default tab width for proportional fonts set to 20 points and the default tab width for monospace fonts set to 4 spaces. Most people use a monospace font for procedure windows, in which case the default tab width is 4 spaces. If you share the procedure file with other users, this pragma increases the likelihood that aligned comments will remain aligned.

You can remove or edit the automatically inserted DefaultTab pragma. However, we recommend that you leave it as is.

We also recommend that you manually add this pragma to existing procedure files. You will then need to re-align comments.

Procedure Window Text Format Settings

You can specify the font, text size, style and color using items in the Procedure menu. Since procedure windows are always plain text windows (as opposed to notebooks, which can be formatted text) these text settings are the same for all characters in the window.

Igor does not store text format settings for plain text files. When you open a procedure file, these settings are determined by preferences. You can capture preferences by choosing Procedure→Capture Procedure Prefs.

Syntax Coloring

Procedure windows and the command window colorize comments, literal strings, flow control, and other syntax elements. Colors of various elements can be adjusted in two ways.

The first is from the Miscellaneous Settings dialog. Select the Text Editing category and then select the Color tab.

The second is by executing the following **SetIgorOption** colorize commands::

Command	Effect
`SetIgorOption colorize,doColorize=<1 or 0>`	Turn all colorize on or off
`SetIgorOption colorize,OpsColorized=<1 or 0>`	Turn operation keyword colorization on or off
`SetIgorOption colorize,BIFuncsColorized=<1 or 0>`	Turn function keyword colorization on or off
`SetIgorOption colorize,keywordColor=(`*r*`,`*g*`,`*b*`[,`*a*`])`	Color for language keywords
`SetIgorOption colorize,commentColor=(`*r*`,`*g*`,`*b*`[,`*a*`])`	Color for comments
`SetIgorOption colorize,stringColor=(`*r*`,`*g*`,`*b*`[,`*a*`])`	Color for strings
`SetIgorOption colorize,operationColor=(`*r*`,`*g*`,`*b*`[,`*a*`])`	Color for operation keywords
`SetIgorOption colorize,functionColor=(`*r*`,`*g*`,`*b*`[,`*a*`])`	Color for built-in function keywords
`SetIgorOption colorize,poundColor=(`*r*`,`*g*`,`*b*`[,`*a*`])`	Color for #keywords such as #pragma
`SetIgorOption colorize,UserFuncsColorized=1`	Turn colorizing on for user functions
`SetIgorOption colorize,userFunctionColor=(r,g,b[,`*a*`])`	Color for user-defined functions
`SetIgorOption colorize,SpecialFuncsColorized=1`	Turn colorizing on for special operations (MatrixOP and APMath)
`SetIgorOption colorize,specialFunctionColor=(r,g,b[,a])`	Color for special operations (MatrixOP and APMath)

Values for *r*, *g*, *b*, and the optional alpha range from 0 to 65535. Alpha defaults to 65535 (opaque).

Changes to syntax coloring settings, made via the dialog or via SetIgorOption, are saved to preferences and used for future sessions.

Procedure Window Preferences

The procedure window preferences affect the creation of *new* procedure windows. This includes the creation of auxiliary procedure windows and the initialization of the built-in procedure window that occurs when you create a new experiment.

To set procedure preferences, set the attributes of any procedure window and then choose Procedure→Capture Procedure Prefs.

To determine the current preference settings, you must create a new procedure window and examine its settings.

Preferences are stored in the Igor Preferences file. See Chapter III-18, **Preferences**, for further information on preferences.

Double and Triple-Clicking

Double-clicking a word conventionally selects the entire word. Igor extends this idea a bit. In addition to supporting double-clicking, if you triple-click in a line of text, it selects the entire line. If you drag after triple-clicking, it extends the selection an entire line at a time.

Matching Characters

Igor includes a handy feature to help you check parenthesized expressions. If you double-click a parenthesis, Igor tries to find a matching parenthesis on the same line of text. If it succeeds, it selects all of the text between the parentheses. If it fails, it beeps. Consider the command

```
wave1 = exp(log(x)))
```

If you double-clicked on the first parenthesis, it would select “`log(x)`”. If you double-clicked on the last parenthesis, it would beep because there is no matching parenthesis.

If you double-click in-between adjacent parentheses Igor considers this a click on the outside parenthesis.

Igor does matching if you double-click the following characters:

Left and right parentheses	(xxx)
Left and right brackets	[xxx]
Left and right braces	{xxx}
Plain single quotes	'xxx'
Plain double quotes	"xxx"

Code Comments

The Edit menu for procedures contains two items, Commentize and Decommentize, to help you edit and debug your procedure code when you want to comment out large blocks of code, and later, to remove these comments. Commentize inserts comment symbol at the start of each selected line of text. Decommentize deletes any comment symbols found at beginning of each selected line of text.

Code Marker Comments

You can enter specially-formatted comments in procedure files to mark sections of code and later return to those comments using the Code Markers popup menu in the procedure window's navigation bar.

The navigation bar is visible if Show Navigation Bar is checked in the Editing Behavior tab of the Text Editing section of the Miscellaneous Settings dialog. The Code Markers popup menu appears on the left side of the navigation bar and looks like this this: 📍 .

To get a sense of the purpose of code markers, we will look at the "HDF5 Browser.ipf" procedure file. It is an independent module so you must first execute this:

```
SetIgorOption IndependentModuleDev=1    // Enable editing of independent modules
```

Choose Windows→Procedure Windows→HDF5 Browser.ipf

Click the code markers popup menu and note the list of sections in the file such as "Utility Routines", and "Fill List Routines".

Choose Utility Routines from the popup menu. Igor displays the following line:

```
// *** Utility Routines ***
```

That line contains a code marker comment.

The format of a code marker comment is:

```
// *** <code marker text> ***
```

The code marker text, which appears in the code marker popup menu, is everthing after `"// *** "` and before `" ***"`.

The code marker comment must be flush left with no spaces, tabs, or any other text before `"// *** "`.

You can insert a code marker comment manually or by clicking the code markers popup menu and choosing "Insert Code Marker Comment". You can not insert a code marker comment if the procedure window is write protected or is open for read only.

Aligning Comments

You can align comments introduced by "//" in a procedure window by selecting text and choosing Edit→Align Comments. This aligns subsequent comments with the first comment in the selected text. This feature is available for procedure windows only, not for notebooks.

Aligning comments requires that the procedure window use the spaces mode of defining default tabs. To ensure this, when you choose Edit→Align Comments, Igor adds a DefaultTab pragma to the procedure window if it is not already there. See **The Default Tab Pragma For Procedure Files** on page III-406 for details about this pragma. The DefaultTab pragma is ignored by versions of Igor prior to 9.00 so comments that you align using this technique will not appear aligned in earlier versions.

Aligning comments also requires that the procedure window use a monospace font. It is up to you to make sure that this is the case. The factory default fonts for procedure windows, Monaco on Macintosh and Lucida Console on Windows, are monospace fonts.

When you choose Edit→Align Comments, Igor finds the first selected line containing a comment (introduced by "//"). It then works on each subsequent selected line and attempts to align comments by inserting or removing tabs. During this process, Igor removes spaces that appear between the end of the line's active code and the comment.

Comments that are flush left are left unchanged as they typically appear above function definitions and are already positioned as desired.

By default, other pure comment lines (lines with nothing before the comment symbol except for spaces and tabs) are also left unchanged unless:

- The section consists entirely of pure comment lines
- You press the Option key (*Macintosh*) or Alt key (*Windows*) while choosing Edit→Align Comments to override the default behavior

In some cases, exact alignment is not possible. For example, if the active code of the second selected line extends paste the comment in the first selected line, removing all tabs from the second selected line does not achieve alignment. In such cases, Igor does the best it can.

Procedure File Text Encodings

Igor uses UTF-8, a form of Unicode, internally. Prior to Igor7, Igor used non-Unicode text encodings such as MacRoman, Windows-1252 and Shift JIS.

All new procedure files should use UTF-8 text encoding. When you create a new procedure file using Windows→New→Procedure, Igor automatically uses UTF-8.

Igor must convert from the old text encodings to Unicode when opening old files. It is not always possible to get this conversion right. You may get incorrect characters or receive errors when opening files containing non-ASCII text.

For a discussion of these issues, see **Text Encodings** on page III-459 and **Plain Text File Text Encodings** on page III-466.

Procedure Window Shortcuts

To view text window keyboard navigation shortcuts, see **Text Window Navigation** on page II-51.

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To show the built-in procedure window	Press Command-M.	Press Ctrl+M.
To hide the active procedure file and cycle to the next	Press Command-Shift-Option-M.	Press Ctrl+Shift+Alt+M.
To cycle through the open procedure windows	Press Command-Option-M.	Press Ctrl+Alt+M.
To revisit a previously-viewed location	Press Cmd-Option-Left Arrow (Go Back) or Cmd-Option-Right Arrow (Go Forward).	Press Alt+Left Arrow (Go Back) or Alt+Right Arrow (Go Forward).
To get a contextual menu of commonly-used actions	Press Control and click in the body of the procedure window.	Right-click the body of the procedure window.
To execute commands in a procedure window	Select the commands or click in the line containing the commands and press Control-Return or Control-Enter.	Select the commands or click in the line containing the commands and press Ctrl+Enter.
To insert a template	Type or select the name of an operation, Control-click, and choose Insert Template.	Type or select the name of an operation, right-click, and choose Insert Template.
To get help for a built-in or external operation or function	Type or select the name of an operation, Control-click, and choose Help for <name> or choose Help→Help For <name>.	Type or select the name of an operation, right-click, and choose Help for <name> or choose Help→Help For <name>.
To view the definition of a user-defined function	Type or select the name of an operation, Control-click, and choose Go to <name> or choose Edit→Help For <name>.	Type or select the name of an operation, right-click, and choose Go to <name> or choose Edit→Help For <name>.
To find a procedure in the active procedure window	Click the navigation bar at the top of the procedure window.	Click the navigation bar at the top of the procedure window.
To find the definition of a procedure when you have selected an invocation of it	Control-click the selected invocation and choose “Go to <procedure name>” from the resulting contextual menu.	Right-click the selected invocation and choose “Go to <procedure name>” from the resulting contextual menu.
To find in the procedure window	Press Command-F.	Press Ctrl+F.
To find the same text again	Press Command-G.	Press Ctrl+G.
To find again but in the reverse direction	Press Command-Shift-G.	Press Ctrl+Shift+G.
To find the selected text	Press Command-E and Command-G.	Press Ctrl+H.

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To find the selected text but in the reverse direction	Press Command-E and Command-Shift-G. This shortcut can be changed through the Miscellaneous Settings dialog.	Press Ctrl+Shift+H.
To find a user-defined menu's procedure	Open any procedure window and press Option while selecting a user-defined menu item.	Open any procedure window and press Alt while selecting the user-defined menu item.
To select a word	Double-click.	Double-click.
To select an entire line	Triple-click.	Triple-click.
To save all modified standalone notebook files	When a notebook window is active, choose File→Save All Standalone Notebook Files.	When a notebook window is active, choose File→Save All Standalone Notebook Files.

Chapter

III-14

Controls and Control Panels

Overview

We use the term *controls* for a number of user-programmable objects that can be employed by Igor programmers to create a graphical user interface for Igor users. We call them *controls* even though some of the objects only display values. The term *widgets* is sometimes used by other application programs.

Here is a summary of the types of controls available.

Control Type	**Control Description**
Button	Calls a procedure that the programmer has written.
Chart	Emulates a mechanical chart recorder. Charts can be used to monitor data acquisition processes or to examine a long data record. Programming a chart is quite involved.
CheckBox	Sets an off/on value for use by the programmer's procedures.
CustomControl	Custom control type. Completely specified and modified by the programmer.
GroupBox	An organizational element. Groups controls with a box or line.
ListBox	Lists items for viewing or selecting.
PopupMenu	Used by the user to choose a value for use by the programmer's procedures.
SetVariable	Sets and displays a numeric or string global variable. The user can set the variable by clicking or typing. For numeric variables, the control can include up/down buttons for incrementing/decrementing the value stored in the variable.
Slider	Duplicates the behavior of a mechanical slider. Selects either discrete or continuous values.
TabControl	Selects between groups of controls in complex panels.
TitleBox	An organizational element. Provides explanatory text or message.
ValDisplay	Presents a readout of a numeric expression which usually references a global variable. The readout can be in the form of numeric text or a thermometer bar or both.

The programmer can specify a procedure to be called when the user clicks on or types into a control. This is called the control's *action procedure*. For example, the action procedure for a button may interrogate values in PopupMenu, Checkbox, and SetVariable controls and then perform some action.

Control panels are simple windows that contain these controls. These windows have no other purpose. You can also place controls in graph windows and in panel panes embedded into graphs. Controls are not available in any other window type such as tables, notebooks, or layouts. When used in graphs, controls are not considered part of the *presentation* and thus are **not** included when a graph is printed or exported.

Nonprogrammers will want to skim only the Modes of Operation and Using Controls sections, and skip the remainder of the chapter. Igor programmers should study the entire chapter.

Modes of Operation

With respect to controls, there are two modes of operation: one mode to use the control and another to modify it. To see this, choose Show Tools from the Graph or Panel menu. Two icons will appear in the top-left corner window. When the top icon is selected, you are able to use the controls. When the next icon is selected, the draw tool palette appears below the second icon. To modify the control, select the arrow tool from the draw tool palette.

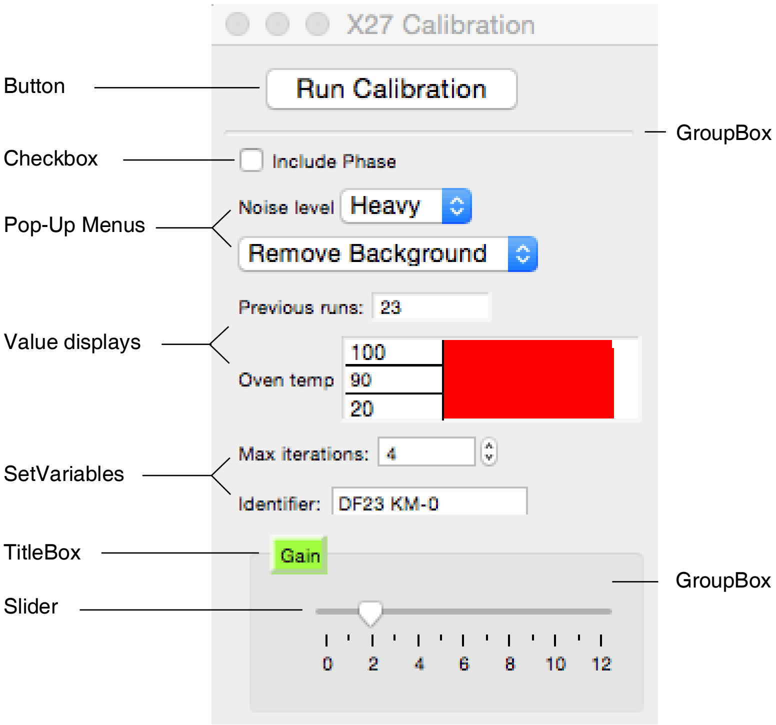
When the top icon is selected or when the icons are hidden, you are in the *use* or *operate* mode. You can momentarily switch to the *modify* or *draw* mode by pressing Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*). Use this to drag or resize a control as well as to double-click it. Double-clicking with the Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*) pressed brings up a dialog that you use to modify the control.

You can also switch to modify mode by choosing an item from the Select Control submenu of the Graph or Panel menu.

Important:To enable the Add Controls submenu in the Graph and Panel menus, you must be in modify mode; either by clicking the second icon or by pressing Command-Option (*Macintosh*) or the Ctrl+Alt (*Windows*) while choosing the Add Controls submenu.

Using Controls

The following panel window illustrates most of the control types.



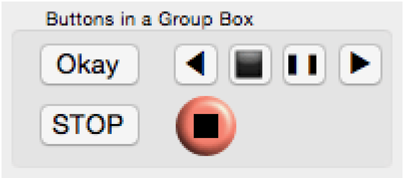
Buttons

When you click a button, it runs whatever procedure the programmer may have specified.

If nothing happens when you click a button, then there is no procedure assigned to the button. If the procedure window(s) haven't been compiled, clicking a button that has an assigned procedure will produce an error dialog.

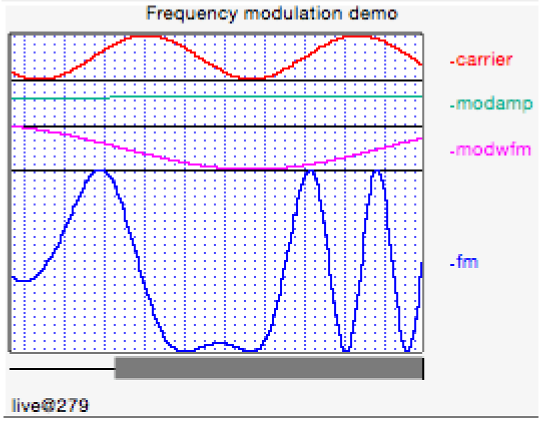
You should choose Compile from the Macros menu to correct this situation. If no error occurs then the button will now be functional.

Buttons usually have a rounded appearance, but a programmer can assign a custom picture so that the button can have nearly any appearance.



Charts

Chart controls can be used to emulate a mechanical chart recorder that writes on paper with moving pens as the paper scrolls by under the pens. Charts can be used to monitor data acquisition processes or to examine a long data record.

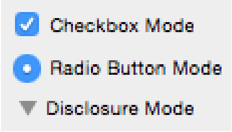


For further discussion of using chart controls, see **Using Chart Recorder Controls** on page IV-317.

Although programming a chart is quite involved, using a chart is actually very easy. See **FIFOs and Charts** on page IV-313 for details.

Checkboxes

Clicking a checkbox changes its selected state and may run a procedure if the programmer specified one. A checkbox may be connected to a global variable. Checkboxes can be configured to look and behave like radio buttons.

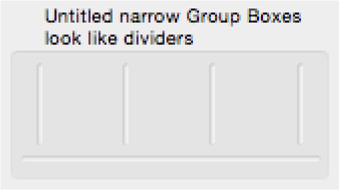


CustomControl

CustomControls are used to create completely new types of controls that are custom-made by the programmer. You can define and control the appearance and all aspects of a custom control's behavior. See **Creating Custom Controls** on page III-424 for examples.

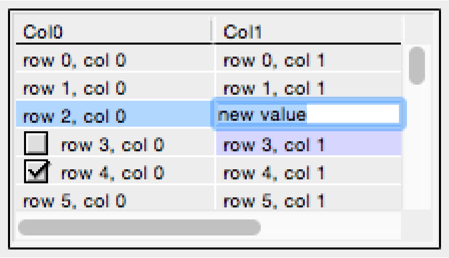
GroupBox

GroupBox controls are organizational or decorative elements. They are used to graphically group sets of controls. They may either draw a box or a separator line and can have optional titles.



ListBox

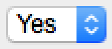
ListBox controls can present a single or multiple column list of items for viewing or selection. ListBoxes can be configured for a variety of selection modes. Items in the list can be made editable and can be configured as checkboxes.

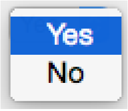


Col0	Col1
row 0, col 0	row 0, col 1
row 1, col 0	row 1, col 1
row 2, col 0	new value
□ row 3, col 0	row 3, col 1
☑ row 4, col 0	row 4, col 1
row 5, col 0	row 5, col 1

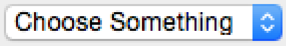
Pop-Up Menus

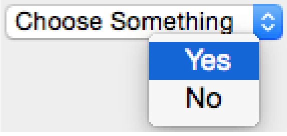
These controls come in two forms: one where the current item is shown in the pop-up menu box:





and another where there is no current item and a title is shown in the box:





The first form is usually used to choose one of many items while the second is used to run one of many commands.

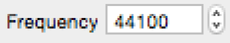
Pop-up menus can also be configured to act like Igor’s color, line style, pattern, or marker pop-up menus. These always show the current item.

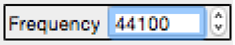
SetVariable

SetVariable controls also can take on a number of forms and can display numeric values. Unlike Value Display controls that display the value of an expression, SetVariable controls are connected to individual global variables and can be used to set or change those variables in addition to reading out their current value. SetVariable controls can also be used with global string variables to display or set short one line strings. SetVariable controls are automatically updated whenever their associated variables are changed.

When connected to a numeric variable, these controls can optionally have up or down arrows that increment or decrement the current value of the variable by an amount specified by the programmer. Also, the programmer can set upper and lower limits for the numeric readouts.

New values for both numeric and string variables can be entered by directly typing into the control. If you click the control once you will see a thick border form around the current value.


Frequency 44100


Frequency 44100

You can then edit the readout text using the standard techniques including Cut, Copy, and Paste. If you want to discard changes you have made, press Escape. To accept changes, press Return, Enter, or Tab or click anywhere outside of the control. Tab enters the current value and also takes you to the next control if any. Shift-Tab is similar but takes you to the previous control if any.

If the control is connected to a numeric variable and the text you have entered can not be converted to a number then a beep will be emitted when you try to enter the value and no change will be made to the value of the variable. If the value you are trying to enter exceeds the limits set by the programmer then your value will be replaced by the nearest limit.

When a numeric control is selected for editing, the Up and Down Arrow keys on the keyboard act like the up and down buttons on the control.

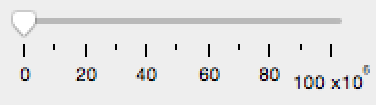
Changing a value in a SetVariable control may run a procedure if the programmer has specified one.

SetVariable Controls and Data Folders

SetVariable controls remember the data folder in which the variable exists, and continue to function properly when the current data folder is different than the controlled variable. See **SetVariable** on page III-417.

The system variables (K0 through K19) belong to no particular data folder (they are available from any data folder), and there is only *one* copy of these variables. If you create a SetVariable controlling K0 while the current data folder is “aFolder”, and another SetVariable controlling K0 while the current data folder is “bFolder”, *they are actually controlling the same K0.*

Sliders



Slider controls can be used to graphically select either discrete or continuous values. When used to select discrete values, a slider is similar to a pop-up menu or a set of radio buttons. Sliders can be live, updating a variable or running a procedure as the user drags the slider, or they can be configured to wait until the user finishes before performing any action.

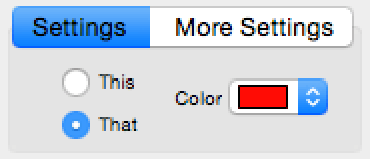
Repeating Sliders

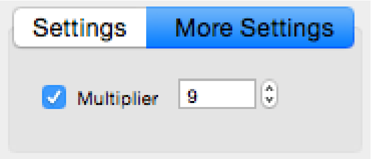
Sliders can be configured to call your action procedure repeatedly while the user is clicking on the thumb. They can be configured to operate at a constant rate or at a rate proportional to the value, and optionally to spring back to a resting value when released. This feature was added in Igor Pro 8.00.

To implement a repeating slider, use the repeat keyword with the **Slider** operation. For a demonstration, see the Slider Repeat Demo demo experiment.

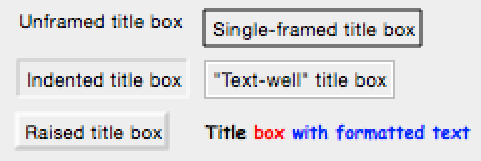
TabControl

TabControls are used to create complex panels containing many more controls than would otherwise fit. When the user clicks on a tab, the programmers procedure runs and hides the previous set of controls while showing the new set.





TitleBox

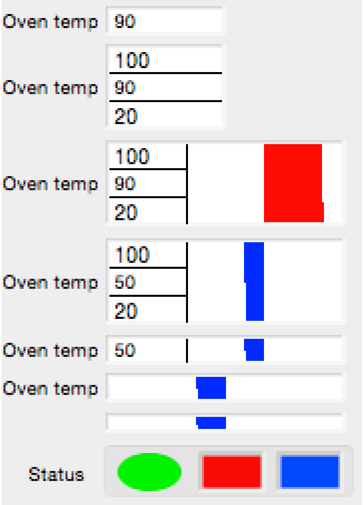


TitleBox controls are mainly decorative elements. They are used provide explanatory text in a control panel. They may also be used to display textual results. The text can be unchanging, or can be the contents of a global string variable. In either case, the user can't inadvertently change the text.

ValDisplays

ValDisplay controls display numeric or string values in a variety of forms ranging from a simple numeric readout to a thermometer bar. Regardless of the form, ValDisplays are just readouts. There is no interaction with the user. They display the current value of whatever expression the programmer specified. Often this will be just the value of a numeric variable, but it can be any numeric expression including calls to user-defined functions and external functions.

Here is a sampling of the forms that ValDisplay controls can assume.



When a thermometer bar is shown, the left edge of the thermometer region represents a low limit set by the programmer while the right edge represents a high limit. The low and high limits appear in some of the above examples. The bar is drawn from a nominal value set by the programmer and will be red if the current value exceeds the nominal value and will be blue if it is less than the nominal value. In the above examples the nominal value is 60. There is no numeric indication of the nominal value. If the nominal value is less than the low limit then the bar will grow from the left to the right. If the nominal value is greater than the high limit then the bar will grow from the right to the left.

If you carefully observe a thermometer bar that is connected to an expression whose value is slowly changing with time you will see that the bar is drawn in a zig-zag fashion. This provides a much finer resolution than if the bar were to be extended or contracted by an entire column of screen pixels at once.

Creating Controls

The ease of creating the various controls varies widely. Anyone capable of writing a simple procedure can create buttons and checkboxes, but creating charts and custom controls requires more expertise. Most controls can be created and modified using dialogs that you invoke via the Add Controls submenu in the Graph or Panel menu.

The Add Controls and Select Control menus are enabled only when the arrow tool in the tool palette is selected. To do this, choose Show Tools from the Graph or Panel menu and then click the second icon from the top in the graph or panel tool palette.

You can temporarily use the arrow tool without the tool palette showing by pressing Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*). While you press these keys, the normally-disabled Add Controls and Select Control submenus are enabled.

When you click a control with the arrow tool, small handles are drawn that allow you to resize the control. Note that some controls can not be resized in this way and some can only be resized in one dimension. You will know this when you try to resize a control and it doesn't budge. You can also use the arrow tool to reposition a control. You can select a control by name with the Select Control submenu in the Graph or Panel menu.

With the arrow tool, you can double-click most controls to get a dialog that modifies or duplicates the control. Charts and CustomControls do not have dialog support.

When you right-click (*Windows*) or Control-click (*Macintosh*) a control, you get a contextual menu that varies depending on the type of control.

You can select multiple controls, mix selections with drawing objects, and perform operations such as move, cut, copy, paste, delete, and align. These operations are undoable. You can't group buttons or use Send to Back as you can with drawing objects.

In panels, when you do a Select All, the selectin includes all controls and drawing objects, but in the case of graphs, only drawing objects are selected. This is because drawing objects in graphs are used for presentation graphics whereas in panels they are used to construct the user interface.

If you want to copy controls from one window to another, simply use the Edit menu to copy and paste. You can also duplicate controls using copy and paste.

When you copy controls to the clipboard, the command and control names are also copied as text. This is handy for programming.

Press Option (*Macintosh*) or Alt (*Windows*) while choosing Edit-Copy to copy the complete commands for creating the copied controls.

If you copy a control from the extreme right side or bottom of a window, it may not be visible when you paste it into a smaller window. Use the smaller window's Retrieve submenu in the Mover tool palette icon to make it visible.

General Command Syntax

All of the control commands use the following general syntax:

```
ControlOperation Name [,keyword[=value] [,keyword[=value]]…]
```

`Name` is the control's name. It must be unique among controls in the window containing the control. If `Name` is not already in use then a new control is created. If a control with the same name already exists then that control is modified, so that multiple commands using the same name result in only one control. This is useful for creating controls that require many keywords.

All keywords are optional. Not all controls accept all keywords, and some controls accept a keyword but do not actually use the value. The value for a keyword with one control can have a different form from the value for the same keyword used with a different type of control. See the specific control operation documentation in Chapter V-1, **Igor Reference** for details.

Some controls utilize a format keyword to set a format string. The format string can be any printf-style format that expects a single numeric value. Think of the output as being the result of the following command:

```
Printf formatString, value_being_displayed
```

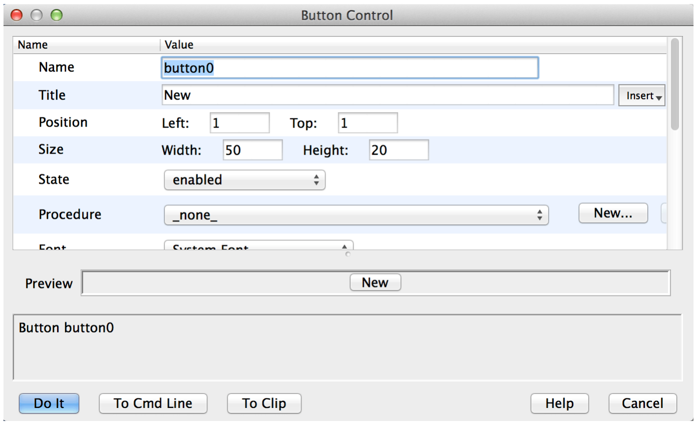
See the **printf** operation on page V-770 for a discussion of printf format strings. The maximum length of the format string is 63 bytes. The format is used only for controls that display numeric values.

All of the clickable controls can optionally call a user-defined function when the user releases the mouse button. We use the term *action procedure* for such a function. Each control passes one or more parameters to the action procedure. The dialogs for each control can create a blank user function with the correct parameters.

Creating Button Controls

The **Button** operation (page V-55) creates or modifies a rounded-edge or custom button with the title text centered in the button. The default font depends on the operating system, but you can change the font, font size, text color and use annotation-like escape codes (see **Annotation Escape Codes** on page III-53). The amount of text does not change the button size, which you can set to what you want.

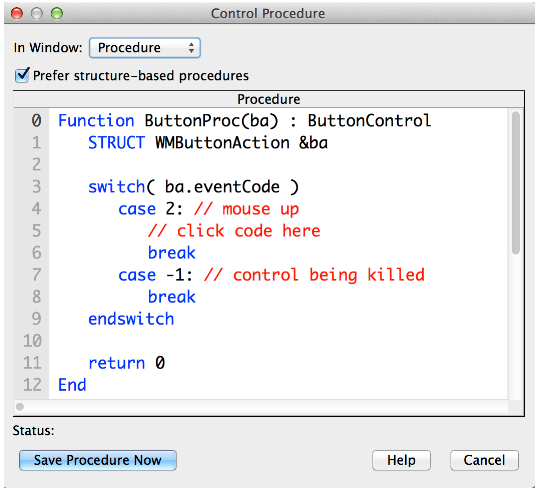
Here we create a simple button that will just emit a beep when pressed. Start by choosing the Add Button menu item in the Graph or Panel menu to invoke the Button Control dialog:



Name	Value
Name	button0
Title	New
Position	Left: 1 Top: 1
Size	Width: 50 Height: 20
State	enabled
Procedure	_none_
Font	System Font

Button button0

Clicking the procedure's New button brings up a dialog containing a procedure template that you can edit, rename, and save. Here we started with the standard ButtonControl template, replaced the default name with `MyBeepProc`, and added the `Beep` command:



In Window: Procedure

☑ Prefer structure-based procedures

```
Function ButtonProc(ba) : ButtonControl
	STRUCT WMButtonAction &ba

	switch( ba.eventCode )
		case 2: // mouse up
			// click code here
			break
		case -1: // control being killed
			break
	endswitch

	return 0
End
```

The controls can work with procedures using two formats: the old procedure format used in Igor 3 and 4, and the "structure-based" format introduced in Igor 5.

Selecting the "Prefer structure-based procedures" checkbox creates new procedure templates using the structure-based format.

The "Prefer structure-based procedures" checkbox is checked by default because structure-based procedures are recommended for all new code. If you uncheck this checkbox before editing the template, Igor switches the template to the old procedure format. Use of the old format is discouraged.

Click Help to get help about the Button operation. In the Details section of the help, you can find information about the WMButtonAction structure.

The fact that you can create the action procedure for a control in a dialog may lead you to believe that the procedure is stored with the button. This is not true. The procedure is actually stored in a procedure window. This way you can use the same action procedure for several controls. The parameters which are passed to a given procedure can be used to differentiate the individual controls.

For more information on using action procedures, see **Control Structures** on page III-437, the **Button** operation (page V-55), and **Using Structures with Windows and Controls** on page IV-103.

Button Example

Here is how to make a button whose title alternates between Start and Stop.

Enter the following in the procedure window:

```
Function MyStartProc()
   Print "Start"
End

Function MyStopProc()
   Print "Stop"
End

Function StartStopButton(ba) : ButtonControl
   STRUCT WMButtonAction &ba

   switch(ba.eventCode)
      case 2:                       // Mouse up
         if (CmpStr(ba.ctrlName,"bStart") == 0)
            Button $ba.ctrlName,title="Stop",rename=bStop
            MyStartProc()
         else
            Button $ba.ctrlName,title="Start",rename=bStart
            MyStopProc()
         endif
         break
   endswitch

   return 0
End
```

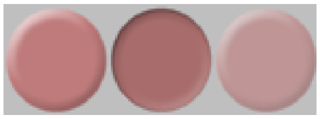
Now execute:

```
NewPanel
Button bStart,size={50,20},proc=StartStopButton,title="Start"
```

Custom Button Control Example

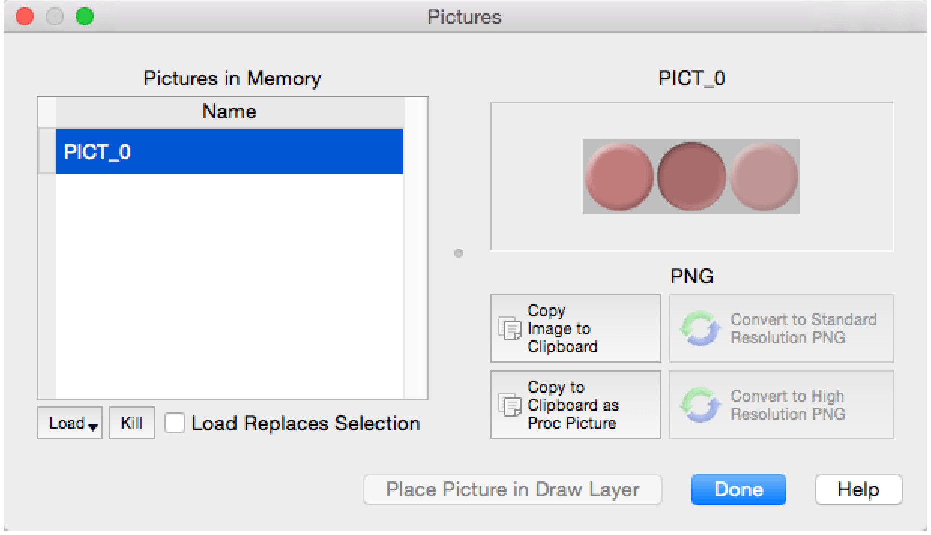
You can create custom buttons by following these steps:

1. Using a graphics-editing program, create a picture that shows the button in its normal (“relaxed”) state, then in the pressed-in state, and then in the disabled state. Each portion of the picture should be the same size:

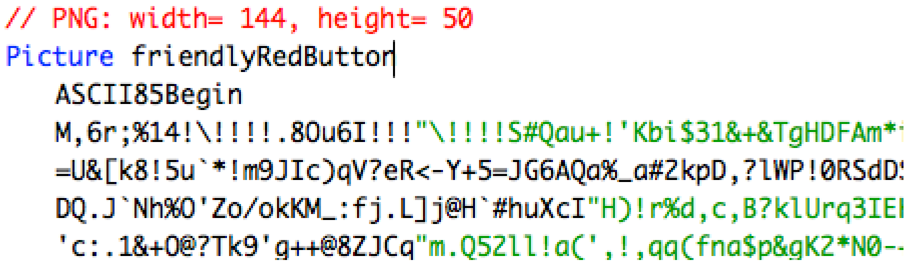


If the button blends into the background it will look better if the buttons are created on the background you will use in the panel. Igor looks at the pixels in the upper left corner, and if they are a light neutral color, Igor omits those pixels when the button is drawn.

2. Copy the picture to the clipboard.
3. Switch to Igor,choose Misc→Pictures, click Load and then From Clipboard.

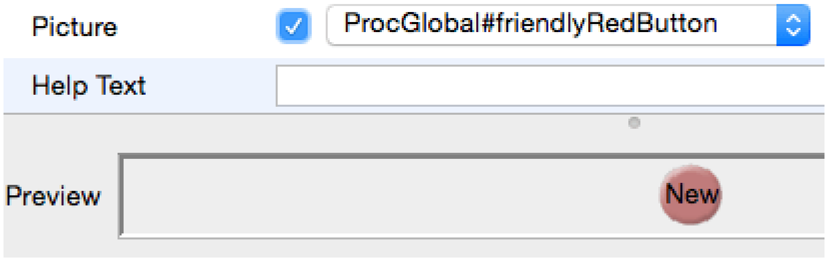

Pictures
Pictures in Memory
Name
PICT_0
Load
Kill
□ Load Replaces Selection
PICT_0
PNG
Copy Image to Clipboard
Convert to Standard Resolution PNG
Copy to Clipboard as Proc Picture
Convert to High Resolution PNG
Place Picture in Draw Layer
Done
Help

4. Click Copy to Clipboard as Proc Picture to create Proc Picture text on the Clipboard.
5. Click Done.
6. Open a procedure window, paste the text, and give a suitable name to the picture:



```
// PNG: width= 144, height= 50
Picture friendlyRedButton
    ASCII85Begin
    M,6r;%14!\!!!!.8Ou6I!!!"\!!!!S#Qau+!'Kbi$31&+&TgHDFAm*
    =U&[k8!5u`*!m9JIc)qV?eR<-Y+5=JG6AQa%_a#2kpD,?lWP!0RSdD
    DQ.J`Nh%O'Zo/okKM_:fj.L]j@H`#huXcI"H)!r%d,c,B?klUrq3IE
    'c:.1&+O@?Tk9'g++@8ZJCq"m.Q52ll!a(',!,qq(fna$p&gK2*N0-
```

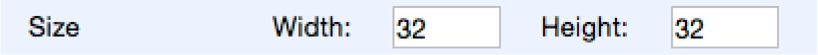
7. Activate an existing control panel or create a new one.
8. If the tool palette is not showing, choose Panel→Show Tools.
9. Choose Panel-Add Control→Add Button to display the Button Control dialog.
10. Locate the Picture setting in the dialog, check the checkbox, and select the proc picture from the pop-up menu:



Picture ☑ ProcGlobal#friendlyRedButton

Help Text

11. Locate the Size setting in the dialog and set the appropriate size for the button:



Size Width: 32 Height: 32

Creating Chart Controls

The **Chart** operation creates or modifies a chart control. There is no dialog support for chart controls. You need at least intermediate-level Igor programming skills to create a functional chart control.

For further information, see **FIFOs and Charts** on page IV-313.

Creating Checkbox Controls

The **CheckBox** creates or modifies a checkbox or a radio button.

CheckBox controls automatically size themselves in both height and width. They can optionally be connected to a global variable.

For an example of using checkbox controls as radio buttons, see the reference documentation for the **CheckBox** operation.

The user-defined action procedure that you will need to write for CheckBoxes must have the following form:

```
Function CheckProc(cba) : CheckBoxControl
   STRUCT WMCheckboxAction &cba

   switch(cba.eventCode)
      case 2:             // Mouse up
         Variable checked = cba.checked
         break
      case -1:            // Control being killed
         break
   endswitch

   return 0
End
```

The checked structure member is set to the new checkbox value:; 0 or 1.

You often do not need an action procedure for a checkbox because you can read the state of the checkbox with the **ControlInfo** operation.

You can create custom checkboxes by following steps similar to those for custom buttons (see **Custom Button Control Example** on page III-422), except that the picture has six states side-by-side instead of three. The checkbox states are:

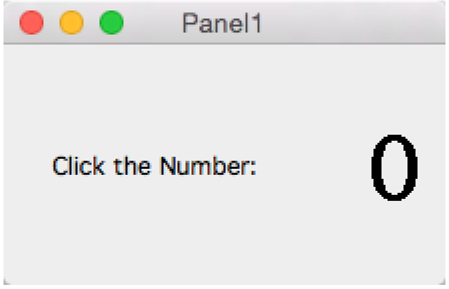
Image Order	Control State
Left	Deselected enabled.
	Deselected enabled and clicked down (about to be selected).
	Deselected disabled.
	Selected enabled.
	Selected enabled and clicked down (about to be deselected).
Right	Selected disabled.

Creating Custom Controls

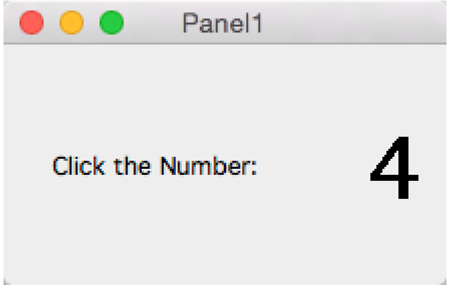
The CustomControl operation creates or modifies a custom control, all aspects of which are completely defined by the programmer. See the **CustomControl** operation on page V-134 for a complete description.

The examples in this section are also available in the Custom Controls demo experiment. Choose File→Example Experiments→Feature Demos 2→Custom Control Demo.

What you can create with a CustomControl can be fairly simple such as this counter that increments when you click on it.



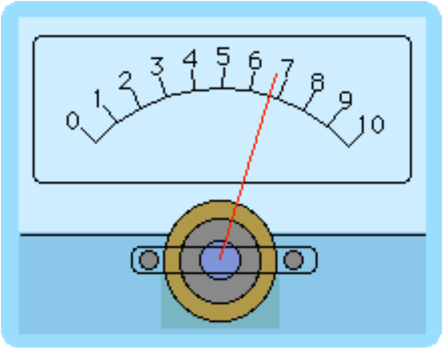
Four clicks later:



The following code implements the counter custom control using the `kCCE_frame` event. In the panel, click on the number to increment the counter; also try clicking and then dragging outside the control.

```
static constant kCCE_mouseup= 2
static constant kCCE_frame= 12

// PNG: width= 280, height= 49
Picture Numbers0to9
	ASCII85Begin
	M,6r;%14!\!!!!.8Ou6I!!!$:!!!!R#Qau+!00#^OT5@]&TgHDFAm*iFE_/6AH5;7DfQssEc39jTBQ
	=U"5QO:5u`*!m@2jnj"La,mA^'a?hQ[Z.[.,Kgd(1o5*(PSO8oS[GX%3u'11dTl)fII/"f-?Jq*no#
	Qb>Y+UBKXKHQpQ&qYW88I,Ctm(`:C^]$4<ePf>Y(L\U!R2N7CEAn![N1I+[hTtr.VepqSG4R-;/+$3
	IJE.V(>s0B@E@"n"ET+@5J9n_E:qeR_8:Fl?m1=DM;mu.AEj!)]K4CUuCa4T=W)#(SE>uH[A4\;IG/
	e]FqJ4u,2`*p=N5sc@qLD5bH89>gIBdF-1i6SF28oH@"3c2m)bDr&,UB$]i]/0bA.=qbR2#\-D9E?O
	2>3D>`($p(Kn)F8aF@)LYiXn[h2K):5@^kF?94)j*1Xtq1U2oFZmY.te?0G)EQ%5,RVT-c)DVa+%mP
	%+bS*_hN$hC*8uCJuIWqTHJR.U?32`_B)(g_8e#*YXa>=faEdJsF]6iJlrQ@QAX7huJUmXj8:PBTb2
	Y:DYf*Sci'Q"3_;@RDQA:A/([2sO8r$hW)\B$XBGASJ:6OpC+GL<FjVfeNm20U<l<9J%cndX3'HP+k
	R.IV?U>ns*_;Zt[]6G6"Rb-*'Nm-E8]LXXXo7Ub>A**7Bm5cS*">HbQ&_RhmUe]$iu@T?Cci:e-_`k
	sE+H.GRSMT(9to;IZuH`T4%Yt<jF$+W?Yh6Q*_`C4sGig=L@DKoT%.H=#e_H"QEeeBVNTWBSMYr3dj
	O=T%d&4kT9#cWPHS>kAG;3=or2(IK*IBF$^qK,+m0NSDK_!+e0#3fAI>HfKa<sk0641u\W@r+Y:$.i
	i$grCPR#&6,;+>nTs_IKS6XcYR)A$fJiC6Z_d2S!$R>_ZH+[<p:JI0ub]\BhE(0RP@((KTRTGo;#SY
	LT^9;D7X#km%UV20?$RS"FZoIF!(`FY-iL?n$%#o;-Wj(\PaBS6ZRQe@:kC>%ULrhTWLNM=n@fUbRp
	SKkLe\kJ)Sd]u7!?pRJk-!XL[/MZX'"n4?a?JIKO0k'KUm1IZ+roB=:Bq'$&E<#$Krp%p,E"4sI>[-
	0F#^ff5SN':2fO)LNC?L4(2ga=!aLm8)tVbGAM?L`l^=$D_YP7Z(sOFs)BL5er5G95p3?m%hM^lSr'
	*E^O@8=u6hL`L$mPcq!Bl-iHuGA6hiip%`cFjl9>W?'E-&5T%Y.]i2A@1i%p8XJ5[khb:&"JXYSC\r
	10Ss8<Ye;S^"Nc0%-DFouAiPQ9OemnR!"sHH$JKt@!"d0E"'M(P%:`p'15_10`!<nVt"TALQ>PF8WL
	Z:#f!!!!j78?7R6=>B
	ASCII85End
End

Structure CC_CounterInfo
	Int32 theCount		// current frame of 10 frame sequence of numbers in
EndStructure

Function MyCC_CounterFunc(s)
	STRUCT WMCustomControlAction &s

	STRUCT CC_CounterInfo info

	if( s.eventCode==kCCE_frame )
		StructGet/S info,s.userdata
		s.curFrame= mod(info.theCount+(s.curFrame!=0),10)
	elseif( s.eventCode==kCCE_mouseup )
		StructGet/S info,s.userdata
		info.theCount= mod(info.theCount+1,10)
		StructPut/S info,s.userdata	// will be written out to control
	endif

	return 0
End

Window Panel0() : Panel
	PauseUpdate; Silent 1		// building window...
	NewPanel /W=(69,93,271,252)
	CustomControl cc2,pos={82,46},proc=MyCC_CounterFunc,picture=
{ProcGlobal#Numbers0to9,10}
EndMacro
```

You can create even more sophisticated controls, such as this voltage meter control.



Choose File→Example Experiments→Feature Demos 2→Custom Control Demo to try this control and see the code that implements it.

Creating GroupBox Controls

The GroupBox operation creates or modifies a listbox control. See the **GroupBox** operation on page V-334 for a complete description and examples.

Creating ListBox Controls

The **ListBox** operation creates or modifies a listbox control.

We will illustrate listbox creation by example.

1. Create a panel with a listbox control:

   ```
   NewPanel
   ListBox list0 size={200,60}, mode=1
   ```

   The simplest functional listbox needs at least one text wave to contain the list items. Without the text wave, a listbox control has no list items. In this state, the listbox is drawn with a red X over the control.
2. We need a text wave to contain the list items:

   ```
   Make/O/T textWave0 = {"first list item", "second list item", "etc..."}
   ```

3. Choose Panel→Show Tools.

   This puts the panel in edit mode so you can modify controls.
4. Double-click the listbox control to invoke the ListBox Control dialog.
5. For the List Text Wave property, select the wave you created to assign it as the list’s text wave.
6. Click Do It.

   You now have a functional listbox control.
7. Click the operate (top) icon, or choose Panel→Hide Tools, so you can use, rather than edit, the list.

In this example, we created a single-selection list. You can query the selection by calling **ControlInfo** and checking the V_Value output variable.

See the **ListBox** for a complete description and further examples.

Right-clicking (*Windows*) or Control-clicking (*Macintosh*) a listbox shows a contextual menu with commands for editing the list waves and action procedure, and for creating a numeric selection wave, if the control is a multi-selection listbox.

Creating PopupMenu Controls

The **PopupMenu** creates or modifies a pop-up menu control. Pop-up menus are usually used to provide a choice of text items but can also present colors, line styles, patterns, and markers.

The control automatically sizes itself as a function of the title or the currently selected menu item. You can specify the bodyWidth keyword to force the body (non-title portion) of the pop-up menu to be a fixed size. You might do this to get a set of pop-up menus of nicely aligned with equal width. The bodywidth keyword also affects the non-text pop-up menus.

The font and fsize keywords affect only the title of a pop-up menu. The pop-up menu itself uses standard system fonts.

Unlike color, line style, pattern, or marker pop-up menus, text pop-up menu controls can operate in two distinct modes as set by the mode keyword's value.

If the argument to the mode keyword is nonzero then it is considered to be the number of the menu item to be the initial current item *and* displays the current item in the pop-up menu box. This is the *selector mode*. There is often no need for an action procedure since the value of the current item can be read at any time using the **ControlInfo** operation (page V-89).

If mode is zero then the title appears inside the pop-up menu box, hence the name *title-in-box mode*. This mode is generally used to select a command for the action procedure to execute. The current item has no meaning except when the pop-up menu is activated and the selected item is passed to the action procedure.

The menu that pops up when the control is clicked is determined by a string expression that you pass as the argument to the value keyword. For example:

```
PopupMenu name value="Item 1;Item 2;Item 3;"
```

To create the color, line style, pattern or marker pop-up menus, set the string expression to one of these fixed values:

```
"*COLORPOP*"
"*LINESTYLEPOP*"
"*MARKERPOP*"
"*PATTERNPOP*"
```

For text pop-up menus, the string expression must evaluate to a list of items separated by semicolons. This can be a fixed literal string or a dynamically-calculated string. For example:

```
PopupMenu name value="Item 1;Item 2;Item 3;"
PopupMenu name value="_none_;" + WaveList("*",";","")
```

It is possible to apply certain special effects to the menu items, such as disabling an item or marking an item with a check. See **Special Characters in Menu Item Strings** on page IV-133 for details.

The literal text of the string expression is stored with the control rather than the results of the evaluation of the expression. Igor evaluates the expression when the PopupMenu value=<value> command runs and reevaluates it every time the user clicks on the pop-up menu box. This reevaluation ensures that dynamic menus, such as created by the WaveList example above, reflect the conditions at click time rather than the conditions that were in effect when the PopupMenu control was created.

When the user clicks and Igor reevaluates the string expression, the procedure that created the pop-up menu is no longer running. Consequently, its local variables no longer exist, so the string expression can not reference them. To incorporate the value of local variables in the value expression use the **Execute** operation:

```
String str = <code that generates item list>    // str is a local variable
Execute "PopupMenu name value=" + str
```

Igor evaluates the string expression as if it were typed on the command line. You can not know what the current data folder will be when the user clicks the pop-up menu. Consequently, if you want to refer to objects in specific data folders, you must use full paths. For example:

```
PopupMenu name value=#"func(root:DF234:wave0, root:gVar)"
```

Because of click-time reevaluation, the pop-up menu does not automatically update if the value of the string expression changes. Normally this is not a problem, but you can use the **ControlUpdate** operation (page V-94) to force the pop-up menu to update. Here is an example:

```
NewPanel/N=PanelX
String/G gPopupList="First;Second;Third"
PopupMenu oneOfThree value=gPopupList  // pop-up shows “First”
gPopupList="1;2;3"                     // pop-up is unchanged
ControlUpdate/W=PanelX oneOfThree      // pop-up shows “1”
```

If the string expression can not be evaluated at the time the command is compiled, you can defer the evaluation of the expression by enclosing the value this way:

In some cases, the string expression can not be compiled at the time the PopupMenu command is compiled because it references a global object that does not yet exist. In this case, you can prevent a compile-time error by using this special syntax:

```
PopupMenu name value= #"pathToNonExistentGlobalString"
```

If a deferred expression has quotes in it, they need to be escaped with backslashes:

```
PopupMenu name value= #"\"_none_;\"+UserFunc(\"foo\")"
```

The optional user defined action procedure is called after the user makes a selection from the popup menu. Popup menu procedures have the following form:

```
Function PopMenuProc(pa) : PopupMenuControl
   STRUCT WMPopupAction pa

   switch(pa.eventCode)
      case 2:        // Mouse up
         Variable popNum = pa.popNum      // 1-based item number
         String popStr = pa.popStr        // Text of selected item
         break
      case -1:       // Control being killed
         break
   endswitch
End
```

`pa.popNum` is the item number, *starting from one*, and pa.`popStr` is the text of the selected item.

For the color pop-up menus the easiest way to determine the selected color is to use the **ControlInfo**.

Creating SetVariable Controls

The **SetVariable** operation (page V-854) creates or modifies a SetVariable control. SetVariable controls are useful for both viewing and setting values.

SetVariable controls are tied to numeric or string global variables, to a single element of a wave, or to an internal value stored in the control itself. To minimize clutter, you should use internal values in most cases.

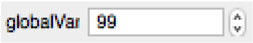
When used with numeric variables, Igor draws up and down arrows that the user can use to increment or decrement the value.

You can set the width of the control but the height is determined from the font and font size. The width of the readout area is the width of the control less the width of the title and up/down arrows. However, you can use the bodyWidth keyword to specify a fixed width for the body (nontitle) portion of the control.

For example, executing the commands:

```
Variable/G globalVar=99
SetVariable setvar0 size={120,20},frame=1,font="Helvetica", value=globalVar
```

creates the following SetVariable control:



To associate a SetVariable control with a variable that is not in the current data folder at the time SetVariable runs, you must use a data folder path:

```
Variable/G root:Packages:ImagePack:globalVar=99
SetVariable setvar0 value=root:Packages:ImagePack:globalVar
```

Unlike PopupMenu controls, SetVariable controls remember the current data folder when the SetVariable command executes. Thus an equivalent set of commands is:

```
SetDataFolder root:Packages:ImagePack
Variable/G globalVar=99
SetVariable setvar0 value=globalVar
```

Also see **SetVariable Controls and Data Folders** on page III-417.

You can control the style of the numeric readout via the format keyword. For example, the string "%.2d" will display the value with 2 digits past the decimal point. You should not use the format string to include text in the readout because Igor has to read back the numeric value. You may be able to add suffixes to the readout but prefixes will not work. When used with string variables the format string is not used.

Often it is sufficient to query the value using **ControlInfo** and you there is no need for an action procedure. If you want to do something every time the value is changed, then you need to create an action procedure of the following form:

```
Function SetVarProc(sva) : SetVariableControl
   STRUCT WMSetVariableAction sva

   switch(sva.eventCode)
      case 1:                      // Mouse up
      case 2:                      // Enter key
      case 3:                      // Live update
         Variable dval = sva.dval
         String sval = sva.sval
         break
      break
      case -1:                     // Control being killed
         break
   endswitch

End
```

varName will be the name of the variable being used. If the variable is a string variable then varStr will contain its contents and varNum will be set to the results of an attempt to convert the string to a number. If the variable is numeric then varNum will contain its contents and varStr will be set to the results of a number to string conversion.

If the value is a string, then sva.sval contains the value. If it is numeric, then sva.dval contains the value. sva.isStr is 0 for numeric values and non-zero for string values.

When the user presses and holds in the up or down arrows then the value of the variable will be steadily changed by the increment value but your action procedure will not be called until the user releases the mouse button.

Creating Slider Controls

The **Slider** creates or modifies a slider control.

A slider control is tied to a numeric global variables or to a numeric internal value stored in the control itself. To minimize clutter, you should use internal values in most cases. The value is changed by dragging the “thumb” part of the control.

There are many options for labelling the numeric range such as setting the number of ticks.

You can also provide custom labels in two waves, one numeric and another providing the corresponding text label. For example:

```
NewPanel
Make/O tickNumbers= {0,25,60,100}
Make/O/T tickLabels= {"Off","Slow","Medium","Fast"}
Slider speed,pos={86,28},size={74,73}
Slider speed,limits={0,100,0},value= 40
Slider speed,userTicks={tickNumbers,tickLabels}
```

Often it is sufficient to query the value using **ControlInfo** and you there is no need for an action procedure. If you want to do something every time the value is changed, or to implement a repeating slider, then you need to create an action procedure.

Igor calls the action procedure when the user drags the thumb, when the user clicks the thumb, when the user clicks on either side of the thumb, and when a procedure modifies the slider’s global variable, if any. For a repeating slider, it calls the action procedure periodically while the user clicks the thumb.

See the **Slider** operation on page V-874 and **Repeating Sliders** on page III-418 for further information.

Handling Slider Events

A slider can call your action procedure only when the mouse is released (live mode off) or it can call it each time the slider position changes while the mouse is pressed (live mode on). This section demonstrates how to create both kinds of sliders.

Enter this code in the procedure window of a new experiment. Then execute

```
SliderDemoPanel()
```

in the command line and play with both sliders.

```
Window SliderDemoPanel() : Panel
   PauseUpdate; Silent 1         // building window...
   NewPanel /W=(262,115,665,287)

   TitleBox Title0,pos={46,21},size={139,15},fSize=12,frame=0,fStyle=1
   TitleBox Title0,title="Live Mode Off"
   Slider slider0,pos={197,23},size={150,44},proc=Slider0Proc
   Slider slider0,limits={0,2,0},value=0,live=0,vert=0

   TitleBox Title1,pos={52,114},size={135,15},fSize=12,frame=0,fStyle=1
   TitleBox Title1,title="Live Mode On"
   Slider slider1,pos={197,113},size={150,44},proc=Slider1Proc
   Slider slider1,limits={0,2,0},value=0,live=0,vert=0
EndMacro

Function Slider0Proc(sa) : SliderControl  // Action procedure for slider0
   STRUCT WMSliderAction &sa

   switch(sa.eventCode)
      case -3:       // Control received keyboard focus
      case -2:       // Control lost keyboard focus
      case -1:       // Control being killed
         break
      default:
         if (sa.eventCode & 1) // Value set
            Printf "Value = %g, event code = %d\r", sa.curval, sa.eventCode
         endif
         break
   endswitch

   return 0
```

```
End

Function Slider1Proc(sa) : SliderControl   // Action procedure for slider1
   STRUCT WMSliderAction &sa

   switch(sa.eventCode)
      case -3:       // Control received keyboard focus
      case -2:       // Control lost keyboard focus
      case -1:       // Control being killed
         break
      default:
         if (sa.eventCode & 1)    // Value set
            Printf "Value = %g, event code = %d\r", sa.curval, sa.eventCode
         endif
         if (sa.eventCode & 8)    // Mouse moved or arrow key moved the slider
            Printf "Value = %g, event code = %d\r", sa.curval, sa.eventCode
         endif
         break
   endswitch

   return 0
End
```

Creating TabControl Controls

The **TabControl** creates or modifies a TabControl control. Tabs are used to group controls into visible and hidden groups.

The tabs are numbered. The first tab is tab 0, the second is tab 1, etc.

A default tab control has one tab:

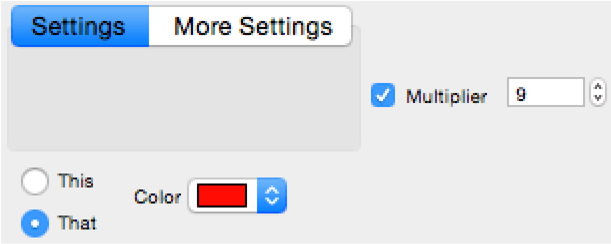
```
NewPanel/W=(150,50,650,400)
TabControl tb, tabLabel(0)="Settings", size={400,250}
```

You add tabs to the control by providing additional tab labels:

```
TabControl tb, tabLabel(1)="More Settings"
```

When you click on a tab, the control’s action procedure receives the number of the clicked-on tab.

The showing and hiding of the controls are accomplished by your action procedure. In this example, the This, That, and Color controls are shown when the Settings tab is clicked, and the Multiplier checkbox is hidden. When the More Settings tab is clicked, the action procedure makes the opposite occur.



The simplest way to create a tabbed user interface is to create an over-sized panel with all the controls visible and outside of the tab control. Place controls in their approximate positions relative to one another. By positioning the controls this way you can more easily modify each control until you are satisfied with them.

Before you put the controls into the tab control, get a list of the non-tab control names:

```
Print ControlNameList("" ,"\r", "!tb")       // all but “tb”
thisCheck
thatCheck
colorPop
multCheck
multVar
```

Determine which controls are to be visible in which tabs:

Tab 0: Settings	Tab 1: More Settings
thisCheck	multCheck
thatCheck	multVar
colorPop	

Write the action procedure for the tab control to show and hide the controls:

```
Function TabProc(tca) : TabControl

   STRUCT WMTabControlAction &tca

   switch (tca.eventCode)
      case 2:                               // Mouse up
         Variable tabNum = tca.tab          // Active tab number
         Variable isTab0 = tabNum==0
         Variable isTab1 = tabNum==1

         ModifyControl thisCheck disable=!isTab0       // Hide if not Tab 0

         ModifyControl thatCheck disable=!isTab0       // Hide if not Tab 0
         ModifyControl colorPop disable=!isTab0        // Hide if not Tab 0

         ModifyControl multCheck disable=!isTab1       // Hide if not Tab 1
         ModifyControl multVar disable=!isTab1         // Hide if not Tab 1
         break
   endswitch

   return 0
End
```

A more elegant method, useful when you have many controls, is to systematically name the controls inside each tab using a prefix or suffix that is unique to that tab, such as tab0_thisCheck, tab0_thatCheck, tab1_multVar. Then use the ModifyControlList operation to show and hide the controls. See the **ModifyControlList** operation for an example.

Assign the action procedure to the tab control:

```
TabControl tb, proc=TabProc
```

Click on the tabs to see whether the showing and hiding is working correctly.

Verify that the action procedure correctly shows and hides controls as you click the tabs. When this works correctly, move the controls into their final positions, inside the tab control.

During this process, the "temporary selection" shortcut comes in handy. While you are in operate mode, pressing Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*) temporarily switchs to select mode, allowing you to select and drag controls.

Save the panel as a recreation macro (Windows→Control→Window Control) to record the final control positions. Rewrite the macro as a function that initially creates the panel:

```
Function CreatePanel()
   KillWindow/Z TabPanel
   NewPanel/N=TabPanel/W=(596,59,874,175) as "Tab Demo Panel"
   TabControl tb,pos={15,19},size={250,80},proc=TabProc
   TabControl tb,tabLabel(0)="Settings"
   TabControl tb,tabLabel(1)="More Settings",value= 0
   CheckBox thisCheck,pos={53,52},size={39,14},title="This"
   CheckBox thisCheck,value= 1,mode=1
   CheckBox thatCheck,pos={53,72},size={39,14},title="That"
   CheckBox thatCheck,value= 0,mode=1
   PopupMenu colorPop,pos={126,60},size={82,20},title="Color"
   PopupMenu colorPop,mode=1,popColor= (65535,0,0)
   PopupMenu colorPop,value= #"\"*COLORPOP*\""
   CheckBox multCheck,pos={50,60},size={16,14},disable=1
   CheckBox multCheck,title="",value= 1
   SetVariable multVar,pos={69,60},size={120,15},disable=1
   SetVariable multVar,title="Multiplier",value=multiplier
End
```

See the **TabControl** operation on page V-1011 for a complete description and examples.

Creating TitleBox Controls

The TitleBox operation creates or modifies a TitleBox control. The control's text can be static or can be tied to a global string variable. See the **TitleBox** operation on page V-1038 for a complete description and examples.

Creating ValDisplay Controls

The **ValDisplay** operation (page V-1060) creates or modifies a value display control.

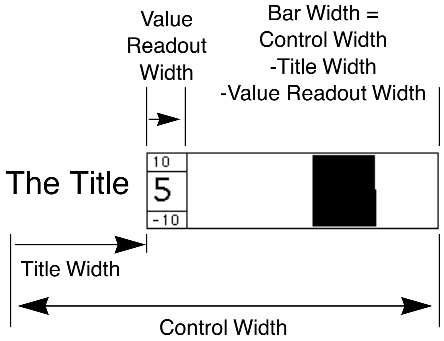
ValDisplay controls are very flexible and multifaceted. They can range from simple numeric readouts to thermometer bars or a hybrid of both. A ValDisplay control is tied to a numeric expression that you provide as an argument to the value keyword. Igor automatically updates the control whenever anything that the numeric expression depends on changes.

ValDisplay controls evaluate their value expression in the context of the root data folder. To reference a data object that is not in the root, you must use a data folder path, such as "root:Folder1:var1".

Here are a few selected keywords extracted from the **ValDisplay** operation on page V-1060:

```
size={width,height}
barmisc={lts, valwidth}
limits={low,high,base}
```

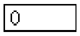
The size and appearance of the ValDisplay control depends primarily on the *valwidth* and size parameters and the width of the title. However, you can use the bodyWidth keyword to specify a fixed width for the body (non-title) portion of the control. Essentially, space for each element is allocated from left to right, with the title receiving first priority. If the control width hasn't all been used by the title, then the value readout width is the smaller of *valwidth* points or what is left. If the control width hasn't been used up, the bar is displayed in the remaining control width:



Here are the various major possible forms of ValDisplay controls. Some of these examples modify previous examples. For instance, the second bar-only example is a modification of the valdisp1 control created by the first bar-only example.

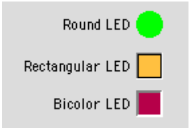
Numeric Readout Only

```
// Default readout width (1000) is >= default control width (50)
ValDisplay valdisp0 value=K0
```



LED Display

```
// Create the three LED types
ValDisplay led1,pos={67,17},size={75,20},title="Round LED"
ValDisplay led1,limits={-50,100,0},barmisc={0,0},mode=1
ValDisplay led1,bodyWidth= 20,value= #"K1",zeroColor=(0,65535,0)

ValDisplay led2,pos={38,48},size={104,20},title="Rectangular LED"
ValDisplay led2,frame=5,limits={0,100,0},barmisc={0,0},mode=2
ValDisplay led2,bodyWidth= 20,value= #"K2"
ValDisplay led2,zeroColor= (65535,49157,16385)

ValDisplay led3,pos={60,76},size={82,20},title="Bicolor LED"
ValDisplay led3,limits={-40,100,-100},barmisc={0,0},mode= 2
ValDisplay led3,bodyWidth= 20,value= #"K3"
```



Bar Only

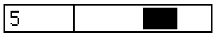
```
// Readout width = 0
ValDisplay valdisp1,frame=1,barmisc={12,0},limits={-10,10,0},value=K0
K0= 5                 // halfway from base of 0 to high limit of 10.
```

The nice thing about a bar-only ValDisplay is that you can make it 5 to 200 points tall whereas with a numeric readout, the height is set by the font sizes of the readout and printed limits.

```
// Set control height= 80
ValDisplay valdisp1, size={50,80}
```

Numeric Readout and Bar

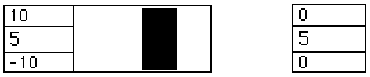
```
// 0 < readout width (50) <  control width (150)
ValDisplay valdisp2 size={150,20},frame=1,limits={-10,10,0}
ValDisplay valdisp2 barmisc={0,50},value=K0  // no limits shown
```



Optional Limits

Whenever the numeric readout is visible, the optional limit values may be displayed too.

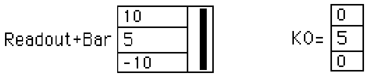
```
// Set limits font size to 10 points. Readout widths unchanged.
ValDisplay valdisp2 barmisc={10,50}
ValDisplay valdisp0 barmisc={10,1000}
```



Optional Title

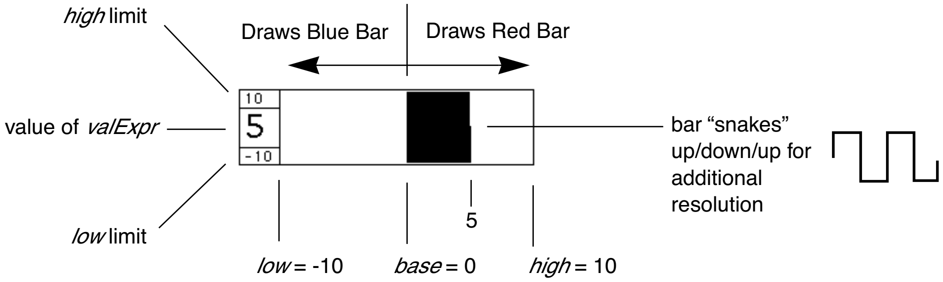
The control title steals horizontal space from the numeric readout and the bar, pushing them to the right. You may need to increase the control width to prevent them from disappearing.

```
// Add titles. Readout widths, control widths unchanged.
ValDisplay valdisp2 title="Readout+Bar"
ValDisplay valdisp0 title="K0="
```



The limits values *low*, *high*, and *base* and the value of *valExpr* control how the bar, if any, is drawn. The bar is drawn from a starting position corresponding to the *base* value to an ending position determined by the value of *valExpr*, *low* and *high*. *low* corresponds to the left side of the bar, and *high* corresponds to the right. The position that corresponds to the *base* value is linearly interpolated between *low* and *high*.

For example, with *low* = -10, *high*=10, and *base*= 0, a *valExpr* value of 5 will draw from the center of the bar area (0 is centered between -10 and 10) to the right, halfway from the center to the right of the bar area (5 is halfway from 0 to 10):



You can force the control to not draw bars with fractional parts by specifying mode=3.

Killing Controls

You can kill (delete) a control from within a procedure using the **KillControl** operation (page V-468). This might be useful in creating control panels that change their appearance depending on other settings.

You can interactively kill a control by selecting it with the arrow tool or the Select Control submenu and press Delete.

Getting Information About Controls

You can use the **ControlInfo** operation (page V-89) to obtain information about a given control. This is useful to obtain the current state of a checkbox or the current setting of a pop-up menu.

ControlInfo is usually used for control panels that have a Do It button or equivalent. When the user clicks the button, its action procedure calls ControlInfo to query the state of each relevant control and acts accordingly.

ControlInfo is generally not used for the other style of control panel in which the action procedure for each control acts as soon as that control is clicked.

Updating Controls

You can use the **ControlUpdate** operation (page V-94) to cause a given control to redraw with its current value. You would use this in a procedure after changing the value or appearance of a control to display the changes before the normal update occurs.

Help Text for User-Defined Controls

Each control type has a help text property, set using the help keyword, through which you add a help tip. In Igor Pro 9.00 and later, tips are limited to 1970 bytes. Previously they were limited to 255 bytes.

Here is an example:

```
Button button0 title="Beep", help={"This button beeps."}
```

The tip appears when the user moves the mouse over the control, if tooltips are enabled in the Help section of the Miscellaneous Settings dialog.

You can use a limited set of HTML tags for formatting. See **HTML Tags in Tooltips** on page IV-311.

Modifying Controls

The control operations create a new control if the name parameter doesn't match a control already in the window. The operations modify an existing control if the name does match a control in the window, but generate an error if the control kind doesn't match the operation.

For example, if a panel already has a button control named `button0`, you can modify it with another `Button button0` command:

```
Button button0 disable=1            // hide
```

However, if you use a Checkbox instead of Button, you get a "button0 is not a Checkbox" error.

You can use the **ModifyControl** operation (page V-606) and **ModifyControlList** operation (page V-608) to modify a control without needing to know what kind of control it is:

```
ModifyControl button0 disable=1     // hide
```

This is especially handy when used in conjunction with tab controls.

Disabling and Hiding Controls

All controls support the keyword "disable=*d*" where *d* can be:

0:	Normal operation
1:	Hidden
2:	User input disabled
3:	Hidden and user input disabled

Charts and ValDisplays do not change appearance when disable=2 because they are read-only.

SetVariables also have the noedit keyword. This is different from disable=2 mode in that noedit allows user input via the up or down arrows but disable=2 does not.

Control Background Color

The background color of control panel windows and the area at the top of a graph as reserved by the **ControlBar** operation (page V-88) is a shade of gray chosen to match the operating system look. This gray is used when the control bar background color, as set by ModifyGraph cbRGB or ModifyPanel cbRGB, is the default pure white, where the red, green and blue components are all 65535. Any other cbRGB setting, including not quite pure white, is honored. However, some controls or portions of controls are drawn by the operating system and may look out of place if you choose a different background color.

For special purposes, you can specify a background color for an individual control using the labelBack keyword. See the reference help of the individual control types for details.

Control Structures

Control action procedures can take one of two forms: structure-based or an old form that is not recommended. This section assumes that you are using the structure-based form.

The action procedure for a control uses a predefined, built-in structure as a parameter to the function. The procedure has this format:

```
Function ActionProcName(s)
   STRUCT <WMControlTypeActio>& s    // <WMControlTypeActio> is one of the
   …                                 // structures listed below
End
```

The names of the various control structures are:

Control Type	Structure Name
Button	**WMButtonAction**
CheckBox	**WMCheckboxAction**
CustomControl	**WMCustomControlAction**
ListBox	**WMListboxAction**
PopupMenu	**WMPopupAction**
SetVariable	**WMSetVariableAction**
Slider	**WMSliderAction**
TabControl	**WMTabControlAction**

Action functions should respond only to documented `eventCode` values. Other event codes may be added along with more fields in the future. Although the return value is not currently used, action functions should always return zero.

The constants used to specify the size of structure `char` arrays are internal to Igor Pro and may change.

You can use the same action procedure for different controls of the same type, for all the buttons in one window, for example. Use the ctrlName field of the structure to identify the control and the win field to identify the window containing the control.

Control Structure Example

This example illustrates the extended event codes available for a button control. The function prints various text messages to the history area of the command window, depending what actions you take while in the button area.

```
Function ControlStructureTest()
   NewPanel
   Button b0,proc= NewButtonProc
End

Structure MyButtonInfo
   Int32 mousedown
   Int32 isLeft
EndStructure

Function NewButtonProc(s)
   STRUCT WMButtonAction &s

   STRUCT MyButtonInfo bi
   Variable biChanged= 0

   StructGet/S bi,s.userdata
   if( s.eventCode==1 )
      bi.mousedown= 1
      bi.isLeft= s.mouseLoc.h < (s.ctrlRect.left+s.ctrlRect.right)/2
      biChanged= 1
   elseif( s.eventCode==2 || s.eventCode==3 )
      bi.mousedown= 0
      biChanged= 1
   elseif( s.eventCode==5 )
      print "Enter button"
   elseif( s.eventCode==6 )
      print "Leave button"
   endif

   if( s.eventCode==4 )             //  mousemoved
      if( bi.mousedown )
         if( bi.isLeft )
            printf "L"
         else
            printf "R"
         endif
      else
         printf "*"
      endif
   endif
   if( biChanged )
      StructPut/S bi,s.userdata     // written out to control
   endif

   return 0
End
```

Control Structure eventMod Field

The eventMod field appears in the built-in structure for each type of control. It is a bitfield defined as follows:

EventMod Bit	Meaning
Bit 0	A mouse button is down.
Bit 1	Shift key is down.
Bit 2	Option (Macintosh) or Alt (Windows) is down.
Bit 3	Command (Macintosh) or Ctrl (Windows) is down.
Bit 4	Contextual menu click occurred.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Control Structure blockReentry Field

Long ago, on Macintosh version 6 and before, if a control action procedure took a long time to run, it was possible to repeat the action and start the action procedure again before the previous invocation was finished (that's called reentrancy). That is, if a button starts a long computation, like a lengthy curve fit, it was possible to start a new fit before the first was done, if you clicked the button again too soon. To control that, the blockReentry member was added.

In Igor 7 and later, code was added that prevents reentry, so the blockReentry field is obsolete. It is retained for backward compatibility and has no effect.

If you find a case in which reentry is a problem, please contact support@wavemetrics.com with a report. We will need an example that we can use to reproduce the problem.

Control Structure blockReentry Advanced Example

This example further illustrates the use of the blockReentry field. It is of interest only to those who want to experiment with this issue.

The ReentryDemoPanel procedure below creates a panel with two buttons. Each button prints a message in the history area when the action procedure receives the "mouse up" message, then pauses for two seconds, and then prints another message in the history before returning. The pause is a stand-in for a procedure that takes a long time.

The top button does not block reentry so, if you click it twice in quick succession, the action procedure is reentered and you get nested messages in the history area.

The bottom button does block reentry so, if you click it twice in quick succession, the action procedure is not reentered.

Because of architectural differences, reentry of the action procedure occurs on Macintosh only in Igor6 and does not occur at all in Igor7 or later. Because reentry may be affected by internal changes in Igor or by operating system changes, it may reappear as an issue in the future.

```
Function ButtonProc(ba) : ButtonControl
   STRUCT WMButtonAction &ba

   switch( ba.eventCode )
      case 2: // mouse up
         // Block bottom button only
         ba.blockReentry= CmpStr(ba.ctrlName,"Block") == 0
         print "Start button ",ba.ctrlName
         Variable t0= ticks
         do
            DoUpdate
         while(ticks < (t0+120) )
         Print "Finish button",ba.ctrlName
         break
   endswitch

   return 0
End

Window ReentryDemoPanel() : Panel
   PauseUpdate; Silent 1// building window...
   NewPanel /K=1 /W=(322,55,622,255)
   Button NoBlock,pos={25,10},size={150,20},proc=ButtonProc,title="No Block Reentry"
   Button Block,pos={25,50},size={150,20},proc=ButtonProc,title="Block
Reentry"
End
```

User Data for Controls

You can store arbitrary data with a control using the userdata keyword. You can set user data for the following controls: Button, CheckBox, CustomControl, ListBox, PopupMenu, SetVariable, Slider, and TabControl.

Each control has a primary, unnamed user data string that is used by default. You can also store an unlimited number of additional user data strings by specifying a name for each one. The name can be any legal standard Igor name.

You can retrieve information from the default user data using the **ControlInfo** operation (page V-89), which returns such information in the S_UserData string variable. To retrieve any named user data, you must use the **GetUserData** operation (page V-316).

Although there is no size limit to how much user data you can store, it does have to be generated as part of the recreation macro for the window when experiments are saved. Consequently, huge user data strings can slow down experiment saving and loading

User data is intended to replace or reduce the usage of global variables for maintaining state information related to controls.

Control User Data Examples

:Here is a simple example of a button with user data:

```
NewPanel
Button b0, userdata="user data for button b0"
Print GetUserData("","b0","")
```

Here is a more complex example.

Copy the following code into the procedure window of a new experiment and run the `Panel0` macro. Then click the buttons.

```
Structure mystruct
    Int32 nclicks
    double lastTime
EndStructure

Function ButtonProc(ctrlName) : ButtonControl
    String ctrlName

    STRUCT mystruct s1
    String s= GetUserData("", ctrlName,"")
    if( strlen(s) == 0 )
        print "first click"
    else
        StructGet/S s1,s
        // Warning: Next command is wrapped to fit on the page.
        printf "button %s clicked %d time(s), last click = %s\r",ctrlName, s1.nclicks,
Secs2Date(s1.lastTime, 1 )+" "+Secs2Time(s1.lastTime,1)
    endif
    s1.nclicks += 1
    s1.lastTime= datetime
    StructPut/S s1,s
    Button $ctrlName,userdata= s
End

Window Panel0() : Panel
    PauseUpdate; Silent 1           // building window...
    NewPanel /W=(150,50,493,133)
    SetDrawLayer UserBack
    Button b0,pos={12,8},size={50,20},proc=ButtonProc,title="Click"
    Button b1,pos={65,8},size={50,20},proc=ButtonProc,title="Click"
    Button b2,pos={119,8},size={50,20},proc=ButtonProc,title="Click"
    Button b3,pos={172,8},size={50,20},proc=ButtonProc,title="Click"
    Button b4,pos={226,8},size={50,20},proc=ButtonProc,title="Click"
EndMacro
```

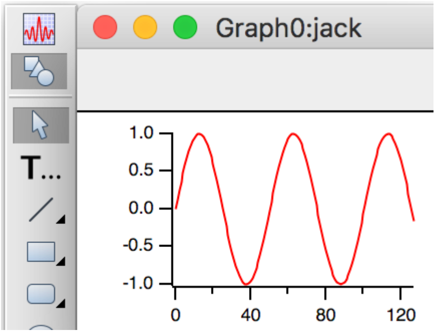
Action Procedures for Multiple Controls

You can use the same action procedure for different controls of the same type, for all the buttons in one window, for example. The name of the control is passed to the action procedure so that it can know which control was clicked. This is usually the name of the control *in the target/active window*, which is what most control operations assume.

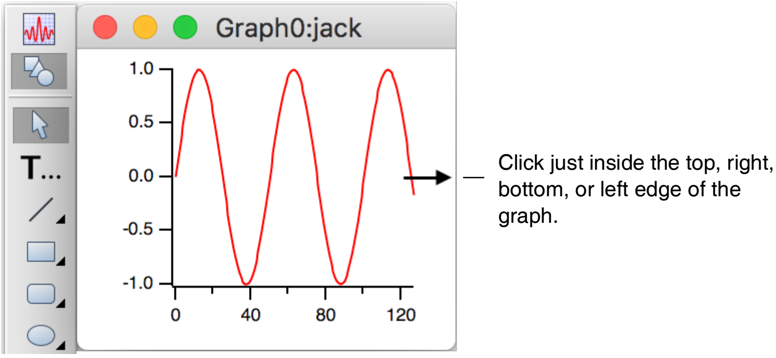
Controls in Graphs

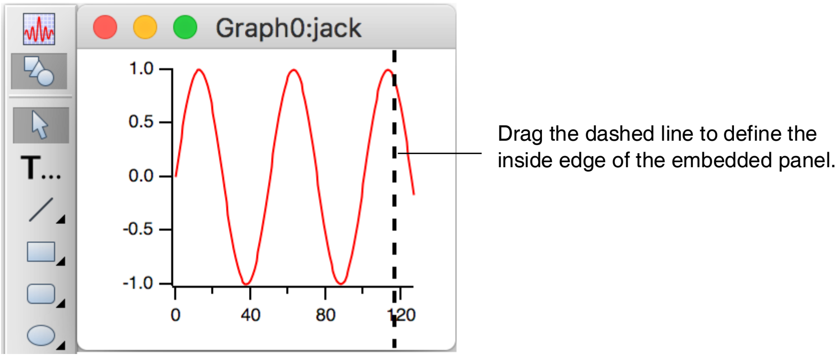
The combination of controls and graphs provides a nice user interface for tinkering with data. You can create such a user interface by embedding controls in a graph or by embedding a graph in a control panel. This section explains the former technique, but the latter technique is usually recommended. See Chapter III-4, **Embedding and Subwindows** for details.

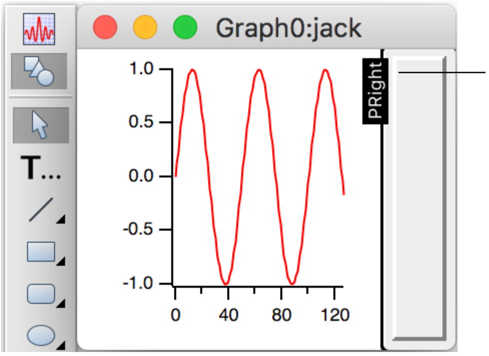
Although controls can be placed anywhere in a graph, you can and should reserve an area just for controls at the edge of a graph window. Controls in graphs operate much more smoothly if they reside in these reserved areas. The **ControlBar** operation (page V-88) or the Control Bar dialog can be used to set the height of a nonembedded control area at the top of the graph.



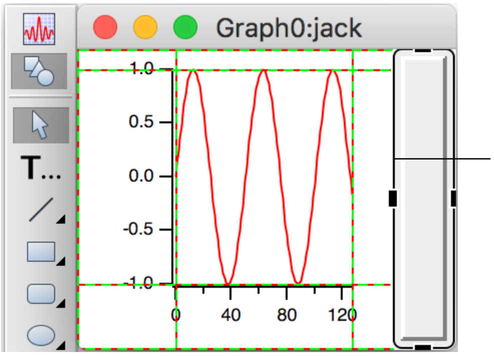
The simplest way to add a panel is to click near the edge of the graph and drag out a control area:


Graph0:jack
T...
1.0
0.5
0.0
-0.5
-1.0
0
40
80
120
Click just inside the top, right, bottom, or left edge of the graph.


Graph0:jack
T...
1.0
0.5
0.0
-0.5
-1.0
0
40
80
120
Drag the dashed line to define the inside edge of the embedded panel.



`PRight` is the name of the resulting embedded panel subwindow. The label disappears in “operate” mode.



Adjust the position of the embedded window by clicking the subwindow frame and dragging its handles. The dashed lines represent the edges of the plot and graph areas, and the subwindow frame snaps and attaches to them.

The background color of a control area or embedded panel can be set by clicking the background to exit any subwindow layout mode, then Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the background, and then selecting a color from the contextual menu’s pop-up color palette. See **Control Background Color** on page III-437 for details.

The contextual menu adjusts the style of the frame around the panel.

You can use the same contextual menu to remove an embedded panel, leaving only the bare control area underneath. Remove the control area by dragging the inside edge back to the outside edge of the graph.

Drawing Limitations

The drawing tools can not be used in bare control areas of a graph. If you want to create a fancy set of controls with drawing tools, you have to embed a panel subwindow into the graph.

Updating Problems

You may occasionally run into certain updating problems when you use controls in graphs. One class of update problems occurs when the action procedure for one control changes a variable used by a ValDisplay control in the same graph and also forces the graph to update while the action procedure is being executed. This short-circuits the normal chain of events and results in the ValDisplay not being updated.

You can force the ValDisplay to update using the **ControlUpdate** operation (page V-94). Another solution is to use a control panel instead of a graph.

The ControlUpdate operation can also solve problems in updating pop-up menus. This is described above under **Creating PopupMenu Controls** on page III-426.

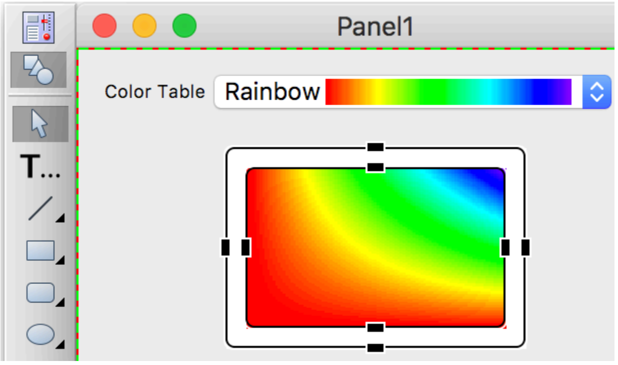
Control Panels

Control panels are windows designed to contain controls. The **NewPanel** creates a control panel.

Drawing tools can be used in panel windows to decorate control panels. Control panels have two drawing layers, UserBack and ProgBack, behind the controls and one layer, Overlay, in front of the controls. See **Drawing Layers** on page III-68 for details.

A panel window's background color can be set by Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the background and then selecting a color from the pop-up color palette. See **Control Background Color** on page III-437 for details.

Embedding into Control Panels



You can embed a graph, table, notebook, or another panel into a control panel window. See Chapter III-4, **Embedding and Subwindows** for details. This technique is cleaner than adding control areas to a graph. It also allows you to embed multiple graphs in one window with controls.

Use the contextual menu while in drawing mode to add an embedded window. Click on the frame of the embedded window to adjust the size and position.

You can use a notebook subwindow in a control panel to display status information or to accept lengthy user input. See **Notebooks as Subwindows in Control Panels** on page III-91 for details.

Exterior Control Panels

Exterior subwindows are panels that act like subwindows but live in their own windows attached to a host window. The host window can be a graph, table, panel or Gizmo plot. The host window and its exterior subwindows move together and, in general, act as a single window. Exterior subwindows have the advantage of not disturbing the host window and, unlike normal subwindows, are not limited in size by the host window.

To create an exterior subwindow panel, use **NewPanel** with the /EXT flag in combination with /HOST.

Floating Control Panels

Floating control panels float above all other windows, except dialogs. To create a floating panel, use **NewPanel** with the /FLT flag.

Control Panel Expansion

In Igor Pro 9.00 and later, you can set an expansion factor for a control panel using the Expansion submenu in the Panel menu or the panel's contextual menu which you summon by right-clicking. You can set it for all control panels using the Panel section of the Miscellaneous Settings dialog. As explained below, setting it for all panels is recommended in most cases.

The factory default control panel expansion factor is 1.0 which means that control panels and their controls are displayed at "normal" size. The normal size depends on the pixel size of your screen and on various complex software factors. It may be too small or too large for your taste. If so, you can use the control panel expansion factor to adjust.

When you change the expansion factor, the size of the panel window and any controls and subwindows in the panel changes accordingly. Additionally, elements drawn in the panel using drawing tools are also expanded.

If normal size is not ideal for you, you will usually want to change the expansion for all control panels, not for just one specific control panel. You can do this by changing the default control panel expansion factor using the Panel section of the Miscellaneous Settings dialog. For most uses, changing the default is recommended over changing the expansion for a specific control panel.

When creating a control panel programmatically, you can set the expansion factor using the NewPanel /EXP=<factor> flag which was added in Igor Pro 9.00. However, for most uses, you should omit /EXP and allow the user's default control panel expansion factor to take effect.

When a control panel is recreated, such as when you open an experiment with a control panel, if the /EXP=<factor> flag is present in the recreation macro, Igor applies the specified expansion factor to the panel. If the /EXP flag is omitted, Igor applies the user's default control panel expansion factor.

When Igor generates a control panel recreation macro, if the control panel uses other than the default expansion as set in the Panel section of the Miscellaneous Settings dialog, Igor uses the NewPanel /EXP flag to specify the desired expansion. If the experiment is opened on a different machine, the specific expansion factor is applied. This overrides the user's default control panel expansion factor. You can avoid overriding the user's preference by using the default control panel expansion rather than setting it for a specific panel.

It is rarely necessary, but you can programmatically set the expansion factor of a control panel window or subwindow using a ModifyPanel command with the expand keyword which was added in Igor Pro 9.00. For main control panel windows, this works the same as setting the expansion using the Expansion submenu. For control panel subwindows, it affects the size of the controls but not the size of the subwindow itself. For the reasons explained above, it is usually best to refrain from programmatically setting the expansion so that the user's default takes effect.

You can add controls to a graph by creating a control bar using the **ControlBar** operation. The default control panel expansion factor affects the size of the control bar in a graph and the sizes of the controls in the control bar.

Control Panel Units

The size and position of a control panel window on screen is set by the NewPanel /W=(*left*,*top*,*right*,*bottom*) flag. The size and position of controls within a control panel are set by the pos={*left*,*top*} and size={*width*,*height*} keywords of control operations like Button and TitleBox.

For historical reasons, the interpretation of these parameters depends on the operating system; see **Control Panel Resolution on Windows** on page III-456 for details.

On Macintosh the parameters are always interpreted as points (1/72 of an inch).

On Windows, they are interpreted as points except if the screen resolution is 96 DPI in which case, for compatibility with existing experiments and procedures, they are interpreted as pixels.

To allow users to control the size of control panels and their contents, Igor Pro 9.00 introduced **Control Panel Expansion** (see page III-443). The rest of this section discusses how control panel expansion affects the interpretation of the NewPanel /W and control operation keyword parameters.

A parameter that is expressed in normal control panel units (points or pixels as described above) to which Igor applies the control panel expansion factor is said to be expressed in *control panel units*.

Interpretation of NewPanel Coordinates

First we consider the NewPanel /W=(*left*,*top*,*right*,*bottom*) flag.

When control panel expansion is other than the factory default 1.0, Igor scales the width (computed as *right-left*) and height (computed as *bottom-top*) of the panel by the expansion factor but does not change the interpretation of left and top.

Consider these examples assuming that the default control panel expansion is 1.0:

```
NewPanel/N=Panel1/W=(100,100,300,200)
NewPanel/N=Panel2/EXP=2/W=(100,100,300,200)
```

The first NewPanel command creates a panel at the specified coordinates which are interpreted as points or pixels as described in the preceding section.

In the second command, the expansion is 2.0 because of the /EXP flag. This does not affect the position of the window as specified by the left and top parameters. It does affect the width and height of the panel as computed by Igor from width=*right-left* and height=*bottom-top*. After computing the width and height, Igor applies the expansion factor. The result is that the top/left corner of Panel2 is the same as Panel1 but Panel2 is twice as wide and twice as tall. We wind up with width=200 and height=100 in both cases but, because of panel expansion, Panel2 is twice as large as Panel1.

The *left*, *top*, *right*, and *bottom* parameters are in normal panel units but NewPanel interprets the internally computed width and height as control panel units (i.e., normal units to which control panel expansion is to be applied).

If you use /I (inches) or /M (centimeters) before /W then the interpretation of the parameters is different. They are converted from inches or centimeters to points and the result is treated as points, as with the Display operation.

Interpretation of Control Operation Coordinates

Now we consider the pos={*left*,*top*} and size={*width*,*height*} keywords of control operations like Button and TitleBox.

All of these parameters are treated as control panel units. This means that they are first treatedas points or pixels as described above and then the expansion factor of the targeted panel, if other than 1.0, is applied.

Consider these examples which use Panel1 and Panel2 from the preceding section:

```
AutopositionWindow /R=Panel1 Panel2       // Make panels side-by-side
Button button0 win=Panel1, pos={100,50}, size={100,20}, title="Button"
Button button0 win=Panel2, pos={100,50}, size={100,20}, title="Button"
```

We created buttons that are one-half the width of the panel and with the top/left corner of the button one-half of the way from the top to the bottom and one-half of the way from the left to the right. We did this using the same parameters in both cases. This gives different results because the Button operation interprets the parameters as being expressed in control panel units.

Interpretation of Drawing Coordinates

When you use drawing tools and drawing operations in control panels, coordinates are interpreted as control panel units the same as for control operations such as Button.

See also **Control Panel Resolution on Windows** on page III-456, **ScreenResolution** on page V-832, **PanelResolution** on page V-732

Control Panel Preferences

Control panel preferences allow you to control what happens when you create a new control panel. To set preferences, create a panel and set it up to your taste. We call this your prototype panel. Then choose Capture Panel Prefs from the Panel menu.

Preferences are normally in effect only for manual operations, not for automatic operations from Igor procedures. This is discussed in more detail in Chapter III-18, **Preferences**.

When you initially install Igor, all preferences are set to the factory defaults. The dialog indicates which preferences you have changed.

The preferences affect the creation of new panels only.

Selecting the Show Tools category checkbox captures whether or not the drawing tools palette is initially shown or hidden when a new panel is created.

See also **Control Panel Expansion** on page III-443.

Controls Shortcuts

Action	**Shortcut** (*Macintosh*)	**Shortcut** (*Windows*)
To show or hide a panel's or graph's tool palette	Press Command-T.	Press Ctrl+T.
To move or resize a user-defined control without using the tool palette	Press Command-Option and click the control. With Command-Option still pressed, drag or resize it.	Press Ctrl+Alt and click the control. With Ctrl+Alt still pressed, drag or resize it.
To add a user-defined control without using the tool palette	Press Command-Option and choose from the Panel or Graph menu's Add Control submenu.	Press Ctrl+Alt and choose from the Panel or Graph menu's Add Control submenu.
To modify a user-defined control	Press Command-Option and double-click the control. This displays a dialog for modifying all aspects of the control. If the control is already selected, you don't need to press Command-Option.	Press Ctrl+Alt and double-click the control. This displays a dialog for modifying all aspects of the control. If the control is already selected, you don't need to press Ctrl+Alt.
To edit a user-defined control's action procedure	With the panel in modify mode (tools showing, second icon from the top selected) press the Control key and click the control. This displays a contextual menu with a "Go to <action procedure>" item.	With the panel in modify mode (tools showing, second icon from the top selected) right-click the control. This displays a contextual menu with a "Go to <action procedure>" item.
To create an embedded graph or table in the panel	With the panel in modify mode, press the Control key and click the panel background. Choose the subwindow type from the resulting contextual menu's "New" submenu.	With the panel in modify mode, right-click the panel background. Choose the subwindow type from the resulting contextual menu's "New" submenu.
To change an embedded window's border style	With the panel in modify mode, press the Control key and click the embedded window. Choose the border style from the resulting contextual menu's "Frame" and "Style" submenus.	With the panel in modify mode, right-click the embedded window. Choose the border style from the resulting contextual menu's "Frame" and "Style" submenus.
To remove an embedded window	With the panel in modify mode, press the Control key and click the embedded window. Choose the Delete from the resulting contextual menu.	With the panel in modify mode, right-click the embedded window. Choose the Delete from the resulting contextual menu.
To eliminate a control area at the edge of a graph	In modify mode or while pressing Command-Option, click the inside edge of the control area and drag it to the outside edge of the graph.	In modify mode or while pressing Ctrl+Alt, click the inside edge of the control area and drag it to the outside edge of the graph.
To nudge a user-defined control's position	Select the control and press arrow keys. Press Shift to nudge faster.	Select the control and press arrow keys. Press Shift to nudge faster.

Chapter

III-15

Platform-Related Issues

Platform-Related Issues

Igor Pro runs on Macintosh and Windows. This chapter contains information that is platform-specific and also information for people who use Igor on both platforms.

Windows-Specific Issues

On Windows, the name of the Igor program file must be “Igor64.exe” for the 64-bit version of Igor or “Igor.exe” for the 32-bit version, exactly. If you change the name, Igor extensions will not work because they will be unable to find Igor.

Cross-Platform File Compatibility

Version 3.1 was the first version of Igor Pro that ran on Windows as well as Macintosh.

Crossing Platforms

When crossing from one platform to another, page setups are only partially translated. Igor tries to preserve the page orientation and margins.

When crossing platforms, Igor attempts to do font substitution where necessary. If Igor can not determine an appropriate font it displays the font substitution dialog where you can choose the font.

Platform-specific picture formats are displayed as gray boxes when you attempt to display them on the non-native platform. Windows Metafile, Enhanced Metafile, and Windows bitmap (BMP) pictures are supported on Windows only and appear as gray boxes on Macintosh.

The EPS, PNG, JPEG, TIFF, and SVG formats are platform-independent and are displayed on both platforms.

Prior to Igor Pro 9.00, PDF pictures were supported on Macintosh only and appeared as gray boxes on Windows. Now Igor can display PDF pictures on both platforms.

Transferring Files Using File Transfer Programs

Some transfer programs offer the option of translating file formats as they transfer the program from one computer to another. This translation usually consists of replacing each carriage return character with a carriage return/linefeed pair (Macintosh to Windows) or vice-versa (Windows to Macintosh). This is called a “text mode” transfer, as opposed to a “binary mode” transfer. This translation is appropriate for plain text files only. In Igor, plain text notebooks, procedure files, and Igor Text data files are plain text. All other files are not plain text and will be corrupted if you transfer in text mode. If you get flaky results after transferring a file, transfer it again making sure text mode is off.

If you have a problem opening a binary file after doing a transfer, compare the number of bytes in the file on both computers. If they are not the same, the transfer has corrupted the file.

File Name Extensions, File Types, and Creator Codes

This table shows the file name extension and corresponding Macintosh file type for Igor Pro files:

Extension	File Type	What’s in the File
.pxp	IGsU	Packed experiment file
.pxt	IGsS	Packed experiment template (stationery)
.uxp	IGSU	Unpacked experiment file
.uxt	IGSS	Unpacked experiment template (stationery)
.ifn	WMT0	Igor formatted notebook (last character is zero)

Extension	File Type	What's in the File
.txt	TEXT	Igor plain notebook
.ihf	WMT0	Igor help file
.ibw	IGBW	Igor binary data file

The Macintosh creator code for Igor is 'IGR0' (last character is zero).

Experiments and Paths

An Igor experiment sometimes refers to wave, notebook, or procedure files that are stored separate from the experiment file itself. This is discussed under **References to Files and Folders** on page II-24. In this case, Igor creates a symbolic path that points to the folder containing the referenced file. It writes a **NewPath** command in the experiment file to recreate the symbolic path when the experiment is opened. When you move the experiment to another computer or to another platform, this path may not be valid. However, Igor goes to great lengths to find the folder, if possible.

Igor stores the path to the folder containing the file as a relative path, relative to the experiment file, if possible. This means that Igor will be able to find the folder, even on another computer, if the folder's relative location in the disk hierarchy is the same on both computers. You can minimize problems by using the same disk hierarchy on both computers.

If the folder is not on the same volume as the experiment file, then Igor can not use a relative path and must use an absolute path. Absolute paths cause problems because, although your disk hierarchy may be the same on both computers, often the name of the root volume will be different. For example, on the Macintosh your hard disk may be named "hd" while on Windows it may be named "C:".

If Igor can not locate a folder or file needed to recreate an experiment, it displays a dialog asking you to locate it.

Picture Compatibility

Igor displays pictures in graphs, page layouts, control panels and notebooks. The pictures are stored in the Pictures collection (Misc→Pictures) and in notebooks. Graphs, page layouts and control panels reference pictures stored in the Pictures collection while notebooks store private copies of pictures.

This table shows the graphic formats that Igor can use to store pictures:

Format	How To Create	Notes
PDF	Paste or use Misc→Pictures	Cross-platform high resolution vector format. See **Importing PDF Pictures** on page III-510.
EMF (Enhanced Metafile)	Paste or use Misc→Pictures	Windows only See **Graphics Technology on Windows** on page III-506 for information about different types of EMF pictures.
BMP (bitmap)	Use Misc→Pictures	Windows Only. BMP also called DIB (device-independent bitmap).
PNG (Portable Network Graphics)	Use Misc→Pictures	Cross-platform bitmap format
JPEG	Use Misc→Pictures	Cross-platform bitmap format
TIFF (Tagged Image File Format)	Use Misc→Pictures	Cross-platform bitmap format

Format	How To Create	Notes
EPS (Encapsulated PostScript)	Use Misc→Pictures	High resolution vector format. EPS is largely obsolete. Use PDF instead.
SVG (Scalable Vector Graphics)	Use Misc→Pictures	Cross-platform high resolution vector format.

Formats that are not supported on the current platform are drawn as gray boxes.

Although Igor does not display non-native graphic formats, it does preserve them. For example, you can create an experiment on Windows and paste a Windows metafile into a page layout, graph, or notebook window. If you save the experiment and open it on Macintosh, the Windows metafile is displayed as a gray box. If you now save and open the experiment on Windows again, the Windows metafile is displayed correctly.

Converting to PNG

If you want platform-specific pictures to be displayed correctly on both platforms, you must convert the pictures to PNG. To convert to PNG, use the Pictures dialog (Misc menu) for pictures in graphs and page layouts, or for pictures in notebooks, use the Special submenu in the Notebook menu.

Converting a picture to PNG makes it a bitmap format and may degrade resolution. This is fine for graphics intended to be viewed on the screen but not for graphics intended to be printed at high resolution. You can convert to a high resolution PNG without losing much picture quality.

Page Setup Compatibility

Page setup records store information regarding the size and orientation of the page. Prior to Igor Pro 7, page setups contained platform-depedent and printer-depedent data. This is no longer the case, but as a consequence, only minimal information is stored.

In each experiment file, Igor stores a separate page setup for each page layout, notebook, and procedure window, and stores a single page setup for all graphs and a single page setup for all tables.

File System Issues

This section discusses file system issues that you need to take into account if you use Igor on both Macintosh and Windows.

File and Folder Names

On Windows, the following characters are illegal in file and folder names: backslash (\), forward slash (/), colon (:), asterisk (*), question mark (?), double-quote ("), left angle bracket (<), right angle bracket (>), vertical bar (|). On Macintosh, the only illegal character is colon.

This means, for example, that you can not create a file with a name like “Data 1/23/98” on Windows. You can create a file with this name on Macintosh. If you write an Igor procedure that generates a file name like this, it will run on Macintosh but fail on Windows.

Therefore, if you are concerned about cross-platform compatibility, you must not use any of the Windows illegal characters in a file or folder name, even if you are running on Macintosh. Also, don’t use period except before a file name extension.

File and folder names in Windows can theoretically be up to 255 characters in length. Because of some limitations in Windows and also in Igor, you will encounter errors if you use file names that long. However, both Igor and Windows are capable of dealing with file names up to about 250 characters in length. It is unlikely that you will approach this limit.

An exception to this is that Igor limits names of XOP files to 31 characters, plus the ".xop" extension. Igor will not recognize an XOP file with a longer name.

Paths in Windows are limited to 259 characters in length. Neither Windows nor Igor can deal with a path that exceeds this limit. For example, if you create a directory with a 250 character name and try to create a file with a 15 character name, neither Windows nor Igor will permit this.

This boils down to the following: Feel free to use long file and directory names, but expect to see errors if you use outrageously long names or if you have directories so deeply nested that paths approach the theoretical limit.

Path Separators

The Macintosh HFS file system uses a colon to separate elements in a file path. For example:

```
hd:Igor Pro Folder:Examples:Sample Graphs:Contour Demo.pxp
```

The Windows file system uses a backslash to separate elements in a file path. For example:

```
C:\Igor Pro Folder\Examples\Sample Graphs:Contour Demo.pxp
```

Some Igor operations (e.g., LoadWave) allow you to enter file paths. Igor accepts Macintosh-style or Windows-style paths regardless of the platform on which you are running.

Note: Igor uses the backslash character as an escape character in literal strings. This can cause problems when using Windows-style paths.

For example, the following command creates a textbox with two lines of text. "\r" is an escape code inserts a carriage return character:

```
Textbox "This is line 1.\rThis is line 2."
```

Because Igor interprets a backslash as an escape character, the following command will not execute properly:

```
LoadWave "C:\Data Files\really good data.ibw"
```

Instead of loading a file named "really good data.ibw", Igor would try to load a file named "Data Files<CR>eally good data.ibw", where <CR> represents the carriage return character. This happens because Igor interprets "\r" in literal strings to mean carriage return.

To solve this problem, you must use "\\" instead of "\" in a file path. Igor will correctly execute the following:

```
LoadWave "C:\\Data Files\\really good data.ibw"
```

This works because Igor interprets "\\" as an escape sequence that means "insert a backslash character here".

Another solution to this problem is to use a Macintosh HFS-style path, even on Windows:

```
LoadWave "C:Data Files:really good data.ibw"
```

Igor converts the Macintosh HFS-style path to a Windows-style path before using it. This avoids the backslash issue.

For a complete list of escape sequences, see **Escape Sequences in Strings** on page IV-14.

If you are writing procedures that need to extract sections of file paths or otherwise manipulate file paths, the **ParseFilePath** function on page V-733 may come in handy.

UNC Paths

"UNC" stands for "Universal Naming Convention". This is a Windows convention for identifying resources on a network. One type of network resource is a shared directory. Consequently, when running under Windows, in order to reference a network directory from an Igor command, you need to use a UNC path.

The format of a UNC path that points to a file in a folder on a shared server volume or directory is:

```
    "\\server\share\directory\filename"
```

“server” is the name of the file server and “share” is the name of the top-level shared volume or directory on that server.

Because Igor treats a backslash as an escape character, in order to reference this from an Igor command, you would have to write:

```
"\\\\server\\share\\directory\\filename"
```

As described in the preceding section, you could also use Macintosh HFS path syntax by using a colon in place of two backslashes. However, you can not do this for the “\\server\share” part of the path. Thus, using Macintosh HFS syntax, you would write:

```
"\\\\server\\share:directory:filename"
```

Unix Paths

Unix paths use the forward slash character as a path separator. Igor does not recognize Unix paths. Use Macintosh HFS paths instead.

Keyboard and Mouse Usage

This section describes how keyboard and mouse usage differs on Macintosh versus Windows. It is intended to help Igor users more easily adapt when switching platforms.

There are three main differences between Macintosh and Windows input mechanisms:

1. The Macintosh mouse may have one button and the Windows mouse has two.
2. The Macintosh keyboard has four modifier keys (Shift, Command, Option, Control) while the Windows keyboard has three (Shift, Ctrl, Alt).
3. The Macintosh keyboard has Return and an Enter keys while the Windows keyboard (usually) has two Enter keys.

For the most part, Igor maps between Macintosh and Windows input as follows:

Macintosh	**Windows**
Shift	Shift
Command	Ctrl
Option	Alt
Control	<not mapped>

Macintosh	**Windows**
Return	Enter
Enter	Enter
Control-click	Right-click

In notebooks, procedure windows and help windows, pressing Control-Return or Control-Enter executes the selected text or, if there is no selection, to execute the line of text containing the caret.

Command Window Input

This table compares command window mouse actions:

Action	Macintosh	Windows
Copy history selection to command line	Option-click	Alt+click
Copy history to command and start execution	Command-Option-click	Ctrl+Alt+click
Invoke contextual menu	Control-click	Right-click

Cross-Platform Text and Fonts

Text Encoding Compatibility

Prior to Igor7, Igor used system text encoding. On Macintosh, this was usually MacRoman. On Windows, it was usually Windows-1252. On Japanese systems, it was Shift JIS on both platforms.

As of Igor7, Igor uses UTF-8 text encoding internally on both Macintosh and Windows.

When opening old files, Igor must convert from the file's text encoding to UTF-8 for storage in memory.

Dealing with various text encodings is a complex issue. See **Text Encodings** on page III-459 for details.

Carriage Returns and Linefeeds

The character or character pattern that marks the end of a line of text in a plain text file is called the "line terminator". There are three common line terminators, carriage return (CR, ASCII 13, used on old Macintosh systems), linefeed (LF, ASCII 10, used on Unix) and carriage return plus linefeed (CRLF, used on Windows).

When Igor Pro opens a text file (procedure file, plain text notebook or plain text data file), it accepts CR, LF or CRLF as the line terminator.

If you create a new procedure file or plain text notebook, Igor writes LF on Mac OS and CRLF on Windows. If you open an existing plain text file, edit it and then save it, Igor preserves the original terminator as determined by examining the first line in the file.

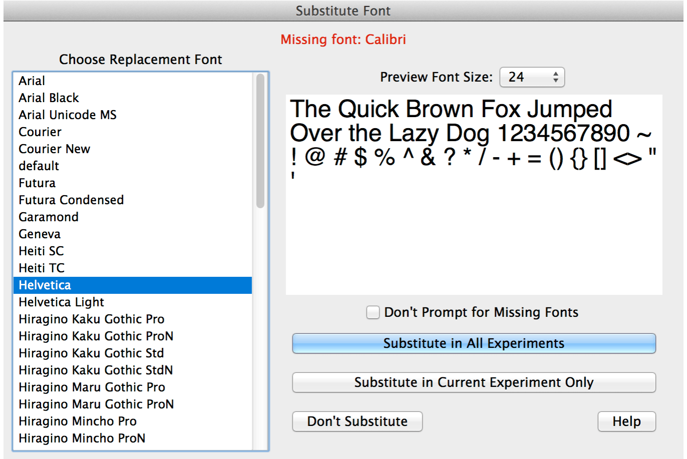
By default, the **FReadLine** operation treats CR, LF, or CRLF as terminators. Use this to write a procedure that can read lines from a text file without caring whether it is a Macintosh, Windows, or Unix file.

Font Substitution

When a font specified in a command or document is not installed, Igor applies font substitution to choose an installed font to use in place of the missing font. Dealing with these missing fonts often occurs when transferring a Windows-originated document to Macintosh or vice versa.

Igor employs two levels of font substitution: user-level editable substitution and built-in uneditable substitution.

The first level is an optional user-level font substitution facility that you will usually encounter for the first time when Igor displays the Substitute Font Dialog while opening an experiment or file. Use the dialog to choose a temporary or permanent replacement for the missing font:



Missing font: Calibri

Preview Font Size: 24

The Quick Brown Fox Jumped Over the Lazy Dog 1234567890 ~ ! @ # $ % ^ & ? * / - + = () {} [] <> " '

The second level is Igor’s built-in substitution table which substitutes between fonts normally installed on various Macintosh and Windows operating systems. For example, it substitutes Arial (a standard Windows font) for Geneva (a standard Macintosh font) if Geneva is not installed (which it usually isn’t on a Windows computer).

The user-level substitutions have priority over the built-in substitutions.

The Substitute Font dialog appears when neither level of font substitution specifies a replacement for the missing font. You can prevent this dialog from appearing by selecting the Don’t Prompt for Missing Fonts checkbox in the Substitute Font dialog.

You can manage font substitutions without waiting for a missing font situation to occur by choosing Misc-Edit Font Substitutions.

The user-level font substitution table is maintained in the “Igor Font Substitutions.txt” text file in Igor’s preferences folder. The file format is:

```
<name of missing font to replace> = <name of font to use instead>
```

one entry per line. For example:

```
Palatino=New Times Roman
```

spaces or tabs are allowed around the equals sign.

When a missing font is replaced, Igor uses the name of the replacement font instead of the name of the font in the command.

The name of the missing font is replaced only in the sense that the altered or created object (window, control, etc.) uses and remembers only the name of the replacement font. Recreation macros, including experiment recreation procedures, use the name of the replacement font when the experiment is saved. The command, however, is unaltered and still contains the name of the missing font.

Cross-Platform Procedure Compatibility

Igor procedures are about 99.5% platform-independent. For the other 0.5%, you need to test which platform you are running on and act accordingly. You can use ifdefs to achieve this. For example:

```
Function Demo()
   #ifdef MACINTOSH
      Print "We are running on Macintosh"
   #else
      #ifdef WINDOWS
         Print "We are running on Windows"
      #else
```

```
         Print "Unknown OS"
      #endif
   #endif
End
```

File Paths

As described under **Path Separators** on page III-451, Igor accepts paths with either colons or backslashes on either platform.

The use of backslashes is complicated by the fact that Igor uses the backslash character as an escape character in literal strings. This is also described in detail under **Path Separators** on page III-451. The simplest solution to this problem is to use a colon to separate path elements, even when you are running on Windows.

If you are writing procedures that need to extract sections of file paths or otherwise manipulate file paths, the **ParseFilePath** function on page V-733 may come in handy.

Igor supports paths up to 2000 bytes but operating system limits may apply. See **File Names and Paths** on page II-21 for details.

File Types and Extensions

On Mac OS 9, all files had a file type property. This property is a four letter code that is stored with the file by the Macintosh file system. For example, plain text files have the file type TEXT. Igor binary wave files have the file type IGBW. The file type property controlled the icon displayed for the file and which programs could open the file.

The file type property is no longer used on Macintosh. On Mac OS X, as well as on Windows, the file type is indicated by the file name extension.

For backward compatibility, some Igor operations and functions, such as **IndexedFile**, still accept Macintosh file types. New code should use extensions instead.

Points Versus Pixels

A pixel is the area taken up by the smallest displayable dot on an output device such as a display screen or a printer. The physical width and height of a pixel depend on the device.

In Igor, most measurements of length are in terms of points. A point is roughly 1/72 of an inch. 72 points make up 1 “logical inch”. Because of hardware differences and system software adjustments, the actual size of a logical inch varies from screen to screen and system to system.

Window Position Coordinates

With one exception, Igor stores and interprets window position coordinates in units of points. For example the command

```
Display/W=(5, 42, 405, 242)
```

specifies the left, top, right, and bottom coordinates of the window in points relative to a reference point which is, roughly speaking, the top/left corner of the menu bar. Other Igor operations that use window position coordinates in points include Edit, Layout, NewNotebook, NewGizmo and MoveWindow.

The exception is the control panel window when running on a standard resolution screen. This is explained under **Control Panel Resolution on Windows** on page III-456.

Most users do not need to worry about the exact meaning of these coordinates. However, for the benefit of programmers, here is a discussion of how Igor interprets them.

On Macintosh, the reference point, (0, 0), is the top/left corner of the menu bar on the main screen. On Windows, the reference point is 20 points above the bottom/left corner of the main Igor menu bar. This difference is designed so that a particular set of coordinates will produce approximately the same effect on both platforms, so that experiments and procedures can be transported from one platform to another.

The coordinates specify the location of the content region, in Macintosh terminology, or the client area, in Windows terminology, of the window. They do not specify the location of the window frame or border.

On Macintosh, a point is always interpreted to be one pixel except on Retina displays where it is two.

On Windows, the correspondence between a point and a pixel can be controlled by the user using system settings. Since Igor stores window positions in units of points, if the user changes the number of pixels per point, the size of Igor windows in pixels will change.

Control Panel Resolution on Windows

In the past, by default, Windows screens ran at 96 DPI (dots-per-inch) resolution. In recent years, high-resolution displays, also called UltraHD displays and 4K displays, have become common.

In Igor6 and before, commands that create control panels and controls use screen pixel units to specify sizes and coordinates. This results in tiny controls on high-resolution displays.

In Igor7 and later, control panels and control sizes and coordinates can be specified in units of points instead of pixels. Since a point is a resolution-independent unit, this results in controls that are the same logical size regardless of the physical resolution of the screen.

By default, panels are drawn using pixels if the screen resolution is 96 DPI but using points for higher-DPI settings. This gives backward compatibility on standard screens and reasonably-sized controls on high-resolution screens.

You can change the way control panel coordinates and control sizes are interpreted using two different methods:

1. Control Panel Expansion (Igor Pro 9.00 and later)

2. You can set control panel expansion using the Expansion submenu in the Panel menu or using the Expansion setting in the Panel section of the Miscellaneous Settings dialog. If you are using Igor Pro 9.00 or later, this method is preferred.

   See the **Control Panel Expansion** on page III-443 for further discussion.

3. SetIgorOption with the PanelResolution keyword.

   This method requires executing commands as described in the next section. It can make panels larger than normal but not smaller. It is not recommended in Igor Pro 9.00 or later.

See also **Control Panel Units** on page III-444.

SetIgorOption PanelResolution

In Igor Pro 9.00 and later we recommend that you use **Control Panel Expansion** instead of SetIgorOption PanelResolution.

You set this option by executing a command of the form:

```
SetIgorOption PanelResolution = <resolution>
```

The <resolution> parameter controls how Igor interprets coordinates and sizes in control panels. It must be one of these values:

0: Coordinates and sizes are treated as points regardless of the screen resolution.

1: Coordinates and sizes are treated as pixels if the screen resolution is 96 DPI, points otherwise. This is the default setting in effect when Igor starts.

72: Coordinates and sizes are treated as pixels regardless of the screen resolution (Igor6 mode).

>96 If <resolution> is greater than 96, panel coordinates and sizes are treated as points regardless of the screen resolution. Larger values make larger panels and controls. For example on a 96 DPI screen `SetIgorOption PanelResolution = 192` makes them twice as large as normal.

This setting is not remembered across Igor sessions.

The resolution for a panel is set when the panel is created so changing the panel resolution does not affect already-existing control panels.

Programmers are encouraged to test their control panels at various resolution settings.

For the most part, Igor6 control panels will work fine in Igor7 and later. However, if you have panels that rearrange their components depending on the window size, you probably have some code that uses the ratio of 72 and ScreenResolution. For example, here is a snippet from the WaveMetrics procedure Axis Slider.ipf:

```
   case "Resize":
      GetWindow kwTopWin,gsize         // Returns points
                                       // Controls are positioned in pixels
      grfName= WinName(0, 1)
      V_left *= ScreenResolution / 72  // Convert points to pixels
      V_right *= ScreenResolution / 72
      Slider WMAxSlSl,size={V_right-V_left-kSliderLMargin,16}
      break
```

To make this work properly at variable resolution, we use the **PanelResolution** function, added in Igor Pro 7.00:

```
   case "Resize":
      GetWindow kwTopWin,gsize         // Returns points
      grfName= WinName(0, 1)
      V_left *= ScreenResolution / PanelResolution(grfName)      // Variable
      V_right *= ScreenResolution / PanelResolution(grfName)     // resolution
      Slider WMAxSlSl,size={V_right-V_left-kSliderLMargin,16}
      break
```

The PanelResolution function, when called with "" as the parameter returns the current global default setting for panel resolution in pixels per inch or, if in Igor6 mode, 72. When called with the name of a control panel or graph, it returns the current resolution of the specified window in pixels per inch.

If you want your procedures to work with Igor6, you should insert this into each procedure file that needs the PanelResolution function:

```
#if Exists("PanelResolution") != 3
Static Function PanelResolution(wName)    // For compatibility with Igor7
   String wName
   return 72
End
#endif
```

To find other examples, use the Help Browser to search procedure files for PanelResolution.

See also **Control Panel Units** on page III-444.

Pictures in Notebooks

If you want to display notebook pictures on both platforms, they must be in one of these cross-platform formats: PDF, PNG, JPEG, TIFF, SVG.

PDF pictures are displayed correctly on Windows in Igor Pro 9.00 or later. In earlier versions on Windows, PDF pictures are displayed as gray boxes.

If you have pictures in other formats and you want them to be viewable on both platforms, you should convert them to PNG.

There are two ways to create a PNG picture in an Igor notebook. You can load it from a file using Misc→Pictures and then place it in a notebook or you can convert a picture that you have pasted into a notebook using Notebook→Special→Convert to PNG.

The Convert to PNG command in the Notebook→Special menu converts the selected picture or pictures into PNG. It skips selected pictures that are already PNG or that are foreign (not native to the platform on which you are running). You can determine the type of a picture in a notebook by clicking in it and looking at the notebook status line.

The Convert to PNG dialog allows you to choose the desired resolution. Usually 4x or 8x is best.

When you create a PNG file from within Igor, from a graph window for example, you can create it at screen resolution or at higher resolution. For good quality when viewed at higher magnifications and when printed, use 4x or 8x.

Chapter

III-16

Text Encodings

Overview

Starting with Igor Pro 7, Igor stores text internally as UTF-8, a Unicode text encoding format. Using Unicode improves Igor's interoperability with other software and gives you easy access to a wide array of characters including Greek, mathematical and special characters.

Previous versions of Igor used non-Unicode text encodings such as MacRoman, Windows-1252, and Shift JIS (Japanese), depending on the operating system you were running. Consequently Igor needs to do text encoding conversions when opening files from earlier versions.

If a file contains accented characters, special symbols, and other non-ASCII text, it is not always possible to get this conversion right. You may see incorrect characters or receive “Unicode conversion errors” or other types of errors.

Text encoding is not an issue with files that contain only ASCII characters as these characters are represented using the same codes in UTF-8 as inn other text encodings.

This chapter provides information that a technically-oriented Igor user can use to understand the conversions and to deal with issues that can arise. To understand these issues, it helps to understand what text encodings are and how they are used in Igor, so we start with an overview.

Text Encoding Overview

A text encoding is a mapping from a set of numbers to a set of characters. The terms "text encoding", "character encoding" and "code page" represent the same thing.

In most commonly-used text encodings, text is stored as an array of bytes. For example, the text "ABC" is stored in memory as three bytes with the numeric values 0x41, 0x42 and 0x43. (The 0x notation means the number is expressed in hexadecimal, also called base 16.)

In the 1960's the ASCII text encoding was defined. It specifies the mapping of 128 numbers, from 0x00 to 0x7F, to characters. ASCII defines mappings for:

- Unaccented letters (A-Z and a-z)
- Numerals (0-9)
- Common punctuation marks (comma, period, dash, question mark ...)
- Other printable characters (space, brackets, slash, backslash, ...)
- Control characters (carriage-return, linefeed, tab, ...)

The "ABC" example above uses the ASCII codes for A, B and C.

Over time, other text encodings were created to permit the representation of characters not supported by ASCII such as:

- Accented western letters
- Asian characters
- Mathematical symbols

Hundreds of text encodings were created for encoding characters used in different languages. Even for a given language, different operating systems often adopted different text encodings.

In ASCII, each character is represented by a 7-bit code which is typically stored in a single 8-bit byte with one bit unused. 7 bits can represent 128 different characters. To represent more characters, other text encodings use 8 bits per character, multiple bytes per character, 16-bit codes or 32-bit codes.

We use the term "byte-oriented text encoding" to distinguish text encodings that represent characters using one byte or multiple bytes per character from those that use 16-bit or 32-bit codes.

The characters represented by the ASCII codes 0x00 to 0x7F are called "ASCII characters" with all other characters called "non-ASCII characters". All byte-oriented text encodings that you are likely to encounter are supersets of ASCII.

Multiple-byte text encodings, such as Shift-JIS and UTF-8, typically use one byte for ASCII characters, and for some frequently-used non-ASCII characters, and two or more bytes for other non-ASCII characters.

In order to properly display and process text, a program must convert any text it reads from outside sources (e.g., files) into whatever text encoding the program uses for internal storage if the source text encoding and the internal text encoding are different.

Since all byte-oriented text encodings that you are likely to encounter are supersets of ASCII, ASCII characters do not require conversion when treated as any byte-oriented text encoding. Non-ASCII characters require conversion and problems arise if the conversion is not done or not done correctly.

Text Encodings Commonly Used in Igor

In Igor the most commonly-used text encodings are:

- MacRoman (Macintosh Western European)
- Windows-1252 (Windows Western European)
- Shift JIS (Japanese)
- UTF-8 (Unicode using a byte-oriented encoding form)

Igor6 and earlier versions of Igor stored text in "system text encoding". For western users this means MacRoman on Macintosh and Windows-1252 on Windows. For Japanese users it means Shift JIS on both platforms.

Igor now uses UTF-8 exclusively.

The system text encoding is determined by the system locale. The system locale determines the text encoding used by non-Unicode programs such as Igor6. It has no effect on Igor7 or later.

On Macintosh you set the system locale through the preferred languages list in the Language & Region preference panel. On Windows you set it through the Region control panel. Most users never have to set the system locale because their systems are initially configured appropriately for them.

Because Igor now uses UTF-8 internally, when it loads a pre-Igor7 file, it must convert the text from system text encoding to UTF-8. If the file contains ASCII text only this conversion is trivial. But if the file contains non-ASCII characters, such as accented letters, Greek letters, math symbols or Japanese characters, issues may arise, as explained in the following sections.

In order to convert text to UTF-8, Igor must know the text encoding of the file from which it is reading the text. This is tricky because plain text files, including computer program source files, data files, and plain text documentation files, usually provide no information that identifies the text encoding they use. As explained below, Igor employs several strategies for determining a file's text encoding so that it can correctly convert the file's text to UTF-8 for internal storage and processing.

Western Text Encodings

MacRoman and Windows-1252 (also called "Windows Latin 1") are used for Western European text and are collectively called "western".

In MacRoman and Windows-1252, the numeric code for each character is stored in a single byte. A byte can represent 256 values, from 0x00 to 0xFF. The 128 values 0x00 through 0x7F represent the same characters as ASCII in both text encodings. The remaining 128 values, from 0x80 to 0xFF, represent different characters in each text encoding. For example the value 0xA5 in MacRoman represents a bullet character while in Windows-1252 it represents a yen symbol.

When Igor6, running on Macintosh, loads an experiment written on Windows, it converts text from Windows-1252 to MacRoman. The opposite occurs if, running on Windows, it loads an experiment written on Macintosh.

When Igor loads an Igor6 experiment written on Macintosh, it converts from MacRoman to UTF-8. If the experiment was written on Windows it converts from Windows-1252 to UTF-8.

Asian Text Encodings

Asian languages have thousands of characters so it is not possible to map one character to one byte as in western text encodings. Consequently various multi-byte text encodings were devised for different Asian languages. Such systems are sometimes called MBCS for "multi-byte character system".

Shift JIS is the predominant MBCS text encoding for Japanese. It uses a single byte for all ASCII characters and some Japanese characters and two bytes for all other Japanese characters.

In Shift JIS, the codes from 0x00 to 0x7F have the same meaning as in ASCII except for 0x5C which represents a backslash in ASCII but a yen symbol in Shift JIS. The codes from 0xA1 to 0xDF represent single-byte katakana characters. The codes from 0x81 to 0x9F and 0xE0 to 0xEF are the first bytes of double-byte characters.

Chinese and Korean text encodings also use MBCS.

Unlike the situation for western text, Macintosh and Windows both use the same, or practically the same, text encodings for Japanese. The same is true for Traditional Chinese and Simplified Chinese.

The MacJapanese and Windows-932 text encodings are variants of Shift JIS. In this documentation, "Shift JIS" and "Japanese" mean "Shift JIS or its variants MacJapanese and Windows-932".

When Igor loads an Igor6 experiment written in Shift JIS it converts text from Shift JIS to UTF-8.

Unicode

The proliferation of text encodings for different languages and even for the same language on different operating systems complicated the exchange and handling of text. This led to the creation of Unicode - a system for representing virtually all characters in all languages with a single text encoding. The first iteration of Unicode was released in the early 1990's. It was based on 16-bit numbers rather than the 8-bit bytes of previous text encodings.

In 1993 Microsoft released Windows NT, one of the first operating systems to use Unicode internally. Mac OS X, developed in the late 1990's and released in 2001, also uses Unicode internally. Most applications at the time still used system text encoding and the operating systems transparently translated between it and Unicode as necessary.

In the mid-1990's the developers of Unicode realized that 16-bits, which can represent 65,536 distinct numbers, were not enough to represent all characters. They extended Unicode so that it can represent over one million characters by using two 16-bit numbers for some characters. However nearly all of the commonly-used characters can still be represented by a single 16-bit number.

Over time most applications migrated to Unicode for internal storage of text. Igor7 was the first version of Igor that uses Unicode internally.

Unicode Character Encoding Schemes

Unicode maps each character to a unique number between 0 and 0x10FFFF (1,114,111 decimal). Each of these numbers is called a "code point".

There are several ways of storing a code point in memory or in a file. Each of them is called a "character encoding scheme". Each character encoding scheme is based on a "code unit" which may be a byte, a 16-bit value or a 32-bit value.

The most straight-forward character encoding scheme is UTF-32. The code unit is a 32-bit value and each code point is directly represented by a single code unit. UTF-32 is not widely used for historical reasons and because it is not memory-efficient.

UTF-16 is a widely-used character encoding scheme in which the code unit is a 16-bit value and each code point is represented by either one or two code units. Nearly all of the commonly-used characters require just one. Both Mac OS X and Windows use UTF-16 internally.

UTF-8 is a widely-used character encoding scheme in which the code unit is a byte and each code point is represented by one, two, three or four code units. The ASCII characters are represented by a single byte using the same code as in ASCII itself. All other characters are represented by two, three or four bytes. UTF-8 is the predominant character encoding scheme on the Worldwide Web and in email. It is the scheme used internally by Igor7 and later.

Problems Caused By Text Encodings

The use of different text encodings on different operating systems, for different languages, and in different versions of Igor requires that Igor convert between text encodings when reading files. In Igor7 and later this means converting from the file's text encoding to UTF-8 which Igor now uses for internal storage. Since ASCII text is the same in UTF-8 as in other byte-oriented text encodings, conversion and the problems it can cause are issues only for files that contain non-ASCII text.

Broadly speaking, when Igor opens an Igor6 file, three types of text encoding-related problems can occur:

- Igor's idea of the file's text encoding might be wrong (text encoding misidentification)
- The file may contain invalid byte sequences for its text encoding (invalid text)
- The file may contain a mixture of text encodings (mixed text encodings)

Text Encoding Misidentification Problems

This problem stems from the fact that it is often unclear what text encoding a given file uses. The clearest example of this is the plain text file - the format used for text data files, Igor procedure files and plain text notebooks. A plain text file usually stores just the text and nothing else. This means that there is often nothing in a plain text file that tells Igor what text encoding the file uses.

The problem is similar for plain text items stored within Igor experiment files. These include:

- wave names, wave units, wave notes, wave dimension labels
- text wave contents
- the experiment recreation procedures stored in an experiment file
- the history area contents
- the built-in procedure window's contents
- packed procedure files
- packed plain text notebook files

Igor Pro 6.30 and later, which we will call Igor6.3x for short, store text encoding information in the experiment file for these items but earlier versions of Igor did not.

The text encoding information stored by Igor6.3x and later is not foolproof. Igor6.3x determines the text encoding for a procedure file or a plain text notebook based on the font used to display it. Usually this is correct but it is not guaranteed as the Igor6 user can change a font at any time.

Waves created pre-Igor6.3x have unknown text encodings for their various items.

Igor6.3x determines the text encoding for wave items when the wave is created based on the text encoding used to enter text into the user interface (e.g., into dialogs). This will be the system text encoding except that in IgorJ (the Japanese version of Igor), it is Japanese regardless of the system text encoding. Usually this produces the correct text encoding but it is not guaranteed as it is possible in Igor6 to make waves using any text encoding. For example, when running IgorJ you can switch the command line to an English font and create a wave whose name contains non-ASCII English characters. This name will therefore use MacRoman on Macintosh and Windows-1252 on Windows but IgorJ63x will record the wave name's text encoding as Shift JIS because that is the user interface text encoding. It is similarly possible to create a Japanese wave name when running an English Igor, causing Igor to record the wave name's text encoding as MacRoman or Windows-1252 even though the wave name is really Japanese.

Igor6.3x sets a wave's text encoding information when the wave is created. If a wave was created prior to Igor6.3x and is opened in Igor6.3x, its text encoding is unknown. Saving the experiment in Igor6.3x does not change this - the wave's text encoding is still unknown. So an Igor6.3x experiment can contain a mix of Igor6.3x waves with known text encodings and pre-Igor6.3x waves with unknown text encodings.

The three main causes of text encoding misidentification are:

- The file is a plain text file which contains no text encoding information
- The file is a pre-Igor6.3x experiment file which contains no text encoding information
- The Igor6.3x assumption that the text encoding can be deduced from the text encoding used to enter text into the user interface was incorrect

In order to convert text to UTF-8, Igor has to know the text encoding used when the text was created. If it gets this wrong then errors or garbled text will result. Some strategies for fixing such problems are described below.

Invalid Text Problems

A file that contains text in a particular text encoding can be damaged, creating sequences of bytes that are illegal in that text encoding.

In Shift JIS, for example, there are rules about how double-byte characters are formed. The first byte must be in a certain range of values and a second byte must be in a different range of values. If, by error, a line break is inserted between the first and second bytes, the resulting text is not valid as Shift JIS.

Damage like this can happen in Igor6 with long Japanese textboxes. When Igor generates a recreation macro for a graph, it generates TextBox commands for textboxes. If the text is long, Igor breaks it up into an initial TextBox command and subsequent AppendText commands. Igor6 is not smart enough to know that it must keep the two bytes of a double-byte Japanese characters together. This problem is asymptomatic in Igor6 because the two bytes of the double-byte character are reassembled when the macro runs. But it causes an error if you load the Igor6 file into Igor7 or later because Igor must convert the entire file to UTF-8 for internal storage before any reassembly occurs.

There are many other ways to damage a Shift JIS file. Similar damage can occur with other multi-byte text encodings such as UTF-8.

When a file contains invalid text but you suspect that it is mostly OK, you can use the Choose Text Encoding dialog to change the text encoding conversion error handling mode. This allows you to open the file despite the invalid text.

Mixed Text Encoding Problems

In Igor6, you can enter text in different text encodings by merely using different fonts. For example:

1. Run Igor6.3x with an English font controlling the built-in procedure window.
2. Create a graph.
3. Create an annotation using a Japanese font and Japanese characters in the graph.
4. Save the graph as a recreation macro.
5. Save the experiment.

Because an English font controls the built-in procedure window, Igor6.3x thinks that the text encoding for the procedure window is western (MacRoman or Windows-1252) and records this information in the experiment.

When Igor7 or later opens this experiment, it must convert the procedure window text to UTF-8. Since the Igor6.3x recorded the file's text encoding as western (MacRoman or Windows-1252), Igor attempts to convert the text from western to UTF-8. You will wind up with gibberish western characters where the Japanese text should be.

In this example Igor mistook Japanese text for western and the result was gibberish. If Igor mistakes western text for Japanese, you can either wind up with gibberish, or you may get an error, or you may see the Unicode replacement character (a white question mark on a black background) where the misidentified text was.

Fortunately this kind of problem is relatively rare. To fix it you need to manually repair the gibberish text.

The Experiment File Text Encoding

Experiments saved by Igor6.3x or later record the "experiment file text encoding". This is the text encoding used by the built-in procedure window at the time the experiment was saved. As explained below, Igor uses the experiment file text encoding as the source text encoding when converting a wave element to UTF-8 if the element's text encoding is unknown.

Prior to version 6.30 Igor did not record the experiment file text encoding so it defaults to unknown when a pre-Igor6.3x experiment is loaded into Igor7 or later. It also defaults to unknown when you first launch Igor and when you create a new experiment.

You can display the experiment file text encoding for the currently open experiment by right-clicking the status bar and choosing Show Experiment Info. Igor will display something like "Unknown / 6.22" or "Mac Roman / 6.36". The first item is the experiment file text encoding while the second is the version of Igor that saved the experiment file. This information is usually of interest only to experts investigating text encoding issues.

You can also see the experiment file text encoding by choosing File→Experiment Info.

The Default Text Encoding

Because it is not always possible for Igor to correctly determine a file's text encoding, it sometimes needs additional information. You supply this information through the Default Text Encoding submenu in the Text Encoding submenu in the Misc menu. Igor uses your selected default text encoding when it can not otherwise determine the text encoding of a file or wave.

You can display the default text encoding for the currently open experiment by right-clicking the status bar and choosing Show Default Text Encoding. Displaying this information in the status bar is usually of interest only to experts investigating text encoding issues.

Igor experiment files record the platform (Macintosh or Windows) on which they were last saved. In Igor6.3x and later, Igor records text encoding information for most of the items stored in an experiment file. Neither platform information nor text encoding information is stored in standalone plain text files. Consequently determining the text encoding of an experiment file is different from determining the text encoding of a standalone file. This makes Igor's handling of text encodings and the meaning of the default text encoding setting more complicated than we would like.

The Default Text Encoding submenu contains the following items:

```
UTF-8
Western
MacRoman
Windows-1252
Japanese
Simplified Chinese
```

```
Traditional Chinese
<Other text encodings>
Override Experiment
```

When you open an experiment, Western means MacRoman if the experiment was last saved on Macintosh and Windows-1252 if it was last saved on Windows. When you open a standalone file, Western means MacRoman if you are running on Macintosh and Windows-1252 if you are running on Windows.

The remaining text encoding items, such as MacRoman, Windows-1252 and Japanese, specify the text encoding to use for conversions to UTF-8 without regard to the platform on which the file was last saved or on which you are currently running.

When opening an experiment saved in Igor6.3x or later, Igor normally uses the text encoding information in the experiment file. As explained above under **Text Encoding Misidentification Problems** on page III-463, occasionally this information will be wrong. The Override Experiment item gives you a way to force Igor to use your selected text encoding even when opening experiments that contain text encoding information.

When Igor first runs it sets the default text encoding to Western or Japanese based on your system locale. This will give the right behavior when opening Igor6 experiments in most cases. Igor stores the default text encoding in preferences so when your launch Igor again it is restored.

For normal operation, leave the default text encoding setting on Western if you use a Western language or on the appropriate Asian language if you use an Asian language and leave Override Experiment unchecked.

Change these only if a problem arises. For example, if you are running an English system but you are opening a file containing Japanese, or if you are running a Japanese system but you are opening a file containing non-ASCII western text, you may get incorrect text or errors. To fix this, change the settings in this menu and reload the file. After loading the problematic file, reset the menu items for normal operation.

Igor sets the Override Experiment setting to unchecked each time Igor is launched since this is less likely to result in undetected erroneous text conversion than if you turned it on and unintentionally left it turned on.

In addition to affecting text encoding conversion during the loading of files, the default text encoding setting is used as follows:

- When you choose New Experiment, it determines the text encoding of the history area and built-in procedure window. This determines the text encoding used for these items when the experiment is saved to disk.
- When you create a new procedure or notebook window, if a text encoding was not explicitly specified, it determines the text encoding used when the file is saved to disk.
- When you insert text into an existing document using Edit→Insert File, it determines the source text encoding for conversion to UTF-8.
- When Igor loads a wave, it determines the source text encoding for conversion to UTF-8 if the source text encoding is not otherwise specified. This will be the case for waves created before Igor6.3x.
- When the help browser searches procedure files, notebook files and help files, it determines the source text encoding for conversion to UTF-8 if it is not otherwise known.

Plain Text File Text Encodings

Plain text files include procedure files, plain text notebook files, the built-in procedure window text and history text. Igor uses UTF-8 when writing new instances of these items.

When reading existing files, Igor must convert text that it reads from plain text files to UTF-8 for storage in memory. In order to do this, it needs to know what text encoding to convert from. Most plain text files contain no indication of their text encoding so it can be tricky to determine the source text encoding. If Igor gets the source text encoding wrong, you will get errors or see incorrect characters in the text.

Starting with Igor6.3x, Igor stores text encoding information for plain text files stored in or accessed by experiment files. Experiment files saved before Igor6.3x do not contain text encoding information. Standalone plain text files not part of an experiment file, such as Igor6 global procedure files, plain text notebook files and data files, also do not contain text encoding information.

Igor looks for the optional byte order mark (see **Byte Order Marks** on page III-471) that some programs write at the start of a Unicode plain text file. If present, the byte order mark unambiguously identifies the file as Unicode and identifies which encoding scheme (UTF-8, UTF-16 or UTF-32) the file uses.

Starting with Igor7, Igor supports a TextEncoding pragma in procedure files that looks like this:

```
#pragma TextEncoding = "<text encoding name>"
```

This is described under **The TextEncoding Pragma** on page IV-55. The TextEncoding pragma allows you to unambiguously specify the text encoding for an Igor procedure file. It is ignored by Igor6.

Starting with Igor9, the text encoding for procedure files defaults to UTF-8.

Igor must deal with a variety of circumstances including:

- Standalone files versus experiment files
- Pre-Igor6.3x experiment files versus experiment files from Igor6.3x and later
- Macintosh versus Windows versus Japanese experiment files
- Procedure files with and without a TextEncoding pragma

Because of these complications there is no simple way for Igor to determine the source text encoding for any given plain text file. To cope with this situation, it employs rules described in the next section.

The status bar of a plain text file's window includes a text encoding button which shows you the "file text encoding" - the text encoding that Igor used to convert the file's text to UTF-8. When the file is saved, Igor converts from UTF-8 back to the file text encoding. You can change the file text encoding by clicking the text encoding button to display the Text Encoding dialog.

Determining the Text Encoding for a Plain Text File

This section describes the rules that Igor uses to determine the source text encoding for a plain text file.

The rules depend on whether Override Experiment (Misc→Default Text Encoding→Override Experiment) is checked.

Override Experiment Unchecked

If Override Experiment is unchecked, Igor tries the following text encodings, one-after-another, until it succeeds (i.e., it is able to convert to UTF-8 without error), in this order:

```
Byte order mark
TextEncoding pragma if present (for procedure files only)
The specified text encoding (described below)
UTF-8 if the text contains non-ASCII characters
The default text encoding
Choose Text Encoding dialog (described below)
```

Note that a conversion can "succeed" without being correct. For example, Japanese can be successfully interpreted as MacRoman or Windows but you will get the wrong characters.

The "specified text encoding" is the explicit text encoding if present (e.g., specified by the /ENCG flag for the OpenNotebook operation) or the item text encoding (e.g., a text encoding stored in an experiment file for its built-in procedure window text).

Starting with Igor Pro 9, during experiment loading, if text is valid as UTF-8, including all ASCII text, it is loaded as UTF-8 regardless of the text encoding stored in the experiment file. The purpose is to transition away from obsolete text encodings.

Also starting with Igor Pro 9, if a plain text file has a UTF-8 byte order mark it is loaded as UTF-8 even if it contains invalid byte sequences. Invalid bytes are displayed using the Unicode replacement character (U+FFFD).

If Igor can not otherwise find a valid text source encoding it displays the Choose Text Encoding dialog. This dialog asks you to choose the text encoding to use for the plain text file being opened.

Override Experiment Checked

If the Misc→Default Text Encoding→Override Experiment menu item is checked then the order changes to this:

```
Byte order mark
TextEncoding pragma if present (for procedure files only)
The default text encoding
The specified text encoding (described below)
UTF-8 if the text contains non-ASCII characters
Choose Text Encoding dialog (described below)
```

In this mode the default text encoding, as set in the Misc→Default Text Encoding submenu, is given a higher priority. This should be used only if the normal mode fails to give the correct results. If you turn Override Experiment on, make sure to select the appropriate default text encoding from the submenu. You should turn Override Experiment on only in emergencies and turn it off for normal use.

Starting with Igor Pro 9, during experiment loading, if text is valid as UTF-8, including all ASCII text, it is loaded as UTF-8 regardless of the Override Experiment setting. The purpose is to transition away from obsolete text encodings.

Potential Problems Determining Plain Text File Text Encodings

The rules listed above have the following potential problems:

TextEncoding pragma (for procedure files only)

The pragma could be wrong. For example, if you set TextEncoding="MacRoman" and then change the file to UTF-8 in an external editor, it will still succeed in Igor but give the wrong characters. This is a programmer error and it is up to you to fix it.

The specified text encoding

There is no specified text encoding for plain text files in a pre-Igor6.3x experiment.

In Igor6.3x and later, Igor includes text encoding information in the experiment restart procedures for each plain text file to be opened. This information is the "specified text encoding".

In Igor6.3x the specified text encoding can be wrong. For example, if the user adds some Japanese characters to a graph annotation, the recreation macro will include Shift-JIS. If the font controlling the procedure window is MacRoman or Windows, the specified text encoding will be MacRoman or Windows and the Japanese characters will show up wrong in Igor7 or later.

UTF-8 if the text contains non-ASCII characters

MacRoman, Windows and ShiftJIS can all masquerade as UTF-8 though this will be rare. By “masquerade” we mean that the non-ASCII bytes in the text are legal as UTF-8 but the text really is encoded using another text encoding.

For a procedure file you can fix this by adding a TextEncoding pragma.

There is currently no fix for plain text notebooks other than setting "Override Experiment" and setting the default text encoding to the correct encoding and reloading or converting the file to UTF-8 with a byte order mark.

The default text encoding

If set to MacRoman or Windows, the default text encoding will always succeed but it might be wrong. For example, Japanese can be successfully interpreted as MacRoman or Windows but you will get the wrong characters.

For a procedure file you can fix this by adding a TextEncoding pragma.

There is currently no fix for plain text notebooks other than setting "Override Experiment" and setting the default text encoding to the correct encoding and reloading or converting the file to UTF-8.

Plain Text Text Encoding Conversion Errors

When Igor opens a plain-text file or an Igor6-compatible experiment file, it must convert plain text to UTF-8. In order to do this, it needs to determine the text encoding of the plain text file or the plain text item within an experiment file.

As the previous section briefly outlined, it is possible that Igor might get this wrong. To illustrate the types of problems that can occur in this situation, we will consider the following scenario:

> In Igor6.x you have created a standalone (not part of an experiment) plain text file on Windows with English as the system locale so the text in the file is encoded as Windows-1252. The file contains some non-ASCII characters. You transfer the file to Macintosh. Your default text encoding, as selected in the Default Text Encoding submenu of the Misc menu, is Western. You open the plain text file as a notebook in Igor.

Since the file is is a plain text file, it contains nothing but plain text; it does not contain information indicating the platform of origin or the text encoding. Consequently Igor uses your selected default text encoding - Western. In the case of a plain text file, Western means "MacRoman when running on Macintosh, Windows-1252 when running on Windows", so Igor uses MacRoman to convert the text to UTF-8. This does not cause an error but gives an incorrect translation of the non-ASCII characters in the file.

Unfortunately, Igor has no way to detect this kind of misidentification error. The conversion completes without error but produces some incorrect characters and there is no way for Igor to know that this has happened. This is an example of a conversion that succeeds but is incorrect.

To get the correct characters you must close the file, choose Windows-1252 from the Default Text Encoding menu, and reopen the file. Now Igor will correctly translate the text to UTF-8. You should restore the default text encoding to its normal value for future use.

An alternative solution is to execute OpenNotebook/ENCG=3. This explicitly tells Igor that the file's text encoding is Windows-1252. Igor's text encoding codes are listed under **Text Encoding Names and Codes** on page III-490.

Now let's change the scenario slightly:

> Instead of your default text encoding being set to Windows-1252, it is set to Japanese. You open the standalone plain text file containing Windows western non-ASCII characters as a notebook on either Macintosh or Windows.

Since the file does not contain any text encoding information, Igor tries to convert the text to UTF-8 using default text encoding, Japanese, as the original text encoding. Two things may happen:

- The conversion may succeed. In this case you will see some seemingly random Japanese characters in your notebook because, although the conversion succeeded, it incorrectly interpreted the text.
- The conversion may fail because the file contains byte patterns that are illegal in the Shift JIS text encoding. In this case Igor will display the Choose Text Encoding dialog and you can choose the

correct text encoding.

The best solution for text encoding issues is to convert the text to valid UTF-8. Once you have successfully opened the file in Igor, you can convert it to UTF-8 by clicking the text encoding button in the status bar. You can convert all open non-UTF-8 text files to UTF-8 using the Convert to UTF-8 dialog; see **Converting to UTF-8** on page III-485 for details.

UTF-8 Procedure Files Containing Invalid UTF-8

Though rare, it is possible to create a UTF-8 procedure file containing invalid UTF-8. This occurs if you use invalid UTF-8 in a window and save the window as a recreation macro. For example:

```
Display/TestGraph
Textbox "\xC5"              // Invalid UTF-8 text in annotation
DoWindow/R TestGraph        // Save graph as recreation macro
```

This creates a TestGraph window macro that recreates the annotation which contains invalid UTF-8. This is a contrived example, but the situation occurs in real life if, for example, you obtain the text for the annotation from a source (e.g., database) that provides non-ASCII text in a text encoding other than UTF-8 and you fail to convert it to UTF-8.

If you now save the experiment to disk, you will have invalid UTF-8 procedure text in the experiment's built-in procedure window and also in the experiment's recreation macros that Igor executes to recreate the graph window when you reopen the experiment.

Prior to Igor Pro 9.00, you would get confusing errors on reopening the experiment. In Igor Pro 9.00 and later, if a procedure file contains a UTF-8 byte order mark or UTF-8 TextEncoding pragma, the file is loaded as UTF-8 even if it contains byte sequences that are not valid UTF-8. The invalid bytes are preserved in the text and are displayed as Unicode replacement characters (U+FFFD). This allows Igor to recreate the experiment as it was, invalid UTF-8 and all.

Some Plain Text Files Must Be Saved as UTF-8

When Igor saves a plain text file, it tries to save using the file's original text encoding. Sometimes this is not possible. For example:

You open a file that contains MacRoman text including non-ASCII characters as an Igor procedure file and Igor converts it to UTF-8 for internal storage. Now you add a Japanese character to the procedure file and save it.

The file can not be saved as MacRoman because MacRoman can not represent Japanese characters. In this event, Igor saves the file in UTF-8 and displays an alert informing you of that fact. If you open this file in Igor6, any non-ASCII characters will be incorrect.

If you tell Igor to save a plain text file in a specific text encoding, for example via `SaveNotebook/ENCG=<textEncoding>`, and if it is not possible to save in that text encoding, Igor returns a text encoding conversion error. You can then explicitly tell Igor to save as UTF-8 using `SaveNotebook/ENCG=1`.

History Text Encoding

When an experiment is saved, Igor saves the history text as plain text.

When you create a new experiment, Igor sets the history area text encoding to your chosen default text encoding. When you save the experiment, Igor tries to save the history text using that text encoding. If the history text contains characters that can not be represented in your default text encoding, Igor then saves the history text as UTF-8 and sets the history area text encoding to UTF-8. If you open this experiment in Igor6, any non-ASCII characters in the history, including the bullet characters used to mark commands, will appear as garbage.

The Shift JIS (Japanese) text encoding does not have a character that maps to the bullet character (U+2022). Consequently, if your default text encoding is Shift JIS, when you save an experiment, Igor will be unable to save the history text as Shift JIS and will save it as UTF-8 instead.

Byte Order Marks

A byte order mark, called a BOM for short, is a special character sometimes used in Unicode plain text files to identify the file as Unicode and to identify the Unicode encoding form used by the file. The BOM, if present, must appear at the beginning of the file.

A FAQ about BOMs can be found at http://www.unicode.org/faq/utf_bom.html#BOM.

Most Unicode plain text files omit the BOM. However, including it has the advantage of unambiguously identifying the file's text encoding.

The BOM is the Unicode "zero width no-break space" character. It's code point is U+FEFF which is represented by the following bytes:

UTF-8	0xEF 0xBB 0xBF
UTF-16 Little Endian	0xFF 0xFE
UTF-16 Big Endian	0xFE 0xFF
UTF-32 Little Endian	0x00 0x00 0xFF 0xFE
UTF-32 Big Endian	0xFE 0xFF 0x00 0x00

If a Unicode file contains a BOM, Igor preserves it when saving the file.

By default Igor writes a BOM when writing a Unicode plain text file and removes the BOM, if present, when reading a plain text file into memory.

When Igor creates a new plain text file, it sets an internal writeBOM flag for that file to true. If the file is later saved to disk as Unicode and if the writeBOM flag is still set, Igor writes the BOM to the file.

When Igor opens a plain text file, it checks to see if the file contains a BOM. If not it clears the file's writeBOM flag. If the file does contain a BOM, Igor sets the writeBOM flag for the file and removes the BOM from the text as loaded into memory. When Igor writes the text back disk, if the writeBOM flag is set, Igor writes the BOM and then the text.

You can see the state of the writeBOM flag using the File Information dialog which you access via the Notebook or Procedure menu. If the file's text encoding is a form of Unicode and the writeBOM flag is set, the Text Encoding section of the File Information dialog will say "with byte order mark".

You can specify the value of the writeBOM flag for a plain text notebook file using **NewNotebook** with the /ENCG flag.

You can set or clear the writeBOM flag for a plain text notebook or procedure file using the Text Encoding dialog accessed via the Notebook or Procedure menu. The Write Byte Order Mark checkbox is visible only if a Unicode text encoding is selected. You can also set the writeBOM flag for a plain text notebook using the **Notebook** operation, writeBOM keyword.

The built-in procedure window is a special case. Its writeBOM flag defaults to false and it is set to false each time you do New Experiment. We make this exception to allow Igor6 to open an experiment whose procedure window text encoding is UTF-8 without generating an error. Igor6 does not know about UTF-8 so non-ASCII characters will be wrong, but at least you will be able to open the experiment.

If you modify a plain text file that is open as a notebook or procedure file using an external editor, Igor reloads the text from the modified file. This sets the internal writeBOM flag for the notebook or procedure file to true if the modified text includes a BOM or false otherwise. This then determines whether Igor writes the BOM if you later save the file. This process of reloading an externally-modified plain text file overwrites the writeBOM flag that you may have set using the Text Encoding dialog or Notebook operation.

BOMs are irrelevant for formatted text notebooks.

The **FReadLine** operation looks for and skips UTF-8 byte order marks and UTF-16 byte order marks if you specify UTF-16 using the /ENCG flag.

Formatted Text Notebook File Text Encodings

Formatted notebook files use an Igor-specific file format. You can open such a file as a formatted notebook or as an Igor help file.

Unlike plain text files, in which all text must be encoded using a single text encoding, Igor6 formatted notebook files can use multiple text encodings in a single file. In Igor6, the text encoding for a given format run is determined by the font controlling that run. For example, a single Igor6 formatted text file can contain western text and Shift JIS (Japanese) text encodings. In fact a single paragraph can contain multiple text encodings.

We use the term MBCS (multi-byte character system) to distinguish these Igor6 formatted text files from formatted text files saved using UTF-8 by Igor7 or later.

Igor now stores text internally (in memory) as UTF-8. Consequently it must convert MBCS text into UTF-8 when opening Igor6 formatted files. When doing this conversion, Igor uses text encoding information written to the file by IP6.3x and later. Pre-IP6.3x formatted text notebook files lack this text encoding information and so may be misinterpreted. If you open a formatted text notebook and you get incorrect characters or text encoding conversion errors, try opening and resaving the file in Igor Pro 6.3x, and then reopen it.

In Igor Pro 8.00 or later, Igor writes new formatted notebooks and formatted notebooks previously written using UTF-8 as UTF-8. It writes in Igor6-compatible MBCS format only when saving a file that was previously written as MBCS and only if all of the file’s characters can be represented in MBCS. If you open a UTF-8 formatted notebook file in Igor6, any non-ASCII characters will appear garbled.

To determine how a particular file is saved, click the page icon in the lower-lefthand corner of a formatted notebook or help window. The resulting File Information dialog will say either "MBCS" or "UTF-8" in the Text Encoding area.

If you open a UTF-8-formatted text notebook in Igor Pro 6.36 or before and then save the notebook, it is marked as an Igor6 notebook using MBCS even though it really contains UTF-8 text. If you subsequently open the notebook in Igor7 or later, all non-ASCII characters will be incorrectly interpreted as MBCS and will be wrong.

If you open a UTF-8-formatted text notebook in Igor Pro 6.37 or a later 6.x version and then save the notebook, it is marked as using UTF-8 text encoding. Igor6 will still display the wrong characters for non-ASCII text, but if you subsequently open the notebook in Igor7 or later, the non-ASCII characters will be correctly interpreted as UTF-8.

Wave Text Encodings

Each wave internally stores several plain text elements. They are:

- wave name
- wave units
- wave note
- wave dimension labels
- text wave content (applies to text waves only)

Each wave stores five settings corresponding to these five plain text elements. The settings record the text encoding used for the corresponding element.

You can inspect the text encodings of the wave elements of a particular wave using the Data Browser. To do so, you must first enable the Text Encoding checkbox in the Info pane of the Data Browser section of the Miscellaneous Settings dialog.

For waves created prior to Igor6.3x, all of the settings are set to 0 which means "unknown". This is because the notion of wave text encodings was added only in Igor Pro 6.30, in anticipation of Igor7 and its use of Unicode as the primary text storage format.

When you make a wave in Igor6.3x the text encoding for each element is set to the system text encoding (typically MacRoman, Windows-1252 or Japanese) except for the text wave content element which is set to "unknown". The reason for treating the text wave content element differntly is because the data may be binary data as explained below under **Text Waves Containing Binary Data** on page III-475.

When you make a wave in Igor7 or later, including when you overwrite an existing wave, the text encoding for each element, including the text wave content element, is set to UTF-8.

These defaults are set when you make or overwrite a wave using Make, Duplicate, LoadWave and any other operation that creates a wave. After creating the wave, you can change them explicitly using **SetWaveTextEncoding**, but only advanced users should do this.

If you overwrite an existing text wave, the text encoding for each element, including the text wave content element, is set to UTF-8, the same as when you first create the wave. However, the wave's text content is not cleared or converted to UTF-8 because it is presumed that you will be completely rewriting the content and, if Igor cleared or converted it on overwriting, it would be a waste of time. If the wave contains non-ASCII text and its previous text content text encoding was not UTF-8, this leaves the wave with invalid text content. This is normally not a problem since you will completely rewrite the text anyway. But if you want to make sure that there is no invalid text, you can clear the wave's text content in the Make/T/O statement. For example:

```
Make/T/O textWave = ""
```

In addition, the text encoding setting for the wave name is set if you rename the wave and the text encoding for the wave note is set if you change the wave note text. However the text encoding for the wave units is not set if you set the units using SetScale, the text encoding for the dimension labels is not set if you set a dimension label using SetDimLabel, and the text encoding for the text content of a text wave is not set if you store text in an element of the wave.

Opening an experiment does not change the text encoding settings for the waves in the experiment. Neither does saving an experiment.

When Igor uses one of these wave elements, it must convert the text from its original text encoding into UTF-8. It uses the rules listed in the next section to determine the original text encoding.

Determining the Text Encoding for Wave Elements

This section describes the rules that Igor uses to determine the source text encoding for a wave element when loading a wave from an Igor binary wave file or from a packed experiment file. You don't need to know these rules. We present them for technically-oriented users who are interested or who need to understand the details to facilitate troubleshooting.

1. If the element's text encoding is set to UTF-8, Igor uses UTF-8 as the source text encoding.

   Igor saves text as UTF-8 if it can not be represented in another text encoding. This rule ensures that Igor correctly loads such text.

2. Otherwise if Override Experiment is turned on, Igor uses the selected default text encoding as the source text encoding.

This provides a way for you to override incorrect text encoding settings when loading an experiment. For example, if you know that an experiment uses Japanese but you get gibberish or Unicode conversion errors when loading it, you can choose Japanese as your default text encoding, select Override Experiment, and reload the experiment. You should restore the default text encoding to its normal value and turn Override Experiment off for future use.

3. Otherwise if the text encoding for the specific element being converted is known, because it was previously set to a value other than "unknown", Igor uses it as the source text encoding.

   This is the effective rule for waves created by Igor6.3x and later.

   This is also the effective rule if you explicitly set a wave's text encoding settings using the **SetWaveTextEncoding** operation.

4. Otherwise if the experiment file text encoding is known, Igor uses it as the source text encoding.

   This rule takes effect for waves created in earlier versions if the experiment was later saved from Igor6.3x or later. Wave text encoding settings would still be unknown but the experiment file text encoding would reflect the text encoding of the experiment's procedure window when it was saved. This will give correct results for most experiments.

5. Otherwise Igor uses the selected default text encoding as the source text encoding.

   This rule takes effect for experiments saved before Igor6.3x in which both the experiment file text encoding and the element-specific text encodings are unknown. This is the effective rule for experiments of this vintage.

Wave Text Encoding Problems

This section describes the problems that may occur when loading a wave from an Igor binary wave file or from a packed experiment file.

If Igor is not able to correctly determine a wave element's text encoding and if the element contains non-ASCII text you may see gibberish text or Igor may generate a text encoding conversion error.

If Igor is not able to correctly determine the text encoding of a wave name you will see gibberish for the name and Igor will be unable to find the wave when it is referenced from procedures such as experiment recreation procedures. This may cause errors when the experiment is loaded.

Igor’s handling of invalid text in tables is described under **Editing Invalid Text** on page II-259.

The **SetWaveTextEncoding** operation allows you to set the text encodings for one or more waves or for all waves in the experiment. This is especially useful for pre-IP6.3x experiments. You can run it in Igor6.3x or later. It is discussed below under **Manually Setting Wave Text Encodings** on page III-477.

If you get text encoding conversion errors when opening an experiment you can try the following:

- Choose a different text encoding from the Default Text Encoding menu and reopen the experiment
- Select Override Experiment in the Default Text Encoding menu so that it is checked and reopen the experiment
- Manually fix the errors after opening the experiment
- Use the **SetWaveTextEncoding** operation to specify the text encodings used by waves
- For pre-IP63x experiments, open the experiment in Igor6.3x, resave it, and reopen it

After attempting to fix a text encoding problem you should restore your default text encoding to the most reasonable setting for your situation and turn Override Experiment off.

If you are unable to solve the problem you can send the experiment to WaveMetrics support with the following information:

- What operating system you are running

- What Igor version you are running
- What incorrect output you are seeing and what you expect to see

LoadWave Text Encodings for Igor Binary Wave Files

The rules that LoadWave uses to determine the source text encoding of an Igor binary wave file are the same as the rules described under **Determining the Text Encoding for Wave Elements** on page III-473. The LoadWave /ENCG flag is ignored when loading an Igor binary wave file.

LoadWave Text Encodings for Plain Text Files

This section describes the rules that the LoadWave operation uses to determine the source text encoding when loading a plain text file. This includes general text, delimited text and Igor text files but not Igor binary wave files.

The rules that LoadWave uses to determine the source text encoding of a plain text file are the same as the rules described under **Determining the Text Encoding for a Plain Text File** on page III-467. The "specified text encoding", which is one of the factors used by the rules, is unknown unless you use the /ENCG flag to tell LoadWave the file's text encoding.

For further details see **LoadWave Text Encoding Issues** on page II-149.

Text Waves Containing Binary Data

You can set a text wave's content text encoding to the special value 255 using **SetWaveTextEncoding**. This marks a text wave as really containing binary data, not text.

Text is data that is represented by numeric codes which map to characters using some recognized text encoding such as MacRoman, Windows-1252, Shift JIS or UTF-8. By contrast, binary data does not follow any recognized text encoding and generally does not map to characters. Whereas text data is typically intended to be readable by humans using a text editor, binary data is intended to be processed only by programs.

You can mark any text wave element as binary but it is normally done only for text wave content because it is rare to store binary data in wave units, wave notes or wave dimension labels.

Igor text waves were designed to store text but are able to store any collection of bytes.

While a numeric wave stores a fixed number of bytes in each element, a text wave has the ability to store a different number of bytes in each point. For example:

```
Make /O /T /N=2 twave0
twave0[0] = "Hello"                  // 5 bytes
twave0[1] = "Goodbye"                // 7 bytes
```

This capability makes a text wave a convenient way to store a collection of binary arrays of different length. For example, you could store the contents of a different downloaded file in each element of a text wave:

```
Make /O /T /N=2 twave0
twave0[0] = FetchURL("http://www.wavemetrics.net/images/tbg.gif")
twave0[1] = FetchURL("http://www.wavemetrics.net/images/mbg.gif")
```

While most text waves contain text data, clever programmers sometimes use text waves to store binary data. Since binary data does not follow any recognized text encoding, it must not be treated as text. For example, you can convert a text wave from Shift JIS to UTF-8 and preserve the meaning of the text; you are merely changing how the characters are encoded. By contrast, if you attempt to convert binary data which you have mistaken for text, you turn your data into garbage.

The **SetWaveTextEncoding** operation can convert text waves from one text encoding to another but you never want to convert binary text waves. SetWaveTextEncoding conveniently skips text waves marked as binary when doing a conversion but it is up to you to mark such waves appropriately. Here is an example that illustrates the utility of marking a text wave as containing binary data:

```
// Mark contents as binary
SetWaveTextEncoding 255, 16, root:MyBinaryTextWave
// Convert contents to UTF-8 skipping binary waves
SetWaveTextEncoding /DF={root:,1} /CONV=1 1, 16
```

The first command marks a particular wave's content (16) as containing binary (255) as opposed to text. The second command converts all text wave text to UTF-8 (1) except that it automatically skips waves marked as binary.

This is fine but it would be tedious to identify and mark each text wave containing binary data. So SetWaveTextEncoding provides a way to do it automatically:

```
// Mark all binary waves as binary
SetWaveTextEncoding /BINA=1 /DF={root:,1} 255, 16
// Convert to UTF-8 skipping binary waves
SetWaveTextEncoding /DF={root:,1} /CONV=1 1, 16
```

The first command goes through each wave in the experiment. If it is a text wave, it examines its contents. If the wave contains control characters (codes less than 0x20) other than carriage-return (0x0D), linefeed (0x0A) and tab (0x09), it marks the wave as containing binary. Then the second command converts all non-binary text waves to UTF-8. These commands can be combined into one:

```
// Mark as binary then convert non-binary to UTF-8
SetWaveTextEncoding /BINA=1 /DF={root:,1} /CONV=1 1, 16
```

NOTE: The heuristic used by the /BINA flag is not foolproof. While most binary data will contain 0x00 bytes or other bytes that flag it as binary, it is possible to have binary data that contains no such bytes. In that case, the first command would fail to mark the text waves as containing binary and the second command would either return an error or clobber the binary data. For this reason, it is imperative that you back up any data before doing text encoding conversions.

PROGRAMMER'S NOTE: If you are a programmer and you create text waves to store binary data, be sure to mark them as binary. For example:

```
Wave/T bw = <path to some text wave used to contain binary data>
SetWaveTextEncoding 255, 16, bw        // Mark text wave as binary
```

Doing this will ensure that your binary data is not "converted" to some text encoding, thereby clobbering it.

Handling Text Waves Containing Binary Data

Usually you use a text wave containing binary data only as a container to be accessed by procedures that understand how to interpret the contents. You normally should not treat it as text. For example, you should not print its contents to the history because this treat the wave's binary contents as text. If you do treat it as text you are likely to get gibberish or a Unicode conversion error.

You can view a text wave marked as binary in a table and you can cut and paste its cells but you can not edit the binary data in the table’s entry line. You can dump the contents of a cell as hex to the history area by pressing Cmd (*Macintosh*) or Ctrl (*Windows*) and double-clicking the cell.

Igor normally does automatic text encoding conversion when fetching data from a text wave or storing data into a text wave. For example:

```
// Make a wave to contain text in UTF-8 text encoding
Make /O /T UTF8 = {"<some Japanese text>"}      // UTF-8 by default

// Make a wave to contain text in Shift JIS text encoding
Make /O /T /N=1 ShiftJIS                        // UTF-8 by default
SetWaveTextEncoding 4, 16, ShiftJIS             // Now Shift JIS
```

```
ShiftJIS = UTF8
```

The last statement triggers a fetch from the wave UTF8 and a store into the wave ShiftJIS. A fetch converts a text wave's data to UTF-8. In this case it is already UTF-8 so no conversion is done. A store converts the data to the text encoding of the destination wave - Shift JIS in this case. You wind up with the same characters in ShiftJIS and in UTF8 but not the same raw bytes because they are governed by different text encodings.

Igor does not do text encoding conversion when fetching from a wave or storing into a wave if the wave is marked as binary. For example:

```
// Make a wave to contain binary data
Make /O /T /N=1 Binary                          // UTF-8 by default
SetWaveTextEncoding 255, 16, Binary             // Now binary
Binary = UTF8
```

The last statement's storing action does not do text encoding conversion because the destination wave is marked as binary. Thus Binary will wind up with the same bytes as in UTF8.

Now consider this:

```
Binary = ShiftJIS
```

Once again the last statement's storing action does not do text encoding conversion because the destination wave is marked as binary. You might think that Binary would wind up with the same bytes as in ShiftJIS but you would be wrong. The reason is that the fecthing action triggered by evaluating the righthand side of the assignment statement converts the Shift JIS text in ShiftJIS to Igor's internal standard, UTF-8. Consequently the same bytes wind up being stored in Binary after each of the two previous statements.

The moral of the story is that, if you want to transfer binary data between text waves, make sure that both waves are marked as binary. It will also work if the source wave is marked as UTF-8 since the fetch conversion does nothing if the source is UTF-8.

Manually Setting Wave Text Encodings

NOTE: Back up your experiment before fiddling with text encoding settings.

With the concepts of wave plain text elements, their text encoding settings, and text waves containing binary data in mind, we can now consider a general approach for setting wave text encodings. This is a task for advanced users only.

As a test case, we will assume that you created an experiment using Igor Pro 6.22 on Windows with English as the system locale. Consequently the experiment contains waves with all text encoding settings set to unknown. You then opened the experiment in Igor Pro 6.30, again on Windows with English as the system locale, and created additional waves. These added waves have text encoding settings set to Windows-1252 except for the text wave content element which is set to unknown. Finally, somewhere along the line, some text waves were created to store binary data, possibly by a package that you use and without your knowledge.

We now open the experiment in Igor7 or later and try to set all of the text encoding settings to correct known values using the SetWaveTextEncoding operation. The magic numbers used in the following commands are detailed below under **Text Encoding Names and Codes** on page III-490 and **SetWaveTextEncoding**.

```
// Mark any text waves containing binary data as such.
// 255 means "binary". 16 means "text wave content".
// /DF={root:,1} means "all waves in all data folders".
// /BINA=1 means "automatically mark text waves containing binary data".
SetWaveTextEncoding /DF={root:,1} /BINA=1 255, 16

// Mark any wave text elements currently set to unknown as Windows-1252.
// 0 means "unknown". 3 means Windows-1252. 31 means "all wave elements".
// /ONLY=0 means apply the command only to wave elements currently marked as
unknown.
SetWaveTextEncoding /DF={root:,1} /ONLY=0 3, 31
```

Now all elements of all waves are set to Windows-1252 except for the content element of text waves containing binary data which are so marked. In other words, there are no unknown text encodings and all elements are correctly marked. Marking all waves correctly ensures that Igor can correctly convert the wave plain text elements to UTF-8 for internal use.

If you were starting from a MacRoman experiment (Macintosh western text), you would use 2 instead of 3. If you were starting from a Japanese experiment (Shift JIS), you would use 4 instead of 3.

NOTE: The heuristic used by the /BINA flag is not foolproof. See **Text Waves Containing Binary Data** on page III-475 for details.

At this point you may want to convert your waves from Windows-1252 to UTF-8. This provides two advantages. First, Igor will not need to convert to UTF-8 when fetching the text waves' contents since the waves will already be UTF-8. Second, you will be able to use a wider repertoire of characters. The disadvantage is that you will get gibberish for non-ASCII characters if you open the experiment in any pre-Igor7 version. Assuming that you do want to convert to UTF-8, you would execute this:

```
// 1 means UTF-8. 31 means "all wave elements".
// /CONV means "convert text encoding". It automatically skips text wave
// content marked as binary.
SetWaveTextEncoding /DF={root:,1} /CONV=1 1, 31
```

Data Folder Name Text Encodings

Igor Pro 7.00 through 7.01 were unable to correctly load non-ASCII data folder names from Igor6 packed experiment files. Starting with version 7.02, in most cases, Igor correctly loads such data folder names.

If Igor mangles a non-ASCII data folder name when loading an Igor6 experiment, try choosing the correct text encoding from the Misc→Text Encoding→Default Text Encoding menu and turn Misc→Text Encoding→Default Text Encoding→Override Experiment on. Remember to restore these settings to their original settings after loading the experiment.

Starting with Igor Pro 7.02, Igor writes non-ASCII data folder names to packed experiment files using the text encoding of the built-in procedure window if possible. This will be Igor6-compatible if that text encoding is MacRoman on Macintosh or Windows-1252 on Windows. Igor Pro 7.00 through 7.01 expect UTF-8 and will therefore misinterpret such data folder names unless the text encoding of the built-in procedure window is UTF-8.

String Variable Text Encodings

Unlike the plain text elements of waves, there is no text encoding setting for string variables. Consequently Igor treats a string variable as if it contains UTF-8 text when printing it to the history area or when displaying it in an annotation or control panel or otherwise treating it as text.

If you have string variables in Igor6 experiments that contain non-ASCII characters, they will be misinterpreted by Igor7 or later. This may cause gibberish text or it may cause a Unicode conversion error. In either case you must manually fix the problem by storing valid UTF-8 text in the string variable.

For a local string variable, you can achieve this by assigning a new value to the string variable or by converting its contents from its original text encoding to UTF-8 using the **ConvertTextEncoding** function.

For a global string variable, you can achieve this by converting its contents from its original text encoding to UTF-8 using the **ConvertGlobalStringTextEncoding** operation. You need to know the original text encoding. It will typically be MacRoman if the Igor6 experiment was created on Macintosh, Windows-1252 if it was created on Windows, or Shift JIS if it was created using a Japanese version of Igor.

ConvertGlobalStringTextEncoding can convert all global string variables in the current experiment at once. Choose Misc→Text Encoding→Convert Global String Text Encoding to generate a ConvertGlobalStringTextEncoding command.

If you don't print or display a string variable or compare it to other text then its text encoding does not matter. For example, if you have a string variable that contains binary data and it is treated by your procedures as binary data, then everything will work out.

String Variable Text Encoding Error Example

This section gives a concrete example of a text encoding error which may help you to understand text encoding issues. As you read the example, keep these thoughts in mind:

- A text wave stores the raw bytes representing its text and an explicit setting that tells Igor the text encoding of the raw bytes. When you access an element of a text wave, Igor converts it to UTF-8 if it is not already stored as UTF-8 and is not marked as binary. During this conversion Igor uses the wave's explicit text encoding setting.
- String variables and other string expressions do not have text encoding settings and are always assumed to contain text encoded as UTF-8.

This example shows how you can create a text encoding problem and how you can avoid it.

```
Function TextEncodingErrorExample()
   // Make a text wave. Its text encoding defaults to UTF-8.
   Make /O/T/N=1 textWave0

   // Create a string variable containing non-ASCII text (the character ¹).
   // Igor converts literal text ("¹ != 3" in this case) to UTF-8
   // and stores it in the string variable.
   String text = "¹ != 3"                    // Stored as UTF-8

   // Convert the string variable contents to MacRoman.
   // Since the string has no text encoding setting, Igor still assumes it
   // contains UTF-8 but we know it really contains MacRoman.
   Variable textEncodingUTF8 = TextEncodingCode("UTF-8")
   Variable textEncodingMacRoman = TextEncodingCode("MacRoman")
   text = ConvertTextEncoding(text,textEncodingUTF8,textEncodingMacRoman,1,0)

   // Store the MacRoman text in the text wave. This creates bad data
   // because Igor assumes you are storing UTF-8 text in a UTF-8 text wave
   // when in fact you are storing MacRoman into a UTF-8 text wave.
   textWave0 = text                          // Store incorrect data

   // Print - an incorrect character will be printed because the wave's
   // text encoding setting is incorrect for the raw data bytes stored in it.
   Print textWave0[0]

   // Here is one way to fix the problem - convert the string variable to UTF-8.
   String textUTF8 = ConvertTextEncoding(text, textEncodingMacRoman,
                                          textEncodingUTF8, 1, 0)
   textWave0 = textUTF8                      // Store correct data

   // Print again - this time it prints correctly because the wave's
   // raw text bytes are consistent with the wave text encoding setting.
   Print textWave0[0]

   // Create the problem again so we can see another way to fix it.
   textWave0 = text                          // Store incorrect data

   // Here is another way to fix the problem - by correcting Igor's idea
   // of what text encoding the wave is using.
   // Tell Igor that the text wave stores text encoded as MacRoman.
   SetWaveTextEncoding textEncodingMacRoman, 16, textWave0

   // Print again - this time it prints correctly because Igor knows
   // that the wave contains MacRoman and so converts from MacRoman
   // to UTF-8 before sending the text to the history.
   Print textWave0[0]
End
```

Text Encoding Programming Issues

Igor6 and earlier versions of Igor stored text in "system text encoding". For western users this means MacRoman on Macintosh and Windows-1252 on Windows. For Japanese users it means Shift JIS on both platforms.

Igor now uses UTF-8, a form of Unicode, exclusively.

If you are an Igor programmer attempting to support Igor6, and if you or your users use non-ASCII text, the introduction of Unicode in Igor7 entails compatibility challenges.

If you fall in this category, we highly recommend that you abandon the idea of supporting Igor6 and later versions with the same procedure files. Instead, freeze your Igor6 code, and do all new development in a new fork. Here are the reasons for not attempting to support Igor6 and later versions with the same code base:

1. You will be unable to use new features without using ifdefs and other workarounds.
2. The use of ifdefs and workarounds will clutter your code and introduce bugs in formerly working Igor6 code.
3. You will have to deal with tricky text encoding issues, as explained in the next section.
4. Eventually Igor6 will be ancient history.

The alternative to supporting both Igor6 and later versions with the same procedure files is to create two versions of your procedures, one version for Igor6 and another for Igor7 and beyond. The benefits of this approach are:

1. You avoid cluttering your Igor6 code or introducing new bugs in it.
2. It will be much easier to write new code and it will be cleaner.
3. Your new code can use the full range of Unicode characters.

The costs of this approach are:

1. If you fix a bug in the Igor6 version of your code, you will have to fix it in the new version also.
2. New features of your code will be unavailable to Igor6 users.

The next section gives a taste of the text encoding issues raised by the use of Unicode starting with Igor7.

Literal Strings in Igor Procedures

Consider this function, which prints a temperature with a degree sign in Igor's history area:

```
Function PrintTemperature(temperature)
   Variable temperature

   // The character after %g is the degree sign
   Printf "The temperature is %g°\r", temperature
End
```

The degree sign is a non-ASCII character. It is encoded as follows:

MacRoman	0xA1
Windows-1252	0xB0
Shift JIS (Japanese)	0x81 0x8B
UTF-8	0xC2 0xB0

To simplify the discussion, we will ignore the issue of Japanese.

If you created the procedure file in Igor6 on Macintosh, it would work correctly in Igor6 on Macintosh but would print the wrong character in Igor6 on Windows. If you opened the procedure file in Igor7 or later with "Western" selected in Misc→Text Encoding→Default Text Encoding, it would work correctly on Macintosh but would print the wrong character on Windows. If you added this to the procedure file:

```
#pragma TextEncoding = "MacRoman"
```

it would work correctly in Igor7 or later on both Macintosh and Windows.

Literal Strings in Igor6/Igor7 Procedures

If you want it to work in Igor6 and Igor7 or later, on Macintosh and Windows, and irrespective of the text encoding of the procedure file and the Misc→Text Encoding→Default Text Encoding setting, you will need to do this:

```
#if IgorVersion() >= 7.00
   static StrConstant kDegreeSignCharacter = "\xC2\xB0"
#else
   #ifdef MACINTOSH
      static StrConstant kDegreeSignCharacter = "\241"
   #else
      static StrConstant kDegreeSignCharacter = "\260"
   #endif
#endif

Function PrintTemperature(temperature)
   Variable temperature

   // The character after %g is the degree sign
   Printf "The temperature is %g%s\r", temperature, kDegreeSignCharacter
End
```

This code uses escape sequences instead of literal characters to keep non-ASCII characters out of the procedure file. This allows it to work regardless of the text encoding of the procedure file in which it appears. For Igor7 or later, it uses the \xXX hexadecimal escape sequence. Igor6 does not support \xXX so it uses octal (\OOO) instead.

The section **Determining the Encodings of a Particular Character** on page III-482 shows how we determined the hex and octal codes.

The reason for making the StrConstants static is that other programmers may include the same code, causing a compile error if a user uses both sets of procedures.

The string constant kDegreeSignCharacter produces the correct string in Igor6 on Macintosh and Windows, and in Igor7 or later on Macintosh and Windows.

This will not print the right character for an Igor6 user using a Japanese font for the history area. Since you can not know the user's history area font, there is no way to cope with that.

If you have code that assumes that degree sign is one byte, you will need to modify it, since it is two bytes in UTF-8.

If the degree sign is the only non-ASCII character that you need to represent in Igor6 and Igor7 or later, this problem is manageable. If not, you will need similar code for other characters. The additional complexity and kludginess will get out of hand. That's why we recommend against supporting Igor6 and Igor7 or later with the same set of procedures.

Literal Strings in Igor7-only Code

If your procedure file requires Igor7 or later, you can write the PrintTemperature function in the obvious way:

```
#pragma TextEncoding = "UTF-8"
#pragma IgorVersion = 7.00

Function PrintTemperature(temperature)
   Variable temperature

   // The character after %g is the degree sign
   Printf "The temperature is %g°\r", temperature
End
```

This will work in Igor7 or later on Macintosh and Windows, and without regard to the user's history font or the Misc→Text Encoding→Default Text Encoding setting.

Unlike the previous approaches, this technique allows you to use any Unicode character, not just the small subset available in MacRoman or Windows-1252.

Determining the Encodings of a Particular Character

This section shows a method for determining how a given character is encoded in MacRoman, Windows-1252, and UTF-8. The commands must be executed in Igor7 or later because they use the ConvertTextEncoding function which does not exist in Igor6.

```
// Print MacRoman code for degree sign as octal - Prints 241
Printf "%03o\r", char2num(ConvertTextEncoding("°", 1, 2, 1, 0)) & 0xFF

// Print Windows-1252 code for degree sign as octal - Prints 260
Printf "%03o\r", char2num(ConvertTextEncoding("°" , 1, 3, 1, 0)) & 0xFF

// Print number of bytes of UTF-8 degree sign character - Prints 2
Printf "%d\r", strlen("°")

// Print first byte of UTF-8 for degree sign as hex - Prints C2
Printf "%02X\r", char2num("°"[0]) & 0xFF

// Print second byte of UTF-8 for degree sign as hex - Prints B0
Printf "%02X\r", char2num("°"[1]) & 0xFF
```

Using the printed information, we created the following Igor6/Igor7 code. It uses a hex escape sequence in Igor7 or later and an octal escape sequence in Igor6 because Igor6 does not support hex escape sequences.

```
#if IgorVersion() >= 7.00
   static StrConstant kDegreeSignCharacter = "\xC2\xB0"
#else
   #ifdef MACINTOSH
      static StrConstant kDegreeSignCharacter = "\241"
   #else
      static StrConstant kDegreeSignCharacter = "\260"
   #endif
#endif

Function PrintTemperature(temperature)
   Variable temperature
   // The character after %g is the degree sign
   Printf "The temperature is %g%s\r", temperature, kDegreeSignCharacter
End
```

This web page is a useful resource for comparing MacRoman and Windows-1252 text encodings:
https://kb.iu.edu/d/aesh

The MacRoman column is labeled "Mac OS" and the Windows-1252 column is labeled "Windows". The Latin1 column is a subset of Windows-1252.

You can see from the table that there are many characters that are available in MacRoman but not in Windows-1252 (e.g., Greek small letter pi) and vice-versa (e.g., multiplication sign). This is one of the reasons for switching to Unicode.

The left column of the table shows UTF-16 character codes. The table does not show UTF-8. You can convert a UTF-16 character code to UTF-8 text in Igor7 or later like this:

```
String piCharacter = "\u03C0"
```

For details on "\u", see **Escape Sequences in Strings** on page IV-14.

Other Text Encodings Issues

This section discusses miscellaneous issues relating to text encodings.

Characters Versus Bytes

In olden times, each character was represented in memory by a single byte. For example, in ASCII, A was represented by 0x41 (0x means hexadecimal) and B was represented by 0x42. Life was simple.

A single byte can represent 256 unique values so it was possible to represent 256 unique characters using a single byte. This was enough for western languages.

There came a time when people wanted to represent Asian text in computers. There are far more than 256 Asian characters, so multi-byte character sets (MBCS), such as Shift JIS, were invented. In MBCS, the ASCII characters are represented by one byte but most Asian characters are represented by two or more bytes. At this point, the equivalence of character and byte was no more and it became important to distinguish the two.

For example, in Igor6, the documentation for strlen said that it returns the number of characters in a string. This was true for western text but not for Asian text. The documentation was changed in Igor7 to say that strlen returns the number of bytes in a string.

In Igor7, Igor was changed to store text internally as UTF-8, a byte-oriented Unicode encoding form. In UTF-8, ASCII characters are represented by the same single-byte codes as in ASCII, thus providing complete backward compatibility for ASCII text. However, all non-ASCII characters, including accented characters, Greek letters, math symbols and special symbols, are represented by multiple bytes. A given character can consist of 1, 2, 3 or 4 bytes. The payback for this complexity is that UTF-8 can represent nearly every character of nearly every language.

In Igor6, this command prints 1 but in Igor7 or later, it prints 2:

```
Print strlen("µ")
```

This distinction is immaterial when you treat a string as a single entity, which is most of the time. For example, this code works correctly regardless of how characters are represented:

```
String units = "µs"
Printf "The units are %s\r", units
```

It is when you tinker with the contents of a string that it makes a difference. This works correctly in Igor6 and incorrectly in Igor7 or later:

```
String units = "µs"
Printf "The unit prefix is %s\r", units[0]
```

This fails in Igor7 or later, and prints a missing character symbol, because the expression "units[0]" returns a single byte, but in UTF-8, "µ" is represented by two bytes. Therefore "units[0]" returns the first byte of a two-byte character and that is an invalid character in UTF-8.

If you are tinkering with the contents of a string, and if the string may contain non-ASCII text, you need to be clear when you want to go byte-by-byte and when you want to go character-by-character. To go byte-by-byte, you merely use sequential indices. At present, there is no built-in support in Igor for going character-by-character. See **Character-by-Character Operations** on page IV-173 for an example of stepping through characters using user-defined functions.

Automatic Text Encoding Detection

Techniques exist for attempting to determine a file's text encoding from the codes it contains. Such techniques are unreliable, especially with the kind of text commonly used in Igor. Consequently Igor does not use them.

Much of Igor plain text is procedure text. Procedure text is mostly ASCII, sometimes with a smattering of non-ASCII characters. The non-ASCII characters may be, for example, MacRoman, Windows-1252, or Japanese.

There is no reliable way to distinguish MacRoman from Windows-1252 or to reliably distinguish a procedure file that contains a smattering of non-ASCII western text from one that contains a smattering of Japanese.

For example, consider an Igor procedure file that contains just one Japanese two-byte character encoded in Shift JIS by the bytes 0x95 and 0x61. This can be interpreted as:

Shift JIS:	CJK UNIFIED IDEOGRAPH-75C5
MacRoman:	LATIN SMALL LETTER I WITH DIERESIS, LOWERCASE LETTER A
Windows-1252:	BULLET, LOWERCASE LETTER A

All three of these interpretations are possible and choosing among them is guesswork so Igor does not attempt to do it.

Shift JIS Backslash Issue

Because Igor now internally process all text as UTF-8, it must convert text that it reads from files that use other encodings. A special issue arises with Japanese text stored in Shift JIS format.

In Shift JIS, the single-byte code 0x5C is hijacked to represent the half-width yen symbol. In ASCII, 0x5C represents the backslash character. When Shift JIS text is converted to UTF-8, Igor's conversion software leaves 0x5C unchanged. Thus its interpretation changes from a half-width yen symbol to a backslash symbol.

In most cases this is the desired behavior because the half-width yen symbol is used in Japanese like the backslash in ASCII - to separate elements of Windows file system paths and to introduce escape sequences in literal strings.

If you are using a half-width yen symbol as a currency symbol then this conversion will be wrong and you will have to manually convert it to a half-width yen symbol by editing the text.

In UTF-8, the half-width yen currency symbol is represented by the code units 0xC2 and 0xA5. It can be entered in a literal text string as "\xC2\xA5".

Converting to UTF-8

In Igor Pro 7 and later, Igor uses UTF-8 text encoding, a form of Unicode, to store text.

Non-Unicode text encodings, including MacRoman, Windows-1252, and Shift JIS, which were used in Igor Pro 6 and before, are antiquated. During the transition from Igor6 to Igor7, backward compatibility with Igor6 was needed so that Igor7 users could share files with Igor6 users. Starting with Igor9, we operate on the premise that compatibility with Igor6 is no longer needed.

Although non-Unicode text encodings are antiquated, Igor can continue to open old files that use them and it is often best to leave those files as they are. Nonetheless, there may be situations where it is beneficial to convert old files to Unicode. For example, if you are exporting data to another program, you may want to convert to UTF-8 for compatibility. This is especially true when exporting as HDF5 because HDF5 has no support for text encodings other than ASCII and UTF-8.

In Igor, text encoding is an issue primarily for three types of objects:

- Waves (for both wave properties and text wave data)
- Text files (plain text notebooks and procedure files)
- String variables

Igor stores text encoding settings for waves and text files. In Igor Pro 9 and later, Igor defaults to UTF-8 for new waves and text files. Igor7 and Igor8 defaulted to UTF-8 for new waves but for new text files it defaulted to the user-specified default text encoding set using the Misc→Text Encoding→Default Text Encoding submenu.

Unlike waves and text files, Igor does not store text encoding settings for string variables. Consequently Igor assumes that a string variable is encoded as UTF-8 when printing it to the history area or when displaying it in an annotation or control panel or otherwise treating it as text. This assumption will be correct for string variables containing only ASCII text and for non-ASCII string variables created by Igor7 or later. It will be incorrect for non-ASCII string variables created by Igor6 or before; these need to be converted to UTF-8 for proper display.

As explained under **Text Encoding Overview** on page III-460, the UTF-8 text encoding includes the ASCII text encoding as a subset. Consequently, converting ASCII to UTF-8 is a trivial operation that does not change the stored bytes. Converting non-ASCII to UTF-8 changes both the number of bytes stored and their values.

To facilitate conversion to UTF-8, Igor9 and later provide the Convert to UTF-8 Text Encoding dialog which you can invoke via the Misc→Text Encoding submenu.

The Convert to UTF-8 Text Encoding Dialog

The Convert to UTF-8 Text Encoding dialog facilitates conversion of waves, global string variables, and text files to UTF-8. The dialog presents four tabs - a Summary tab plus a tab for each type of object that can be converted.

When you click the Convert Waves, Convert Strings, or Convert Text Files buttons, conversions are done on objects in memory only. If you then save the experiments, these objects are written to disk.

Back Up Before Converting to UTF-8

You should back up all data before using the Convert to UTF-8 Text Encoding dialog. Possible problems include:

- Converting objects to UTF-8 that you need to access using Igor Pro 6
- Choosing the wrong source text encoding when converting strings and waves whose text encoding is unknown

- Converting wave text data or strings which contain binary data, not text

Although these problems should be rare, they are possible, so you should back up any data that may be affected before doing conversions.

The Summary Tab

The Summary tab provides an overview of what conversions the other tabs can perform.

The Waves Tab

The Waves tab lists all of the waves in the experiment that use non-UTF-8 text encodings or which appear to contain binary data but are not so marked.

As described under **Wave Text Encodings** on page III-472, each wave has separate text encoding settings for each of five elements: the wave name, the wave units, the wave note, the wave dimension labels, and the wave text content (applies to text waves only). However, for the vast majority of waves, all of these elements will use the same text encoding.

In general, waves created in Igor7 or later use UTF-8 text encoding. Such waves do not need to be converted to UTF-8 and do not appear in the Waves tab unless they appear to contain binary data (details below).

Waves created in Igor6 or before typically use MacRoman, Windows-1252, or Shift JIS (Japanese) text encodings. Such waves can be converted to UTF-8 and do appear in the Waves tab.

A wave element's text encoding may be "Unknown". This is the case for waves written by Igor versions prior to 6.30. If such a wave contains non-ASCII text, to convert it to UTF-8, Igor needs to know what text encoding to convert from. This is controlled by the text encoding pop-up menu to the right of the Convert Waves button. Choose a text encoding from the pop-up menu. Then hover the mouse over the Conversions Required column to see snippets of the wave's non-ASCII text rendered using the text encoding you chose. If the snippets appear incorrect, try choosing a different text encoding from the pop-up menu.

As described under **Text Waves Containing Binary Data** on page III-475, some text waves may be used to contain binary data rather than text. Such waves are created by Igor packages for specialized purposes. Waves that appears to contain binary but are not so marked are included in the list of waves needing conversion. On conversion, Igor marks their text content as binary.

The Waves tab is built on the SetWaveTextEncoding operation. Because it is intended for use by experts or by users instructed by experts, the SetWaveTextEncoding operation does not respect the lock state of waves. That is, it will change waves even if they are locked using SetWaveLock. The Waves tab inherits this behavior and so also does not respect the lock state of waves.

The list in the Waves tab comprises five columns:

Wave: Lists the full data folder path to each wave.

Kind: Indicates if a given wave is a "home" wave, meaning that it is part of the current experiment, or a "shared" wave, meaning that it is stored outside of the current experiment.

A wave that is stored in a packed experiment file or in the experiment folder of an unpacked experiment is a home wave. Otherwise it is a shared wave. A wave that is shared may be used by multiple experiments. See **Home Versus Shared Waves** on page II-87 for background information.

Encoding: Shows the text encoding governing the wave or "Mixed" if the wave uses multiple text encodings. Hover the mouse over "Mixed" to see which wave elements are governed by which text encodings.

Conversions Required: Shows which wave elements contain text that can be converted to UTF-8. Hover the mouse over an item to see a snippet of the non-ASCII text which the wave contains. This column also identifies text waves whose data appears to be binary but which are not yet marked as binary. Before the conversion, the column shows conversions that are required for each wave. After the conversion, it remains unchanged and should be understood as "conversions that were required" rather then "conversions that are required".

Status: After you click the Convert Waves button, this column shows the outcome of the conversion - if the conversion succeeded or if there was an error.

You can double-click a cell to display a modal Data Browser dialog for closer inspection of a wave.

Select one or more cells in the list and click Edit Waves to close the dialog and display a table showing those waves. If you click Edit Waves when no cells are selected in the list, all of the waves are displayed in the table.

Two checkboxes under the list of waves control whether waves requiring non-trivial changes and trivial changes are displayed in the list.

Waves containing non-ASCII text using a text encoding other than UTF-8 are deemed to require a non-trivial change. Also, text waves in which the text data content appears to contain binary data but which are not yet marked as binary are deemed to require a non-trivial change.

Waves containing ASCII text only but which are marked as using a non-UTF-8 text encoding are deemed to require a trivial change. Because the text is ASCII only, and because ASCII text is the same in all text encodings used in Igor waves, no actual text conversion is required. The wave elements just need to be marked as UTF-8.

You may find it convenient to convert the waves requiring trivial changes only first to get them out of the way so that you can focus on those that require non-trivial conversion.

Two checkboxes under the list of waves control whether home waves and shared waves are displayed in the list. Converting shared waves increases the possiblity that the conversion may adversely affect other experiment but this is usually a concern only if you need Igor6 compatibility.

Below the checkboxes, Igor displays an explanation of what waves need conversion.

If no items in the list are selected, clicking the Convert Waves button converts all of the waves in the list. The Status column then indicates if the conversion succeeded or failed.

If items are selected in the list, the Convert Waves button changes to Convert Selected Waves. Clicking the button converts the selected waves only.

After doing a conversion, the Convert Waves button changes to Refresh. Clicking Refresh refreshes the list, showing waves remaining to be converted, if any.

The Strings Tab

The Strings tab lists the global string variables in the experiment and allows you to convert those that need conversion.

As described under **String Variable Text Encodings** on page III-478, unlike waves, there is no text encoding setting for string variables. Consequently Igor treats each string variable as if it contains UTF-8 text when printing it to the history area or when displaying it in an annotation or control panel or otherwise treating it as text.

If a string was created in Igor6 or before and contains non-ASCII characters, treating it as UTF-8 results in incorrect characters being displayed. The Strings tab allows you to fix this by converting such strings to UTF-8.

The following types of strings do not need to be converted:

- Strings containing only ASCII characters
- Strings that are already valid as UTF-8
- Strings that appear contain binary data (typically created by procedure packages)

This leaves non-ASCII, non-binary strings that are not valid as UTF-8 to be converted.

To convert text to UTF-8, Igor needs to know the text encoding in effect when the text was entered. For string variables created in Igor7 and later, this will be UTF-8, and such string variables require no conversion. String variables created in Igor6 or before typically use MacRoman, Windows-1252, or Shift JIS (Japanese) text encodings and require conversion if they contain non-ASCII text.

An experiment may contain a combination of non-UTF-8 strings and UTF-8 strings. This happens if you create non-ASCII strings in Igor6 or before and then create more non-ASCII strings in the same experiment in Igor7 or later.

Since string variables have no text encoding settings, there is no way for Igor to know what text encoding was used to create them. Instead, in the Strings tab of the dialog, Igor uses the text encoding selected in the pop-up menu below the list of strings. When you enter the dialog, the pop-up menu is set to the text encoding governing the experiment, if it is known. As described below, you may need to choose a different text encoding from the pop-up menu.

The list in the Strings tab comprises three columns:

String: Lists the full data folder path to the string variable.

Contents: Shows the contents of the string variable.

The contents of strings that need to be converted are displayed assuming that their text was entered using the text encoding selected in the pop-up menu. If the displayed contents appear incorrect then you probably need to select another text encoding.

Strings that appear to contain binary are not converted and are displayed using hex escape codes for bytes that don't appear in normal ASCII text, such as "\x00" to represent a null byte.

Strings that are already valid as UTF-8, which includes ASCII strings since ASCII is a subset of UTF-8, also do not need to be converted and are displayed as UTF-8.

Status: The Status column displays the following:

Can be converted	The string can be converted to UTF-8 using the text encoding selected in the popup menu. The string will be converted when you click the Convert Strings button if it is selected or if no strings are selected.
Converted to UTF-8	The string was converted to UTF-8 using the text encoding selected in the popup menu. This appears after you click the Convert Strings button.
<An error message>	The string needs to be converted but can not be converted using the text encoding selected in the popup menu.
ASCII	The string is plain ASCII and does not need to be converted.
Valid non-ASCII UTF-8	The string contains non-ASCII text that is valid as UTF-8 and does not need to be converted.
Binary detected	The string appears to contain binary data and will not be converted.

Two checkboxes under the list of strings control whether strings that require conversion or do not require conversion are displayed in the list.

Strings that contain non-ASCII text that is not valid as UTF-8 and which do not appear to contain binary data need to be converted to UTF-8. Such strings appear in the list when the Show Strings That Need to be Converted checkbox is checked.

Strings that that appear to contain binary data, are all ASCII, or contain non-ASCII text that is already valid as UTF-8 do not need to be converted to UTF-8. Such strings appear in the list when the Show Strings That Do Not Need to be Converted checkbox is checked. You may want to display these strings to verify that their contents looks right.

To prevent confusion, it is usually best to examine these two classes of strings separately. Display the strings that do not need conversion only and examine them. When you are satisfied that they are correct, display the strings that need conversion only and proceed.

All string variables are stored as part of the current experiment so there are no "shared" string variables. Consequently, the Home and Shared checkboxes that appear in the Waves and Text Files tabs are not present in the Strings tab.

Inspect the text in the Contents column. If it appears correct, click the Convert Strings button to convert the strings to UTF-8. If it appears incorrect, choose a different text encoding from the pop-up menu and inspect the Contents column again.

Below the checkboxes, Igor displays an explanation of what strings need conversion.

When you click Convert Strings, Igor converts text strings from the selected text encoding to UTF-8. It skips strings that do not need conversion, if they are displayed in the table.

If no items in the list are selected, clicking the Convert Strings button converts all of the strings in the list that need to be converted. If items are selected in the list, the button changes to Convert Selected Strings and clicking it converts the selected strings only and only if they need to be converted. Strings that do not need to be converted are skipped in either case.

The Text Files Tab

The Text Files tab lists all of the text files (plain text notebooks and procedure files) in the experiment that are marked as using a non-UTF-8 text encoding, whether they contain non-ASCII text or not. It also lists "History" if the history area of the command window is marked as using a non-UTF-8 text encoding.

Global procedure files and #included procedure files are not considered part of the current experiment and are not displayed in the list of text files. The programmer responsible for these files should convert them to UTF-8.

The list in the Text Files tab comprises five columns:

Window Title: The title of the notebook or procedure window, or "History" for the history area of the command window.

Window Name: The window name for notebooks. The history area of the command window and procedure windows have no names.

Window Type: The type of window: procedure, notebook, or history.

This column also indicates if the text file is a "home" text file, meaning that it is part of the current experiment, or a "shared" text file, meaning that it is stored outside of the current experiment. See **Home Versus Shared Text Files** on page II-56 for background information.

File Name: The name of the file associated with the window if it has been saved to a standalone file.

Text Encoding: The text encoding currently associated with the window.

Two checkboxes under the list of text files control whether home text files and shared text files are displayed in the list. Converting shared text files increases the possiblity that the conversion may adversely affect other experiment but this is usually a concern only if you need Igor6 compatibility.

If no items in the list are selected, clicking the Convert Text Files button converts all of the files in the list, except for files open for read-only which are skipped. If items are selected in the list, the button changes to Convert Selected Text Files and clicking it converts the selected files only, except for files open for read-only which are skipped.

Unlike read-only files, write-protected files are converted, because write-protect is intended to protect against inadvertent manual editing only.

When a given text file is successfully converted, its entry in the Text Encoding column changes to "UTF-8".

After doing a conversion, the Convert Text Files button changes to Refresh. Clicking Refresh refreshes the list, showing text files remaining to be converted, if any.

Text Encoding Names and Codes

Igor operations and functions such as **ConvertTextEncoding**, **SaveNotebook**, **WaveTextEncoding** and **SetWaveTextEncoding** accept text encoding codes as parameters or return them as results.

The functions **TextEncodingName** and **TextEncodingCode** provide conversion between names and codes. For most Igor programming purpose, you need the code, not the name.

For each text encoding supported by Igor there is one text encoding code. This code may correspond to one or more text encoding names. For example, code 2 is the Macintosh Roman text encoding and corresponds to the text encoding names "macintosh", "MacRoman" and "x-macroman".

Because spelling of text encoding names is inconsistent in practice, Igor recognizes variant spellings such as "Shift JIS", "ShiftJIS", "Shift_JIS" and "Shift-JIS". When comparing text encoding names, Igor ignores all non-alphanumeric characters. It also ignores leading zeros in numbers embedded in text encoding names so that "ISO-8859-1" and "ISO-8859-01" refer to the same text encoding. It uses a case-insensitive comparison.

Some of the entries in the table below are marked NOT SUPPORTED. These are not supported because the ICU (International Components for Unicode) library, which Igor uses for text encoding conversions, does not support it. "Not supported" means that the TextEncodingName and TextEncodingCode functions do not recognize these text encodings and Igor can not convert to them or from them.

Text Encoding Name	**Code**	**Notes**
None	0	Means the text encoding is unknown
UTF-8	1	Unicode UTF-8
macintosh, MacRoman, x-macroman	2	Macintosh Western European
Windows-1252	3	Windows Western European
Shift_JIS	4	Predominant Japanese text encoding
MacJapanese, x-mac-japanese	4	Virtually the same as Shift_JIS
Windows-932	4	Virtually the same as Shift_JIS
EUC-JP	5	Japanese, typically used on Unix
Big5	20	Traditional Chinese
x-mac-chinesetrad	20	Virtually the same as Big5
Windows-950	20	Virtually the same as Big5
EUC-CN	21	Simplified Chinese
x-mac-chinesesimp	21	Simplified Chinese
Windows-936	21	Simplified Chinese
ISO-2022-CN	22	Simplified Chinese
GB18030	23	Official text encoding of the PRC. Encompasses Traditional Chinese and Simplified Chinese. Compatible with Windows-936.
EUC-KR, x-mac-korean	40	Macintosh Korean

Text Encoding Name	Code	Notes
Windows-949, ks_c_5601-1987	41	Windows Korean
ISO-2022-KR	42	Korean text encoding not used on Macintosh or Windows
x-mac-arabic	50	Macintosh Arabic. NOT SUPPORTED.
Windows-1256	51	Windows Arabic
x-mac-hebrew	55	Macintosh Hebrew
Windows-1255	56	Windows Hebrew
x-mac-greek	60	Macintosh Greek
Windows-1253	61	Windows Greek
x-mac-cyrillic	65	Macintosh Cyrillic
Windows-1251	66	Windows Cyrillic
x-mac-thai	70	Macintosh Thai. NOT SUPPORTED.
Windows-874	71	Windows Thai
x-mac-ce	80	Macintosh Central European
Windows-1250	81	Windows Central European
x-mac-turkish	90	Macintosh Turkish
Windows-1254	91	Windows Turkish
UTF-16BE	100	UTF-16, big-endian
UTF-16LE	101	UTF-16, little-endian
UTF-32BE	102	UTF-32, big-endian
UTF-32LE	103	UTF-32, little-endian
ISO-8859-1, Latin1	120	ISO standard western European
Symbol	150	Used by Symbol font
Binary	255	Indicates data is really binary, not text

Binary is not a real text encoding. Rather it marks data stored in a text wave as containing binary rather than text. See **Text Waves Containing Binary Data** on page III-475 for details.

Symbol Font

Igor Pro 6 and before used "system text encoding", typically MacRoman on Macintosh and Windows-1252 on Windows. These text encodings can represent a maximum of 256 characters because each character is represented by a single byte and a single byte can represent only 256 unique values from 0 to 255.

Most Greek letters can not be represented in MacRoman or Windows-1252. Because of that, people resorted to Symbol font when they wanted to display special characters, such as Greek letters. In system text encoding, Symbol font is treated specially. For example, the byte value 0x61 (61 hex, 97 decimal), which is normally interpreted as "small letter a", is interpreted in Symbol font as "Greek small letter alpha".

Thus, in Igor6, to create a textbox that displayed a Greek small letter alpha, you needed to execute this:

```
TextBox "\\F'Symbol'a"
```

Because Igor now uses Unicode, you have at your disposal a wide range of characters of all types without changing fonts. To create a textbox that displays a Greek small letter alpha, you simply execute this:

```
TextBox "α"
```

Also, Symbol font is no longer treated specially. Consequently this:

```
TextBox "\\F'Symbol'a"
```

displays a "small letter a", not a "small Greek letter alpha". This is a departure from Igor6 and creates compatibility issues.

You can copy Symbol font characters to the clipboard as Unicode using the **Symbol Font Characters** table. You can insert Unicode characters by choosing Edit→Characters for commonly-used characters such as Greek letters and math symbols or Edit→Special Characters for other characters. In the Add Annotation dialog and in the Axis Label tab of the Modify Axis dialog, you can click Special and choose a character from the Character submenu.

Symbol Font Backward Compatibility

The use of Unicode greatly simplifies the display of Greek letters and other special characters. But it creates an incompatibility with Igor6. Without special handling, when Igor loads an Igor6 file, Symbol font text meant to display Greek letters would display Roman letters instead. This section explains how Igor deals with this issue to provide backward compatibility.

When loading Igor6-compatible procedure files, which use system text encoding, including the experiment recreation procedures that execute when you open an Igor6-compatible experiment file, Igor attempts to convert incoming Symbol font characters to Unicode. For compatibility with Igor6, the reverse is done on writing procedure files, including experiment recreation procedures, using system text encoding. The conversion is also done for Igor6-compatible formatted notebook files which use system text encoding. This conversion is done using a heuristic that involves scanning for certain patterns and consequently is not, and can not be, perfect.

There may be cases where this attempt at backward compatibility creates problems, for example, if you don't care about Igor6 compatibility. If you want to turn Symbol font compatibility off, you can execute

```
SetIgorOption EnableSymbolToUnicode=0
```

and reload the experiment.

If you encounter Symbol font problems and the information below does not provide the solution, please let us know, and provide the original Igor6 text that caused the problem.

Symbol Font Backward Compatibility Limitations

An important limitation is that, in the original Igor6 experiment, Symbol font specifications in annotations must not include anything other than the Symbol characters. No escape sequences, such as font size specifications, are allowed. For example, Igor will not handle this Igor6 text:

```
TextBox/C/N=text0/F=0/A=MC "\\Z18A\\F'Symbol'\\Bq\\F]0"
```

because the subscript escape code, "\\B", is inside the Symbol font escape sequence and prevents Igor from recognizing this as a Symbol font sequence. As a result, you will see a "q" character instead of the intended small Greek letter theta character.

To enable Igor to recognize this as Symbol text, move the subscript escape code outside the Symbol font escape sequence, like this:

```
TextBox/C/N=text0/F=0/A=MC "\\Z18A\\B\\F'Symbol'q\\F]0"
```

When creating graphics for compatibility with Igor6 or for EPS export, use Unicode characters, like this:

```
TextBox/C/N=text0/F=0/A=MC "\\Z18A\\B\\F'Symbol'θ\\F]0"
```

When Igor writes this out to a procedure file, it will convert the Symbol font sequence into an Igor6-compatible sequence, by replacing the Unicode theta character with "q".

If you don't care about compatibility with Igor6 or EPS export, you don't need to use Symbol font. You can instead write:

```
TextBox/C/N=text0/F=0/A=MC "\\Z18A\\Bθ"
```

On Macintosh, you might still want to specify Symbol font because it provides almost all of the Symbol characters and you may prefer its style compared to other fonts.

Zapf Dingbat Font

Igor deals with Zapf Dingbats font in the same way. Zapf Dingbats was widely available on Macintosh in previous millennium.

Symbol Tips

On Windows, because Igor now uses Unicode, Symbol font does not appear to be useful. Consequently, Igor substitutes another font when Symbol is encountered.

To insert frequently-used symbol characters, such as Greek characters and math symbols, choose Edit→Characters.

In the Add Annotation dialog and in the Axis Label tab of the Modify Axis dialog, you can click Special and choose a character from the Character submenu.

Symbol Font Characters

The “Text Encoding.ihf” help file includes a list of Symbol font characters. To display it, execute:

```
DisplayHelpTopic "Symbol Font Characters"
```

You can copy the characters as Unicode from that section of the help file and paste them into another window.

Symbols with EPS and Igor PDF

Both EPS and PDF include Symbol font as one of their standard supported fonts. When inserting a character from the above list, you can either specify Symbol font or you can use a Unicode font that supports those characters.

In the case of Symbol font, Igor translates the Unicode code point to the corresponding Symbol single byte code (unless you have specified that even standard fonts be embedded) and such characters in the resulting file will be editable in a program such as Adobe Illustrator. The alternative is to specify a font such as Lucida Sans Unicode in which case the characters are embedded using an outline font and will not be editable.

Be sure to specify either Symbol or a font that will be embedded because it is likely that the current default font is one of the EPS or PDF standard supported fonts. These are not Unicode and the result will not be what you expect.

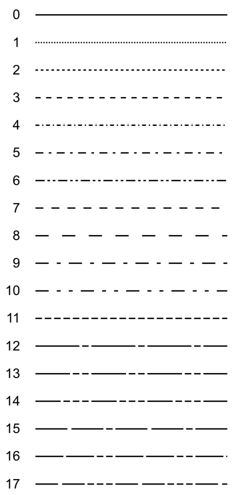
Chapter

III-17

Miscellany

Dashed Lines

You can display traces in graphs, as well as drawn lines, rectangles, ovals, and polygons, using various line styles. This table, generated from the ColorsMarkersLinesPatterns.pxp example experiment, shows Igor’s default line styles:



It is usually not necessary, but you can edit the built-in dashed line styles using the Dashed Lines dialog by choosing Misc→Dashed Lines.

Dashed line 0 (the solid line) cannot be edited. If you need to create a custom dashed line pattern, we recommend that you modify the high numbered dashed lines, leaving the low number ones in their default state. This ensures that the low numbered patterns will be the same for everyone.

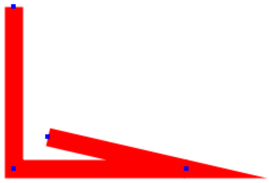
You can also change dashed lines with the **SetDashPattern** operation.

Dashed lines are stored with the experiment, so each experiment can have different dashed lines. You can capture the current dashed lines as the preferred dashed lines for new experiments.

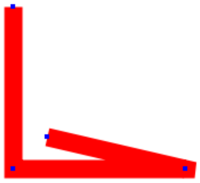
Line Join and End Cap Styles

When graph traces or drawn lines are very wide, you may want to control the appearance of the joins between line segments and the ends of line segments. The ModifyGraph lineJoin keyword provides control of line joins for graph traces in lines-between-points mode. The SetDrawEnv lineJoin keyword provides control for drawn lines.

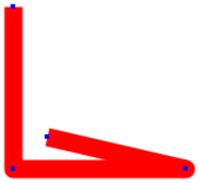
Line joins can have miter, round or bevel styles. For a miter join, you can specify the miter limit to control the length of miters on very acute angles. In these pictures, the blue dots show the position of the ends and join points of the lines:



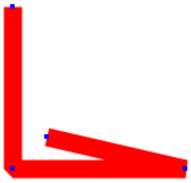
lineJoin={0, 10}
miter, w/ miter limit=10



lineJoin={0, sqrt(2)}
miter, limit=sqrt(2)



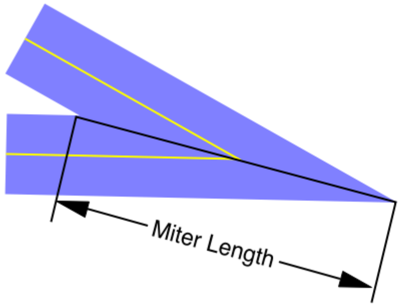
lineJoin={1, 0}
round

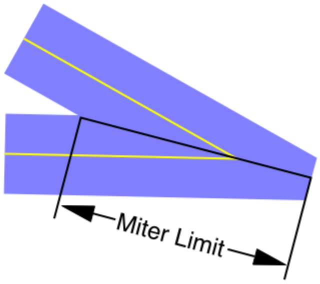


lineJoin={2, 0}
bevel

Miter joins extend the outside edges of the line segments until they intersect. Acute angles may result in very long miters. Round joins draw a circular arc around the join point, and bevel joins truncate the intersection with a bevel at the intersection point.

You can set the miter limit to avoid very long miter extensions:

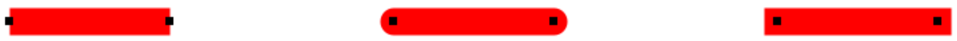




When the miter limit is sqrt(2), any line join that is more acute than 90 degrees is truncated. Unfortunately, the nature of the truncation depends on the Graphics Technology. Core Graphics (Macintosh), GDI (Windows), PDF and Postscript truncate the miter by reverting to a bevel join; Qt graphics and GDI+ (Windows) truncate the miter by beveling at the miter limit. If you export graphics using a bitmap format such as PNG or JPG, the miter limit is controled by the graphics technology you have chosen (usually Qt graphics) in the Miscellaneous Settings dialog. The second picture above was drawn with Qt graphics as a PNG bitmap; the very acute intersection is truncated at the miter limit.

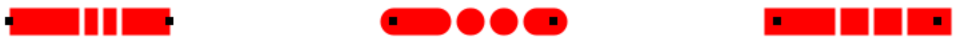
You can control the way line ends are drawn using ModifyGraph lOptions keyword, or for drawn lines using SetDrawEnv lineCap.

Line caps can be flat, round or square. These pictures show the appearance of each option; the dots show where the geometric end of each line is:



Flat	Round	Square
ModifyGraph lOptions=0	lOptions=1	lOptions=2
SetDrawEnv lineCap=0	lineCap=1	lineCap=2

The end cap style is applied at the end of each segment of a dashed line:



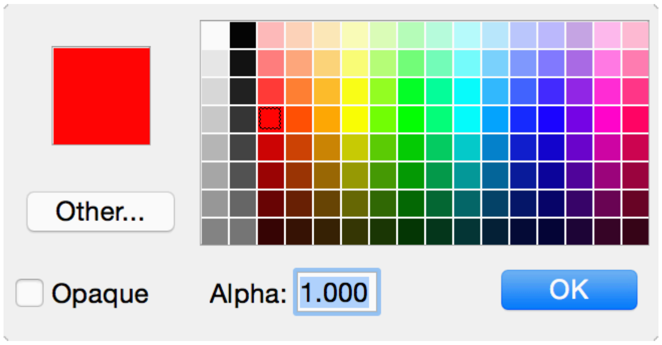
Square end caps extend the square end of the line beyond the geometric end and in Igor are probably not useful.

The Color Environment

Igor has a main color palette that contains colors that you can use for traces in graphs, text, axes, backgrounds and so on. The main color palette appears as a pop-up menu in a number of dialogs, such as the Modify Trace Appearance dialog. This section discusses this palette.

Igor also has color tables and color index waves you can select among when displaying contour plots and images. These are discussed in Chapter II-15, **Contour Plots**, and Chapter II-16, **Image Plots**.

You can select a color from the colors presented in a color palette:



□ Opaque

Alpha: 1.000

You can use the Other button to select colors that are not in the palette. As you use Igor, colors are added to the palette in the Recent Colors area.

The recent colors are remembered by Igor when it quits and restored when it restarts if you have selected the “Save and restore color palette’s recent colors” checkbox in the Miscellaneous Settings dialog’s Color Settings category.

Color Blending

In places where you can supply a color in RGB format, you can optionally provide a fourth parameter called "alpha". Alpha, like the R, G and B values, can range from 65535 (100% opaque) down to 0 (100% transparent). Intermediate values provide translucency. For example:

```
Make jack=sin(x/8),sam=cos(x/6); Display jack, sam
ModifyGraph mode=7,hbFill=2
ModifyGraph rgb(jack)=(65535,32768,32768)
ModifyGraph rgb(sam)=(0,0,65535,40000)       // Translucent
```

Color blending was added in Igor Pro 7.

Color blending does not work on Windows in the old GDI graphics mode explained under **Graphics Technology on Windows** on page III-506.

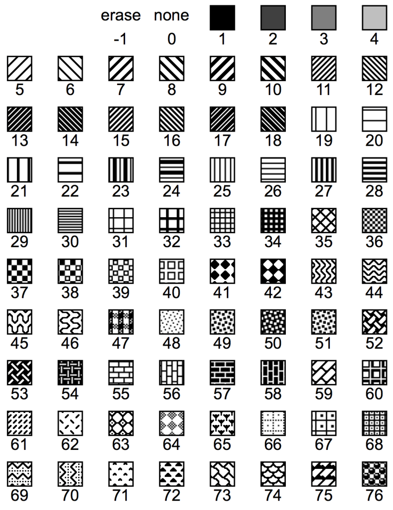
Color blending does not work in graphics exported as EPS. Use PDF instead.

Fill Patterns

Fill patterns can be used in graphs and drawing layers. This table, generated from the ColorsMarkersLinesPatterns.pxp example experiment, shows the available fill patterns.

The fill pattern codes shown below are appropriate for drawing commands. For graph-related commands, such as ModifyGraph hbFill, make the following adjustments:

- Erase is 1 instead of -1
- Add 1 to all pattern codes greater than 0



Gradient Fills

You can specify gradient fills for drawing elements, graph trace fills, and graph window and plot area background fills. A gradient fill is a gradual change of color.

Programmatically you specify gradients using two keywords, gradient and gradientExtra, when calling the ModifyGraph, ModifyLayout or SetDrawEnv operations.

The syntax for the gradient keyword has two forms. The first form provides overall control of the gradient while the second form controls the color details.

The first form for the gradient keyword is:

```
gradient = 0, 1, 2, or 3
```

0 deletes the gradient entirely.

1 turns on an existing gradient or creates a new one with default values.

2 turns off a gradient but keeps the settings.

3 is used in combination with the ModifyLayout operation and signals that a particular page should not display a gradient even when a layout global gradient is set.

The second form for the gradient keyword is:

```
gradient = {type, x0, y0, x1, y1 [,color0 [, color1 ] ]}
```

type is a bitfield.

Bits 0 through 3 specify the gradient mode. 0 is linear, 1 is radial and other values are reserved.

Bits 4 through 7 specify the coordinate mode. 0 is object rect, 1 is window rect.

To construct a type parameter signifying a radial gradient in window rect mode you could write:

```
Variable gradientType = 1 | (1*16)
```

The following commands illustrate the difference between object rect and window rect mode. Execute these commands and then drag the rectangles around:

```
Display /W=(150,50,474,365)
SetDrawLayer UserFront
SetDrawEnv xcoord=abs,ycoord=abs
SetDrawEnv save
SetDrawEnv fillfgc=(16385,16388,65535),fillbgc=(65535,16385,16385)
SetDrawEnv gradient={16, 0, 0, 1, 1}      // Window rect mode
DrawRect 25,38,95,138
SetDrawEnv gradient={1, 0.5, 0.5, 0, 1, (0,65535,0), (65535,0,65535)}
DrawRect 130,150,273,289
ShowTools/A
```

The *x0*, *y0*, *x1*, and *y1* parameters specify normalized locations that define the gradient. (0,0) is the upper left corner of the bounding rectangle while (1,1) is the lower right corner. For a linear gradient, (*x0*, *y0*) defines the start while (*x1*, *y1*) defines the end. Only the slope of the line is used, not the actual extent. For a radial gradient, (*x0*, *y0*) defines the center of a bounding circle while (*x1*, *y1*) is not used at present.

The optional color0 and color1 keywords are specified as (r,g,b) or (r,g,b,a) but use of alpha is not fully supported on all platforms (Quartz PDF on Macintosh for example). When omitted or with a value of (0,0,0) the actual color used is a default for the given circumstance. The default values are specified in the individual operation documentation.

The gradientExtra keyword adds an additional color change point. You can add as many change points as desired.The syntax for the gradientExtra keyword is:

```
gradientExtra = {loc, color}
```

loc is in the range 0.0 to 1.0. Values of exactly 0 or 1 replace the original *color0* or *color1* values as specified by the gradient keyword.

color is specified as (r,g,b) or (r,g,b,a) but use of alpha is not fully supported on all platforms (Quartz PDF on Macintosh for example).

The following operations support the gradient and gradientExtra keywords.

SetDrawEnv

If you specify a gradient it overrides the fill for drawing elements.

If you omit the *color0* parameter or specify (0,0,0,0) then the current pattern background color is used.

If you omit the *color1* parameter or specify (0,0,0,0) then the current foreground color is used.

All gradientExtra keywords must be on the same line as the gradient keyword itself.

ModifyLayout

You can specify gradients for individual pages by combining the gradient and gradientExtra keywords with the /PAGE=*pageNum* flag.

Page numbers start from 1. Use /PAGE=0 to use the currently active page.

/PAGE=-1 causes the operation to modify the layout global gradient which applies to any pages in the targeted page layout for which no explicit gradient has been set.

ModifyGraph (traces)

If you specify a gradient it overrides the fill for traces.

The gradient and gradientExtra keywords apply to all traces unless you specify a trace name like this:

```
gradient(<trace name>) = {...}
gradientExtra(<trace name>) = {...}
```

If you omit the *color0* parameter or specify (0,0,0,0) then the current pattern background color is used.

If you omit the *color1* parameter or specify (0,0,0,0) then the plusRGB color is used.

When the *type* parameter specifies a window rect, the plus and neg areas are used automatically and the *color1* is away from the zero level.

ModifyGraph (colors)

The wbGradient and wbGradientExtra keywords control the window background gradient, if any.

The gbGradient and gbGradientExtra keywords control the graph plot area background gradient, if any.

The "Demo Experiment #1" and "Demo Experiment #2" example experiments demonstrate these gradients. You can turn the on and off using the Macros menu.

Miscellaneous Settings

You can customize many aspects of how Igor works using the Miscellaneous Settings dialog. Choose Misc→Miscellaneous Settings to display the dialog.

Most of the settings are self-explanatory. Many have tooltips that describe what they do.

Object Names

Every Igor object has a name which you give to the object when you create it or which Igor automatically assigns. You use an object's name to refer to it in dialogs, from commands and from Igor procedures. The named objects are:

Waves
Windows
Controls
Symbolic paths
FIFOs
Data folders
Axes
Rulers
Pictures
FIFO channels
Variables (numeric and string)
Annotations
Special characters (in notebooks)
XOPs

In Igor Pro, the rules for naming waves and data folders are not as strict as the rules for naming all other objects, including string and numeric variables, which are required to have standard names. These sections describe the standard and liberal naming rules.

Standard Object Names

Here are the rules for standard object names:

- May be 1 to 255 bytes in length.

  Prior to Igor Pro 8.00, names were limited to 31 bytes. See **Long Object Names** on page III-502 for a discussion of issues related to names longer than 31 bytes.
- Must start with an alphabetic character (A-Z or a-z).
- May include ASCII alphabetic or numeric characters or the underscore character.
- Must not conflict with other names (of operations, functions, etc.).

All names in Igor are case insensitive. wave0 and WAVE0 refer to the same wave.

Characters other than letters and numbers, including spaces and periods, are not allowed. We put this restriction on names so that Igor can identify them unambiguously in commands, including waveform arithmetic expressions.

Liberal Object Names

The rules for liberal names are the same as for standard names except that almost any character can be used in a liberal name. Liberal name rules are allowed for waves and data folders only.

If you are willing to expend extra effort when you use liberal names in commands and waveform arithmetic expressions, you can use wave and data folder names containing almost any character. If you create liberal names then you will need to enclose the names in single (not curly) quotation marks whenever they are used in commands or waveform arithmetic expressions. This is necessary to identify where the name ends. Liberal names have the same rules as standard names except you may use any character except control characters and the following:

```
   "  '  :  ;
```

Here is an example of the creation and use of liberal names:

```
Make 'wave 0';     // 'wave 0' is a liberal name
'wave 0' = p
Display 'wave 0'
```

Note: Providing for liberal names requires extra effort and testing on the part of Igor programmers (see **Programming with Liberal Names** on page IV-168) so you may occasionally experience problems using liberal names with user-defined procedures.

Namespaces

When you refer to an object by name, in a user function for example, each object must be referenced unambiguously. In general, an object must have a unique name so. Sometimes the object type can be inferred from the context, in which case the name can be the same as objects of other types. Objects whose names can be the same are said to be in different namespaces.

Data folders are in their own name space. Therefore the name of a data folder can be the same as the name of any other object, except for another data folder at the same level of the hierarchy.

Waves and variables (numeric and string) are in the same name space and so Igor will not let you create a wave and a variable in a single data folder with the same name.

An annotation is local to the window containing it. Its name must be unique only among annotations in the same window. The same applies for controls and rulers. Data folders, waves, variables, windows, symbolic paths and pictures are global objects, not associated with a particular window.

The names of global objects, except for data folders, are required to be distinct from the names of macros, functions (built-in, external or user-defined) and operations (built-in or external).

Here is a summary of the four global namespaces:

Name Space	Requirements
Data folders	Names must be distinct from other data folders at the same level of the hierarchy.
Waves, variables, windows	Names must be distinct from other waves, variables (numeric and string), windows.
Pictures	Names must be distinct from other pictures.
Symbolic paths	Names must be distinct from other symbolic paths.

Object Name Functions

These functions are useful for programmatically generating object names: **CreateDataObjectName**, **CheckName**, **CleanupName**, **UniqueName**.

Long Object Names

Prior to Igor Pro 8.00, names were limited to 31 bytes. Now names can be up to 255 bytes in length.

The following types of objects support long names in Igor Pro 8.00 or later:

- Data folders
- Waves
- Variables
- Windows
- Axes
- Annotations
- Controls
- Special characters in formatted notebooks
- Symbolic paths
- XOPs, external operations, external functions

In addition, in Igor8, wave dimension labels and procedure names can be up to 255 bytes in length.

The following types of objects are still limited to names of 31 bytes or less:

- Rulers in formatted notebooks

- Global picture names
- Page setup names
- FIFOs
- FIFO channels

If you do not create objects with names longer than 31 bytes, wave and experiment files that you create will be compatible with earlier versions of Igor. However, if you do create objects with long names, older versions of Igor will report errors when opening wave and experiment files containing long names.

NOTE: If you use long names, your wave and experiment files will require Igor Pro 8.00 or later and will return errors when opened by earlier versions of Igor.

You can choose File→Experiment Info to determine if the current experiment uses long object names or has waves with long dimension labels. You can also use the **ExperimentInfo** operation programmatically. These check only wave, variable, data folder, target window, and symbolic path names, and wave dimension labels. They do not check axis, annotation, control, procedure or other names.

If you attempt to save an experiment file that uses long wave, variable, data folder, target window or symbolic path names, or that has waves with long dimension labels, Igor displays a warning dialog telling you that the experiment will require Igor Pro 8.00 or later. The warning dialog is presented only when you save an experiment interactively, not if you save it programmatically using SaveExperiment. You can suppress the dialog by clicking the "Do not show this message again" checkbox.

Global picture names (see **The Picture Gallery** on page III-510) are limited to 31 bytes but names of Proc Pictures (**Proc Pictures** on page IV-56) are not.

Page setup names are used behind the scenes to save a page setup record for each page layout window. The experiment file format limits the name of a page setup record to 31 bytes. If a layout window name exceeds 31 bytes, when you save the experiment, the page setup record for that window is not written to the experiment file. When you reopen the experiment, the layout window receives a default page setup. Since long page layout names are rare and page setups affect printing but not the dimensions of the page (see **Page Layout Page Sizes** on page II-478), this issue will have little impact.

An XOP name is the name of the XOP file without the ".xop" extension. In Igor8, XOP names can be up to 255 bytes. However, if an XOP name exceeds 31 bytes, Igor does not send the SAVESETTINGS message to the XOP. Most XOP names are shorter than 31 bytes and most XOPs do not save experiment settings, so this is not likely to cause a problem.

Long Object Names With Old Igor Versions

If you use long names, your wave and experiment files will require Igor Pro 8.00 or later and will return errors when opened by earlier versions of Igor.

You can choose File→Experiment Info to determine if the current experiment uses long object names or has waves with long dimension labels. You can also use the **ExperimentInfo** operation programmatically.

If you open an experiment file that uses long wave, variable, data folder, window or symbolic path names while running Igor Pro 7.xx, where xx is 01 or later, the old Igor version displays an error dialog explaining that the experiment requires Igor Pro 8.00 or later. This mechanism for informing you that a later version of Igor is required works for long wave, variable, data folder, window and symbolic path names only. It does not work for long axis, annotation, control, special character, procedure or XOP names. For those object types, you will get an error later, when the old version of Igor first encounters the long name.

If you open an experiment file that uses long names of any kind in Igor Pro 7.00 or before, you will get an error such as "name too long" or "incompatible Igor binary version" or some other error.

In Igor Pro 6.38 and Igor Pro 7.01, a bug was fixed that cause Igor to crash if you load a file containing long wave names and the file contains wave reference waves or data folder reference waves. You are very unlikely to have such files.

In Igor Pro 6.38 and Igor Pro 7.01, a bug was fixed that cause Igor to crash if you load a formatted notebook file containing a long special action name and you attempt to modify that special action.

Programming With Long Object Names

If your code must run with Igor7 or before, the best strategy is to avoid using long object names. Attempting to conditionally support long object names will make your code complex and fragile. A better approach is to freeze your Igor7 code and add new features to an Igor8 or later branch.

If you must use conditional programming, you can test whether the running version of Igor supports long object names like this:

```
Variable maxObjectNameLength = 31
if (IgorVersion() >= 8.00)
   maxObjectNameLength = 255
endif
```

The names of functions, constants, variables and other programming entities can be up to 255 bytes, but if you use names longer than 31 bytes, your procedures will require Igor Pro 8.00 or later.

Package names can be up to 255 bytes, but if you use a name longer than 31 bytes, your package will require Igor Pro 8.00 or later.

Long Object Names and XOPs

For most of recorded history, XOPs supported a maximum object name length of 31. Igor now support long object names but pre-existing XOPs continue to work as before. However, if you attempt to use a long object name with a pre-existing XOP, you will get an error. For example:

```
NewPath SymbolicPathWithANameThatExceeds31Bytes, <path>
OldFileLoader /P=SymbolicPathWithANameThatExceeds31Bytes <path>
```

This returns an error because the hypothetical OldFileLoader XOP has not been updated to work with long object names. It continues to work with short object names.

If a pre-existing XOP attempts to retrieve the name of a wave or data folder which has a long name, Igor returns an error to the XOP. If the XOP is properly written, it will generate a "name too long" error and Igor will report the error.

As of XOP Toolkit 8, the XOP Toolkit supports creating XOPs that work with long object names. Supporting long names requires that the XOP programmer modify and recompile the XOP. XOPs compiled to support long object names will require Igor Pro 8 or later. Consequently, many XOPs will not support them.

As of this writing, if the name of the XOP itself, without the ".xop" extension, exceeds 31 bytes in length, the XOP will not be able to save settings using the SAVESETTINGS message or load settings using the LOADSETTINGS message. Consequently it is best to avoid creating an XOP whose name exceeds 31 bytes.

Renaming Objects

You can use Misc→Rename Objects or Data→Rename to rename waves, variables, strings, symbolic paths, and pictures. Both of these invoke the Rename Objects dialog.

Graphs, tables, page layouts, notebooks, control panels, Gizmo windows, and XOP target windows are renamed using the **DoWindow** operation (see page V-168) which you can build using the Window Control dialog (see **The Window Control Dialog** on page II-49).

You can use the Data Browser to rename data folders, waves, and variables. See **The Data Browser** on page II-114.

Object Name Conflicts

In general, Igor does not allow two objects in the same namespace to have the same name. (To see a list of Igor namespaces, see **Namespaces** on page III-502.) The rest of this section discusses some exceptions to this rule and related issues. Most users will not need to know this material.

Object Name Conflicts That Igor Allows

Igor allows you to create name conflicts between waves, variables, and data folders with certain exceptions listed below. For example:

```
Function DemoAllowedNameConflicts()     // Execute this in a new experiment
   Make/O root:test                     // to demonstrate that Igor allows
   Variable/G root:test                 // name conflicts in some cases
   NewDataFolder/O root:test
End
```

This shows that Igor allows you to create a wave and a global variable with the same name and to create a data folder with the same name as a wave or a global variable.

Allowing a wave and a global variable to have the same name is a historical accident that should not have been allowed because waves and variables are in the same namespace, called the "main namespace". This may be disallowed in a future version of Igor.

Allowing a data folder to have the same name as a wave or variable is intended because data folders are in their own namespace.

The following exceptions specify the name conflicts in the main namespace that are not allowed:

1. It is an error to create a variable with the same name as an existing wave from a macro, proc, or from the command line (but it is not an error from a user-defined function as shown above).

2. It is an error to create a wave with the same name as an existing variable from any context (function, macro, proc, or from the command line).

The table below expresses those rules. The left column shows the type of object created first and the top row shows the type of object created second.

F means that Igor returns an error if you attempt to create the objects in a function.

M means that Igor returns an error if you attempt to create the objects in a macro, proc, or from the command line.

--- means Igor returns no error from a function, macro, proc, or from the command line.

	Wave	**Variable**	**Data Folder**
Wave	N/A	M	---
Variable	F, M	N/A	---
Data Folder	---	---	N/A

Object Name Conflicts and HDF5 Files

When saving an experiment in packed or unpacked format, these name conflicts do not cause a problem. If you save and reload the experiment, you get back to where you started.

However, name conflicts do cause problems with HDF5 files, saved via Save Experiment or via the HDF5SaveGroup operation, because HDF5 does not allow a group and a dataset to have the same name.

- If there is a conflict between a wave or variable and a data folder, you get an error when you attempt to save as HDF5.

- If there is a conflict between a wave and a variable, if overwrite is off, you get an error; if overwrite is on, the dataset for the variable overwrites the dataset for the wave so the wave is not saved to the resulting HDF5 file. In Igor Pro 9.00, when writing an HDF5 packed experiment file, Igor turned overwrite on; in Igor Pro 9.01 we changed overwrite to off so that an error will be flagged in this situation.

It is conceptually possible for Igor to handle these conflicts by changing the name of one of the conflicting objects when the file is saved, somehow marking the fact that this change was made, and reversing the process when the file is loaded. When we tried to implement this scheme, we found that it added a significant amount of complexity to already complex code. Adding complexity introduces the possibility of creating new bugs and slowing operations down. Since these name conflicts are rare, the downside of implementing such a workaround outweighed the benefits so we decided to decline to support name conflicts in HDF5 files.

Graphics Technology

As of version 7, Igor Pro is based on the cross-platform Qt framework and, by default, Igor uses Qt for graphics. However, for special purposes, Igor provides access to platform-native graphics.

You should avoid native graphics and stick with Qt graphics if possible because it is the focus for future development. Native graphics is provided mainly for emergency use.

You can select native graphics on a global basis (all windows are affected) in two ways:

1. From the Miscellaneous category of the Miscellaneous Settings dialog (Misc→Miscellaneous Settings menu item).
2. By executing

   ```
   SetIgorOption GraphicsTechnology=n
   ```

   where *n* is interpreted as follows:

n=0	Default (currently Qt)
n=1	Quartz on Macintosh, GDI+ on Windows
n=2	Not used on Macintosh, GDI on Windows
n=3	Qt

   Unlike most other SetIgorOption cases, this change is saved to preferences on disk and applies to future Igor sessions.

In addition to changing the global graphics technology setting, you can change individual windows using

```
SetWindow winName, graphicsTech=n
```

where n is the same as above. *winName* can be kwTopWin or the actual name of a window. Currently this setting is saved for graphs only but that is subject to change.

Over time, it is our intention that Qt graphics will replace the other technologies and the SetIgorOption GraphicsTechnology option may no longer be supported.

Graphics Technology on Windows

In general, you should use the default graphics (GraphicsTechnology=0) which is currently Qt graphics. If you experience problems with graphics, you might try GraphicsTechnology=1 (GDI+) or 2 (GDI). High-resolution displays are supported using Qt graphics only.

On Windows, GraphicsTechnology=1 utilizes the GDI+ interface. This provides support for transparency and other advanced features. Unfortunately, GDI+ is very slow, especially when rendering text.

If speed is an issue, you can try GraphicsTechnology=2 to use the older GDI interface. However, transparency is not honored - colors will be opaque regardless of the alpha value (see **Color Blending** on page III-498 for a discussion of alpha).

In GDI+ mode, EMF export uses a hybrid format, called "dual EMF", that contains both GDI+ and, for compatibility, the older GDI. In GraphicsTechnology=2 mode, only the older GDI-only style is exported. In Qt graphics mode, GDI+ is used for rendering the EMF for export, resulting in the dual EMF format. Some applications don't work well with EMF+ or dual EMF. In that case, try setting GraphicsTechnology=2 (old GDI mode) to export an EMF without the EMF+ component.

Our experience indicates that plain EMF is rendered better by GDI mode. In Qt graphics, the best rendering technology is chosen based on the contents of the picture you are pasting.

Regardless of the selected technology, exporting as EMF on Windows uses native interfaces.

Graphics Technology on Macintosh

On Macintosh, there is only one native format, Core Graphics, also known as Quartz (GraphicsTechnology=1), and it works the same as in Igor Pro 6.3. The native picture format is PDF.

High-resolution screen graphics on Retina displays is supported in Qt graphics technology mode only.

PDF pictures pasted into Igor are also rendered using Quartz. This produces the same picture regardless of the chosen Igor graphics technology, except on a Retina display, in which case the PDF is rendered on-screen at high-resolution only when using Qt graphics.

SVG Graphics

In Qt mode (GraphicsTechnology=3), SVG (Scalable Vector Graphics) is available as an alternative object-based picture format. This can replace EMF and PDF whenever the destination program supports it. In Qt mode, other imported object-based pictures (PDF on Macintosh, EMF on Windows) are rendered as high-resolution bitmap images.

In native graphics modes, imported SVG pictures are rendered as high-resolution bitmap images using Qt graphics.

Regardless of the selected technology, exporting as SVG format is rendered using Qt.

High-Resolution Displays

High-resolution screen graphics is supported in Qt graphics technology mode only.

Macintosh Retina and Windows high-resolution displays, also called Retina, 4K, 5K, Ultra HD, and Quad HD, bring a more pleasing visual experience but also present performance challenges. Depending on the operating system, graphics technology, and your actual data, graph updates can be thousands of times slower. This situation is expected to be temporary as operating systems optimize high-resolution graphics. See the "Graphs and High-Resolution Displays" topic in the release notes for the latest information.

On Windows, prior to Igor Pro 7, control panels were drawn using pixels for coordinates. Now, when the resolution exceeds 96 DPI, those coordinates are taken to be points. This prevents panels from appearing tiny on high-resolution displays. See **Control Panel Resolution on Windows** on page III-456 for more information.

Graphs and High-Resolution Displays

Macintosh Retina and Windows high-resolution displays bring a more pleasing visual experience but also present performance challenges. Previously, one point lines were one pixel wide. Now, on high-resolution displays, such lines are two pixels wide and, depending on the operating system, graphics technology and the actual data, drawing can be thousands of times slower.

Since Igor Pro 8, we now use fast custom code when drawing lines that are two or more pixels wide. However, because the custom code lines are not as pleasing as the system code, it is used only under conditions where the system code is slow, particularly when very long traces are used.

When you need the fastest update speed, you can specify that a trace should use the fast line draw code by using the ModifyGraph live keyword with a value of 2. This will cause even one pixel thick lines to be drawn with the new code. The speed improvement is on the order of a factor of two.

You can use the Live Mode demo experiment to see how different settings affect the speed of graph updates.

Windows High-DPI Recommendations

High-DPI (dots-per-inch) displays have resolutions substantially higher than the Windows standard 96 DPI. These displays sometimes go by the names Retina, 4K, 5K, Ultra HD and Quad HD.

Since the pixels of a high-DPI monitor are typically about one-half the size of pixels on standard monitors, software running on a high-DPI monitor must make adjustments. These adjustments are made by some combination of the operating system and the program itself.

On Macintosh, with its “Retina” high-DPI screens, the operating system handles most resolution issues and there are few problems with Igor on Retina monitors.

On Windows, both the operating system and the Qt framework that Igor uses introduce complexities that sometimes result in less than perfect behavior. This section provides guidance for Windows user to minimize these issues.

To get the best experience when using a high-DPI display, we recommend that you use Windows 10 or later. We do not test or explicitly support high-DPI features on older Windows versions.

If you only have one display, such as a laptop with a high-DPI display, and no external displays are connected, you should not need to make any changes to Igor's default configuration to achieve the correct behavior.

If you have multiple displays with different resolutions, such as a high-resolution laptop display and a standard-resolution external monitor, you may see problems such as text, menus, windows or icons too big or too small. In our experience with Igor8, most or all of these issues are resolved if you make your high-resolution display your main display using the Windows 10 Display control panel.

As an example, here are instructions for configuring a typical mixed-resolution system - a laptop with a built-in high-resolution display and an external standard-resolution monitor:

1. Connect the external standard-resolution monitor.
2. Open the Display settings page. You can do this by right-clicking the Start menu, choosing Settings, clicking System, and selecting Display in the lefthand pane.
3. In the Multiple Displays setting, near the bottom of the pane, select Extend These Displays.
4. Click the Identify link below the diagram of your displays at the top of the pane to confirm which display is which. For these instructions, we assume that display #1 is the high-resolution built-in display and display #2 is a standard-resolution external monitor.
5. Click the box representing the built-in high-resolution display (display #1 for this example).
6. Set Resolution to the recommended value, such as 3840 x 2160.
7. Set Scale and Layout to the recommended value, which may be 200%, 225%, or 250% depending on your hardware.
8. Check the Make This My Main Display checkbox near the bottom of the pane.
9. Close the settings control panel.

10. Reboot your machine twice. To get the best results, you must reboot your machine 2 times after any changes to the display settings.

11. After your machine restarts twice, start Igor. Except for occasional minor glitches, it should work properly.

If the preceding instructions do not give useable results, you may find it necessary to fall back to standard resolution for the high-resolution display. To do this, set Resolution to standard resolution and set Scale and Layout to 100%. Standard resolution is usually half the recommended resolution - for example 1920 x 1080 instead of 3140 x 2160. If you do this, you can make either display the main display.

Pictures

Igor can import pictures from other programs for display in graphs, page layouts and notebooks. It can also export pictures from graphs, page layouts and tables for use in other programs. Exporting is discussed in Chapter III-5, **Exporting Graphics (Macintosh)**, and Chapter III-6, **Exporting Graphics (Windows)**. This section discusses how you can import pictures into Igor, what you can do with them and how Igor stores them.

For information on importing images as data rather than as graphics, see **Loading Image Files** on page II-157.

Importing Pictures

There are three ways to import a picture.

- Pasting from the Clipboard into a graph, layout, or notebook
- Using the Pictures dialog (Misc menu) to import a picture from a file or from the Clipboard
- Using the **LoadPICT** operation (see page V-506) to import a picture from a file or from the Clipboard

Each of these methods, except for pasting into a notebook, creates a *named*, global picture object that you can use in one or more graphs or layouts. Pasting into a notebook creates a picture that is local to the notebook.

This table shows the types of graphics formats from which Igor can import pictures:

Format	Notes
PDF (Portable Document Format)	Macintosh: Supported in native graphics only.
	Windows: Supported in Igor Pro 9.00 and later.
	See **Importing PDF Pictures**.
EMF (Enhanced Metafile)	Supported in Windows native graphics only.
	See **Graphics Technology on Windows** on page III-506 for information about different types of EMF pictures.
BMP (Windows bitmap)	Supported in on Windows only.
	BMP is sometimes called DIB (Device Independent Bitmap).
PNG (Portable Network Graphics)	Lossless cross-platform bitmap format
JPEG	Lossy cross-platform bitmap format
TIFF (Tagged Image File Format)	Lossless cross-platform bitmap format

Format	Notes
EPS (Encapsulated PostScript)	High resolution vector format. Requires PostScript printer. On Windows and on Macintosh in Qt graphics mode, a screen preview is displayed on screen.
SVG (Scalable Vector Graphics)	Cross-platform vector and bitmap format. Always rendered using Qt graphics. In other graphics technology modes, it is drawn into a high-resolution btmap which is then drawn on the screen, exported, or printed.

Formats that are not supported on the current platform are drawn as gray boxes.

See also **Picture Compatibility** on page III-449 for a discussion of Macintosh graphics on Windows and vice-versa.

Importing PDF Pictures

PDF (Portable Document Format) is Adobe's platform-independent vector graphics format. However, not all programs can import PDF.

On Macintosh, Igor has supported importing PDF graphics since Igor Pro 5. PDF pictures are using bitmaps except in Macintosh native graphics mode (see **Graphics Technology** on page III-506).

On Windows, Igor supports importing PDF graphics in Igor Pro 9 or later. PDF pictures are drawn using bitmaps.

The Picture Gallery

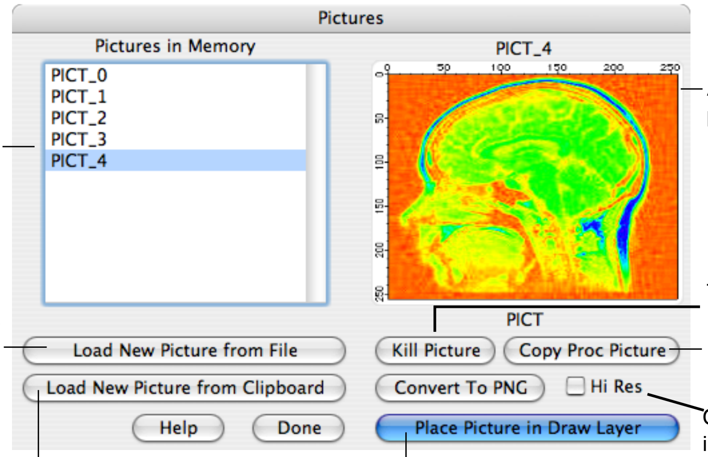
When you create a named picture using one of the techniques listed above, Igor stores it in the current experiment’s **picture gallery**. When you save the experiment, the picture gallery is stored in the experiment file. You can inspect the collection using the Pictures dialog via the Misc menu.

Igor gives names to pictures so they can be referenced from an Igor procedure. For example, if you paste a picture into a layout, Igor assigns it a name of the form “PICT_0” and stores it in the picture gallery. If you then close the layout and ask Igor to create a recreation macro, the macro will reference the picture by name.

You can rename a named picture using the Pictures dialog in the Misc menu, the Rename Objects dialog in the Misc menu, the Rename dialog in the Data menu, or the **RenamePICT** operation (see page V-797). You can kill a named picture using the Pictures dialog or the **KillPICTs** operation (see page V-470).

Pictures Dialog

The Pictures dialog permits you to view the picture gallery, to add pictures, to remove pictures and to place a picture into a graph or page layout. It also can place a copy of a picture into a formatted notebook. To invoke it, choose Pictures from the Misc menu.



Lists the named pictures in the picture gallery. Click to select a picture.

Loads a new picture from a file, usually created in a drawing program.

Loads a new picture that you have copied to the Clipboard, usually from a drawing program.

Click to place the picture in a graph or page layout.

A preview of the selected picture.

Removes the selected picture, from the collection.

Creates an ASCII representation for use in procedures.

Converts the selected picture into a platform-independent PNG format bitmap. Select the box for high resolution

The Kill This Picture button will be dimmed if the selected picture is used in a currently open graph or layout.

Note: Igor determines if a picture is in use by checking to see if it is used in an *open* graph or layout window.

If you kill a graph or layout that contains a picture and create a recreation macro, the recreation macro will *refer* to the picture by name. However, Igor does not check for this. It will consider the picture to be unused and will allow you to kill it. If you later run the recreation macro, an error will occur when the macro attempts to append the picture to the graph or layout. Therefore, don't kill a picture unless you are sure that it is not needed.

NotebookPictures

When you paste a picture into a formatted notebook, you create a notebook picture. These work just like pictures in a word processor document. You can copy and paste them. These pictures will not appear in the Pictures or Rename Objects dialogs.

Igor Extensions

Igor includes a feature that allows a C or C++ programmer to extend its capabilities. An Igor extension is called an "XOP" (short for "external operation"). The term XOP comes from that fact that, originally, adding a command line operation was all that an extension could do. Now extensions can add operations, functions, menus, dialogs and windows.

XOPs come in 32-bit and 64-bit varieties. Because almost all Igor users now run the 64-bit version of Igor, this section focuses on 64-bit XOPs.

WaveMetrics XOPs

The "Igor Pro Folder/Igor Extensions (64-bit)" and "Igor Pro Folder/More Extensions (64-bit)" folders contain XOPs that we developed at WaveMetrics. These add capabilities such as file-loading and instrument control to Igor and also serve as examples of what XOPs can do. These XOPs range from very simple to rather elaborate. Most XOPs come with help files that describe their operation.

The WaveMetrics XOPs are described in the XOP Index help file, accessible through the Help→Help Windows submenu.

Third Party Extensions

A number of Igor users have written XOPs to customize Igor for their particular fields. Some of these are freeware, some are shareware and some are commercial programs. WaveMetrics publicizes third party XOPs through our Web page. User-developed XOPs are available from http://www.igorexchange.com.

See **Creating Igor Extensions** on page IV-208 if you are a programmer interested in writing your own XOPs.

Activating 64-bit Extensions

To activate a 64-bit extension, you put it, or an alias or shortcut pointing to it, in the "Igor Extensions (64-bit)" folder in your "Igor Pro User Files" folder.

For illustration purposes, we show here the steps you need to take to activate Igor's SQL XOP which provides access to databases.

1. In Igor, choose Help->Show Igor Pro User Files to open your "Igor Pro User Files" folder on the desktop.

2. In Igor, choose Help->Show Igor Pro Folder to open your "Igor Pro Folder" on the desktop.

3. Open the "Igor Pro Folder\More Extensions (64-bit)\Utilities" folder on the desktop.

4. Make an alias (*Macintosh*) or shortcut (*Windows*) for "SQL64.xop" and put it in the "Igor Extensions (64-bit)" folder inside your "Igor Pro User Files" folder.

5. Restart Igor64.

If you want to activate an XOP for all users on a given machine, you can put the alias or shortcut in the "Igor Extensions (64-bit)" folder inside your Igor Pro Folder. You may need to run as administrator to do this.

Changes that you make to either "Igor Extensions (64-bit)" folder take effect the next time Igor is launched.

Activating 32-bit Extensions

If you are running the 32-bit version of Igor on Windows, you can activate the 32-bit version of an XOP following the instructions in the preceding section with a few modifications.

Use "Igor Extensions" instead of "Igor Extensions (64-bit)" and "More Extensions" instead of "More Extensions (64-bit)". Activate the 32-bit version of the XOP, for example SQL.xop, instead of SQL64.xop

XOPs on MacOS 10.15 and Later

In MacOS 10.15 (Catalina), Apple added strict security features that prevent downloaded XOPs from running without special security certification ("notarization").

WaveMetrics XOPs that ship with Igor Pro 8.04 or later have the required notarization and so work.

XOPs that you compiled on your own machine are likely to run correctly unless they depend on libraries that are not compatible with Catalina.

Most third-party XOPs that you downloaded will not run on Catalina.

For further information, see https://www.wavemetrics.com/news/igor-pro-macos-1015-catalina and https://www.wavemetrics.com/node/21088 for a workaround.

Memory Management

Igor comes in 32-bit and 64-bit versions called IGOR32 and IGOR64 respectively. IGOR32 is available on Windows only. The only reason to run IGOR32 is if you depend on 32-bit XOPs that have not been ported to 64 bits.

IGOR32 can theoretically address 4GB of virtual address space. For the vast majority of Igor applications, this is more than sufficient. If you must load gigabytes of data into memory at one time, you may run out of memory. This may happen long before you load 4GB of data into memory, because, to allocate a wave for example, you not only need free memory, but the free memory must also be continguous, and memory becomes fragmented over time.

IGOR64 can theoretically address about a billion gigabytes. However, actual operating systems impose far lower limits. On Windows 10, 64-bit programs can address between 128 GB (home edition) and 512 GB (professional edition). On Mac OS X, 64-bit programs can theoretically address the full 64-bit address space.

If you load more data than fits in physical memory, the system starts using "virtual memory", meaning that it swaps data between physical memory and disk, as needed. This is very slow. Consequently, you should avoid loading more data into memory than can fit in physical memory.

Even if your data fits in physical memory, graphing and manipulating very large waves, such as 10 million, 100 million, or 1 billion points, will be slow.

All of this boils down to the following rules:

1. If you don't need to load gigabytes of data into memory at one time then you don't need to worry about memory management.

2. Run IGOR64 unless you rely on 32-bit XOPs that can not be ported to 64 bits. If you are running on Macintosh and rely on 32-bit XOPs, you must run Igor7.

3. Install enough physical memory to avoid the need for virtual memory swapping.

For further information about very large waves, see **IGOR64 Experiment Files** on page II-35.

Macintosh System Requirements

Igor Pro requires Mac OS X 10.9.0 or later.

Windows System Requirements

Igor Pro requires Windows 7 or later.

Chapter III-18

Preferences

Overview

Preferences affect the creation of *new* graphs, panels, tables, layouts, notebooks, and procedure windows, and the *appending* of traces to graphs and columns to tables. In addition, preferences affect the command window, and default font for graphs.

You can turn preferences off or on using the Misc menu. Normally you will run with preferences on.

Preferences are automatically off while a procedure is running so that the effects of the procedure will be the same for all users. See the **Preferences** operation (see page V-768) for further information.

When preferences are off, factory default values are used for settings such as graph size, position, line style, size and color. When preferences are on, Igor applies your preferred values for these settings.

Preferences differ from *settings*, as set in the Miscellaneous Settings dialog, in that settings generally take effect immediately, while preferences are used when something is created.

Igor Preferences Directory

Igor preferences are stored in a per-user directory. The location of this directory depends on your operating system. You generally don't need to access your Igor preferences directory, but you can open it in the desktop by pressing the Shift key while choosing Help→Show Igor Preferences Folder.

You can also determine the location of your Igor preferences directory by executing this command:

```
Print SpecialDirPath("Preferences", 0, 0, 0)
```

For technical reasons, most of the built-in preferences are stored not in the Igor preferences directory but rather one level up in a file named "Igor Pro 9.ini".

Deleting the preferences directory and the "Igor Pro 9.ini" file effectively reverts all preferences to factory defaults. You should use the Capture Prefs dialogs, described in **Captured Preferences** operation on page III-516, to revert preferences more selectively.

Other information is stored in preferences, such as the screen position of dialogs, a few dialog settings, colors recently selected in the color palette, window stacking and tiling information, page setups, font substitution settings, and dashed line settings.

How to Use Preferences

Preferences are always on when Igor starts up. You can turn preferences off by choosing Preferences Off in the Misc menu.

You can also turn preferences on and off with the **Preferences** operation (see page V-768).

Preferences are set by Capture Preferences dialogs, the Tile or Stack Windows dialog, and some dialogs such as the Dashed Lines dialog.

In general, *preferences are applied only when something new is created* such as a new graph, a new trace in a graph, a new notebook, a new column in a table, and then only if preferences are on.

Preferences are normally in effect only for *manual* ("point-and-click") operation, not for user-programmed operations in Igor procedures. See **Procedures and Preferences** on page IV-203.

Captured Preferences

You set most preference values by capturing the current settings of the active window with the Capture Prefs item in the menu for that window. (The dialog to capture the Command Window preferences is found in the Command/History Settings submenu of the Misc menu.) The dialogs are described in more detail in the chapter that discusses each type of window. For instance, see **Graph Preferences** on page II-348.

Volume IV Programming

Table of Contents

IV-1 Working with Commands IV-1
IV-2 Programming Overview IV-23
IV-3 User-Defined Functions IV-29
IV-4 Macros IV-115
IV-5 User-Defined Menus IV-125
IV-6 Interacting with the User IV-143
IV-7 Programming Techniques IV-165
IV-8 Debugging IV-211
IV-9 Dependencies IV-229
IV-10 Advanced Topics IV-235

Chapter

IV-1

Working with Commands

Overview

You can execute commands by typing them into the command line and pressing Return or Enter.

You can also use a notebook for entering commands. See **Notebooks as Worksheets** on page III-4.

You can type commands from scratch but often you will let Igor dialogs formulate and execute commands. You can view a record of what you have done in the history area of the command window and you can easily reenter, edit and reexecute commands stored there. See **Command Window Shortcuts** on page II-13 for details.

Multiple Commands

You can place multiple commands on one line if you separate them with semicolons. For example:

```
wave1= x; wave1= wave2/(wave1+1); Display wave1
```

You don’t need a semicolon after the last command but it doesn’t hurt.

Comments

Comments start with //, which end the executable part of a command line. The comment continues to the end of the line.

Maximum Length of a Command

The total length of the command line can not exceed 2500 bytes.

There is no line continuation character in the command line. However, it is nearly always possible to break a single command up into multiple lines using intermediate variables. For example:

```
Variable a = sin(x-x0)/b + cos(y-y0)/c
```

can be rewritten as:

```
Variable t1 = sin(x-x0)/b
Variable t2 = cos(y-y0)/c
Variable a = t1 + t2
```

Parameters

Every place in a command where Igor expects a numeric parameter you can use a numeric expression. Similarly for a string parameter you can use a string expression. In an operation flag (e.g., /N=<number>), you must parenthesize expressions. See **Expressions as Parameters** on page IV-12 for details.

Liberal Object Names

In general, object names in Igor are limited to a restricted set of characters. Only letters, digits and the underscore character are allowed. Such names are called “standard names”. This restriction is necessary to identify where the name ends when you use it in a command.

For waves and data folders only, you can also use “liberal” names. Liberal names can include almost any character, including spaces and dots (see **Liberal Object Names** on page III-501 for details). However, to define where a liberal name ends, you must quote them using single quotes.

In the following example, the wave names are liberal because they include spaces and therefore they must be quoted:

```
'wave 1' = 'wave 2'      // Right
wave 1 = wave 2          // Wrong - liberal names must be quoted
```

(This syntax applies to the command line and macros only, not to user-defined functions in which you must use **Wave References** to read and write waves.)

Note: Providing for liberal names requires extra effort and testing by Igor programmers (see **Programming with Liberal Names** on page IV-168) so you may occasionally experience problems using liberal names with user-defined procedures.

Commands and Data Folders

Data folders provide a way to keep separate sets of data from interfering with each other. You can examine and create data folders using the Data Browser (Data menu). There is always a root data folder and this is the only data folder that many users will ever need. Advanced users may want to create additional data folders to organize their data.

You can refer to waves and variables either in the current data folder, in a specific data folder or in a data folder whose location is relative to the current data folder:

```
// wave1 is in the current data folder
wave1 = <expression>

// wave1 is in a specific data folder
root:'Background Curves':wave1 = <expression>

// wave1 is in a data folder inside the current data folder
:'Background Curves':wave1 = <expression>
```

(This syntax applies to the command line and macros only, not to user-defined functions in which you must use **Wave References** to read and write waves.)

In the first example, we use an object name by itself (wave1) and Igor looks for the object in the current data folder.

In the second example, we use a full data folder path (root:'Background Curves':) plus an object name. Igor looks for the object in the specified data folder.

In the third example, we use a relative data folder path (:'Background Curves':) plus an object name. Igor looks in the current data folder for a subdata folder named Background Curves and looks for the object within that data folder.

Important: The right-hand side of an assignment statement (described under **Assignment Statements** on page IV-4) is evaluated in the context of the data folder containing the destination object. For example:

```
root:'Background Curves':wave1 = wave2 + var1
```

For this to work, wave2 and var1 must be in the Background Curves data folder.

Examples in the rest of this chapter use object names alone and thus reference data in the current data folder. For more on data folders, see Chapter II-8, **Data Folders**.

Command Tooltips

In procedure windows and the command window, if you hover the mouse cursor over a command, such as an operation or function, Igor displays a tooltip that contains information about the command. If you hover over the name of a built-in structure, such as WMButtonAction, Igor displays the structure definition.

You can turn off this feature by choosing Misc→Miscellaneous Settings, selecting the Text Editing section, selecting the Editor Behavior tab, and unchecking the Show Context-sensitive Tooltips checkbox.

Command Completion

Procedure windows and the command line support automatic command completion.

When you type the first few letters of a command, a popup appears and allows you to quickly select a command. Use the arrow keys to move up and down the list of completion options. Press the Enter key or Tab key to insert the highlighted text in the document.

If you hover the mouse cursor over a completion option or use the arrow keys to change the highlighted option, a tool tip is displayed that shows the template of the selected option.

You can adjust Command completion settings in Completion tab of the Text Editing category of the Miscellaneous Settings dialog. You can separately enable or disable completion in procedure windows and the command line. You can control the delay between the last keypress and the display of the completion options popup and whether an opening parenthesis should be appended when you insert an item that requires parentheses.

Completion is currently not supported for objects such as waves and variables. Support for these types may be added in a future version of Igor.

Command completion is not context sensitive. This means that you will sometimes be offered completion options that do not make sense in the context of the text you are entering. For example, if you type "Display/HID", you might get the following completion options: HideIgorMenus, HideInfo, HideProcedures, HideTools". Those options are not valid as a flag for the Display operation, but the command completion algorithm isn't able to filter them out.

Types of Commands

There are three fundamentally different types of commands that you can execute from the command line:

- assignment statements
- operation commands
- user-defined procedure commands

Here are examples of each:

```
wave1 = sin(2*pi*freq*x)       // assignment statement

Display wave1,wave2 vs xwave   // operation command

MyFunction(1.2,"hello")        // user-defined procedure command
```

As Igor executes commands you have entered, it must determine which of the three basic types of commands you have typed. If a command starts with a wave or variable name then Igor assumes it is an assignment statement. If a command starts with the name of a built-in or external operation then the command is treated as an operation. If a command begins with the name of a user-defined macro, user-defined function or external function then the command is treated accordingly.

Note that built-in functions can only appear in the right-hand side of an assignment statement, or as a parameter to an operation or function. Thus, the command:

```
sin(x)
```

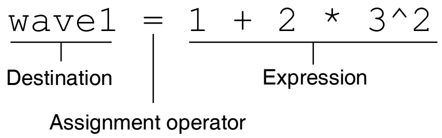
is not allowed and you will see the error, “Expected wave name, variable name, or operation.” On the other hand, these commands are allowed:

```
Print sin(1.567)               // sin is parameter of print operation
wave1 = 5*sin(x)               // sin in right side of assigment
```

If, perhaps due to a misspelling, Igor can not determine what you want to do, it will display an error dialog and the error will be highlighted in the command line.

Assignment Statements

Assignment statement commands start with a wave or variable name. The command assigns a value to all or part of the named object. An assignment statement consists of three parts: a destination, an assignment operator, and an expression. For example:



This assigns 19 to every point in wave1.

The spaces in the above example are not required. You could write:

```
wave1=1+2*3^2
```

See **Waveform Arithmetic and Assignments** on page II-74 for details on wave assignment statements.

In the following examples, str1 is a string variable, created by the String operation, var1 is a numeric variable, created by the Variable operation, and wave1 is a wave, created by the Make operation.

```
str1 = "Today is " + date()                // string assignment

str1 += ", and the time is " + time()      // string concatenation

var1 = strlen(str1)                        // variable assignment

var1 = pnt2x(wave1,numpnts(wave1)/2)       // variable assignment

wave1 = 1.2*exp(-0.2*(x-var1)^2)           // wave assignment

wave1[3] = 5                               // wave assignment

wave1[0,;3]= wave2[p/3] *exp(-0.2*x)       // wave assignment
```

These all operate on objects in the current data folder. To operate on an object in another data folder, you need to use a data folder path:

```
root:'run 1':wave1[3] = 5         // wave assignment
```

(This syntax applies to the command line and macros only, not to user-defined functions in which you must use **Wave References** to read and write waves.)

See Chapter II-8, **Data Folders**, for further details.

If you use liberal wave names (see **Object Names** on page III-501), you must use quotes:

```
'wave 1' = 'wave 2'          // Right

wave 1 = wave 2              // Wrong
```

(This syntax applies to the command line and macros only, not to user-defined functions in which you must use **Wave References** to read and write waves.)

Assignment Operators

The assignment operator determines the way in which the expression is combined with the destination. Igor supports the following assignment operators:

Operator	Assignment Action
=	Destination contents are set to the value of the expression.
+=	Expression is added to the destination.
-=	Expression is subtracted from the destination.
*=	Destination is multiplied by the expression.
/=	Destination is divided by the expression.
:=	Destination is dynamically updated to the value of the expression whenever the value of any part of the expression changes. The := operator is said to establish a “dependency” of the destination on the expression.

For example:

```
wave1 = 10
```

sets each Y value of the wave1 equal to 10, whereas:

```
wave1 += 10
```

adds 10 to each Y value of wave1. This is equivalent to:

```
wave1 = wave1 + 10
```

The assignment operators =, := and += work with string assignment statements but -=, *= and /= do not. For example:

```
String str1; str1 = "Today is "; str1 += date(); Print str1
```

prints something like “Today is Fri, Mar 31, 2000”.

For more information on the := operator, see Chapter IV-9, **Dependencies**.

Operators

Here is a complete list of the operators that Igor supports in the expression part of an assignment statement in order of precedence:

Operator	Effect
`++ --`	Prefix and postfix increment and decrement. Require Igor Pro 7 or later. Available only for local variables in user-defined functions.
`^ << >>`	Exponentiation, bitwise left shift, bitwise right shift. Shifts require Igor Pro 7 or later.
`- ! ~`	Negation, logical complement, bitwise complement.
`* /`	Multiplication, division.
`+ -`	Addition or string concatenation, subtraction.
`== != > < >= <=`	Comparison operators.
`& | %^`	Bitwise AND, bitwise OR, bitwise XOR.
`&& || ? :`	Logical AND, logical OR, conditional operator.
`$`	Substitute following string expression as name.

Comparison operators do not work with NaN parameters because, by definition, NaN compared to anything, even another NaN, is false. Use numtype to test if a value is NaN.

Comparison operators, bitwise AND, bitwise OR, bitwise XOR associate right to left. Therefore a==b>=c means (a==(b>=c)). For example, 2==1>= 0 evaluates to 0, not 1.

All other binary operators associate left to right.

Aside from the precedence of common operators that everyone intuitively knows, it is useful to include parentheses to avoid confusion. You may know the precedence and associativity but someone reading your code may not.

Unary negation changes the sign of its operand. Logical complementation changes nonzero operands to zero and zero operands to 1. Bitwise complementation converts its operand to an unsigned integer by truncation and then changes it to its binary complement.

Exponentiation raises its left-hand operand to a power specified by its right-hand operand. That is, 3^2 is written as 3^2. In an expression a^b, if the result is assigned to a real variable or wave, then a must not be negative if b is not an integer. If the result is used in a complex expression, any combination of negative a, fractional b or complex a or b is allowed.

If the exponent is an integer, Igor evaluates the expression using only multiplication. There is no need to write a^2 as a*a to get efficient evaluation — Igor does the equivalent automatically. If, on the other hand, the exponent is not an integer then the evaluation is performed using logarithms, hence the restriction on negative a in a real expression.

Logical OR (||) and logical AND (&&) determine the truth or falseness of pairs of expressions. The AND operation returns true only when both expressions are true; OR will return true if *either* is true. As in C, true is *any* nonzero value, and false is zero. The operations are undefined for NaNs. These operators are not available in complex expressions.

The logical operators are evaluated from left to right, and an operand will not be evaluated if it is not necessary. For the example:

```
if(MyFunc1() && MyFunc2())
```

when MyFunc1() returns false (zero), then MyFunc2() will not be evaluated because the entire expression is already false. This can produce unexpected consequences when the right-hand expression has side effects, such as creating waves or setting global values.

Bitwise AND (&), OR (|), and XOR (%^) convert their operands to an unsigned integer by truncation and then return their binary AND, OR or exclusive OR.

Bitwise shifts, << and >>, return the lefthand operand shifted by the specified number of bits. They require Igor Pro 7 or later. The lefthand operation is truncated to an integer before shifting.

The conditional operator (? :) is a shorthand form of an if-else-endif expression. In the statement:

```
<expression> ? <TRUE> : <FALSE>
```

the first operand, <expression>, is the test condition; if it is nonzero then Igor evaluates the <TRUE> operand; otherwise <FALSE> is evaluated. Only one operand is evaluated according to the test condition. This is the same as if you had written:

```
if( <expression> )
   <TRUE>
else
   <FALSE>
endif
```

The “:” character in the conditional operator must always be separated from the two adjacent operands with a space. If you omit either space, you will get an error (“No such data folder”) because the expression can also be interpreted as part of a data folder path. To be safe, always separate the operands from the operator symbols with a space.

The operands must be numeric; for strings, use the **SelectString** function. When using complex expressions with the conditional operator, only the real portion is used when the operator evaluates the expression.

The conditional operator can easily cause confusion, so you should exercise caution when using it. For example, it is unclear from simple inspection what Igor may return for

```
1 ? 2 : 3 ? 4 : 5
```

(4 in this case), whereas

```
1 ? 2 : (3 ? 4 : 5)
```

will return 2. Always use parentheses to remove any ambiguity.

The comparison operators return 1 if the result of the comparison is true or 0 if it is false. For example, the == operator returns 1 if its operands are equal or 0 if they are not equal. The != operator returns the opposite. Because comparison operators return the values 1 or 0 they can be used in interesting ways. The assignment:

```
wave1 = sin(x)*(x<=50) + cos(x)*(x>50)
```

sets wave1 so that it is a sine wave below x=50 and a cosine wave above x=50. See also **Example: Comparison Operators and Wave Synthesis** on page II-82.

Note that the double equal sign, ==, is used to mean equality while the single equal sign, =, is used to indicate assignment.

Because of roundoff error, using == to test two numbers for equality may give incorrect results. It is safer to use <= and >= to see if a number falls in a narrow range. For example, imagine that you want to compare a variable to see if it is equal to one-third. The expression:

```
(v1 == 1/3)
```

is subject to failure because of roundoff. It is safer to use something like

```
((v1 > .33332) && (v1 < .33334))
```

If the numbers are integers then the use of == is safe because integers smaller than 2^{53} (approximately 10^{16}) are represented precisely in double-precision floating point numbers.

The previous discussion on operators has assumed numeric operands. The + operator is the only one that works with both numeric *and* string operands. For example, if str1 is a string variable then the assignment statement

```
str1 = "Today is " + "a nice day"
```

assigns the value “Today is a nice day” to str1. The other string operator, $ is discussed in **String Substitution Using $** on page IV-18.

Unless specified otherwise by parentheses, unary negation or complementation are carried out first followed by exponentiation then by multiplication or division followed by addition or subtraction then by comparison operators. The wave assignment:

```
wave1 = ((1 + 2) * 3) ^ 2
```

assigns the value 81 to every point in wave1, but

```
wave1 = 1 + 2 * 3 ^ 2
```

assigns the value 19.

`-a^b` is an exception to this rule and is evaluated as `-(a^b)`.

The precedence of string substitution, substrings, and wave indexing is somewhat complex. When in doubt, use parenthesis to enforce the precedence you want.

Obsolete Operators

As of Igor Pro 4.0, the old bitwise complement (%~), bitwise AND (%&), and bitwise OR (%|) operators have been replaced by new versions that omit the % character from the operator. These old bitwise operators can still be used interchangeably with the new versions but are not recommended for new code.

Operands

In addition to literal numbers like 3.141 or 27, operators can operate on variables and function values. In the assignment statement:

```
var1 = log(3.7) + var2
```

the operator + operates on the function value returned by log and on the variable var2. Functions and function values are discussed later in this chapter.

Numeric Type

In Igor, each numeric destination object (wave or variable) has its own numeric type. The numeric type consists of the numeric precision (e.g., double precision floating point) and the number type (real or complex). Waves can be single or double precision floating point or various sizes of integer but variables are always double precision floating point.

The numeric precision of the destination does not affect the calculations. With the exception of a few operations that are done in place such as the FFT, all calculations are done in double precision.

Although waves can have integer numeric types, wave expressions are always evaluated in double precision floating point. The floating point values are converted to integers by rounding as the final step before storing the value in the wave. If the value to be stored exceeds the range of values that the given integer type can represent, the results are undefined.

Starting with Igor Pro 7, Igor supports integer local variables and 64-bit integer waves. When one of these is the destination of an assignment statement, Igor performs calculations using integer arithmetic. See **Expression Evaluation** on page IV-38 for details.

The number type of the destination determines the initial number type (real or complex) of the assignment expression. This is important because Igor can not deal with "surprise" or "runtime" changes in number type. An example would be taking the square root of a negative number requiring that all following arithmetic be done using complex numbers.

Here are some examples:

```
Variable a, b, c, var1
Variable/C cvar1
Make wave1

var1= a*b
cvar1= c*cmplx(a+1,b-1)
wave1= var1 + real(cvar1)
```

The first expression is evaluated using the real number type. The second expression contains a mixture of two types. The multiplication of c with the result of the cmplx function is evaluated as complex while the arguments to the cmplx function are evaluated as real. The third example is evaluated as real except for the argument to the real function which is evaluated as complex.

Constant Type

You can define named numeric and string constants in Igor procedure files and use them in the body of user-defined functions.

Constants are defined in procedure files using following syntax:

```
Constant <name1> = <literal number> [, <name2> = <literal number>]

StrConstant <name1> = <literal string> [, <name2> = <literal string>]
```

You can use the static prefix to limit the scope to the given source file. For debugging, you can use the Override keyword as with functions.

These declarations can be used in the following ways:

```
Constant kFoo=1,kBar=2
StrConstant ksFoo="hello",ksBar="there"

static Constant kFoo=1,kBar=2
static StrConstant ksFoo="hello",ksBar="there"

Override Constant kFoo=1,kBar=2
Override StrConstant ksFoo="hello",ksBar="there"
```

Programmers may find that using the "k" and "ks" prefixes will make their code easier to read.

Names for numeric and string constants can conflict with all other names. Duplicate constants of a given type are not allowed except for static constants in different files and when used with Override. The only true conflict is with variable names and with certain built-in functions that do not take parameters such as pi. Variable names override constants, but constants override functions such as pi.

Dependency Assignment Statements

You can set up global variables and waves so that they automatically recalculate their contents when other global objects change. See Chapter IV-9, **Dependencies**, for details.

Operation Commands

An operation is a built-in or external routine that performs an action but, unlike a function, does not directly return a value. Here are some examples:

```
Make/N=512 wave1
Display wave1
Smooth 5, wave1
```

Operation commands perform the majority of the work in Igor and are automatically generated and executed as you work with Igor using dialogs.

You can use these dialogs to experiment with operations of interest to you. As you click in a dialog, Igor composes a command. This provides a handy way for you to check the syntax of the operation or to generate a command for use in a user-defined procedure. See Chapter V-1, **Igor Reference**, for a complete list of all built-in operations. Another way to learn their syntax is to use the Igor Help Browser's Command Help tab.

The syntax of operation commands is highly variable but in general consists of the operation name, followed by a list of flags (e.g., `/N=512`), followed by a parameter list. The operation name specifies the main action of the operation and determines the syntax of the rest of the command. The list of flags specifies variations on the default behavior of the operation. If the default behavior of the operation is satisfactory then no flags are required. The parameter list identifies the objects on which the operation is to operate. Some commands take no parameters. For example, in the command:

```
Make/D/N=512 wave1, wave2, wave3
```

the operation name is "`Make`". The list of flags is "`/D/N=512`". The parameter list is "`wave1, wave2, wave3`".

You can use numeric expressions in the parameter list of an operation where Igor expects a numeric parameter, but in an operation flag you need to parenthesize the expression. For example:

```
Variable val = 1.0
Make/N=(val) wave0, wave1
Make/N=(numpnts(wave0)) wave2
```

The most common types of parameters are literal numbers or numeric expressions, literal strings or string expressions, names, and waves. In the example above, wave1 is a name parameter when passed to the Make operation. It is a wave parameter when passed to the Display and Smooth operations. A name parameter can refer to a wave that may or may not already exist whereas a wave parameter must refer to an existing wave.

See **Parameter Lists** on page IV-11 for general information that applies to all commands.

User-Defined Procedure Commands

User-defined procedure commands start with a procedure name and take a list of parameters in parentheses. Here are a few examples:

```
MyFunction1(5.6, wave0, "igneous")
```

Macro and Function Parameters

You can invoke macros, but not functions, with one or more of the input parameters missing. When you do this, Igor displays a dialog to allow you to enter the missing parameters.

You can add similar capabilities to user-defined functions using the **Prompt** (see page V-782) and **DoPrompt** (see page V-167) keywords.

Macros provide very limited programming features so, with rare exceptions, you should program using functions.

Function Commands

A function is a routine that directly returns a numeric or string value. There are three classes of functions available to Igor users:

- Built-in
- External (XFUNCs)
- User-defined

Built-in numeric functions enjoy one advantage over external or user-defined functions: a few come in real and complex number types and Igor automatically picks the appropriate version depending on the current number type in an expression. External and user-defined functions must have different names when different types are needed. Generally, only real user and external functions need be provided.

For example, in the wave assignment:

```
wave1 = enoise(1)
```

if wave1 is real then the function enoise returns a real value. If wave1 is complex then enoise returns a complex value.

You can use a function as a parameter to another function, to an operation, to a macro or in an arithmetic or string expression so long as the data type returned by the function makes sense in the context in which you use it.

User-defined and external functions can also be used as commands by themselves. Use this to write a user function that has some purpose other than calculating a numeric value, such as displaying a graph or making new waves. Built-in functions cannot be used this way. For instance:

```
MyDisplayFunction(wave0)
```

External and user-defined functions can be used just like built-in functions. In addition, numeric functions can be used in curve fitting. See Chapter IV-3, **User-Defined Functions** and **Fitting to a User-Defined Function** on page III-190.

Most functions consist of a function name followed by a left parenthesis followed by a parameter list and followed by a right parenthesis. In the wave assignment shown at the beginning of this section, the function name is `enoise`. The parameter is 1. The parameter is enclosed by parentheses. In this example, the result from the function is assigned to a wave. It can also be assigned to a variable or printed:

```
K0 = enoise(1)
Print enoise(1)
```

User and external functions, but not built-in functions, can be executed on the command line or in other functions or macros without having to assign or print the result. This is useful when the point of the function is not its explicit result but rather its side effects.

Nearly all functions require parentheses even if the parameter list is empty. For example the function date() has no parameters but requires parentheses anyway. There are a few exceptions. For example the function Pi returns π and is used with no parentheses or parameters.

Igor's built-in functions are described in detail in Chapter V-1, **Igor Reference**.

Parameter Lists

Parameter lists are used for operations, functions, and macros and consist of one or more numbers, strings, keywords or names of Igor objects. The parameters in a parameter list must be separated with commas.

Expressions as Parameters

In an operation, function, or macro parameter list which has a numeric parameter you can always use a numeric expression instead of a literal number. A numeric expression is a legal combination of literal numbers, numeric variables, numeric functions, and numeric operators. For example, consider the command

```
SetScale x, 0, 6.283185, "v", wave1
```

which sets the X scaling for wave1. You could also write this as

```
SetScale x, 0, 2*PI, "v", wave1
```

Literal numbers include hexadecimal literals introduced by “0x”. For example:

```
Printf "The largest unsigned 16-bit number is: %d\r", 0xFFFF
```

Parentheses Required for /N=(<expression>)

Many operations accept flags of the form “/A=n” where A is some letter and n is some number. You can use a numeric expression for n but you must parenthesize the expression.

For example, both:

```
Make/N=512 wave1
Make/N=(2^9) wave1
```

are legal but this isn’t:

```
Make/N=2^9 wave1
```

A variable name is a form of numeric expression. Thus, assuming v1 is the name of a variable:

```
Make/N=(v1)
```

is legal, but

```
Make/N=v1
```

is not. This parenthesization is required only when you use a numeric expression in an operation flag.

String Expressions

A string expression can be used where Igor expects a string parameter. A string expression is a legal combination of literal strings, string variables, string functions, UTF-16 literals, and the string operator + which concatenates strings.

A UTF-16 literal represents a Unicode code value and consists of “U+” followed by four hexadecimal digits. For example:

```
Print "This is a bullet character: ‶ + U+2022
```

Setting Bit Parameters

A number of commands require that you specify a bit value to set certain parameters. In these instances you set a certain bit *number* by using a specific bit *value* in the command. The bit value is 2^n, where *n* is the bit number. So, to set bit 0 use a bit value of 1, to set bit 1 use a bit value of 2, etc.

For the example of the TraceNameList function the last parameter is a bit setting. To select normal traces you must set bit 0:

```
TraceNameList("",";",1)
```

and to select contour traces set bit 1:

```
TraceNameList("",";",2)
```

Most importantly, you can set multiple bits at one time by adding the bit values together. Thus, for TraceNameList you can select both normal (bit 0) and contour (bit 1) traces by using:

```
TraceNameList("",";",3)
```

See also **Using Bitwise Operators** on page IV-42.

RGBA Values

This section explains RGBA values used to specify colors in commands other than Gizmo commands. For Gizmo commands, see **Gizmo Color Specification** on page II-428.

In commands, colors are specified as RGBA values in the form (r,g,b[,a]).

r, g, and b specify the amount of red, green and blue in the color as integers from 0 to 65535.

The optional parameter a specifies "alpha" which represents the opacity of the color as an integer from 0 (fully transparent) to 65535 (fully opaque). a defaults to 65535 (fully opaque).

(0,0,0) represents opaque black and (65535,65535,65535) represents opaque white.

For example:

```
ModifyGraph rgb(wave0)=(0,0,0)                  // Opaque black
ModifyGraph rgb(wave0)=(65535,65535,65535)      // Opaque white
ModifyGraph rgb(wave0)=(65535,0,0,30000)        // Translucent red
```

Working With Strings

Igor has a rich repertoire of string handling capabilities. See **Strings** on page V-11 for a complete list of Igor string functions. Many of the techniques described in this section will be of interest only to programmers.

Many Igor operations require *string* parameters. For example, to label a graph axis, you can use the Label operation:

```
Label left, "Volts"
```

Other Igor operations, such as Make, require *names* as parameters:

```
Make wave1
```

Using the string substitution technique, described in **String Substitution Using $** on page IV-18, you can generate a name parameter by making a string containing the name and using the $ operator:

```
String stringContainingName = "wave1"
Make $stringContainingName
```

String Expressions

Wherever Igor requires a string parameter, you can use a string expression. A string expression can be:

- A literal string (`"Today is"`)
- The output of a string function (`date()`)
- An element of a text wave (`textWave0[3]`)
- A UTF-16 literal (`U+2022`)
- Some combination of string expressions (`"Today is" + date()`)

In addition, you can derive a string expression by indexing into another string expression. For example,

```
Print ("Today is" + date())[0,4]
```

prints “Today”.

A string variable can store the result of a string expression. For example:

```
String str1 = "Today is" + date()
```

A string variable can also be part of a string expression, as in:

```
Print "Hello. " + str1
```

Strings in Text Waves

A text wave contains an array of text strings. Each element of the wave can be treated using all of the available string manipulation techniques. In addition, text waves are commonly used to create category axes in bar charts. See **Text Waves** on page II-86 for further information.

String Properties

Strings in Igor can contain up to roughly two billion bytes.

Igor strings are usually used to store text data but you can also use them to store binary data.

When you treat a string as text data, for example if you print it to the history area of the command window or display its contents in an annotation or control, internal Igor routines treat a null byte as meaning “end-of-string”. Consequently, if you treat a binary string as if it were text, you may get unexpected results.

When you treat a string as text, Igor assumes that the text is encoded using the UTF-8 text encoding. For further discussion, see **String Variable Text Encodings** on page III-478.

Unicode Literal Characters

A Unicode literal represents a character using its UTF-16 code value. It consists of “U+” followed by four hexadecimal digits. For example:

```
Print "This is a bullet character: " + U+2022
```

The list of Unicode character codes is maintained at http://www.unicode.org/charts.

Escape Sequences in Strings

Igor treats the backslash character in a special way when reading literal (quoted) strings in a command line. The backslash is used to introduce an “escape sequence”. This just means that the backslash plus the next character or next few characters are treated like a different character — one you could not otherwise include in a quoted string. The escape sequences are:

`\t`	Represents a tab character
`\r`	Represents a return character (CR)
`\n`	Represents a linefeed character (LF)
`\'`	Represents a ' character (single-quote)
`\"`	Represents a " character (double-quote)
`\\`	Represents a \ character (backslash)
`\ddd`	Represents an arbitrary byte code ddd is a 3 digit octal number
`\xdd`	Represents an arbitrary byte code dd is a 2 digit hex number Requires Igor Pro 7.00 or later
`\udddd`	Represents a UTF-16 code point dddd is a 4 digit hex number Requires Igor Pro 7.00 or later
`\Udddddddd`	Represents a UTF-32 code point dddddddd is an 8 digit hex number Requires Igor Pro 7.00 or later

For example, if you have a string variable called “fileName”, you could print it in the history area using:

```
fileName = "Test"
Printf "The file name is \"%s\"\r", fileName
```

which prints

```
The file name is "Test"
```

In the Printf command line, \" embeds a double-quote character in the format string. If you omitted the backslash, the " would end the format string. The \r specifies that you want a carriage return in the format string.

Unicode Escape Sequences in Strings

Igor represents text internally in UTF-8 format. UTF-8 is a byte-oriented encoding format for Unicode. In this section we see how to use the \u, \U and \x escape sequences to represent Unicode characters in literal strings.

The use of UTF-8 and these escape sequences requires Igor Pro 7.00 or later. If you use them, your will get incorrect data or errors in Igor Pro 6.x.

In the first set of examples, we use the hexadecimal code 0041. This is hexadecimal for the Unicode code point representing CAPITAL LETTER A.

```
String str
str = "\x41"        // Specify code point using UTF-8 escape sequence
Print str
str = "\u0041"      // Specify code point using UTF-16 escape sequence
Print str
str = "\U0000041"   // Specify code point using UTF-32 escape sequence
Print str
```

When you use `\xdd`, Igor replaces the escape sequence with the byte specified by dd. Therefore you must specify byte values representing a valid UTF-8 character. 41 is hexadecimal for the UTF-8 encoding for CAPITAL LETTER A. If the code point that you specify is not a valid UTF-16 or UTF-32 code point, Igor replaces the escape sequence with the Unicode replacement character.

When you use `\udddd` or `\Udddddddd`, Igor replaces the escape sequence with the UTF-8 bytes for the corresponding Unicode code point as represented in UTF-16 or UTF-32 format.

Now we specify the characters for "Toyota" in Japanese Kanji:

```
str = "\xE8\xB1\x8A\xE7\x94\xB0"        // UTF-8
Print str
str = "\u8C4A\u7530"                    // UTF-16
Print str
str = "\U00008C4A\U00007530"            // UTF-32
Print str
```

In order to see the correct characters you must be using a font that includes Japanese characters.

\U is mainly of use to enter rare Unicode characters that are not in the Basic Multilingual Plane. Such characters can not be specified with single 16-bit UTF-16. Here is an example:

```
str = "\U00010300"                      // UTF-32 for OLD ITALIC LETTER A
Print str
str = "\xF0\x90\x8C\x80"                // UTF-8 for OLD ITALIC LETTER A
Print str
```

OLD ITALIC LETTER A is located in the Unicode Supplementary Multilingual Plane (plane 1).

See Also

http://en.wikipedia.org/wiki/Unicode

http://en.wikipedia.org/wiki/UTF-16

http://en.wikipedia.org/wiki/UTF-8

Embedded Nulls in Literal Strings

A null in a byte-oriented string is a byte with the value 0.

It is possible to embed an null byte in a string:

```
String test = "A\x00B"                         // OK in Igor Pro 7 or later
Print strlen(text)                                        // Prints 3
Print char2num(test[0]),char2num(test[1]),char2num(test[2]) // Prints 65 0 66
```

Here Igor converts the escape sequence \x00 to a null byte.

You typically have no need to embed a null in an Igor string because strings are usually used to store readable text and null does not represent a readable character. The need might arise, however, if you are using the string to store binary rather than text data. For example, if you need to send a small amount of binary data to an instrument, you can do so using \x escape sequences to represent the data in a literal string.

Although Igor allows you to embed nulls in literal strings, other parts of Igor are not prepared to handle them. For example:

```
Print test           // Prints "A", not "A<null>B"
```

In C null is taken to mean "end-of-string". Because of the use of C strings and C library routines in Igor, many parts of Igor will treat an embedded null as end-of-string. That is why the Print statement above prints just "A".

The bottom line is: You can store binary data, including nulls, in an Igor string but many parts of Igor that expect readable text will treat a null as end-of-string. See **Working With Binary String Data** for further discussion.

String Indexing

Indexing can extract a part of a string. This is done using a string expression followed by one or two numbers in brackets. The numbers are byte positions. Zero is the byte position of the first byte; n-1 is the byte position of the last byte of an n byte string value. For example, assume we create a string variable called s1 and assign a value to it as follows:

```
String s1="hello there"
```

h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10

Then,

```
Print s1[0,4]               prints   hello
Print s1[0,0]               prints   h
Print s1[0]                 prints   h
Print s1[1]+s1[2]+s1[3]     prints   ell
Print (s1+" jack")[6,15]    prints   there jack
```

A string indexed with one index, such as s1[p], is a string with one byte in it if p is in range (i.e. 0 ≤ p ≤ n-1). s1[p] is a string with no bytes in it if p is not in range. For example:

```
Print s1[0]                 prints   h
Print s1[-1]                prints   (nothing)
Print s1[10]                prints   e
Print s1[11]                prints   (nothing)
```

A string indexed with two indices, such as s1[p1,p2], contains all of the bytes from s1[p1] to s1[p2]. For example:

```
Print s1[0,10]              prints   hello there
Print s1[-1,11]             prints   hello there
Print s1[-2,-1]             prints   (nothing)
Print s1[11,12]             prints   (nothing)
Print s1[10,0]              prints   (nothing)
```

You can use INF to mean “end of string”:

```
Print s1[10,INF]            prints   (nothing)
```

The indices in these examples are byte positions, not character positions. See **Characters Versus Bytes** on page III-483 for a discussion of this distinction. See **Character-by-Character Operations** on page IV-173 for an example of stepping through characters using user-defined functions.

Because the syntax for string indexing is identical to the syntax for wave indexing, you have to be careful when using text waves. For example:

```
Make/T textWave0 = {"Red", "Green", "Blue"}
```

```
Print textWave0[1]             prints   Green
Print textWave0[1][1]          prints   Green
Print textWave0[1][1][1]       prints   Green
Print textWave0[1][1][1][1]    prints   Green
Print textWave0[1][1][1][1][1] prints   r
```

The first four examples print row 1 of column 0. Since waves may have up to four dimensions, the first four [1]’s act as dimension indices. The column, layer, and chunk indices were out of range and were clipped to a value of 0. Finally in the last example, we ran out of dimensions and got string indexing. Do not count on this behavior because future versions of Igor may support more than four dimensions.

The way to avoid the ambiguity between wave and string indexing is to use parentheses like so:

```
Print (textWave0[1])[1]     prints   r
```

The way to avoid the ambiguity between wave and string indexing is to use parentheses like so:

```
String tmp = textWave0[1]
Print tmp[1]                prints   r
```

String Assignment

You can assign values to string variables using string assignment. We have already seen the simplest case of this, assigning a literal string value to a string variable. You can also assign values to a subrange of a string variable, using string indexing. Once again, assume we create a string variable called s1 and assign a value to it as follows:

```
String s1="hello there"
```

Then,

```
s1[0,4]="hi"; Print s1              prints       hi there
s1[0,4]="greetings"; Print s1       prints       greetings there
s1[0,0]="j"; Print s1               prints       jello there
s1[0]="well "; Print s1             prints       well hello there
s1[100000]=" jack"; Print s1        prints       hello there jack
s1[-100]="well ";print s1           prints       well hello there
```

When the s1[p1,p2]= syntax is used, the right-hand side of the string assignment *replaces* the subrange of the string variable identified by the left-hand side, after p1 and p2 are clipped to 0 to n, where n is the number of bytes in the destination.

When the s1[p]= syntax is used, the right-hand side of the string assignment *is inserted before* the byte identified by p after p is clipped to 0 to n.

The subrange assignment just described for string variables is not supported when a text wave is the destination. To assign a value to a range of a text wave element, you will need to create a temporary string variable. For example:

```
Make/O/T tw = {"Red", "Green", "Blue"}
String stmp= tw[1]
stmp[1,2]="XX"
tw[1]= stmp;
```

```
Print tw[0],tw[1],tw[2]                prints        Red GXXen Blue
```

The indices in these examples are byte positions, not character positions. See **Characters Versus Bytes** on page III-483 for a discussion of this distinction.

String Substitution Using $

Wherever Igor expects the literal *name* of an operand, such as the name of a wave, you can instead provide a string expression preceded by the $ character. The $ operator evaluates the string expression and returns the *value* as a *name*.

For example, the Make operation expects the name of the wave to be created. Assume we want to create a wave named wave0:

```
Make wave0                // OK: wave0 is a literal name.

Make $"wave0"             // OK: $"wave0" evaluates to wave0.

String str = "wave0"
Make str                  // WRONG: This makes a wave named str.
Make $str                 // OK: $str evaluates to wave0.
```

$ is often used when you write a function which receives the name of a wave to be created as a parameter. Here is a trivial example:

```
Function MakeWave(wName)
   String wName       // name of the wave

   Make $wName
End
```

We would invoke this function as follows:

```
MakeWave("wave0")
```

We use $ because we need a wave name but we have a string containing a wave name. If we omitted the $ and wrote:

```
Make wName
```

Igor would make a wave whose name is wName, not on a wave whose name is wave0.

String substitution is capable of converting a string expression to a *single* name. It can not handle multiple names. For example, the following will *not* work:

```
String list = “wave0;wave1;wave2”
Display $list
```

See **Processing Lists of Waves** on page IV-198 for ways to accomplish this.

See **Converting a String into a Reference Using $** on page IV-62 for details on using $ in a user-defined function.

$ Precedence Issues In Commands

There is one case in which string substitution does not work as you might expect. Consider this example:

```
String str1 = "wave1"
wave2 = $str1 + 3
```

(The uses of $ in this section apply to the command line and macros only, not to user-defined functions in which you must use **Wave References** to read and write waves.)

You might expect that this would cause Igor to set wave2 equal to the sum of wave1 and 3. Instead, it generates an “expected string expression” error. The reason is that Igor tries to concatenate str1 and 3 *before* doing the substitution implied by $. The + operator is also used to concatenate two string expressions, and it has higher precedence than the $ operator. Since str1 is a string but 3 is not, Igor cannot do the concatenation.

You can fix this by changing this wave assignment to one of the following:

```
wave2 = 3 + $str1
wave2 = ($str1) + 3
```

Both of these accomplish the desired effect of setting wave2 equal to the sum of wave1 and 3. Similarly,

```
wave2 = $str1 + $str2   // Igor sees "$(str1 + $str2)"
```

generates the same “expected string expression” error. The reason is that Igor is trying to concatenate str1 and $str2. $str2 is a name, not a string. The solution is:

```
wave2 = ($str1) + ($str2)  // sets wave2 to sum of two named waves
```

Another situation arises when using the $ operator and [. The [symbol can be used for either point indexing into a wave, or byte indexing into a string. The commands

```
String wvName = "wave0"
$wvName[1,2] = wave1[p]    // sets two values in wave named "wave0"
```

are interpreted to mean that points 1 and 2 of wave0 are set values from wave1.

If you intended “$wvName[1,2] = wave1” to mean that a wave whose name comes from bytes 1 and 2 of the wvName string (“av”) has all of its values set from wave1, you must use parenthesis:

```
$(wvName[1,2]) = wave1     // sets all values of wave named "av"
```

String Utility Functions

WaveMetrics provides a number of utility functions for dealing with strings. To see a list of the built-in string functions:

1. Open the Igor Help Browser Command Help tab.
2. Open the Advanced Filtering control.
3. Uncheck all checkboxes except for Functions.
4. Choose String from the Functions pop-up menu.

See **Character-by-Character Operations** on page IV-173 for an example of stepping through characters using user-defined functions.

Special Cases

This section documents some techniques that were devised to handle certain specialized situations that arise with respect to Igor’s command language.

Instance Notation

There is a problem that occurs when you have multiple instances of the same wave in a graph or multiple instances of the same object in a layout. For example, assume you want to graph yWave versus xWave0, xWave1, and xWave2. To do this, you need to execute:

```
Display yWave vs xWave0
AppendToGraph yWave vs xWave1
AppendToGraph yWave vs xWave2
```

The result is a graph in which yWave occurs three times. Now, if you try to remove or modify yWave using:

```
RemoveFromGraph yWave
```

or

```
ModifyGraph lsize(yWave)=2
```

Igor will always remove or modify the first instance of yWave.

Instance notation provides a way for you to specify a particular instance of a particular wave. In our example, the command

```
RemoveFromGraph yWave#2
```

will remove instance number 2 of yWave and

```
ModifyGraph lsize(yWave#2)=2
```

will modify instance number 2 of yWave. Instance numbers start from zero so "yWave" is equivalent to "yWave#0". Instance number 2 is the instance of yWave plotted versus xWave2 in our example.

Where necessary to avoid ambiguity, Igor operation dialogs (e.g., Modify Trace Appearance) automatically use instance notation. Operations that accept trace names (e.g., ModifyGraph) or layout object names (e.g., ModifyObject) accept instance notation.

A graph can also display multiple waves with the same name if the waves reside in different datafolders. Instance notation applies to the this case also.

Instance Notation and $

The $ operator can be used with instance notation. The # symbol may be either inside the string operand or may be outside. For example `$"wave0#1"` or `$"wave0"#1`. However, because the # symbol may be inside the string, the string must be parsed by Igor. Consequently, unlike other uses of $, the wave name portion must be surrounded by single quotes if liberal names are used. For example, suppose you have a wave with the liberal name of `'ww#1'` plotted twice. The first instance would be `$"'ww#1'"` and the second `$"'ww#1'#1"` whereas `$"ww#1"` would reference the second instance of the wave ww.

Object Indexing

The ModifyGraph, ModifyTable and ModifyLayout operations, used to modify graphs, tables and page layouts, each support another method of identifying the object to modify. This method, object indexing, is used to generate style macros (see **Graph Style Macros** on page II-350). You may also find it handy in other situations.

Normally, you need to know the name of the object that you want to modify. For example, assume that we have a graph with three traces in it and we want to set the traces' markers from a procedure. We can write:

```
ModifyGraph marker(wave0)=1, marker(wave1)=2, marker(wave2)=3
```

Because it uses the names of particular traces, this command is specific to a particular graph. What do we do if we want to write a command that will set the markers of three traces in *any* graph, regardless of the names of the traces? This is where object indexing comes in.

Using object indexing, we can write:

```
ModifyGraph marker[0]=1, marker[1]=2, marker[2]=3
```

This command sets the markers for the first three traces in a graph, no matter what their names are.

Indexes start from zero. For graphs, the object index refers to traces starting from the first trace placed in the graph. For tables the index refers to columns from left to right. For page layouts, the index refers to objects starting from the first object placed in the layout.

/Z Flag

The ModifyGraph marker command above works fine if you know that there *are* three waves in the graph. It will, however, generate an error if you use it on a graph with fewer than 3 waves. The ModifyGraph operation supports a flag that can be used to handle this:

```
ModifyGraph/Z marker[0]=1, marker[1]=2, marker[2]=3
```

The /Z flag ignores errors if the command tries to modify an object that doesn't exist. The /Z flag works with the SetAxis and Label operations as well as with the ModifyGraph, ModifyTable and ModifyLayout operations. Like object indexing, the /Z flag is primarily of use in creating style macros, which is done automatically, but it may come in handy for other uses.

Chapter

IV-2

Programming Overview

Overview

You can perform powerful data manipulation and analysis interactively using Igor's dialogs and the command line. However, if you want to automate common tasks or create custom data analysis features, then you need to use procedures.

You can write procedures yourself, use procedures supplied by WaveMetrics, or find someone else who has written procedures you can use. Even if you don't write procedures from scratch, it is useful to know enough about Igor programming to be able to understand code written by others.

Programming in Igor entails creating procedures by entering text in a procedure window. After entering a procedure, you can execute it via the command line, by choosing an item from a menu, or using a button in a control panel.

The bulk of the text in a procedure window falls into one of the following categories:

- Pragmas, which send instructions from the programmer to the Igor compiler
- Include statements, which open other procedure files
- Constants, which define symbols used in functions
- Structure definitions, which can be used in functions
- Proc Pictures, which define images used in control panels, graphs, and layouts
- Menu definitions, which add menu items or entire menus to Igor
- Functions — compiled code which is used for nearly all Igor programming
- Macros — interpreted code which, for the most part, is obsolete

Functions are written in Igor's programming language. Like conventional procedural languages such as C or Pascal, Igor's language includes:

- Data storage elements (variables, strings, waves)
- Assignment statements
- Flow control (conditionals and loops)
- Calls to built-in and external operations and functions
- Ability to define and call subroutines

Igor programming is easier than conventional programming because it is much more interactive — you can write a routine and test it right away. It is designed for interactive use within Igor rather than for creating stand-alone programs.

Names in Programming

Functions, constants, variables, structures and all other programming entities have names. Names used in programming must follow the standard Igor naming conventions.

Names can consist of up to 255 characters. Only ASCII characters are allowed. The first character must be alphabetic while the remaining characters can include alphabetic and numeric characters and the underscore character. Names in Igor are case insensitive.

Prior to Igor Pro 8.00, names used in programming were limited to 31 bytes. If you use long names, your procedures will require Igor Pro 8.00 or later.

The names of external operations and external functions are an exception. They are limited to 31 bytes.

Organizing Procedures

Procedures can be stored in the built-in Procedure window or in separate auxiliary procedure files. Chapter III-13, **Procedure Windows**, explains how to edit the Procedure window and how to create auxiliary procedure files.

At first you will find it convenient to do all of your Igor programming in the built-in Procedure window. In the long run, however, it will be useful to organize your procedures into categories so that you can easily find and access general-purpose procedures and keep them separate from special-case procedures.

This table shows how we categorize procedures and how we store and access the different categories.

Category	What	Where	How
Experiment Procedures	These are specific to a single Igor experiment. They include procedures you write as well as window recreation macros created automatically when you close a graph, table, layout, control panel, or Gizmo plot.	Usually experiment procedures are stored in the built-in Procedure window. You can optionally create additional procedure windows in a particular experiment but this is usually not needed.	You create an experiment procedure by typing in the built-in Procedure window.
Utility Procedures	These are general-purpose and potentially useful for any Igor experiment. WaveMetrics supplies utility procedures in the WaveMetrics Procedures folder. You can also write your own procedures or get them from colleagues.	WaveMetrics-supplied utility procedure files are stored in the WaveMetrics Procedures folder. Utility procedure files that you or other Igor users create should be stored in your own folder, in the Igor Pro User Files folder (see **Igor Pro User Files** on page II-31 for details) or at another location of your choosing. Place an alias or shortcut for your folder in "Igor Pro User Files/User Procedures".	Use an include statement to use a WaveMetrics or user utility procedure file. Include statements are described in **The Include Statement** on page IV-166.
Global Procedures	These are procedures that you want to be available from all experiments.	Store your global procedure files in "Igor Pro User Files/Igor Procedures" (see **Igor Pro User Files** on page II-31 for details). You can also store them in another folder of your choice and place an alias or shortcut for your folder in "Igor Pro User Files/Igor Procedures".	Igor automatically opens any procedure file in "Igor Pro 7 Folder/Igor Procedures" and "Igor Pro User Files/Igor Procedures" and subfolders or referenced by an alias or shortcut in those folders, and leaves it open in all experiments.

Following this scheme, you will know where to put procedure files that you get from colleagues and where to look for them when you need them.

Utility and global procedures should be general-purpose so that they can be used from any experiment. Thus, they should not rely on specific waves, global variables, global strings, specific windows or any other objects specific to a particular experiment. See **Writing General-Purpose Procedures** on page IV-167 for further guidelines.

After they are debugged and thoroughly tested, you may want to share your procedures with other Igor users via **IgorExchange**.

WaveMetrics Procedure Files

WaveMetrics has created a large number of utility procedure files that you can use as building blocks. These files are stored in the WaveMetrics Procedures folder. They are described in the WM Procedures Index help file, which you can access through the Help→Help Windows menu.

You access WaveMetrics procedure files using include statements. Include statements are explained under **The Include Statement** on page IV-166.

Using the Igor Help Browser, you can search the WaveMetrics Procedures folder to find examples of particular programming techniques.

Macros and Functions

There are two kinds of Igor procedures: **macros** and **functions**. They use similar syntax. The main difference between them is that Igor compiles user functions but interprets macros.

Because functions are compiled, they are dramatically faster than macros. Compilation also allows Igor to detect errors in functions when you write the function, whereas errors in macros are detected only when they are executed.

Functions provide many programming features that are not available in macros.

Macros are a legacy of Igor's early days. With rare exceptions, **all new programming should use functions**, not macros. To simplify the presentation of Igor programming, most discussion of macros is segregated into Chapter IV-4, **Macros**.

Scanning and Compiling Procedures

When you modify text in a procedure window, Igor must process it before you can execute any procedures. There are two parts to the processing: scanning and function compilation. In the scanning step, Igor finds out what procedures exist in the window. In the compilation step, Igor's function compiler converts the function text into low-level instructions for later execution.

For the sake of brevity, we use the term "compile" to mean "scan and compile" except when we are specifically pointing out the distinction between these two steps.

You can *explicitly* compile the procedures using the Compile button in the Procedure window or the Compile item in the Macros menu.

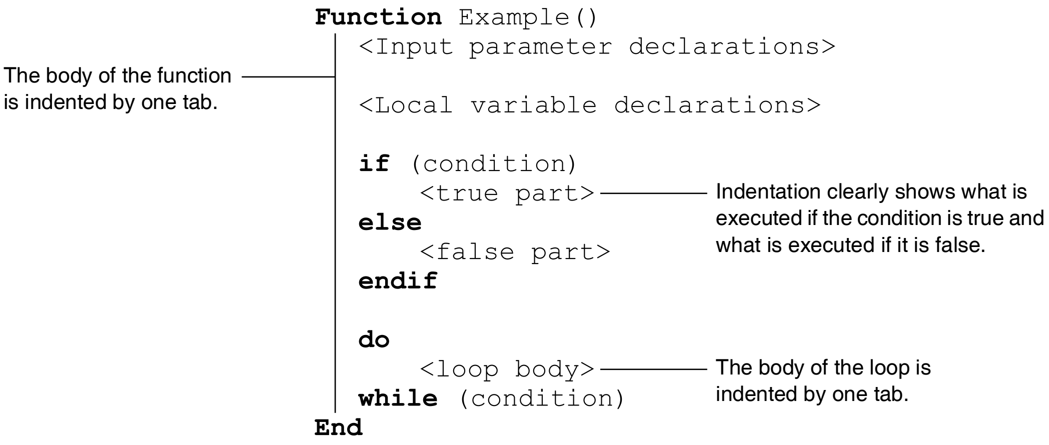
By default, Igor *automatically* compiles the procedure text at appropriate times. For example, if you type in the Procedure window and then hide it by clicking in the close button, Igor will automatically compile.

If you have many procedures that take long to compile, you may want to turn auto-compiling off using the Macros menu.

When Auto-compile is deselected, Igor compiles only when you explicitly request it. Igor will still scan the procedures when it needs to know what macros and functions exist.

Indentation Conventions

We use indentation to indicate the structure of a procedure.



```
Function Example()
    <Input parameter declarations>

    <Local variable declarations>

    if (condition)
        <true part>
    else
        <false part>
    endif

    do
        <loop body>
    while (condition)
End
```

The body of the function is indented by one tab.
Indentation clearly shows what is executed if the condition is true and what is executed if it is false.
The body of the loop is indented by one tab.

The structural keywords, shown in bold here, control the flow of the procedure. The purpose of the indentation is to make the structure of the procedure apparent by showing which lines are within which structural keywords. Matching keywords are at the same level of indentation and all lines within those keywords are indented by one tab.

The Edit menu contains aids for maintaining or adjusting indentation. You can select multiple lines and choose Indent Left or Indent Right. You can have Igor automatically adjust the indentation of a procedure by selecting the whole procedure or a subset and then choosing Adjust Indentation.

Igor does not require that you use indentation but we recommend it for readability.

What’s Next

The next chapter covers the core of Igor programming — writing user-defined functions.

Chapter IV-4, **Macros**, explains macros. Because new programming does not use macros, that chapter is mostly of use for understanding old Igor code.

Chapter IV-5, **User-Defined Menus**, explains user-defined menus. It explains how you can add menu items to existing Igor menus and create entire new menus of your own.

Chapter IV-6, **Interacting with the User**, explains other methods of interacting with the user, including the use of dialogs, control panels, and cursors.

Chapter IV-7, **Programming Techniques**, covers an assortment of programming topics. An especially important one is the use of the include statement, which you use to build procedures on top of existing procedures.

Chapter IV-8, **Debugging**, covers debugging using Igor’s symbolic debugger.

Chapter IV-9, **Dependencies**, covers dependencies — a way to tie a variable or wave to a formula.

Chapter IV-10, **Advanced Topics**, covers advanced topics, such as communicating with other programs, doing FTP transfers, doing data acquisition, and creating a background task.

Chapter

IV-3

User-Defined Functions

Overview

Most of Igor programming consists of writing user-defined functions.

A function has zero or more parameters. You can use local variables to store intermediate results. The function body consists of Igor operations, assignment statements, flow control statements, and calls to other functions.

A function can return a numeric, string, wave reference or data folder reference result. It can also have a side-effect, such as creating a wave or creating a graph.

Before we dive into the technical details, here is an informal look at some simple examples.

```
Function Hypotenuse(side1, side2)
   Variable side1, side2

   Variable hyp
   hyp = sqrt(side1^2 + side2^2)

   return hyp
End
```

The Hypotenuse function takes two numeric parameters and returns a numeric result. “hyp” is a local variable and sqrt is a built-in function. You could test Hypotenuse by pasting it into the built-in Procedure window and executing the following statement in the command line:

```
Print Hypotenuse(3, 4)
```

Now let’s look at a function that deals with text strings.

```
Function/S FirstStr(str1, str2)
   String str1, str2

   String result

   if (CmpStr(str1,str2) < 0)
      result = str1
   else
      result = str2
   endif

   return result
End
```

The FirstStr function takes two string parameters and returns the string that is first in alphabetical order. CmpStr is a built-in function. You could test FirstStr by executing pasting it into the built-in Procedure window the following statement in the command line:

```
Print FirstStr("ABC", "BCD")
```

Now a function that deals with waves.

```
Function CreateRatioOfWaves(w1, w2, nameOfOutputWave)
   WAVE w1, w2
   String nameOfOutputWave

   Duplicate/O w1, $nameOfOutputWave
   WAVE wOut = $nameOfOutputWave
   wOut = w1 / w2
End
```

The CreateRatioOfWaves function takes two wave parameters and a string parameter. The string is the name to use for a new wave, created by duplicating one of the input waves. The “`WAVE wOut`” statement creates a wave reference for use in the following assignment statement. This function has no direct result (no return statement) but has the side-effect of creating a new wave.

Here are some commands to test CreateRatioOfWaves:

```
Make test1 = {1, 2, 3}, test2 = {2, 3, 4}
CreateRatioOfWaves(test1, test2, "ratio")
Edit test1, test2, ratio
```

Function Syntax

The basic syntax of a function is:

```
Function <Name> (<Parameter list> [<Optional Parameters>]) [:<Subtype>]
   <Parameter declarations>

   <Local variable declarations>

   <Body code>

   <Return statement>
End
```

Here is an example:

```
Function Hypotenuse(side1, side2)
   Variable side1, side2            // Parameter declaration

   Variable hyp                     // Local variable declaration

   hyp = sqrt( side1^2 + side2^2 )  // Body code

   return hyp                       // Return statement
End
```

You could test this function from the command line using one of these commands:

```
Print Hypotenuse(3,4)
Variable/G result = Hypotenuse(3,4); Print result
```

As shown above, the function returns a real, numeric result. The Function keyword can be followed by a flag that specifies a different result type.

Flag	Return Value Type
/D	Double precision number (obsolete)
/C	Complex number
/S	String
/WAVE	Wave reference
/DF	Data folder reference

The /D flag is obsolete because all calculations are now performed in double precision. However, it is still permitted.

The Function Name

The names of functions must follow the standard Igor naming conventions. Names can consist of up to 255 characters. Only ASCII characters are allowed. The first character must be alphabetic while the remaining characters can include alphabetic and numeric characters and the underscore character. Names must not conflict with the names of other Igor objects, functions or operations. Names in Igor are case insensitive.

Prior to Igor Pro 8.00, function names were limited to 31 bytes. If you use long function names, your procedures will require Igor Pro 8.00 or later.

The Procedure Subtype

You can identify procedures designed for specific purposes by using a subtype. Here is an example:

```
Function ButtonProc(ctrlName) : ButtonControl
   String ctrlName

   Beep
End
```

Here, " : ButtonControl" identifies a function intended to be called when a user-defined button control is clicked. Because of the subtype, this function is added to the menu of procedures that appears in the Button Control dialog. When Igor automatically generates a procedure it generates the appropriate subtype. See **Procedure Subtypes** on page IV-204 for details.

The Parameter List and Parameter Declarations

The parameter list specifies the name for each input parameter. There is no limit on the number of parameters.

All parameters must be declared immediately after the function declaration. In Igor Pro 7 or later you can use inline parameters, described below.

The parameter declaration declares the type of each parameter using one of these keywords:

Variable	Numeric parameter
Variable/C	Complex numeric parameter
String	String parameter
Wave	Wave reference parameter
Wave/C	Complex wave reference parameter
Wave/T	Text wave reference parameter
DFREF	Data folder reference parameter
FUNCREF	Function reference parameter
STRUCT	Structure reference parameter
int	Signed integer parameter- requires Igor7 or later
int64	Signed 64-bit integer parameter - requires Igor7 or later
uint64	Unsigned 32-bit integer parameter - requires Igor7 or later
double	Numeric parameter - requires Igor7 or later
complex	Complex numeric parameter - requires Igor7 or later

`int` is 32 bits in IGOR32 and 64 bits in IGOR64.

`double` is a synonym for `Variable` and `complex` is a synonym for `Variable/C`.

`WAVE/C` tells the Igor compiler that the referenced wave is complex.

`WAVE/T` tells the Igor compiler that the referenced wave is text.

Variable and string parameters are usually passed to a subroutine *by value* but can also be passed by reference. For an explanation of these terms, see **How Parameters Work** on page IV-58.

Integer Parameters

In Igor Pro 7 or later you can use these integer types for parameters and local variables in user-defined functions:

`int`	32-bit integer in IGOR32; 64-bit integer in IGOR64
`int64`	64-bit signed integer
`uint64`	64-bit unsigned integer

`int` is a generic signed integer that you can use for wave indexing or general use. It provides a speed improvement over Variable or double in most cases.

Signed `int64` and unsigned `uint64` are for special purposes where you need explicit access to bits. You can also use them in structures.

Optional Parameters

Following the list of required function input parameters, you can also specify a list of optional input parameters by enclosing the parameter names in brackets. You can supply any number of optional parameter values when calling the function by using the *`ParamName=Value`* syntax. Optional parameters may be of any valid data type. There is no limit on the number of parameters.

All optional parameters must be declared immediately after the function declaration. As with all other variables, optional parameters are initialized to zero. You must use the **ParamIsDefault** function to determine if a particular optional parameter was supplied in the function call.

See **Using Optional Parameters** on page IV-60 for an example.

Inline Parameters

In Igor Pro 7 or later you can declare user-defined functions parameters inline. This means that the parameter types and parameter names are declared in the same statement as the function name:

```
Function Example(Variable a, [ Variable b, double c ])
   Print a,b,c
End
```

or, equivalently:

```
Function Example2(
      Variable a,// The comma is optional
      [
         Variable b,
         double c
      ]
   )
   Print a,b,c
End
```

Local Variable Declarations

The parameter declarations are followed by the local variable declarations if the procedure uses local variables. Local variables exist only during the execution of the procedure. They are declared using one of these keywords:

Variable	Numeric variable
Variable/C	Complex numeric variable
String	String variable
NVAR	Global numeric variable reference
NVAR/C	Global complex numeric variable reference
SVAR	Global variable reference
Wave	Wave reference
Wave/C	Complex wave reference
Wave/T	Text wave reference
DFREF	Data folder reference
FUNCREF	Function reference
STRUCT	Structure
int	Signed integer - requires Igor7 or later
int64	Signed 64-bit integer - requires Igor7 or later
uint64	Unsigned 32-bit integer - requires Igor7 or later
double	Numeric variable - requires Igor7 or later
complex	Complex numeric variable - requires Igor7 or later

`double` is a synonym for `Variable` and `complex` is a synonym for `Variable/C`.

Numeric and string local variables can optionally be initialized. For example:

```
Function Example(p1)
   Variable p1

   // Here are the local variables
   Variable v1, v2
   Variable v3=0
   Variable/C cv1=cmplx(0,0)
   String s1="test", s2="test2"

   <Body code>
End
```

If you do not supply explicit initialization, Igor automatically initializes local numeric variables with the value zero. Local string variables are initialized with a null value such that, if you try to use the string before you store a value in it, Igor reports an error.

Initialization of other local variable types is discussed below. See **Wave References** on page IV-71, **Data Folder References** on page IV-78, **Function References** on page IV-107, and **Structures in Functions** on page IV-99.

The name of a local variable is allowed to conflict with other names in Igor, but they must be unique within the function. If you create a local variable named “sin”, for example, then you will be unable to use Igor’s built-in sin function within the function.

You can declare local variables anywhere in the function after the parameter declarations.

Body Code

This table shows what can appear in body code of a function.

What	Allowed in Functions?	Comment
Assignment statements	Yes	Includes wave, variable and string assignments.
Built-in operations	Yes, with a few exceptions.	See **Operations in Functions** on page IV-111 for exceptions.
Calls to user functions	Yes	
Calls to macros	No	
External functions	Yes	
External operations	Yes, with exceptions.	

Statements are limited to 2500 bytes per line except for assignment statements which, in Igor Pro 7 or later, can be continued on subsequent lines.

Line Continuation

In user-defined functions in Igor Pro 7 or later, you can use arbitrarily long expressions by including a line continuation character at the very end of a line. The line continuation character is backslash. For example:

```
Function Example1(double v1)
   return v1 + \
   2
End
```

Line continuation is supported for any numeric or string expression in a user-defined function. Here is an example using a string expression:

```
Function/S Example2(string s1)
   return s1 + \
      " " + \
      "there"
End
```

Line continuation is supported for numeric and string expressions only. It is not supported on other types of commands. For example, this generates a compile error:

```
Function Example3(double v1)
   Make wave0        \
        wave1
End
```

The Return Statement

A return statement often appears at the end of a function, but it can appear anywhere in the function body. You can also have more than one return statement.

The return statement immediately stops executing the function and returns a value to the calling function. The type of the returned value must agree with the type declared in the function declaration.

If there is no return statement, or if a function ends without hitting a return statement, then the function returns the value NaN (Not a Number) for numeric functions and null for other types of functions. If the calling function attempts to use the null value, Igor reports an error.

Multiple Return Syntax

In Igor Pro 8 or later, you can create a function with multiple return values using this syntax:

```
Function [ <output parameter list> ] <function name> ( <input parameter list> )
```

The square brackets are part of the syntax.

Both the output parameter list and the input parameter list must be specifed using inline syntax (see **Inline Parameters** on page IV-33). The entire function declaration must appear on one line.

You can return all output values using one return statement:

```
return [ <value list> ]
```

For example:

```
Function [ Variable v, String s ] Subroutine( Variable a )
   return [1+a, "hello"]
End

Function CallingRoutine()
   Variable v1
   String s1
   [v1, s1] = Subroutine(10)
   Print v1, s1
End
```

In Igor Pro 9.00 and later, you can declare the return variables in the destination parameter list like this:

```
Function CallingRoutine()
   [Variable v1, String s1] = Subroutine(10)
   Print v1, s1
End
```

You can set the return values individually without using a return statement like this:

```
Function [ Variable v, String s ] Subroutine( Variable a )
   v = 1 + a
   s = "hello"
End
```

The output parameter list can include numeric and string variables as well as structures. Any type that can be used as a pass-by-reference parameter (see **Pass-By-Reference** on page IV-59) can be used in the output parameter list.

Multiple Return Syntax Example

This example illustrates using structure and WAVE references as output parameters:

```
Structure DemoStruct
   Variable num
   String str
EndStructure

Function [STRUCT DemoStruct sOut, WAVE/T wOut] Subroutine(Variable v1)
   STRUCT DemoStruct s
   s.num = 4 + v1
   s.str = "from Subroutine"

   Make/O/T tw = {"text wave point 0","text wave point 1"}

   return [ s, tw ]
End

Function CallingRoutine()
```

```
   STRUCT DemoStruct s
   WAVE/T w
   [ s, w ] = Subroutine(2)
   Print s, w
End
```

Executing CallingRoutine gives:

```
STRUCT DemoStruct
 num: 6
 str: from Subroutine
tw[0]= {"text wave point 0","text wave point 1"}
```

In Igor Pro 9.00 and later, the CallingRoutine can be rewritten to declare the return variables in the destination parameter list:

```
Function CallingRoutine()
   [ STRUCT DemoStruct s, WAVE/T w ] = Subroutine(2)
   Print s, w
End
```

At runtime, WAVE variables declared in the destination list do not perform the usual time consuming lookup (trying to find a wave named w in this case.) They act as if WAVE/ZZ was specified. See **WAVE** on page V-1069 for details.

Using Multiple Return Syntax Variables As Input and Output

Igor initializes local numeric variables to 0, local string variables to NULL, and local WAVE variables to NULL. If you use a local variable as both and input and an output to a function that uses multiple return syntax, you should assign a value to the variable before using it:

```
Function [String str] AppendToString()
   str += "DEF"
End

Function CallingRoutineBad()
   String str                 // Initialized by Igor to NULL
   [str] = AppendToString()   // Error: Attempt to use a NULL string
   Print str
End

Function CallingRoutineGood()
   String str = "ABC"         // Explicitly initialized
   [str] = AppendToString()
   Print str                  // Prints "ABCDEF"
End
```

CallingRoutineBad passes a NULL string to AppendToString which treats it as both an input and an output. AppendToString attempts to append to the NULL string causing an "attempt to use NULL string" error.

Here is a more subtle way to make this mistake:

```
Function CallingRoutineBad()
   [String str] = AppendToString()  // Error: Attempt to use a NULL string
   Print str
End
```

Again, str is NULL and this results in an "attempt to use NULL string" error.

The same principle applies to local numeric variables and to WAVE references. If you are passing them to an MRS function that uses them as both an input and an output, you must initialize them in the calling function.

Expression Evaluation

An expression is a combination of literal values, variable references, wave references, function calls, parentheses and operators. Expressions appear on the right-hand side of assignment statements and as parameters in commands. For example:

```
Variable v = ((123 + someVariable) * someWave[3]) / SomeFunction()
```

In most cases Igor evaluates expressions using double-precision floating point. However, if the destination is an integer type, then Igor uses integer calculations.

Integer Expressions

Prior to Igor Pro 7, all calculations were performed in double-precision floating point. Double-precision can represent integers of up to 53 bits precisely. Integers larger than 53 bits are represented approximately in double-precision.

Igor Pro 7 or later can perform integer calculations instead of floating point. You trigger this by assigning a value to a local variable declared using the integer types int, int64 and uint64. Calculations are done using 32 bits or 64 bits depending on the type of the integer.

When an integer type variable is the destination in an expression in a function, the right-hand side is compiled using integer math. This avoids the limitation of 53 bits for double precision and may also provide a speed advantage. If you use a function such as sin that is inherently floating point, it is calculated as a double and then converted to an integer. You should avoid using anything that causes a double-to-integer conversion when the destination is a 64-bit integer.

This example shows various ways to use integer expressions:

```
Function Example()
   int a = 0x101              // 0x introduces a hexadecimal literal number
   int b = a<<2
   int c = b & 0x400
   printf "a=%x, b=%x, c=%x\r",a, b, c
End
```

This prints

```
a=101, b=404, c=400
```

To set bits in an integer, use the left-shift operator, <<. For example to set bit 60 in a 64-bit integer, use 1<<60. This is the integer equivalent of 2^60, but you can not use ^ because exponentiation is not supported in integer expressions.

To print all the bits in a 64-bit integer, use Printf with the %x or %d conversion specification. You can also use a Print command as long as the compiler can clearly see the number is an integer from the first symbol. For example:

```
Function Example()
   int64 i = 1<<60   // 1152921504606846976 (decimal), 1000000000000000 (hex)
   Printf "i = %0.16X\r", i
   Printf "i = %d\r", i
   Print "i =", i
   Print "i =", 0+i  // First symbol is not an integer variable or wave
End
```

This prints:

```
  i = 1000000000000000
  i = 1152921504606846976
  i = 1152921504606846976
  i = 1.15292e+18
```

Integer Expressions in Wave Assignment Statements

When compiling a wave assignment statement, the right-hand expression is compiled as 64-bit integer if the compiler knows that the destination wave is 64-bit integer. Otherwise, the expression is compiled as double. For example:

```
Function ExpressionCompiledAsInt64()
   Make/O/N=1/L w64
   w64 = 0x7FFFFFFFFFFFFFFF
   Print w64[0]
End
```

0x7FFFFFFFFFFFFFFF is the largest signed 64-bit integer value expressed in hexadecimal. It can not be precisely represented in double precision.

Here the /L flag tells Igor that the wave is signed 64-bit integer. This causes the compiler to compile the right-hand expression using signed 64-bit signed operations. The correct value is assigned to the wave and the correct value is printed.

In the next example, the compiler does not know that the wave is signed 64-bit integer because the /L flag was not used. Consequently the right-hand expression is compiled using double-precision operations. Because 0x7FFFFFFFFFFFFFFF can not be precisely represented in double precision, the value assigned to the wave is incorrect, as is the printed value:

```
Function ExpressionCompiledAsDouble()
   Make/O/N=1/Y=(0x80) w64    // Wave is int64 but Igor does not know it
   w64 = 0x7FFFFFFFFFFFFFFF   // Expression compiled as double
   Print w64[0]
End
```

You can use the /L flag in a wave declaration. For example:

```
Function ExpressionCompiledAsInt64()
   Make/O/N=1Y=(0x80) w64     // Wave is int64 but Igor does not know it
   Wave/L w = w64             // Igor knows that w refers to int64 wave
   w = 0x7FFFFFFFFFFFFFFF     // Expression compiled as int64
   Print w[0]
End
```

This tells Igor that the wave reference by w is signed 64-bit integer, causing the compiler to use signed 64-bit integer to compile the right-hand expression.

If the wave was unsigned 64-bit, we would need to use Wave/L/U.

In summary, to assign values to a 64-bit integer wave, make sure to use the correct flags so that Igor will compile the right-hand expression using 64-bit integer operations.

To view the contents of an integer wave, especially a 64-bit integer wave, display it in a table using an integer or hexadecimal column format or print individual elements on the command line. Print on an entire wave is not yet integer-aware and uses doubles.

Complex Integer Expressions

In Igor Pro 7, you could write wave assignment statements for complex integer waves up to 32 bits. For example:

```
Make/O/I/C complex32BitWave = cmplx(p,p)
```

In Igor Pro 8 or later, you can also write wave assignment statements for 64-bit complex integer waves. For example:

```
Make/O/L/C complex64BitWave = cmplx(p,p)   // This was not supported in Igor7
```

The cmplx function uses double-precision for its argument. This limits the precision of integer arguments to to cmplx to 53 bits. Consequently, when assigning to a 64-bit complex integer wave, values above 2^53 are imprecise.

In the case of 8-, 16- and 32-bit complex integer waves, math is done in double-precision floating point and then converted to integer. This is fine since double-precision supports up to 2^53 precisely. However, for 64-bit complex integers, doubles are not sufficient. Igor uses special routines for 64-bit complex integer addition and subtraction and returns an error for other operations such as multiplication and division. For example:

```
complex64BitWave += cmplx(-p,-p)     // OK
complex64BitWave *= cmplx(1,1)       // Feature not implemented error
```

Conditional Statements in Functions

Igor Pro supports two basic forms of conditional statements: if-else-endif and if-elseif-endif statements. Igor also supports multiway branching with switch and strswitch statements.

If-Else-Endif

The form of the if-else-endif structure is

```
if ( <expression> )
   <true part>
else
   <false part>
endif
```

`<expression>` is a numeric expression that is considered true if it evaluates to any nonzero number and false if it evaluates to zero. The true part and the false part may consist of any number of lines. If the expression evaluates to true, only the true part is executed and the false part is skipped. If the expression evaluates to false, only the false part is executed and the true part is skipped. After the true part or false part code is executed, execution continues with any code immediately following the if-else-endif statement.

The keyword “else” and the false part may be omitted to give a simple conditional:

```
if ( <expression> )
   <true part>
endif
```

Because Igor is line-oriented, you may not put the `if`, `else` and `endif` keywords all on the same line. They must each be in separate lines with no other code.

If-Elseif-Endif

The if-elseif-endif statement provides a means for creating nested if structures. It has the form:

```
if ( <expression1> )
   <true part 1>
elseif ( <expression2> )
   <true part 2>
[else
   <false part>]
endif
```

These statements follow similar rules as for if-else-endif statements. When any expression evaluates as true (nonzero), the code immediately following the expression is executed. If all expressions evaluate as false (zero) and there is an else clause, then the statements following the else keyword are executed. Once any code in a true part or the false part is executed, execution continues with any code immediately following the if-elseif-endif statement.

Comparisons

The relational comparison operators are used in numeric conditional expressions.

Symbol	Meaning	Symbol	Meaning
==	equal	<=	less than or equal
!=	not-equal	>	greater than
<	less than	>=	greater than or equal

These operators return 1 for true and 0 for false.

The comparison operators work with numeric operands only. To do comparisons with string operands, use the **CmpStr** function.

Comparison operators are usually used in conditional structures but can also be used in arithmetic expressions. For example:

```
wave0 = wave0 * (wave0 < 0)
```

This clips all positive values in wave0 to zero.

See also **Example: Comparison Operators and Wave Synthesis** on page II-82.

Operators in Functions

This section discusses operators in the context of user-defined functions. See **Operators** on page IV-6 for a discussion of operators used in the command line and macros as well as functions.

Bitwise and Logical Operators

The bitwise and logical operators are also used in conditional expressions.

Symbol	Meaning
~	Bitwise complement
&	Bitwise AND
|	Bitwise OR
%^	Bitwise exclusive OR
!	Logical NOT
&&	Logical AND
||	Logical OR
<<	Shift left (requires Igor7 or later)
>>	Shift right (requires Igor7 or later)

The precedence of operators is shown in the table under **Operators** on page IV-6. In the absence of parentheses, an operator with higher precedence (higher in the table) is executed before an operator with lower precedence.

Because the precedence of the arithmetic operators is higher than the precedence of the comparison operators, you can write the following without parentheses:

```
if (a+b != c+d)
   Print "a+b is not equal to c+d"
endif
```

Because the precedence of the comparison operators is higher than the precedence of the logical OR operator, you can write the following without parentheses:

```
if (a==b || c==d)
   Print "Either a equals b or c equals d"
endif
```

The same is true of the logical AND operator, &&.

For operators with the same precedence, there is no guaranteed order of execution and you must use parentheses to be sure of what will be executed. For example:

```
if ((a&b) != (c&d))
   Print "a ANDED with b is not equal to c ANDED with d"
endif
```

See **Operators** on page IV-6 for more discussion of operators.

Using Bitwise Operators

The bitwise operators are used to test, set, and clear bits. This makes sense only when you are dealing with integer operands.

Bit Action	Operation
Test	AND operator (&)
Set	OR operator (|)
Clear	Bitwise complement operator (~) followed by the bitwise AND operator (&)
Shift left	<< operator (requires Igor7 or later)
Shift right	>> operator (requires Igor7 or later)

This function illustrates various bit manipulation techniques.

```
Function DemoBitManipulation(vIn)
   Variable vIn

   vIn = trunc(vIn)               // Makes sense with integers only
   Printf "Original value: %d\r", vIn

   Variable vOut

   if ((vIn & 2^3) != 0)          // Test if bit 3 is set
      Print "Bit 3 is set"
   else
      Print "Bit 3 is cleared"
   endif

   vOut = vIn | (2^3)             // Set bit 3
   Printf "Set bit 3: %d\r", vOut

   vOut = vIn & ~(2^3)            // Clear bit 3
   Printf "Clear bit 3: %d\r", vOut

   vOut = vIn << 3                // Shift three bits left
   Printf "Shift three bits left: %d\r", vOut

   vOut = vIn >> 3                // Shift three bits right
   Printf "Shift three bits right: %d\r", vOut
End
```

For a simple demonstration, try this function passing 1 as the parameter.

Bit Shift Operators

Igor Pro 7 or later supports the bit shift operators << and >>. a<<n shifts the bits of the integer a left by n bits. a>>n shifts the bits of the integer a right by n bits.

Normally you should apply bit shift operators to integer values. If you apply a bit shift operator to a floating point value, any fractional part is truncated.

Increment and Decrement Operators

In Igor7 and later, in a user-defined function, you can use these operators on local variables:

++var	Pre-increment
var++	Post-increment
--var	Pre-decrement
var--	Post-decrement

var must be a real local variable.

The pre-increment functions increment the variable and then return its new value. In this example, a is initialized to 0. ++a then increments a to 1 and returns the new value. Consequently 1 is stored in b.

```
Function PreIncrementDemo()
   int a = 0
   int b = ++a
End
```

The post-increment functions return the variable's original value and then increment it. In this example, a is initialized to 0. a++ returns the original value, 0, and then increments a to 1. Consequently 0 is stored in b.

```
Function PostIncrementDemo()
   int a = 0
   int b = a++
End
```

The pre-decrement and post-decrement work the same but decrement the variable rather than incrementing it.

Switch Statements

The switch construct can sometimes be used to simplify complicated flow control. It chooses one of several execution paths depending on a particular value.

Instead of a single form of switch statement, as is the case in C, Igor has two types: *switch* for numeric expressions and *strswitch* for string expressions. The basic syntax of these switch statements is as follows:

```
switch(<numeric expression>)        // numeric switch
   case <literal number or numeric constant>:
      <code>
      [break]
   case <literal number or numeric constant>:
      <code>
      [break]
   . . .
   [default:
      <code>]
endswitch
```

```
strswitch(<string expression>)      // string switch
   case <literal string or string constant>:
      <code>
      [break]
   case <literal string or string constant>:
      <code>
      [break]
   . . .
   [default:
      <code>]
endswitch
```

The switch numeric or string expression is evaluated. In the case of numeric switches, the result is rounded to the nearest integer. Execution proceeds with the code following the matching case label. When none of the case labels match, execution continues at the default label, if it is present, or otherwise the switch exits with no action taken.

All of the case labels must be numeric or string constant expressions and they must all have unique values within the switch statement. The constant expressions can either be literal values or they must be declared using the Constant and StrConstant keywords for numeric and string switches respectively.

Literal numbers used as case labels are required by the compiler to be integers. Numeric constants used as case labels are rounded by the compiler to the nearest integers.

Execution proceeds within each case until a break statement is encountered or the endswitch is reached. The break statement explicitly exits the switch construct. Usually, you should put a break statement at the end of each case. If you omit the break statement, execution continues with the next case label. Do this when you want to execute a single action for more than one switch value.

The following examples illustrate how switch constructs can be used in Igor:

```
Constant kThree=3
StrConstant ksHow="how"

Function NumericSwitch(a)
   Variable a

   switch(a)                     // numeric switch
      case 1:
         print "a is 1"
         break
      case 2:
         print "a is 2"
         break
      case kThree:
      case 4:
         print "a is 3 or 4"
         break
      default:
         print "a is none of those"
         break
   endswitch
End

Function StringSwitch(a)
   String a

   strswitch(a)                  // string switch
      case "hello":
         print "a is hello"
         break
      case ksHow:
         print "a is how"
         break
      case "are":
      case "you":
         print "a is are or you"
         break
      default:
         print "a is none of those"
         break
   endswitch
End
```

Loops

Igor implements two basic types of looping structures: do-while and for loops.

The do-while loop iterates through the loop code and tests an exit condition at the end of each iteration.

The for loop is more complex. The beginning of a for loop includes expressions for initializing and updating variables as well as testing the loop's exit condition at the start of each iteration.

Do-While Loop

The form of the do-while loop structure is:

```
do
   <loop body>
while(<expression>)
```

This loop runs until the expression evaluates to zero or until a break statement is executed.

This example will always execute the body of the loop at least once, like the do-while loop in C.

```
Function Test(lim)
   Variable lim      // We use this parameter as the loop limit.

   Variable sum=0
   Variable i=0      // We use i as the loop variable.
   do
      sum += i       // This is the body; equivalent to sum=sum+i.
      i += 1         // Increment the loop variable.
   while(i < lim)
   return sum
End
```

Nested Do-While Loops

A nested loop is a loop within a loop. Here is an example:

```
Function NestedLoopTest(numOuterLoops, numInnerLoops)
   Variable numOuterLoops, numInnerLoops

   Variable i, j

   i = 0
   do
      j = 0
      do
         <inner loop body>
         j += 1
      while (j < numInnerLoops)
      i += 1
   while (i < numOuterLoops)
End
```

While Loop

This fragment will execute the body of the loop zero or more times, like the while loop in C.

```
do
   if (i > lim)
      break        // This breaks out of the loop.
   endif
   <loop body>
   i += 1
while(1)           // This would loop forever without the break.
...                // Execution continues here after the break.
```

In this example, the loop increment is 1 but it can be any value.

For Loop

The basic syntax of a for loop is:

```
for(<initialize loop variable>; <continuation test>; <update loop variable>)
   <loop body>
endfor
```

Here is a simple example:

```
Function Example1()
   Variable i

   for(i=0; i<5; i+=1)
      print i
   endfor
End
```

The beginning of a for loop consists of three semicolon-separated expressions. The first is usually an assignment statement that initializes one or more variables. The second is a conditional expression used to determine if the loop should be terminated — if true, nonzero, the loop is executed; if false, zero, the loop terminates. The third expression usually updates one or more loop variables.

When a for loop executes, the initialization expression is evaluated only once at the beginning. Then, for each iteration of the loop, the continuation test is evaluated at the start of every iteration, terminating the loop if needed. The third expression is evaluated at the end of the iteration and usually increments the loop variable.

All three expressions in a for statement are optional and can be omitted independent of the others; only the two semicolons are required. The expressions can consist of multiple assignments, which must be separated by commas.

In addition to the continuation test expression, for loops may also be terminated by break or return statements within the body of the loop.

A continue statement executed within the loop skips the remaining body code and execution continues with the loop's update expression.

Here is a more complex example:

```
Function Example2()
   Variable i,j

   for(i=0,j=10; ;i+=1,j*=2)
      if (i == 2)
         continue
      endif
      Print i,j
      if (i == 5)
         break
      endif
   endfor
End
```

Range-Based For Loop

A range-based for loop iterates over all elements of a wave. The basic syntax is:

```
for(<type> varName : <wave expression>)
   <loop body>
endfor
```

The range-based for loop feature was added in Igor Pro 9.00.

During each iteration of the loop, the specified loop variable contains the value of the corresponding wave element. For example:

```
Function Example1()
   Make/O/T/N=(2,2) tw = "p=" + num2str(p) + ",q=" + num2str(q)
   for (String s : tw)
      Print s
   endfor
End
```

Executing Example1 prints this:

```
p=0,q=0
p=1,q=0
p=0,q=1
p=1,q=1
```

You can omit the loop variable type if the loop variable was defined earlier in the function. You can also omit it if the wave expression is a wave reference defined earlier in the function either explicitly or automatically as in Example1. The for statement in Example1 could be written like this:

```
for (s : tw)      // Omit "String" because the tw is a know wave reference
```

Here is an example in which the wave expression is not a wave reference so the loop variable type is required:

```
Function Example2()
   for (String s : ListToTextWave(IndependentModuleList(";"),";"))
      Print s
   endfor
End
```

Example2 could be rewritten like this:

```
Function Example3()
   WAVE/T tw = ListToTextWave(IndependentModuleList(";"),";")
   for (s : tw)
      Print s
   endfor
End
```

A range-based for loop iterates over all elements of the wave, regardless of its dimensionality. For a multidimensional wave, it iterates in column-major order. For a 2D wave, this means that it iterates over all rows of column 0, then all rows of column 1, and so on for each column.

If the wave is numeric, the type of the loop variable does not need to be an exact match for the wave type. For example, the loop variable can be double even if the wave is an integer wave.

If the wave is a text wave, the loop variable must be a string. If the wave is a data folder reference wave, the loop variable must be a DFREF. If the wave is a wave reference wave, the loop variable must be a WAVE.

The next example illustrates looping over all elements of a wave reference wave:

```
Function WaveWaveExample()
   Make/O wave0 = {0,1,2}
   Make/O/T textWave0 = {"A","B","C"}
   Make/WAVE/FREE ww = {wave0,textWave0}      // Make a wave reference wave
   for (WAVE w : ww)
      String name = NameOfWave(w)
      int dataType = WaveType(w)
      Printf "Name=%s, dataType=%d\r", name, dataType
   endfor
End
```

In this example, the loop variable is a wave reference named w. It holds a reference to a numeric wave (wave0) on the first iteration and to a text wave (textWave0) on the second.

Break Statement

A break statement terminates execution of do-while loops, for loops, and switch statements. The break statement continues execution with the statement after the *enclosing* loop's while, endfor, or endswitch statement. A nested do-while loop example demonstrates this:

```
   …
   Variable i=0, j

   do                // Starts outer loop.
      if (i > numOuterLoops)
         break       // Break #1, exits from outer loop.
      endif
      j = 0
      do             // Start inner loop.
         if (j > numInnerLoops)
            break    // Break #2, exits from inner loop only.
         endif
         j += 1
      while (1)      // Ends inner loop.
      …              // Execution continues here from break #2.
      i += 1
   while (1)         // Ends outer loop.
   …                 // Execution continues here from break #1.
```

Continue Statement

The continue statement can be used in do-while and for loops to short-circuit the loop and return execution back to the top of the loop. When Igor encounters a continue statement during execution, none of the code following the continue statement is executed during that iteration.

Flow Control for Aborts

Igor Pro includes a specialized flow control construct and keywords that you can use to test for and respond to abort conditions. The AbortOnRTE and AbortOnValue keywords can be used to trigger aborts, and the try-catch-endtry construct can be used to control program execution when an abort occurs. You can also trigger an abort by pressing the **User Abort Key Combinations** or by clicking the Abort button in the status bar.

These are advanced techniques. If you are just starting with Igor programming, you may want to skip this section and come back to it later.

AbortOnRTE Keyword

The AbortOnRTE (abort on runtime error) keyword can be used to raise an abort whenever a runtime error occurs. Use AbortOnRTE immediately after a command or sequence of commands for which you wish to catch a runtime error.

See **AbortOnRTE** on page V-18 for further details. For a usage example see **Simple try-catch-endtry Example** on page IV-49.

AbortOnValue Keyword

The AbortOnValue keyword can be used to abort function execution when a specified abort condition is satisfied. When AbortOnValue triggers an abort, it can also return a numeric abort code that you can use to characterize the cause.

See **AbortOnValue** on page V-19 for further details. For a usage example see **Simple try-catch-endtry Example** on page IV-49.

try-catch-endtry Flow Control

The try-catch-endtry flow control construct has the following syntax:

```
try
   <code>
catch
   <code to handle abort>
endtry
```

The try-catch-endtry flow control construct can be used to catch and respond to abnormal conditions in user functions. Code within the try and catch keywords tests for abnormal conditions and calls Abort, AbortOnRTE or AbortOnValue if appropriate. An abort also occurs if the user clicks the Abort button in the status bar or presses the **User Abort Key Combinations**.

When an abort occurs, execution immediately jumps to the first statement after the catch keyword. This catch code handles the abnormal condition, and execution then falls through to the first statement after endtry.

If no abort occurs, then the code between catch and endtry is skipped, and execution then falls through to the first statement after endtry.

When an abort occurs within the try-catch area, Igor stores a numeric code in the V_AbortCode variable. The catch code can use V_AbortCode to determine what caused the abort.

See **try-catch-endtry** on page V-1047 for further details.

Simple try-catch-endtry Example

We start with a simple example that illustrates the most common try-catch use. It catches and handles a runtime error.

```
Function DemoTryCatch0(name)
   String name              // Name to use for a new wave

   try
      Make $name            // Generates error if name is illegal or in use
      AbortOnRTE            // Abort if an error occurred

      // This code executes if no error occurred
      Printf "The wave '%s' was successfully created\r", name
   catch
      // This code executes if an error occurred in the try block
      Variable err = GetRTError(1)      // Get error code and clear error
      String errMessage = GetErrMessage(err, 3)
      Printf "While executing Make, the following error occurred: %s\r", errMessage
   endtry
End
```

Copy the code to the Procedure window of a new experiment and then execute:

```
DemoTryCatch0("wave0")
```

Since wave0 does not exist in a new experiment, DemoTryCatch0 creates it.

Now execute the same command again. This time the call to Make generates a runtime error because wave0 already exists. The call to AbortOnRTE generates an abort, because of the error generated by Make. The abort causes the catch code to run. It reports the error to the user and clears it to allow execution to continue.

Without the clearing of the runtime error, via the GetRTError call, the error would cause procedure execution to halt and Igor would report the error to the user via an error dialog.

Complex try-catch-endtry Example

The following example demonstrates how aborting flow control works. Copy the code to the Procedure window and execute the `DemoTryCatch1` function with 0 to 6 as input parameters. When you execute `DemoTryCatch1(6)`, the function loops until you click the Abort button at the right end of the status bar.

```
Function DemoTryCatch1(a)
   Variable a

   Print "A"
   try
      Print "B1"
      AbortOnValue  a==1 || a==2,33
      Print "B2"
      DemoTryCatch2(a)
      Print "B3"
      try
         Print "C1"
         if( a==4 || a==5 )
            Make $""; AbortOnRTE
         endif
         Print "C2"
      catch
         Print "D1"
         // will be runtime error so pass along to outer catch
         AbortOnValue a==5, V_AbortCode
         Print "D2"
      endtry
      Print "B4"
      if( a==6 )
         do
         while(1)
      endif
      Print "B5"
   catch
      Print "Abort code= ", V_AbortCode
      if( V_AbortCode == -4 )
         Print "Runtime error= ", GetRTError(1)
      endif
      if( a==1 )
         Abort "Aborting again"
      endif
      Print "E"
   endtry
   Print "F"
End
```

```
Function DemoTryCatch2(b)
   Variable b

   Print "DemoTryCatch2 A"
   AbortOnValue b==3,99
   Print "DemoTryCatch2 B"
End
```

User Abort Key Combinations

You can abort procedure execution by clicking the Abort button in the status bar or by pressing the following user abort key combinations:

Command-dot	Macintosh only
Ctrl+Break	Windows only
Shift+Escape	All platforms

Constants

You can define named numeric and string constants in Igor procedure files and use them in the body of user-defined functions.

Constants are defined in procedure files using following syntax:

```
Constant <name1> = <literal number> [, <name2> = <literal number>]
StrConstant <name1> = <literal string> [, <name2> = <literal string>]
```

For example:

```
Constant kIgorStartYear=1989, kIgorEndYear=2099
StrConstant ksPlatformMac="Macintosh", ksPlatformWin="Windows"

Function Test1()
   Variable v1 = kIgorStartYear
   String s1 = ksPlatformMac
   Print v1, s1
End
```

We suggest that you use the “k” prefix for numeric constants and the “ks” prefix for string constants. This makes it immediately clear that a particular keyword is a constant.

Constants declared like this are public and can be used in any function in any procedure file. A typical use would be to define constants in a utility procedure file that could be used from other procedure files as parameters to the utility routines. Be sure to use precise names to avoid conflicts with public constants declared in other procedure files.

If you are defining constants for use in a single procedure file, for example to improve readability or make the procedures more maintainable, you should use the `static` keyword (see **Static** on page V-906 for details) to limit the scope to the given procedure file.

```
static Constant kStart=1989, kEnd=2099
static StrConstant ksMac="Macintosh", ksWin="Windows"
```

Names for numeric and string constants are allowed to conflict with all other names. Duplicate constants of a given type are not allowed, except for static constants in different files and when used with the override keyword. The only true conflict is with variable names and with certain built-in functions that do not take parameters such as pi. Variable names, including local variable names, waves, NVARs, and SVARs, override constants, but constants override functions such as pi.

Pragmas

A pragma is a statement in a procedure file that sets a compiler mode or passes other information from the programmer to Igor. The form of a pragma statement is:

```
#pragma keyword [= parameter]
```

The pragma statement must be flush against the left margin of the procedure window, with no indentation.

Igor ignores unknown pragmas such as pragmas introduced in later versions of the program.

Currently, Igor supports the following pragmas:

```
#pragma rtGlobals = value
#pragma version = versionNumber
#pragma IgorVersion = versionNumber
#pragma hide = value
#pragma ModuleName = name
#pragma IndependentModule = name
#pragma TextEncoding = textEncodingNameStr // Igor Pro 7.00
#pragma DefaultTab = {mode, widthInPoints, widthInSpaces} // Igor Pro 9.00
```

The effect of a pragma statement lasts until the end of the procedure file that contains it.

The rtGlobals Pragma

The rtGlobals pragma controls aspects of the Igor compiler and runtime error checking in user-defined functions.

Prior to Igor Pro 3, to access a global (wave or variable) from a user-defined function, the global had to already exist. Igor Pro 3 introduced "runtime lookup of globals" under which the Igor compiler did not require globals to exist at compile time but rather connected references, declared with WAVE, NVAR and SVAR statements, to globals at runtime. Igor Pro 6.20 introduced stricter compilation of wave references and runtime checking of wave index bounds.

You enable and disable these behaviors using an rtGlobals pragma. For example:

```
#pragma rtGlobals = 3   // Strict wave reference mode, runtime bounds checking
```

A given rtGlobals pragma governs just the procedure file in which it appears. The pragma must be flush left in the procedure file and is typically put at the top of the file.

The rtGlobals pragma is defined as follows:

#pragma rtGlobals=0 Specifies the old, pre-Igor Pro 3 behavior.

This is no longer supported and acts like rtGlobals=1.

#pragma rtGlobals=1 Turns on runtime lookup of globals.

This is the default setting if there is no rtGlobals pragma in a given procedure file.

#pragma rtGlobals=2 Forces old experiments out of compatibility mode. This is superceded by rtGlobals=3. It is described under **Legacy Code Issues** on page IV-113.

#pragma rtGlobals=3 Turns on runtime lookup of globals, strict wave references and runtime checking of wave index bounds. Requires Igor Pro 6.2 or later.

rtGlobals=3 is recommended.

Under strict wave references (rtGlobals=3), you must create a wave reference for any use of a wave. Without strict wave references (rtGlobals=1), you do not need to create a wave reference unless the wave is used in an assignment statement. For example:

```
Function Test()
   jack = 0       // Error under rtGlobals=1 and under rtGlobals=3
   Display jack   // OK under rtGlobals=1, error under rtGlobals=3

   Wave jack      // jack is now a wave reference rather than a bare name
   Display jack   // OK under rtGlobals=1 and under rtGlobals=3
End
```

Even with rtGlobals=3, this compiles without error:

```
Function Test()
   // Make creates an automatic wave reference when used with a simple name
   Make jack
   Display jack   // OK under rtGlobals=1 and rtGlobals=3
End
```

See **Automatic Creation of WAVE References** on page IV-72 for details.

Under runtime wave index checking (rtGlobals=3), Igor reports an error if a wave index is out-of-bounds:

```
Function Test()
   Make/O/N=5 jack = 0  // Creates automatic wave reference

   jack[4] = 123        // OK
   jack[5] = 234        // Runtime error under rtGlobals=3.
                        // Clipped under rtGlobals=1.

   Variable index = 5
   jack[index] = 234    // Runtime error under rtGlobals=3.
                        // Clipped under rtGlobals=1.

   // Create and use a dimension label for point 4 of jack
   SetDimLabel 0,4,four,jack
   jack[%four] = 234    // OK

   // Use a non-existent dimension label.
   jack[%three] = 345   // Runtime error under rtGlobals=3.
                        // Clipped under rtGlobals=1.
   // Under rtGlobals=1, this statement writes to point 0 of jack.
End
```

In Igor Pro 9.00 and later, Igor does runtime wave type checking which reports an error if a wave's type does not match the wave reference's type. In this example, we assign a numeric wave to a text wave reference:

```
#pragma rtGlobals=3
Function DemoRuntimeWaveTypeCheck()
   // WRONG: Assigning a numeric wave to a text wave reference
   WAVE/T tw = NewFreeWave(4,3)

   // Error is detected when the incorrect wave reference is used:
   // "An attempt was made to treat a numeric wave as if it were a text wave"
   String str = tw[0]
End
```

You can disable runtime wave type checking by executing:

```
SetIgorOption ReportWaveTypeMismatch = 0  // Turn runtime type checking off
```

You can ignore it for a specific line of code using GetRTError on the same line as the error:

```
String str = w[0]; int err = GetRTError(1)
```

See also: **Runtime Lookup of Globals** on page IV-65
Automatic Creation of WAVE References on page IV-72
Automatic Creation of WAVE, NVAR and SVAR References on page IV-70
Legacy Code Issues on page IV-113

The version Pragma

The version pragma sets the version of the procedure file. It is optional and is of interest mostly if you are the developer of a package used by a widespread group of users.

For details on the version pragma, see **Procedure File Version Information** on page IV-166.

The IgorVersion Pragma

The IgorVersion pragma is also optional and of interest to developers of packages. It gives you a way to prevent procedures from compiling under versions of Igor Pro older than the specified version number.

For example, the statement:

```
#pragma IgorVersion = 7.00
```

tells Igor that the procedure file requires Igor Pro 7.00 or later.

The hide Pragma

The hide pragma allows you to make a procedure file invisible.

For details on the hide pragma, see **Invisible Procedure Files** on page III-402.

The ModuleName Pragma

The ModuleName pragma gives you the ability to use static functions and proc pictures in a global context, such as in the action procedure of a control or on the command line. Using this pragma entails a two step process: define a name for the procedure file, and then use a special syntax to access objects in the named procedure file.

To define a module name for a procedure file use the format:

```
#pragma ModuleName = name
```

This statement associates the specified module name with the procedure file in which the statement appears.

You can then use objects from the named procedure file by preceding the object name with the name of the module and the # character. For example:

```
#pragma ModuleName = MyModule

Static Function Test(a)
   Variable a

   return a+100
End
```

Then, on the command line or from another procedure file, you can execute:

```
Print MyModule#Test(3)
```

Choose a distinctive module name so that it does not clash with module names used by other programmers. WaveMetrics uses module names with a `WM_` prefix, so you must not use such names.

For further discussion see **Regular Modules** on page IV-236.

The IndependentModule Pragma

The IndependentModule pragma is a way for you to designate groups of one or more procedure files to be compiled and linked separately. Once compiled and linked, the code remains in place and is usable even though other procedures may fail to compile. This allows your control panels, menus, and procedures to continue to work regardless of user programming errors.

A file is designated as an independent module using

```
#pragma IndependentModule=imName
```

This is similar to `#pragma ModuleName=modName` (see **The ModuleName Pragma** on page IV-54) and, just as in the case of calling static functions in a procedure with #pragma ModuleName, calling nonstatic function in an IndependentModule from outside the module requires the use of *imName#functionName*() syntax.

To call any function, static or not, defined in an independent module from outside that module, you must qualify the call with the independent module name:

```
MyIndependentModule#Test()
```

For further discussion see **Independent Modules** on page IV-238.

The rtFunctionErrors Pragma

Normally, when an error occurs in a built-in function, the built-in function does not post an error but rather returns 0, NaN, or an empty string as the function result. For example, by default, the str2num built-in function in this user-defined function returns NaN but does not post an error:

```
Function Test()
   Variable val = str2num("abc")     // Returns NaN
   Print val
End
```

As a debugging aid, you can force Igor to post an error that occurs in a built-in function called from a user-defined function by adding this statement to the procedure file:

```
#pragma rtFunctionErrors = 1
```

Now, if you run the Test function, Igor displays an error dialog telling you that the str2num function expects a number.

The rtFunctionErrors pragma works for most errors in built-in functions called from user-defined functions, but not in all.

The rtFunctionErrors pragma was added in Igor Pro 7.00. Earlier versions of Igor ignore it.

The TextEncoding Pragma

The TextEncoding pragma provides a way for a programmer to mark an Igor procedure file as using a particular text encoding. This pragma was added in Igor Pro 7.00 and is ignored by earlier versions.

Here is an example:

```
#pragma TextEncoding = "UTF-8"
```

This pragma tells Igor7 or later that the procedure file is encoded using UTF-8, a form of Unicode. All new procedure files should use UTF-8.

Like all pragmas, the TextEncoding pragma must be flush against the left margin of the procedure file. It is an error to have more than one TextEncoding pragma in a given procedure file. The TextEncoding pragma can appear anywhere in the file and affects the whole file. By convention it is placed near the top of the file.

If your procedure file uses only ASCII characters then all versions of Igor on all platforms will interpret it correctly and there is no need for a TextEncoding pragma.

If your procedure file uses non-ASCII characters then we recommend that you add a TextEncoding pragma as this will allow Igor to reliably interpret it regardless of the user's operating system, system locale and default text encoding setting.

If the procedure file is to be used in Igor7 and later only then you should use the UTF-8 text encoding, like this:

```
#pragma TextEncoding = "UTF-8"
```

This allows you to use any Unicode character in the file.

The following text encodings are not supported for procedure files:

```
UTF-16LE, UTF-16BE, UTF32-LE, UTF32-BE
```

If you attempt to use these in a procedure file Igor will report an error.

Igor automatically adds a TextEncoding pragma under some circumstances.

It adds a TextEncoding pragma if you set the text encoding of a procedure file by clicking the Text Encoding button at the bottom of the window or by choosing Procedure→Text Encoding.

It adds a TextEncoding pragma if it displays the Choose Text Encoding dialog when you open a procedure file because it is unable to convert the text to UTF-8 without your help.

It is possible for the TextEncoding pragma to be in conflict with the text encoding used to open the file. For example, imagine that you open a file containing

```
#pragma TextEncoding = "Windows-1252"
```

and then you change "Windows-1252" to "UTF-8" by editing the file. You have now created a conflict where the text encoding used to open the file was one thing but the TextEncoding pragma is another. To resolve this conflict, Igor displays the Resolve Text Encoding Conflict dialog. In the dialog you can choose to make the file's text encoding match the TextEncoding pragma or to make the TextEncoding pragma match the file's text encoding. You can also cancel the dialog and manually edit the TextEncoding pragma.

For background information, see **Text Encodings** on page III-459.

The DefaultTab Pragma

The DefaultTab pragma allows you to set the width of a procedure file's default tabs to facilitate aligning comments in the file. It was added in Igor Pro 9.00 and is ignored by earlier versions of Igor. See **The Default Tab Pragma For Procedure Files** on page III-406 for details.

Unknown Pragmas

Igor ignores pragmas that it does not know about. This allows newer versions of Igor to use new pragmas while older versions ignore them. The downside of this change is that, if you misspell a pragma keyword, Igor will not warn you about it.

Proc Pictures

Proc pictures are binary PNG or JPEG images encoded as printable ASCII text in procedure files. They are intended for programmers who need to display pictures as part of the user interface for a procedure package. They can be used with the **DrawPICT** operation and with the picture keyword of the **Button**, **CheckBox**, and **CustomControl** operations.

The syntax for defining and using a proc picture is illustrated in the following example:

```
// PNG: width= 56, height= 44
Picture MyGlobalPicture
   ASCII85Begin
   M,6r;%14!\!!!!.8Ou6I!!!!Y!!!!M#Qau+!5G;q_uKc;&TgHDFAm*iFE_/6AH5;7DfQssEc39jTBQ
   =U!7FG,5u`*!m?g0PK.mR"U!k63rtBW)]$T)Q*!=Sa1TCDV*V+l:Lh^NW!fu1>;(.<VU1bs4L8&@Q_
   <4e(%"^F50:Jg6);j!CQdUA[dh6]%[OkHSC,ht+Q7ZO#.6U,IgfSZ!R1g':oO_iLF.GQ@RF[/*G98D
   bjE.g?NCte(pX-($m^\_FhhfL`D9uO6Qi5c[r4849Fc7+*)*O[tY(6<rkm^)/KLIc]VdDEbF-n5&Am
   2^hbTu:U#8ies_W<LGkp_LEU1bs4L8&?fqRJ[h#sVSSz8OZBBY!QNJ
   ASCII85End
End

Function Demo()
   NewPanel
   DrawPict 0,0,1,1,ProcGlobal#MyGlobalPicture
End
```

The ASCII text in the `MyGlobalPicture` procedure between the `ASCII85Begin` and `ASCII85End` is similar to output from the Unix `btoa` command, but with the header and trailer removed.

You can create proc pictures in Igor Pro from normal, global pictures using the Pictures dialog (Misc menu) which shows the experiment's picture gallery. Select a picture in the dialog and click the Copy Proc Picture button to place the text on the dlipboard. Then paste it in your procedure file. If the existing picture is not a JPEG or PNG, it is converted to PNG.

Proc pictures can be either global or local in scope. Global pictures can be used in all procedure files. Local pictures can be used only within the procedure file in which they are defined. Proc pictures are global by default and the picture name must be unique for all open procedure files.

Proc pictures can be defined in global procedure files (not in a regular module or independent module), in regular modules (see **Regular Modules** on page IV-236), or independent modules (see **Independent Modules** on page IV-238).

Proc Pictures in Global Procedure Files

Here is an example of a proc picture in a global procedure file:

```
Picture MyGlobalPicture
   ASCII85Begin
   ...
   ASCII85End
End
```

To draw a proc picture defined in a global procedure file you must qualify the picture name with the ProcGlobal keyword:

```
DrawPICT 0,0,1,1,ProcGlobal#MyGlobalPicture
```

A proc picture defined in a global procedure file can be used in any procedure file using the qualified name.

Proc Pictures in Regular Modules

Here is an example of a proc picture in a regular module:

```
#pragma ModuleName = MyRegularModule

static Picture MyRegularPicture
   ASCII85Begin
   ...
   ASCII85End
End
```

Notice the use of the static keyword. The puts the picture name in the namespace of the regular module.

To draw a proc picture defined in a regular module you must qualify the picture name with the name of the regular module:

```
DrawPICT 0,0,1,1,MyRegularModule#MyRegularPicture
```

A proc picture defined in a regular module is usually intended to be used in that module only but can also be used from a global procedure file using the qualified name. It can not be used from an independent module.

Proc Pictures in Independent Modules

Here is an example of a proc picture in an independent module:

```
#pragma IndependentModule = MyIndependentModule

Picture MyIndependentPicture
   ASCII85Begin
   ...
   ASCII85End
End
```

The static keyword is not used but the the picture name is still in the namespace of the independent module.

To draw a proc picture defined in an independent module you must qualify the picture name with the name of the independent module:

```
DrawPICT 0,0,1,1,MyIndependentModule#MyIndependentPicture
```

A proc picture defined in an independent module is usually intended to be used in that module only but can also be used from any procedure file using the qualified name.

How Parameters Work

There are two ways of passing parameters from a routine to a subroutine:

- Pass-by-value
- Pass-by-reference

“Pass-by-value” means that the routine passes the *value* of an expression to the subroutine. “Pass-by-reference” means that the routine passes *access to a variable* to the subroutine. The important difference is that, in pass-by-reference, the subroutine can change the original variable in the calling routine.

Like C++, Igor allows either method for numeric and string variables. You should use pass-by-value in most cases and reserve pass-by-reference for those situations in which you need multiple return values.

Example of Pass-By-Value

```
Function Routine()
   Variable v = 4321
   String s = "Hello"

   Subroutine(v, s)
End

Function Subroutine(v, s)
   Variable v
   String s

   Print v, s

   // These lines have NO EFFECT on the calling routine.
   v = 1234
   s = "Goodbye"
End
```

Note that v and s are *local variables* in Routine. In Subroutine, they are *parameters* which act very much like local variables. The names “v” and “s” are local to the respective functions. The v in Subroutine is not the same variable as the v in Routine although it initially has the same value.

The last two lines of Subroutine set the value of the local variables v and s. They have no effect on the value of the variables v and s in the calling Routine. What is passed to Subroutine is the numeric value 4321 and the string value “Hello”.

Pass-By-Reference

You can specify that a parameter to a function is to be passed by reference rather than by value. In this way, the function called can change the value of the parameter and update it in the calling function. This is much like using pointers to arguments in C++. This technique is needed and appropriate only when you need to return more than one value from a function.

Functions with pass-by-reference parameters can only be called from other functions — not from the command line.

Only numeric variables (declared by Variable, double, int, int64, uint64 or complex), string variables, structures, DFREF, and WAVE variables can be passed by reference. Structures are always passed by reference. Pass by reference DFREF and WAVE variables were added in Igor Pro 8.00.

The variable or string being passed must be a local variable and can not be a global variable. To designate a variable or string parameter for pass-by-reference, simply prepend an ampersand symbol before the name in the parameter declaration:

```
Function Subroutine(num1,num2,str1)
   Variable &num1, num2
   String &str1

   num1= 12+num2
   str1= "The number is"
End
```

and then call the function with the name of a local variable in the reference slot:

```
Function Routine()
   Variable num= 1000
   String str= "hello"
   Subroutine(num,2,str)
   Print str, num
End
```

When executed, Routine prints “The number is 14” rather than “hello 1000”, which would be the case if pass-by-reference were not used.

A pass-by-reference parameter can be passed to another function that expects a reference:

```
Function SubSubroutine(b)
   Variable &b
   b= 123
End

Function Subroutine(a)
   Variable &a
   SubSubroutine(a)
End

Function Routine()
   Variable num
   Subroutine(num)
   print num
End
```

You can not pass NVARs, SVARS, or FUNCREFs by reference to a function. You can use a structure containing fields of these types to achieve the same end.

How Waves Are Passed

Here is an example of a function “passing a wave” to a subroutine.

```
Function Routine()
   Make/O wave0 = x
   Subroutine(wave0)
End

Function Subroutine(w)
   WAVE w

   w = 1234    // This line DOES AFFECT the wave referred to by w.
End
```

We are really not passing a wave to Subroutine, but rather we are passing a reference to a wave. The parameter `w` is the wave reference.

Waves are global objects that exist independent of functions. The subroutine can use the wave reference to modify the contents of the wave. Using the terminology of “pass-by-value” and “pass-by-reference”, the wave reference is passed by value, but this has the effect of “passing the wave” by reference.

In Igor Pro 8.00 or later, you can specify that a WAVE reference variable itself be passed by reference. This allows you to change the WAVE reference variable so that it references a different wave. For example:

```
Function Routine()
   Make/O wave0 = {1, 2, 3}
   WAVE w = wave0
   Print "Original:", w
   Subroutine(w)
   Print "New:", w
End

Function Subroutine(WAVE& ww)
   Make/O wave1 = {5, 6}
   WAVE ww = wave1// Changes w variable in Routine to reference wave1
End
```

Executing Routine prints:

```
Original:
wave0[0]= {1,2,3}
  New:
wave1[0]= {5,6}
```

Using Optional Parameters

Following is an example of a function with two optional input parameters:

```
Function OptionalParameterDemo(a,b, [c,d])
   Variable a,b,c,d           // c and d are optional parameters

   // Determine if the optional parameters were supplied
   if( ParamIsDefault(c) && ParamIsDefault(d) )
      Print "Missing optional parameters c and d."
   endif

   Print a,b,c,d
End
```

Executing on the command line with none of the optional inputs:

```
OptionalParameterDemo(8,6)
  Missing optional parameters c and d.
  8  6  0  0
```

Executing with an optional parameter as an input:

```
OptionalParameterDemo(8,6, c=66)
  8  6  66  0
```

Note that the optional parameter is explicitly defined in the function call using the *ParamName=Value* syntax. All optional parameters must come after any required function parameters.

Local Versus Global Variables

Numeric and string variables can be **local** or **global**. “Local” means that the variable can be accessed only from within the procedure in which it was created. “Global” means that the variable can be accessed from the command line or from within any procedure. The following table shows the characteristics of local and global variables:

Local Variables	Global Variables
Are part of a procedure.	Are part of the data folder hierarchy of an Igor experiment.
Created using Variable or String within a procedure.	Created using Variable or String or Variable/G or String/G from the command line or within a procedure.
Used to store temporary results while the procedure is executing.	Used to store values that are saved from one procedure invocation to the next.
Cease to exist when the procedure ends.	Exist until you use KillVariables, KillStrings, or KillDataFolder.
Can be accessed only from within the procedure in which they were created.	Can be accessed from the command line or from within any procedure.

Local variables are private to the function in which they are declared and cease to exist when the function exits. Global variables are public and persistent — they exist until you explicitly delete them.

If you write a function whose main purpose is to display a dialog to solicit input from the user, you may want to store the user’s choices in global variables and use them to initialize the dialog the next time the user invokes it. This is an example of saving values from one invocation of a function to the next.

If you write a set of functions that loads, displays and analyzes experimentally acquired data, you may want to use global variables to store values that describe the conditions under which the data was acquired and to store the results from the analyses. These are examples of data accessed by many procedures.

With rare exceptions, you should not use global variables to pass information to procedures or from one procedure to another. Instead, use parameters. Using global variables for these purposes creates a web of hidden dependencies which makes understanding, reusing, and debugging procedures difficult.

Local Variables Used by Igor Operations

When invoked from a function, many Igor’s operations return results via local variables. For example, the WaveStats operation creates a number of local variables with names such as V_avg, V_sigma, etc. The following function illustrates this point.

```
Function PrintWaveAverage(w)
   WAVE w

   WaveStats/Q w
   Print V_avg
End
```

When the Igor compiler compiles the WaveStats operation, it creates various local variables, V_avg among them. When the WaveStats operation runs, it stores results in these local variables.

In addition to creating local variables, a few operations, such as CurveFit and FuncFit, check for the existence of specific local variables to provide optional behavior. For example:

```
Function ExpFitWithMaxIterations(w, maxIterations)
   WAVE w
   Variable maxIterations

   Variable V_FitMaxIters = maxIterations

   CurveFit exp w
End
```

The CurveFit operation looks for a local variable named V_FitMaxIters, which sets the maximum number of iterations before the operation gives up.

The documentation for each operation lists the special variables that it creates or looks for.

Converting a String into a Reference Using $

The $ operator converts a string expression into an object reference. The referenced object is usually a wave but can also be a global numeric or global string variable, a window, a symbolic path or a function. This is a common and important technique.

We often use a *string* to pass the *name* of a wave to a procedure or to algorithmically generate the name of a wave. Then we use the $ operator to convert the string into a wave reference so that we can operate on the wave.

The following trivial example shows why we need to use the $ operator:

```
Function MakeAWave(str)
   String str

   Make $str
End
```

Executing

```
MakeAWave("wave0")
```

creates a wave named wave0.

Here we use $ to convert the contents of the string parameter str into a name. The function creates a wave whose name is stored in the str string parameter.

If we omitted the $ operator, we would have

```
Make str
```

This would create a wave named str, not a wave whos name is specified by the contents of str.

As shown in the following example, $ can create references to global numeric and string variables as well as to waves.

```
Function Test(vStr, sStr, wStr)
   String vStr, sStr, wStr

   NVAR v = $vStr        // v is local name for global numeric var
   v += 1
   SVAR s = $sStr        // s is local name for global string var
   s  += "Hello"
   WAVE w = $wStr        // w is local name for global wave
   w += 1
End

Variable/G gVar = 0; String/G gStr = ""; Make/O/N=5 gWave = p
Test("gVar", "gStr", "gWave")
```

The NVAR, SVAR and WAVE references are necessary in functions so that the compiler can identify the kind of object. This is explained under **Accessing Global Variables and Waves** on page IV-65.

Using $ to Refer to a Window

A number of Igor operations modify or create windows, and optionally take the name of a window. You need to use a string variable if the window name is not determined until run time but must convert the string into a name using $.

For instance, this function creates a graph using a name specified by the calling function:

```
Function DisplayXY(xWave, yWave, graphNameStr)
   Wave xWave, yWave
   String graphNameStr     // Contains name to use for the new graph

   Display /N=$graphNameStr yWave vs xWave
End
```

The $ operator in /N=$graphNameStr converts the contents of the string graphNameStr into a graph name as required by the Display operation /N flag. If you forget $, the command would be:

```
Display /N=graphNameStr yWave vs xWave
```

This would create a graph literally named graphNameStr.

Using $ In a Data Folder Path

$ can also be used to convert a string to a name in a data folder path. This is used when one of many data folders must be selected algorithmically.

Assume you have a string variable named dfName that tells you in which data folder a wave should be created. You can write:

```
Make/O root:$(dfName):wave0
```

The parentheses are necessary because the $ operator has low precedence.

Using $ Examples

This function illustrates various uses of $ in user-defined functions:

```
Function Demo()
    String s = "wave0"        // A string containing a name

   // Make requires a name, not a string
   Make/O $s

   // A wave declaration requires a name on the righthand side, not a string
   WAVE w = $s                // w is a wave reference

   // Display requires a wave reference
   Display w

   // ModifyGraph requires a trace name, not a string or a wave reference
   ModifyGraph mode($s) = 0

   AppendToGraph w            // Add another trace showing wave0

   // #1 is "instance notation" to distinguish multiple traces from same wave
   String t = "wave0#1"

   // ModifyGraph requires a trace name, not a string or a wave reference
   ModifyGraph mode($t) = 3
End
```

Compile Time Versus Runtime

Because Igor user-defined functions are compiled, errors can occur during compilation (“compile time”) or when a function executes (“runtime”). It helps in programming if you have a clear understanding of what these terms mean.

Compile time is when Igor analyzes the text of all functions and produces low-level instructions that can be executed quickly later. This happens when you modify a procedure window and then:

- Choose Compile from the Macros menu.
- Click the Compile button at the bottom of a procedure window.
- Activate a window other than a procedure or help window.

Runtime is when Igor actually executes a function’s low-level instructions. This happens when:

- You invoke the function from the command line.
- The function is invoked from another procedure.
- Igor updates a dependency which calls the function.
- You use a button or other control that calls the function.

Conditions that exist at compile time are different from those at runtime. For example, a function can reference a global variable. The global does not need to exist at compile time, but it does need to exist at runtime. This issue is discussed in detail in the following sections.

Here is an example illustrating the distinction between compile time and runtime:

```
Function Example(w)
   WAVE w

   w = sin(x)
   FFT w
   w = r2polar(w)
End
```

The declaration “`WAVE w`” specifies that w is expected to be a real wave. This is correct until the FFT executes and thus the first wave assignment produces the correct result. After the FFT is executed at runtime, however, the wave becomes complex. The Igor compiler does not know this and so it compiled the second wave assignment on the assumption that w is real. A compile-time error will be generated complaining that r2polar is not available for this number type — i.e., real. To provide Igor with the information that the wave is complex after the FFT you need to rewrite the function like this:

```
Function Example(w)
   WAVE w

   w= sin(x)
   FFT w
   WAVE/C wc = w
   wc = r2polar(wc)
End
```

A statement like “`WAVE/C wc = w`” has the compile-time behavior of creating a symbol, wc, and specifying that it refers to a complex wave. It has the runtime behavior of making wc refer to a specific wave. The runtime behavior can not occur at compile time because the wave may not exist at compile time.

Accessing Global Variables and Waves

Global numeric variables, global string variables and waves can be referenced from any function. A function can refer to a global that does not exist at compile-time. For the Igor compiler to know what type of global you are trying to reference, you need to declare references to globals.

Consider the following function:

```
Function BadExample()
   gStr1 = "The answer is:"
   gNum1 = 1.234
   wave0 = 0
End
```

The compiler can not compile this because it doesn't know what gStr1, gNum1 and wave0 are. We need to specify that they are references to a global string variable, a global numeric variable and a wave, respectively:

```
Function GoodExample1()
   SVAR gStr1 = root:gStr1
   NVAR gNum1 = root:gNum1
   WAVE wave0 = root:wave0

   gStr1 = "The answer is:"
   gNum1 = 1.234
   wave0 = 0
End
```

The SVAR statement specifies two important things for the compiler: first, that gStr1 is a global string variable; second, that gStr1 refers to a global string variable named gStr1 in the root data folder. Similarly, the NVAR statement identifies gNum1 and the WAVE statement identifies wave0. With this knowledge, the compiler can compile the function.

The technique illustrated here is called "runtime lookup of globals" because the compiler compiles code that associates the symbols gStr1, gNum1 and wave0 with specific global variables at runtime.

Runtime Lookup of Globals

The syntax for runtime lookup of globals is:

```
NVAR <local name1>[= <path to var1>][, <loc name2>[= <path to var2>]]…
SVAR <local name1>[= <path to str1>][, <loc name2>[= <path to str2>]]…
WAVE <local name1>[= <path to wave1>][, <loc name2>[= <path to wave2>]]…
```

NVAR creates a reference to a global numeric variable.

SVAR creates a reference to a global string variable.

WAVE creates a reference to a wave.

At compile time, these statements identify the referenced objects. At runtime, the connection is made between the local name and the actual object. Consequently, the object must exist when these statements are executed.

<local name> is the name by which the global variable, string or wave is to be known within the user function. It does not need to be the same as the name of the global variable. The example function could be rewritten as follows:

```
Function GoodExample2()
   SVAR str1 = root:gStr1      // str1 is the local name.
   NVAR num1 = root:gNum1      // num1 is the local name.
   WAVE w = root:wave0         // w is the local name.

   str1 = "The answer is:"
   num1 = 1.234
   w = 0
End
```

If you use a local name that is the same as the global name, and if you want to refer to a global in the current data folder, you can omit the <path to …> part of the declaration:

```
Function GoodExample3()
   SVAR gStr1           // Refers to gStr1 in current data folder.
   NVAR gNum1           // Refers to gNum1 in current data folder.
   WAVE wave0           // Refers to wave0 in current data folder.

   gStr1 = "The answer is:"
   gNum1 = 1.234
   wave0 = 0
End
```

GoodExample3 accesses globals in the current data folder while GoodExample2 accesses globals in a specific data folder.

If you use <path to …>, it may be a simple name (gStr1) or it may include a full or partial path to the name.

The following are valid examples, referencing a global numeric variable named gAvg:

```
   NVAR gAvg= gAvg
   NVAR avg= gAvg
   NVAR gAvg
   NVAR avg= root:Packages:MyPackage:gAvg
   NVAR avg= :SubDataFolder:gAvg
   NVAR avg= $"gAvg"
   NVAR avg= $("g"+ "Avg")
   NVAR avg= ::$"gAvg"
```

As illustrated above, the local name can be the same as the name of the global object and the lookup expression can be either a literal name or can be computed at runtime using $<string expression>.

If your function creates a global variable and you want to create a reference to it, put the NVAR statement after the code that creates the global. The same applies to SVARs and, as explained next, to WAVEs.

Put WAVE Declaration After Wave Is Created

A wave declaration serves two purposes. At compile time, it tells Igor the local name and type of the wave. At runtime, it connects the local name to a specific wave. In order for the runtime purpose to work, you must put wave declaration after the wave is created.

```
Function BadExample()
   String path = "root:Packages:MyPackage:wave0"
   Wave w = $path          // WRONG: Wave does not yet exist.
   Make $path
   w = p                   // w is not connected to any wave.
End

Function GoodExample()
   String path = "root:Packages:MyPackage:wave0"
   Make $path
   Wave w = $path          // RIGHT
   w = p
End
```

Both of these functions will successfully compile. BadExample will fail at runtime because w is not associated with a wave, because the wave does not exist when the "Wave w = $path" statement executes.

This rule also applies to NVAR and SVAR declarations.

Runtime Lookup Failure

At runtime, it is possible that a NVAR, SVAR or WAVE statement may fail. For example,

```
NVAR v1 = var1
```

will fail if var1 does not exist in the current data folder when the statement is executed. You can use the NVAR_Exists, SVAR_Exists, and WaveExists functions to test if a given global reference is valid:

```
Function Test()
   NVAR/Z v1 = var1
   if (NVAR_Exists(v1))
      <use v1>
   endif
End
```

The /Z flag is necessary to prevent an error if the NVAR statement fails. You can also use it with SVAR and WAVE.

A common cause for failure is putting a WAVE statement in the wrong place. For example:

```
Function BadExample()
   WAVE w = resultWave
   <Call a function that creates a wave named resultWave>
   Display w
End
```

This function will compile successfully but will fail at runtime. The reason is that the `WAVE w = resultWave` statement has the runtime behavior of associating the local name w with a particular wave. But that wave does not exist until the following statement is executed. The function should be rewritten as:

```
Function GoodExample()
   <Call a function that creates a wave named resultWave>
   WAVE w = resultWave
   Display w
End
```

Runtime Lookup Failure and the Debugger

You can break whenever a runtime lookup fails using the symbolic debugger (described in Chapter IV-8, **Debugging**). It is a good idea to do this, because it lets you know about runtime lookup failures at the moment they occur.

Sometimes you may create a WAVE, NVAR or SVAR reference knowing that the referenced global may not exist at runtime. Here is a trivial example:

```
Function Test()
   WAVE w = testWave
   if (WaveExists(testWave))
      Printf "testWave had %d points.\r", numpnts(testWave)
   endif
End
```

If you enable the debugger's WAVE checking and if you execute the function when testWave does not exist, the debugger will break and flag that the WAVE reference failed. But you wrote the function to handle this situation, so the debugger break is not helpful in this case.

The solution is to rewrite the function using WAVE/Z instead of just WAVE. The /Z flag specifies that you know that the runtime lookup may fail and that you don't want to break if it does. You can use NVAR/Z and SVAR/Z in a similar fashion.

Accessing Complex Global Variables and Waves

You must specify if a global numeric variable or a wave is complex using the /C flag:

```
   NVAR/C gc1 = gc1
   WAVE/C gcw1 = gcw1
```

Accessing Text Waves

Text waves must be accessed using the /T flag:

```
   WAVE/T tw= MyTextWave
```

Accessing Global Variables and Waves Using Liberal Names

There are two ways to initialize a reference an Igor object: using a literal name or path or using a string variable. For example:

```
Wave w = root:MyDataFolder:MyWave               // Using literal path
String path = "root:MyDataFolder:MyWave"
Wave w = $path                                  // Using string variable
```

Things get more complicated when you use a liberal name rather than a standard name. A standard name starts with a letter and includes letters, digits and the underscore character. A liberal name includes other characters such as spaces or punctuation.

In general, you must quote liberal names using single quotes so that Igor can determine where the name starts and where it ends. For example:

```
Wave w = root:'My Data Folder':'My Wave'        // Using literal path
String path = "root:'My Data Folder':'My Wave'"
Wave w = $path                                  // Using string variable
```

However, there is an exception to the quoting requirement. The rule is:

> You must quote a literal liberal name and you must quote a liberal path stored in a string variable but you must not quote a simple literal liberal name stored in a string variable.

The following functions illustrate this rule:

```
// Literal liberal name must be quoted
Function DemoLiteralLiberalNames()
   NewDataFolder/O root:'My Data Folder'

   Make/O root:'My Data Folder':'My Wave' // Literal name must be quoted

   SetDataFolder root:'My Data Folder'    // Literal name must be quoted

   Wave w = 'My Wave'                     // Literal name must be quoted
   w = 123

   SetDataFolder root:
End

// String liberal PATH must be quoted
Function DemoStringLiberalPaths()
   String path = "root:'My Data Folder'"
   NewDataFolder/O $path

   path = "root:'My Data Folder':'My Wave'"  // String path must be quoted
   Make/O $path

   Wave w = $path
   w = 123

   SetDataFolder root:
End

// String liberal NAME must NOT be quoted
Function DemoStringLiberalNames()
   SetDataFolder root:

   String dfName = "My Data Folder"       // String name must NOT be quoted
   NewDataFolder/O $dfName

   String wName = "My Wave"               // String name must NOT be quoted
   Make/O root:$(dfName):$wName

   Wave w = root:$(dfName):$wName         // String name must NOT be quoted
   w = 123

   SetDataFolder root:
End
```

The last example illustrates another subtlety. This command would generate an error at compile time:

```
Make/O root:$dfName:$wName        // ERROR
```

because Igor would intepret it as:

```
Make/O root:$(dfName:$wName)      // ERROR
```

To avoid this, you must use parentheses like this:

```
Make/O root:$(dfName):$wName      // OK
```

Runtime Lookup Example

In this example, a function named Routine calls another function named Subroutine and needs to access a number of result values created by Subroutine. To make it easy to clean up the temporary result globals, Subroutine creates them in a new data folder. Routine uses the results created by Subroutine and then deletes the temporary data folder.

```
Function Subroutine(w)
   WAVE w

   NewDataFolder/O/S SubroutineResults    // Results go here

   WaveStats/Q w              // WaveStats creates local variables
   Variable/G gAvg = V_avg    // Return the V_avg result in global gAvg
   Variable/G gMin = V_min
   String/G gWName = NameOfWave(w)

   SetDataFolder ::           // Back to original data folder
End

Function Routine()
   Make aWave= {1,2,3,4}
   Subroutine(aWave)

   DFREF dfr = :SubroutineResults

   NVAR theAvg = dfr:gAvg                    // theAvg is local name
   NVAR theMin = dfr:gMin
   SVAR theName = dfr:gWName
   Print theAvg, theMin, theName
   KillDataFolder SubroutineResults          // We are done with results
End
```

Note that the NVAR statement must appear *after* the call to the procedure (Subroutine in this case) that creates the global variable. This is because NVAR has both a compile-time and a runtime behavior. At compile time, it creates a local variable that Igor can compile (theAvg in this case). At runtime, it actually looks up and creates a link to the global (variable gAvg stored in data folder SubroutineResults in this case).

Often a function needs to access a large number of global variables stored in a data folder. In such cases, you can write more compact code using the ability of NVAR, SVAR and WAVE to access multiple objects in one statement:

```
Function Routine2()
   Make aWave= {1,2,3,4}
   Subroutine(aWave)

   DFREF dfr = :SubroutineResults

   NVAR theAvg=dfr:gAvg, theMin=dfr:gMin     // Access two variables
   SVAR theName = dfr:gWName
   Print theAvg, theMin, theName

   KillDataFolder SubroutineResults          // We are done with results
End
```

Automatic Creation of WAVE, NVAR and SVAR References

The Igor compiler sometimes automatically creates WAVE, NVAR and SVAR references. For example:

```
Function Example1()
   Make/O wave0
   wave0 = p

   Variable/G gVar1
   gVar1= 1

   String/G gStr1
   gStr1= "hello"
End
```

In this example, we did not use WAVE, NVAR or SVAR references and yet we were still able to compile assignment statements referencing waves and global variables. This is a feature of *Make*, `Variable/G` and `String/G` that automatically create local references for simple object names.

Simple object names are names which are known at compile time for objects which will be created in the current data folder at runtime. `Make`, `Variable/G` and `String/G` do not create references if you use $ expression, a partial data folder path or a full data folder path to specify the object unless you include the /N flag, discussed next.

Explicit Creation of NVAR and SVAR References

If you create a global variable in a user-defined function using a path or a $ expression, you can explicitly create an NVAR or SVAR reference like this:

```
Function Example2()
   String path

   Variable/G root:gVar2A = 2
   NVAR gVar2A = root:gVar2A        // Create NVAR gVar2A

   path = "root:gVar2B"
   Variable/G $path = 2
   NVAR gVar2B = $path              // Create NVAR gVar2B

   String/G root:gStr2A = "Two A"
   SVAR gStr2A = root:gStr2A        // Create SVAR gStr2A

   path = "root:gStr2B"
   String/G $path = "Two B"
   SVAR gStr2B = $path              // Create SVAR gStr2B
End
```

In Igor Pro 8.00 or later, you can combine the creation of the reference with the creation of the variable by using the /N flag with **Variable** or **String**:

```
Function Example3()
   String path

   Variable/G root:gVar3A/N=gVar3A = 3

   path = "root:gVar3B"
   Variable/G $path/N=gVar3B = 3

   String/G root:gStr3A/N=gStr3A = "Three A"

   path = "root:gStr3B"
   String/G $path/N=gStr3B = "Three B"
End
```

Wave References

A wave reference is a value that identifies a particular wave. Wave references are used in commands that operate on waves, including assignment statements and calls to operations and functions that take wave reference parameters.

Wave reference variables hold wave references. They can be created as local variables, passed as parameters and returned as function results.

Here is a simple example:

```
Function Test(wIn)
   Wave wIn          // Reference to the input wave received as parameter

   String newName = NameOfWave(wIn) + "_out" // Compute output wave name

   Duplicate/O wIn, $newName                 // Create output wave

   Wave wOut = $newName       // Create wave reference for output wave
   wOut += 1                  // Use wave reference in assignment statement
End
```

This function might be called from the command line or from another function like this:

```
Make/O/N=5 wave0 = p
Test(wave0)                   // Pass wave reference to Test function
```

A Wave statement declares a wave reference variable. It has both a compile-time and a runtime effect.

At compile time, it tells Igor what type of object the declared name references. In the example above, it tells Igor that wOut references a wave as opposed to a numeric variable, a string variable, a window or some other type of object. The Igor compiler allows wOut to be used in a waveform assignment statement (wOut += 1) because it knows that wOut references a wave.

The compiler also needs to know if the wave is real, complex or text. Use Wave/C to create a complex wave reference and Wave/T to create a text wave reference. Wave by itself creates a real wave reference.

At runtime the Wave statement stores a reference to a specific wave in the wave reference variable (wOut in this example). The referenced wave must already exist when the wave statement executes. Otherwise Igor stores a NULL reference in the wave reference variable and you get an error when you attempt to use it. We put the `Wave wOut = $newName` statement *after* the Duplicate operation to insure that the wave exists when the Wave statement is executed. Putting the Wave statement before the command that creates the wave is a common error.

Automatic Creation of WAVE References

The Igor compiler sometimes automatically creates WAVE references. For example:

```
Function Example1()
   Make wave1
   wave1 = x^2
End
```

In this example, we did not declare a wave reference, and yet Igor was still able to compile an assignment statement referring to a wave. This is a feature of the **Make** operation (see page V-526) which automatically creates local references for simple object names. The **Duplicate** operation (see page V-185) and many other operations that create output waves also automatically create local wave references for simple object names.

Simple object names are names which are known at compile time for objects which will be created in the current data folder at runtime. Make and Duplicate do not create references if you use $<name>, a partial data folder path, or a full data folder path to specify the object.

In the case of Make and Duplicate with simple object names, the type of the automatically created wave reference is determined by flags. Make/C and Duplicate/C create complex wave references. Make/T and Duplicate/T create text wave references. Make and Duplicate without type flags create real wave references. See **WAVE Reference Types** on page IV-73 and **WAVE Reference Type Flags** on page IV-74 for a complete list of type flags and further details.

Most built-in operations that create output waves (often called "destination" waves) also automatically create wave references. For example, if you write:

```
DWT srcWave, destWave
```

it is as if you wrote:

```
DWT srcWave, destWave
WAVE destWave
```

After the discrete wavelet transform executes, you can reference `destWave` without an explicit wave reference.

Standalone WAVE Reference Statements

In cases where Igor does not automatically create a wave reference, because the output wave is not specified using a simple object name, you need to explicitly create a wave reference if you want to access the wave in an assignment statement.

You can create an explicit standalone wave reference using a statement following the command that created the output wave. In this example, the name of the output wave is specified as a parameter and therefore we can not use a simple object name when calling Make:

```
Function Example2(nameForOutputWave)
   String nameForOutputWave   // String contains the name of the wave to make

   Make $nameForOutputWave          // Make a wave
   Wave w = $nameForOutputWave      // Make a wave reference
   w = x^2
End
```

If you make a text wave or a complex wave, you need to tell the Igor compiler about that by using Wave/T or Wave/C. The compiler needs to know the type of the wave in order to properly compile the assignment statement.

Inline WAVE Reference Statements

You can create a wave reference variable using `/WAVE=<name>` in the command that creates the output wave. For example:

```
Function Example3(nameForOutputWave)
   String nameForOutputWave

   Make $nameForOutputWave/WAVE=w   // Make a wave and a wave reference
   w = x^2
End
```

Here `/WAVE=w` is an inline wave reference statement. It does the same thing as the standalone wave reference in the preceding section.

Here are some more examples of inline wave declarations:

```
Function Example4()
   String name = "wave1"
   Duplicate/O wave0, $name/WAVE=wave1
   Differentiate wave0 /D=$name/WAVE=wave1
End
```

When using an inline wave reference statement, you do not need to, and in fact can not, specify the type of the wave using `WAVE/T` or `WAVE/C`. Just use `WAVE` by itself regardless of the type of the output wave. The Igor compiler automatically creates the right kind of wave reference. For example:

```
Function Example5()
   Make real1, $"real2"/WAVE=r2     // real1, real2 and r2 are real
   Make/C cplx1, $"cplx2"/WAVE=c2   // cplx1, cplx2 and c2 are complex
   Make/T text1, $"text2"/WAVE=t2   // text1, text2 and t2 are text
End
```

Inline wave reference statements are accepted by those operations which automatically create a wave reference for a simple object name.

Inline wave references are not allowed after a simple object name.

Inline wave references are allowed on the command line but do nothing.

WAVE Reference Types

When wave references are created at compile time, they are created with a specific numeric type or are defined as text. The compiler then uses this type when compiling expressions based on the WAVE reference or when trying to match two instances of the same name. For example:

```
Make rWave           // Creates single-precision real wave reference

Make/C cWave         // Creates single-precision complex wave reference

Make/L int64Wave     // Creates signed 64-bit integer wave reference

Make/L/U int64Wave   // Creates unsigned 64-bit integer wave reference

Make/T tWave         // Creates text wave reference
```

These types then define what kind of right-hand side expression Igor compiles:

```
rWave = expression      // Compiles real expression as double precision

cWave = expression      // Compiles complex expression as double precision
```

```
int64Wave = expression  // Compiles signed 64-bit integer expression

uint64Wave = expression // Compiles unsigned 64-bit integer expression

tWave = expression      // Compiles text expression
```

See also **Integer Expressions in Wave Assignment Statements** on page IV-39.

The compiler is sometimes picky about the congruence between two declarations of wave reference variables of the same name. For example:

```
WAVE aWave
if (!WaveExists(aWave))
   Make/D aWave
endif
```

This generates a compile error complaining about inconsistent types for a wave reference. Because Make automatically creates a wave reference, this is equivalent to:

```
WAVE aWave
if (!WaveExists(aWave))
   Make/D aWave
   WAVE/D aWave
endif
```

This creates two wave references with the same name but different types. To fix this, change the explicit wave reference declaration to:

```
WAVE/D aWave
```

WAVE Reference Type Flags

The **WAVE** reference (see page V-1069) along with certain operations such as Duplicate can accept the following flags identifying the type of WAVE reference:

/B	8-bit signed integer destination waves, unsigned with /U.
/C	Complex destination waves.
/D	Double precision destination waves.
/I	32-bit signed integer destination waves, unsigned with /U.
/L	64-bit signed integer destination waves, unsigned with /U. Requires Igor Pro 7.00 or later.
/S	Single precision destination waves.
/T	Text destination waves.
/U	Unsigned destination waves.
/W	16-bit signed integer destination waves, unsigned with /U.
/DF	Wave holds data folder references.
/WAVE	Wave holds wave references.

These are the same flags used by the **Make**. In the case of WAVE declarations and Duplicate, they do not affect the actual wave but rather tell Igor what kind of wave is expected at runtime. The compiler uses this information to determine what kind of code to compile if the wave is used as the destination of a wave assignment statement later in the function. For example:

```
Function DupIt(wv)
   WAVE/C wv                  // complex wave
   Duplicate/O/C wv,dupWv      // dupWv is complex
   dupWv[0]=cmplx(5.0,1.0)     // no error, because dupWv known complex
   . . .
End
```

If Duplicate did not have the /C flag, you would get a “function not available for this number type” message when compiling the assignment of `dupWv` to the result of the `cmplx` function.

Problems with Automatic Creation of WAVE References

Operations that change a wave's type or which can create output waves of more than one type, such as **FFT**, **IFFT** and **WignerTransform** present special issues.

In some cases, the wave reference automatically created by an operation might be of the wrong type. For example, the FFT operation automatically creates a complex wave reference for the destination wave, so if you write:

```
FFT/DEST=destWave srcWave
```

it is as if you wrote:

```
FFT/DEST=destWave srcWave
WAVE/C destWave
```

However, if a real wave reference for the same name already exists, FFT/DEST does not create a new wave reference. For example:

```
Wave destWave                 // Real wave reference
. . .
FFT /DEST=destWave srcWave    // FFT does not create wave reference
                              // because it already exists.
```

In this case, you would need to create a complex wave reference using a different name, like this:

```
Wave/C cDestWave = destWave
```

The output of the FFT operation can sometimes be real, not complex. For example:

```
FFT/DEST=destWave/OUT=2 srcWave
```

The /OUT=2 flag creates a real destination wave. Thus the complex wave reference automatically created by the FFT operation is wrong and can not be used to subsequently access the destination wave. In this case, you must explicitly create a real wave reference, like this:

```
FFT/DEST=destWave/OUT=2 srcWave
WAVE realDestWave = destWave
```

Note that you can not write:

```
FFT/DEST=destWave/OUT=2 srcWave
WAVE destWave
```

because the FFT operation has already created a complex wave reference named `destWave`, so the compiler will generate an error. You must use a different name for the real wave reference.

The **IFFT** has a similar problem but in reverse. IFFT automatically creates a real wave reference for the destination wave. In some cases, the actual destination wave will be complex and you will need to create an explicit wave reference in order to access it.

WAVE Reference Is Needed to Pass a Wave By Name

As of Igor Pro 6.3, new procedure windows are created with this pragma statement:

```
#pragma rtGlobals=3 // Use modern global access method and strict wave access.
```

The strict wave access mode causes a compile error if you pass a literal wave name as a parameter to an operation or function. Instead you must pass a wave reference.

With rtGlobals=3, this function has errors on both lines:

```
Function Test()
   Display jack                          // Error: Expected wave reference
   Variable tmp = mean(jack,0,100)       // Error: Expected wave reference
End
```

The proper way to do this is to create a wave reference, like this:

```
Function Test()
   WAVE jack
   Display jack                          // OK
   Variable tmp = mean(jack,0,100)       // OK
End
```

The purpose of the strict wave access mode is to detect inadvertent name mistakes. This applies to simple names only, not to full or partial paths. Even with rtGlobals=3, it is OK to use a full or partial path where a wave reference is expected:

```
Function Test()
   Display :jack                         // OK
   Variable tmp = mean(root:jack,0,100)  // OK
End
```

If you have old code that is impractical to fix, you can revert to using rtGlobals=1 or rtGlobals=2.

Wave Reference Function Results

Advanced programmers can create functions that return wave references using Function/WAVE:

```
Function/WAVE Test(wIn) // /WAVE flag says function returns wave reference
   Wave wIn    // Reference to the input wave received as parameter

   String newName = NameOfWave(wIn) + "_out" // Compute output wave name

   Duplicate/O wIn, $newName                 // Create output wave

   Wave wOut = $newName       // Create wave reference for output wave
   wOut += 1                  // Use wave reference in assignment statement

   return wOut                // Return wave reference
End
```

This function might be called from another function like this:

```
Make/O/N=5 wave0 = p
Wave wOut = Test(wave0)
Display wave0, wOut
```

This technique is useful when a subroutine creates a free wave for temporary use:

```
Function Subroutine()
   Make/FREE tempWave = <expression>
   return tempWave
End

Function Routine()
   Wave tempWave = Subroutine()
   <Use tempWave>
End
```

When Routine returns, tempWave is automatically killed because all references to it have gone out of scope.

Wave Reference Waves

You can create waves that contain wave references using the Make /WAVE flag. You can use a wave reference wave as a list of waves for further processing and in multithreaded wave assignment using the **MultiThread** keyword.

Wave reference waves are recommended for advanced programmers only.

Note: Wave reference waves are saved only in packed experiment files. They are not saved in unpacked experiments and are not saved by the SaveData operation or the Data Browser's Save Copy button. In general, they are intended for temporary computation purposes only.

Here is an example:

```
Make/O wave0, wave1, wave2              // Make some waves
Make/O/WAVE wr                          // Make a wave reference wave
wr[0]=wave0; wr[1]=wave1; wr[2]=wave2   // Assign values
```

The wave reference wave wr could now be used, for example, to pass a list of waves to a function that performs display or analysis operations.

Make/WAVE without any assignment creates a wave containing null wave references. Similarly, inserting points or redimensioning to a larger size initializes the new points to null. Deleting points or redimensioning to a smaller size deletes any free waves if the deleted points contained the only reference to them.

To determine if a given wave is a type that stores wave references, use the **WaveType** function with the optional selector = 1.

In the next example, a subroutine supplies a list of references to waves to be graphed by the main routine. A wave reference wave is used to store the list of wave references.

```
Function MainRoutine()
   Make/O/WAVE/N=5 wr        // Will contain references to other waves
   wr= Subroutine(p)         // Fill w with references

   WAVE w= wr[0]             // Get reference to first wave
   Display w                 // and display in a graph

   Variable i
   for(i=1;i<5;i+=1)
      WAVE w= wr[i]          // Get reference to next wave
      AppendToGraph w        // and append to graph
   endfor
End

Function/WAVE Subroutine(i)
   Variable i

   String name = "wave"+num2str(i)

   // Create a wave with a computed name and also a wave reference to it
   Make/O $name/WAVE=w = sin(x/(8+i))

   return w       // Return the wave reference to the calling routine
End
```

As another example, here is a function that returns a wave reference wave containing references to all of the Y waves in a graph:

```
Function/WAVE GetListOfYWavesInGraph(graphName)
   String graphName          // Must contain the name of a graph

   // If graphName is not valid, return NULL wave
   if (strlen(graphName)==0 || WinType(graphName)!=1)
      return $""
   endif

   // Make a wave to contain wave references
   Make /FREE /WAVE /N=0 listWave

   Variable index = 0
   do
      // Get wave reference for next Y wave in graph
      WAVE/Z w = WaveRefIndexed(graphName,index,1)
      if (WaveExists(w) == 0)
         break                // No more waves
      endif

      // Add wave reference to list
      InsertPoints index, 1, listWave
      listWave[index] = w

      index += 1
   while(1)                   // Loop till break above

   return listWave
End
```

The returned wave reference wave is a free wave. See **Free Waves** on page IV-91 for details.

For an example using a wave reference wave for multiprocessing, see **Wave Reference MultiThread Example** on page IV-327.

Data Folder References

The data folder reference is a lightweight object that refers to a data folder, analogous to the wave reference which refers to a wave. You can think of a data folder reference as an identification number that Igor uses to identify a particular data folder.

Data folder reference variables (DFREFs) hold data folder references. They can be created as local variables, passed as parameters and returned as function results.

The most common use for a data folder reference is to save and restore the current data folder during the execution of a function:

```
Function Test()
   DFREF saveDFR = GetDataFolderDFR()  // Get reference to current data folder

   NewDataFolder/O/S MyNewDataFolder   // Create a new data folder
   . . .                               // Do some work in it

   SetDataFolder saveDFR               // Restore current data folder
End
```

Data folder references can be used in commands where Igor accepts a data folder path. For example, this function shows three equivalent methods of accessing waves in a specific data folder:

```
Function Test()
   // Method 1: Using paths
   Display root:Packages:'My Package':yWave vs root:Packages:'My Package':xWave

   // Method 2: Using the current data folder
   DFREF dfSave = GetDataFolderDFR()// Save the current data folder
   SetDataFolder root:Packages:'My Package'
   Display yWave vs xWave
   SetDataFolder dfSave            // Restore current data folder

   // Method 3: Using data folder references
   DFREF dfr = root:Packages:'My Package'
   Display dfr:yWave vs dfr:xWave
End
```

Using data folder references instead of data folder paths can streamline programs that make heavy use of data folders.

Using Data Folder References

In an advanced application, the programmer often defines a set of named data objects (waves, numeric variables and string variables) that the application acts on. These objects exist in a data folder. If there is just one instance of the set, it is possible to hard-code data folder paths to the objects. Often, however, there will be a multiplicity of such sets, for example, one set per graph or one set per channel in a data acquisition application. In such applications, procedures must be written to act on the set of data objects in a data folder specified at runtime.

One way to specify a data folder at runtime is to create a path to the data folder in a string variable. While this works, you wind up with code that does a lot of concatenation of data folder paths and data object names. Using data folder references, such code can be streamlined.

You create a data folder reference variable with a DFREF statement. For example, assume your application defines a set of data with a wave named wave0, a numeric variable named num0 and a string named str0 and that we have one data folder containing such a set for each graph. You can access your objects like this:

```
Function DoSomething(graphName)
   String graphName
   DFREF dfr = root:Packages:MyApplication:$graphName
   WAVE w0 = dfr:wave0
   NVAR n0 = dfr:num0
   SVAR s0 = dfr:str0
   . . .
End
```

Igor accepts a data folder reference in any command in which a data folder path would be accepted. For example:

```
Function Test()
   Display root:MyDataFolder:wave0  // OK

   DFREF dfr = root:MyDataFolder
   Display dfr:wave0                // OK

   String path = "root:MyDataFolder:wave0"
   Display $path                    // OK. $ converts string to path.

   path = "root:MyDataFolder"
   DFREF dfr = $path                // OK. $ converts string to path.
   Display dfr:wave0                // OK

   String currentDFPath
   currentDFPath = GetDataFolder(1) // OK
   DFREF dfr = GetDataFolder(1)     // ERROR: GetDataFolder returns a string
                                    // not a path.

   DFREF dfr = GetDataFolderDFR()   // OK
End
```

The /SDFR Flag

You can also use the /SDFR (source data folder reference) flag in a WAVE, NVAR or SVAR statement. The utility of /SDFR is illustrated by this example which shows three different ways to reference multiple waves in the same data folder:

```
Function Test()
   // Assume a data folder exists at root:Run1

   // Use explicit paths
   Wave wave0=root:Run1:wave0, wave1=root:Run1:wave1, wave2=root:Run1:wave2

   // Use a data folder reference
   DFREF dfr = root:Run1
   Wave wave0=dfr:wave0, wave1=dfr:wave1, wave2=dfr:wave2

   // Use the /SDFR flag
   DFREF dfr = root:Run1
   Wave/SDFR=dfr wave0, wave1, wave2
End
```

Igor Pro 8 and Igor Pro 9 handle invalid data folder references in /SDFR=*dfr* flags differently when /Z is included. Igor Pro 8 incorrectly flags an error on the /SDFR statement despite the /Z. Igor Pro 9 correctly suppresses the error on the /SDFR statement because of /Z.

If you use WAVE/Z, NVAR/Z, or SVAR/Z, this means you want to handle errors yourself so you should follow it with a WaveExists, NVAR_Exists, or SVAR_Exists test.

The DFREF Type

In functions, you can define data folder reference variables using the DFREF declaration:

```
DFREF localname [= <DataFolderRef or path>] [<more defs]
```

You can then use the data folder reference in those places where you can use a data folder path. For example:

```
DFREF dfr = root:df1
Display dfr:wave1             // Equivalent to Display root:df1:wave1
```

The syntax is limited to a single name after the data folder reference, so this is not legal:

```
Display dfr:subfolder:wave1   // Illegal
```

You can use DFREF to define input parameters for user-defined functions. For example:

```
Function Test(df)
   DFREF df
   Display df:wave1
End
```

You can also use DFREF to define fields in structures. However, you can not directly use a DFREF structure field in those places where Igor is expecting a path and object name. So, instead of:

```
Display s.dfr:wave1              // Illegal
```

you would need to write:

```
DFREF dftmp = s.dfr
Display dftmp:wave1              // OK
```

You can use a DFREF structure field where just a path is expected. For example:

```
SetDataFolder s.dfr              // OK
```

Built-in DFREF Functions

Some built-in functions take string data folder paths as parameters or return them as results. Those functions can not take or return data folder references. Here are equivalent DFREF versions that take or return data folder references:

```
   CountObjectsDFR(dfr, type)

   GetDataFolderDFR()

   GetIndexedObjNameDFR(dfr, type, index)

   GetWavesDataFolderDFR(wave)
```

These additional data folder reference functions are available:

```
   DataFolderRefChanges(dfr, changeType)

   DataFolderRefStatus(dfr)

   NewFreeDataFolder()

   DataFolderRefsEqual(dfr1, dfr2)
```

Just as operations that take a data folder path accept a data folder reference, these DFREF functions can also accept a data folder path:

```
Function Test()
   DFREF dfr = root:MyDataFolder
   Print CountObjectsDFR(dfr,1)                  // OK
   Print CountObjectsDFR(root:MyDataFolder,1)    // OK
End
```

Checking Data Folder Reference Validity

The **DataFolderRefStatus** function returns zero if the data folder reference is invalid. You should use it to test any DFREF variables that might not be valid, for example, when you assign a value to a data folder reference and you are not sure that the referenced data folder exists:

```
Function Test()
   DFREF dfr = root:MyDataFolder    // MyDataFolder may or may not exist
   if (DataFolderRefStatus(dfr) != 0)
      . . .
   endif
End
```

For historical reasons, an invalid DFREF variable will often act like root.

Data Folder Reference Function Results

A user-defined function can return a data folder reference. This might be used for a subroutine that returns a set of new objects to the calling routine. The set can be returned in a new data folder and the subroutine can return a reference it.

For example:

```
Function/DF Subroutine(newDFName)
   String newDFName
   NewDataFolder/O $newDFName
   DFREF dfr = $newDFName
   Make/O dfr:wave0, dfr:wave1
   return dfr
End
```

```
Function/DF MainRoutine()
   DFREF dfr = Subroutine("MyDataFolder")
   Display dfr:wave0, dfr:wave1
End
```

Data Folder Reference Waves

You can create waves that contain data folder references using the **Make** /DF flag. You can use a data folder reference wave as a list of data folders for further processing and in multithreaded wave assignment using the MultiThread keyword.

Data folder reference waves are recommended for advanced programmers only.

Note: Data folder reference waves are saved only in packed experiment files. They are not saved in unpacked experiments and are not saved by the SaveData operation or the Data Browser's Save Copy button. In general, they are intended for temporary computation purposes only.

Make/DF without any assignment creates a wave containing null data folder references. Similarly, inserting points or redimensioning to a larger size initializes the new points to null. Deleting points or redimensioning to a smaller size deletes any free data folders if the wave contained the only reference to them.

To determine if a given wave is a type that stores data folder references, use the **WaveType** function with the optional selector = 1.

For an example using a data folder reference wave for multiprocessing, see **Data Folder Reference MultiThread Example** on page IV-325.

Accessing Waves in Functions

To access a wave in a user-defined function, we need to create, one way or another, a wave reference. The section **Accessing Global Variables and Waves** on page IV-65 explained how to access a wave using a WAVE reference. This section introduces several additional techniques.

We can create the wave reference by:

- Declaring a wave parameter
- Using $<string expression>
- Using a literal wave name
- Using a wave reference function

Each of these techniques is illustrated in the following sections.

Each example shows a function and commands that call the function. The function itself illustrates how to deal with the wave within the function. The commands show how to pass enough information to the function so that it can access the wave. Other examples can be found in **Writing Functions that Process Waves** on page III-171.

Wave Reference Passed as Parameter

This is the simplest method. The function might be called from the command line or from another function.

```
Function Test(w)
   WAVE w                  // Wave reference passed as a parameter

   w += 1                  // Use in assignment statement
   Print mean(w)           // Pass as function parameter
   WaveStats w             // Pass as operation parameter
   AnotherFunction(w)      // Pass as user function parameter
End

Make/O/N=5 wave0 = p
Test(wave0)
Test($"wave0")
String wName = "wave0"; Test($wName)
```

In the first call to Test, the wave reference is a literal wave name. In the second call, we create the wave reference using $<literal string>. In the third call, we create the wave reference using $<string variable>. $<literal string> and $<string variable> are specific cases of the general case $<string expression>.

If the function expected to receive a reference to a text wave, we would declare the parameter using:

```
   WAVE/T w
```

If the function expected to be receive a reference to a complex wave, we would declare the parameter using:

```
   WAVE/C w
```

If you need to return a large number of values to the calling routine, it is sometimes convenient to use a parameter wave as an output mechanism. The following example illustrates this technique:

```
Function MyWaveStats(inputWave, outputWave)
   WAVE inputWave
   WAVE outputWave

   WaveStats/Q inputWave

   outputWave[0] = V_npnts
   outputWave[1] = V_avg
   outputWave[2] = V_sdev
End

Function Test()
   Make/O testwave= gnoise(1)

   Make/O/N=20 tempResultWave
   MyWaveStats(testwave, tempResultWave)
   Variable npnts = tempResultWave[0]
   Variable avg = tempResultWave[1]
   Variable sdev = tempResultWave[2]
   KillWaves tempResultWave

   Printf "Points: %g; Avg: %g; SDev: %g\r", npnts, avg, sdev
End
```

If the calling function needs the returned values only temporarily, it is better to return a free wave as the function result. See **Wave Reference Function Results** on page IV-76.

Wave Accessed Via String Passed as Parameter

This technique is of most use when the wave might not exist when the function is called. It is appropriate for functions that create waves.

```
Function Test(wName)
   String wName            // String containing a name for wave

   Make/O/N=5 $wName
   WAVE w = $wName         // Create a wave reference
   Print NameOfWave(w)
End

Test("wave0")
```

This example creates wave0 if it does not yet exist or overwrites it if it does exist. If we knew that the wave had to already exist, we could and should use the wave parameter technique shown in the preceding section. In this case, since the wave may not yet exist, we can not use a wave parameter.

Notice that we create a wave reference immediately after making the wave. Once we do this, we can use the wave reference in all of the ways shown in the preceding section. We can not create the wave reference before making the wave because a wave reference must refer to an existing wave.

The following example demonstrates that $wName and the wave reference w can refer to a wave that is not in the current data folder.

```
NewDataFolder root:Folder1
Test("root:Folder1:wave0")
```

Wave Accessed Via String Calculated in Function

This technique is used when creating multiple waves in a function or when algorithmically selecting a wave or a set of waves to be processed.

```
Function Test(baseName, startIndex, endIndex)
   String baseName
   Variable startIndex, endIndex

   Variable index = startIndex
   do
      String name = baseName + num2istr(index)
      WAVE w = $name
      Variable avg = mean(w)
      Printf "Wave: %s; average: %g\r", NameOfWave(w), avg
      index += 1
   while (index <= endIndex)
End

Make/O/N=5 wave0=gnoise(1), wave1=gnoise(1), wave2=gnoise(1)
Test("wave", 0, 2)
```

We need to use this method because we want the function to operate on any number of waves. If the function were to operate on a small, fixed number of waves, we could use the wave parameter method.

As in the preceding section, we create the wave reference using $<string expression>.

Wave Accessed Via Literal Wave Name

In data acquisition or analysis projects, you often need to write procedures that deal with runs of identically-structured data. Each run is stored in its own data folder and contains waves with the same names. In this kind of situation, you can write a set of functions that use literal wave names specific for your data structure.

```
Function CreateRatio()
   WAVE dataA, dataB
   Duplicate dataA, ratioAB
   WAVE ratioAB
   ratioAB = dataA / dataB
End

Make/O/N=5 dataA = 1 + p, dataB = 2 + p
CreateRatio()
```

The CreateRatio function assumes the structure and naming of the data. The function is hard-wired to this naming scheme and assumes that the current data folder contains the appropriate data.

We don't need explicit wave reference variables because Make and Duplicate creat automatic wave reference for simple wave names, as explained under **Automatic Creation of WAVE References** on page IV-72.

Wave Accessed Via Wave Reference Function

A wave reference function is a built-in Igor function or user-defined function that returns a reference to a wave. Wave reference functions are typically used on the right-hand side of a WAVE statement. For example:

```
WAVE w = WaveRefIndexedDFR(:,i)  // ith wave in current data folder
```

A common use for a wave reference function is to get access to waves displayed in a graph, using the TraceNameToWaveRef function. Here is an example.

```
Function PrintAverageOfDisplayedWaves()
   String list, traceName

   list = TraceNameList("",";",1)   // List of traces in top graph
   Variable index = 0
   do
      traceName = StringFromList(index, list)   // Next trace name
      if (strlen(traceName) == 0)
         break                                  // No more traces
      endif
      WAVE w = TraceNameToWaveRef("", traceName)// Get wave ref
      Variable avg = mean(w)
      Printf "Wave: %s; average: %g\r", NameOfWave(w), avg
      index += 1
   while (1)              // loop till break above
End

Make/O/N=5 wave0=gnoise(1), wave1=gnoise(1), wave2=gnoise(1)
Display wave0, wave1, wave2
PrintAverageOfDisplayedWaves()
```

See **Wave Reference Waves** on page IV-77 for an example using WaveRefIndexed to return a list of all of the Y waves in a graph.

There are other built-in wave reference functions (see **Wave Reference Functions** on page IV-197), but **WaveRefIndexed**, **WaveRefIndexedDFR** and **TraceNameToWaveRef** are the most used.

See **Wave Reference Function Results** on page IV-76 for details on user-defined functions that return wave references.

Destination Wave Parameters

Many operations create waves. Examples are Make, Duplicate and Differentiate. Such operations take "destination wave" parameters. A destination wave parameter can be:

A simple name	`Differentiate fred /D=jack`
A path	`Differentiate fred /D=root:FolderA:jack`

$ followed by a string expression	`String str = "root:FolderA:jack"`
	`Differentiate fred /D=$str`

A wave reference to an existing wave	`Wave w = jack`
	`Differentiate fred /D=w`

The wave reference works only in a user-defined function. The other techniques work in functions, in macros and from the command line.

Using the first three techniques, the destination wave may or may not already exist. It is created if it does not exist and overwritten if it does exist.

In the last technique, the destination wave parameter is a wave reference. This technique works properly only if the referenced wave already exists. Otherwise the destination wave is a wave with the name of the wave reference itself (w in this case) in the current data folder. This situation is further explained below under **Destination Wave Reference Issues** on page IV-86.

Wave Reference as Destination Wave

Here are the rules for wave references when used as destination waves:

1. If a simple name (not a wave reference) is passed as the destination wave parameter, the destination wave is a wave of that name in the current data folder whether it exists or not.
2. If a path or $ followed by a string containing a path is passed as the destination wave parameter, the destination wave is specified by the path whether the specified wave exists or not.
3. If a wave reference is passed as the destination wave parameter, the destination wave is the referenced wave if it exists. See **Destination Wave Reference Issues** on page IV-86 for what happens if it does not exist.

Exceptions To Destination Wave Rules

The Make operation is an exception in regard to wave references. The following statements make a wave named w in the current data folder whether root:FolderA:jack exists or not:

```
Wave/Z w = root:FolderA:jack
Make/O w
```

Prior to Igor Pro 6.20, many operations behaved like Make in this regard. In Igor Pro 6.20 and later, most operations behave like Differentiate.

Updating of Destination Wave References

When a simple name is provided as the destination wave parameter in a user-defined function, Igor automatically creates a wave reference variable of that name at compile time if one does not already exist. This is an implicit automatic wave reference variable.

When a wave reference variable exists, whether implicit or explicit, during the execution of the command, the operation stores a reference to the destination wave in that wave reference variable. You can use the wave reference to access the destination wave in subsequent commands.

Similarly, when the destination is specified using a wave reference field in a structure, the operation updates the field to refer to the destination wave.

Inline Wave References With Destination Waves

When the destination wave is specified using a path or a string containing a path, you can use an inline wave reference to create a reference to the destination wave. For example:

```
String dest = "root:FolderA:jack"
Differentiate input /D=$dest/WAVE=wDest
Display wDest
```

Here the Igor compiler creates a wave reference named wDest. The Differentiate operation stores a reference to the destination wave (root:FolderA:jack in this case) in wDest which can then be used to access the destination wave in subsequent commands.

Inline wave references do not determine which wave is the destination wave but merely provide a wave reference pointing to the destination wave when the command finishes.

Destination Wave Reference Issues

You will get unexpected behavior when a wave reference variable refers to a wave with a different name or in a different data folder and the referenced wave does not exist. For example, if the referenced wave does not exist:

```
Wave/Z w = jack
Differentiate fred /D=w    // Creates a wave named w in current data folder

Wave/Z w = root:FolderA:jack
Differentiate fred /D=w    // Creates a wave named w in current data folder

Wave/Z jack = root:FolderA:jack
Differentiate fred /D=jack // Creates a wave named jack in current data folder

STRUCT MyStruct s          // Contains wave ref field w
Differentiate fred /D=s.w  // Creates a wave named w in current data folder
```

In a situation like this, you should add a test using WaveExists to verify that the destination wave is valid and throw an error if not or otherwise handle the situation. For example:

```
Wave/Z w = root:FolderA:jack
if (!WaveExists(w))
   Abort "Destination wave does not exist"
endif
Differentiate fred /D=w
```

As noted above, when you use a simple name as a destination wave, the Igor compiler automatically creates a wave reference. If the automatically-created wave reference conflicts with a pre-existing wave reference, the compiler generates an error. For example, this function generates an "inconsistent type for wave reference error":

```
Function InconsistentTypeError()
   Wave/C w                   // Explicit complex wave reference
   Differentiate fred /D=w    // Implicit real wave reference
End
```

Another consideration involves loops. Suppose in a loop you have code like this:

```
SetDataFolder <something depending on loop index>
Duplicate/O srcWave, jack
```

You may think you are creating a wave named jack in each data folder but, because the contents of the automatically-created wave refrence variable jack is non-null after the first iteration, you will simply be overwriting the same wave over and over. To fix this, use

```
Duplicate/O srcWave,jack
WaveClear jack
```

or

```
Duplicate/O srcWave,$"jack"/WAVE=jack
```

This creates a wave named jack in the current data folder and stores a reference to it in a wave reference variable also named jack.

Changes in Destination Wave Behavior

Igor's handling of destination wave references was improved for Igor Pro 6.20. Previously some operations treated wave references as simple names, did not set the wave reference to refer to the destination wave on output, and exhibited other non-standard behavior.

Programming With Trace Names

A trace is the graphical representation of a 1D wave or a subset of a multi-dimensional wave. Each trace in a given graph has a unique name within that graph.

The name of a trace is, by default, the same as the name of the wave that it represents, but this is not always the case. For example, if you display the same wave twice in a given graph, the two trace names will be unique. Also, for programming convenience, an Igor programmer can create a trace whose name has no relation to the represented wave.

Trace names are used when changing traces in graphs, when accessing waves associated with traces in graphs, and when getting information about traces in graphs. See **Trace Names** on page II-282 for a general discussion.

These operations take trace name parameters:

ModifyGraph (traces), **ErrorBars**, **RemoveFromGraph**, **ReplaceWave**, **ReorderTraces**, **GraphWaveEdit**

Tag, **TextBox**, **Label**, **Legend**

These operations return trace names:

GetLastUserMenuInfo

These functions take trace name parameters:

GetUserData, **TraceInfo**, **TraceNameToWaveRef**, **XWaveRefFromTrace**

These functions return trace names:

TraceNameList, **CsrInfo**, **CsrWave**, **TraceFromPixel**

Trace Name Parameters

A trace name is not the same as a wave. An example may clarify this subtle point:

```
Function Test()
   Wave w = root:FolderA:wave0
   Display w
   ModifyGraph rgb(w) = (65535,0,0)          // WRONG
End
```

This is wrong because ModifyGraph is looking for the name of a trace in a graph and w is not the name of a trace in a graph. The name of the trace in this case is wave0. The function should be written like this:

```
Function Test()
   Wave w = root:FolderA:wave0
   Display w
   ModifyGraph rgb(wave0) = (65535,0,0)      // CORRECT
End
```

In the next example, the wave is passed to the function as a parameter so the name of the trace is not so obvious:

```
Function Test(w)
   Wave w
   Display w
   ModifyGraph rgb(w) = (65535,0,0)          // WRONG
End
```

This is wrong for the same reason as the first example: w is not the name of the trace. The function should be written like this:

```
Function Test(w)
   Wave w
   Display w

   String name = NameOfWave(w)
   ModifyGraph rgb($name) = (65535,0,0)      // CORRECT
End
```

Trace User Data

For advanced procedures that manage traces in graphs, you can attach user data to a trace using the userData keyword of the ModifyGraph operation. You can retrieve the user data using the GetUserData function.

User-defined Trace Names

As of Igor Pro 6.20, you can provide user-defined names for traces using /TN=<name> with **Display** and **AppendToGraph**. For example:

```
Make/O jack=sin(x/8)
NewDataFolder/O foo; Make/O :foo:jack=sin(x/9)
NewDataFolder/O bar; Make/O :bar:jack=sin(x/10)
Display jack/TN='jack in root', :foo:jack/TN='jack in foo'
AppendToGraph :bar:jack/TN='jack in bar'
ModifyGraph mode('jack in bar')=7,hbFill('jack in bar')=6
ModifyGraph rgb('jack in bar')=(0,0,65535)
```

As of Igor Pro 9.00, you can change the name of an existing trace using ModifyGraph with the traceName keyword.

Trace Name Programming Example

This example illustrates applying some kind of process to each trace in a graph. It appends a smoothed version of each trace to the graph. To try it, copy the code below into the procedure window of a new experiment and execute these commands one-at-a-time:

```
SetupForSmoothWavesDemo()
AppendSmoothedWavesToGraph("", 5)       // Less smoothing
AppendSmoothedWavesToGraph("", 15)      // More smoothing

Function SetupForSmoothWavesDemo()
   Variable numTraces = 3

   Display /W=(35,44,775,522)           // Create graph

   Variable i
   for(i=0; i<numTraces; i+=1)
      String xName, yName
      sprintf xName, "xWave%d", i
      sprintf yName, "yWave%d", i
      Make /O /N=100 $xName = p + 20*i
      Wave xW = $xName
      Make /O /N=100 $yName = p + gnoise(5)
      Wave yW = $yName
      AppendToGraph yW vs xW
   endfor
End

Function CopyTraceOffsets(graphName, sourceTraceName, destTraceName)
   String graphName              // Name of graph or "" for top graph
   String sourceTraceName        // Name of source trace
   String destTraceName          // Name of dest trace

   // info will be "" if no offsets or something like "offset(x)={10,20}"
   String info = TraceInfo(graphName, sourceTraceName, 0)

   String offsetStr = StringByKey("offset(x)", info, "=")   // e.g., "{10,20}"
   Variable xOffset=0, yOffset=0
   if (strlen(offsetStr) > 0)
      sscanf offsetStr, "{%g,%g}", xOffset, yOffset
   endif
   ModifyGraph offset($destTraceName) = {xOffset, yOffset}
End

Function AppendSmoothedWavesToGraph(graphName, numSmoothingPasses)
   String graphName              // Name of graph or "" for top graph
   Variable numSmoothingPasses   // Parameter to Smooth operation, e.g., 15

   // Get list of all traces in graph
   String traceList = TraceNameList(graphName, ";", 3)
   Variable numTraces = ItemsInList(traceList)
   Variable traceIndex

   // Remove traces representing smoothed waves previously added
   for(traceIndex=0; traceIndex<numTraces; traceIndex+=1)
      String traceName = StringFromList(traceIndex, traceList)
      if (StringMatch(traceName, "*_sm"))
         traceList = RemoveFromList(traceName, traceList)
         numTraces -= 1
         traceIndex -= 1
      endif
   endfor

   // Create smoothed versions of the traces
   for(traceIndex=0; traceIndex<numTraces; traceIndex+=1)
      traceName = StringFromList(traceIndex, traceList)

      Variable isXYTrace = 0

      Wave yW = TraceNameToWaveRef(graphName, traceName)
      DFREF dfr = $GetWavesDataFolder(yW, 1)
      String ySmoothedName = NameOfWave(yW) + "_sm"
      // Create smoothed wave in data folder containing Y wave
      Duplicate /O yW, dfr:$ySmoothedName
      Wave yWSmoothed = dfr:$ySmoothedName
      Smooth numSmoothingPasses, yWSmoothed

      Wave/Z xW = XWaveRefFromTrace(graphName, traceName)
      if (WaveExists(xW))        // It is an XY pair?
         isXYTrace = 1
      endif

      // Append smoothed wave to graph if it is not already in it
      CheckDisplayed /W=$graphName yWSmoothed
      if (V_flag == 0)           // Not yet already in graph?
         if (isXYTrace)
            AppendToGraph yWSmoothed vs xW
         else
            AppendToGraph yWSmoothed
         endif
         ModifyGraph /W=$graphName rgb($ySmoothedName) = (0, 0, 65535)
      endif

      // Copy trace offsets from input trace to smoothed trace
      CopyTraceOffsets(graphName, traceName, ySmoothedName)
   endfor
End
```

Free Waves

Free waves are waves that are not part of any data folder hierarchy. They are used mainly for temporary storage in a user function. They are somewhat faster to create than global waves, and when used as temporary storage in a user function, they are automatically killed the end of the function.

Free waves are recommended for advanced programmers only.

A wave that is stored in no data folder is called a “free” wave to distinguish it from a “global” wave which is stored in the root data folder or its descendants and from a “local” wave which is stored in a free data folder or its descendants.

Note: Free waves are saved only in packed experiment files. They are not saved in unpacked experiments and are not saved by the SaveData operation or the Data Browser's Save Copy button. In general, they are intended for temporary computation purposes only. The only way to save a free wave in an experiment file is by storing a wave reference in a wave reference wave.

You most commonly create free waves using the **NewFreeWave** function, or the **Make**/FREE and **Duplicate**/FREE operations. There are some other operations that can optionally make their output waves free waves. By default free waves are given the name '_free_' but NewFreeWave and Make/FREE allow you to specify other names - see **Free Wave Names** (see page IV-95) for details.

Here is an example:

```
Function ReverseWave(w)
   Wave w

   Variable lastPoint = numpnts(w) - 1
   if (lastPoint > 0)
      // This creates a free wave named _free_ and an automatic
      // wave reference named wTemp which refers to the free wave
      Duplicate /FREE w, wTemp

      w = wTemp[lastPoint-p]
   endif
End
```

In this example, wTemp is a free wave. As such, it is not contained in any data folder and therefore can not conflict with any other wave.

As explained below under **Free Wave Lifetime** on page IV-92, a free wave is automatically killed when there are no more references to it. In this example that happens when the function ends and the local wave reference variable wTemp goes out of scope.

You can access a free wave only using the wave reference returned by NewFreeWave, Make/FREE or Duplicate/FREE.

Free waves can not be used in situations where global persistence is required such as in graphs, tables and controls. In other words, you should use free waves for computation purposes only.

For a discussion of multithreaded assignment statements, see **Automatic Parallel Processing with MultiThread** on page IV-323. For an example using free waves, see **Wave Reference MultiThread Example** on page IV-327.

Free Wave Created When Free Data Folder Is Deleted

A wave stored in a free data folder or one of its descendants is called a local wave. This is in contrast to free waves which are stored in no data folder and to global waves which are stored in the main data folder hierarchy (in the root data folder or one of its descendants).

Local waves, like free waves, can not be used in situations where global persistence is required such as in graphs, tables and controls.

A local wave becomes a free wave if the parent data folder is killed while other references to the wave exist, as shown in thus example:

```
Function/WAVE Test()
   DFREF dfSav= GetDataFolderDFR()

   // Create a free data folder. It persists because it is the current
   // data folder (i.e., there is a reference to it in Igor itself).
   SetDataFolder NewFreeDataFolder()

   Make jack={1,2,3}    // jack is not a free wave at this point.
   // This also creates an automatic wave reference named jack.

   // The free data folder created above is killed because there
   // are no more references to it.
   SetDataFolder dfSav

   // The free data folder is now gone but jack persists
   // because of the wave reference to it and is now a free wave.

   return jack
End
```

In this example, jack was created as a local wave, not as a free wave, and consequently has the name jack, not _free_. When a local wave's data folder is killed, but the wave lives on due to a remaining wave reference, the wave retains the name it had when it was a local wave. See **Free Wave Names** (see page IV-95) for details.

Free Wave Created For User Function Input Parameter

If a user function takes a wave reference as in input parameter, you can create and pass a short free wave using a list of values as illustrated here:

```
Function Test(w)
   WAVE/D w
   Print w
End
```

You can invoke this function like this:

```
Test({1,2,3})
```

Igor automatically creates a free wave and passes it to Test. The free wave is automatically deleted when Test returns.

The data type of the free wave is determined by the type of the function's wave parameter. In this case the free wave is created as double-precision floating point because the wave parameter is defined using the /D flag. If /D were omitted, the free wave would be single-precision floating point. Wave/C/D would give a double-precision complex free wave. Wave/T would give a text free wave.

This list of values syntax is allowed only for user-defined functions because only they have code to test for and delete free waves upon exit.

Free Wave Lifetime

A free wave is automatically deleted when the last reference to it disappears.

Wave references can be stored in:

1. Wave reference variables in user-defined functions
2. Wave reference fields in structures
3. Elements of a wave reference wave (created by Make/WAVE)

The first case is the most common.

A wave reference disappears when:

1. The wave reference variable containing it is explicitly cleared using WaveClear.
2. The wave reference variable containing it is reassigned to refer to another wave.
3. The wave reference variable containing it goes out-of-scope and ceases to exist when the function in which it was created returns.
4. The wave reference wave element containing it is deleted or the wave reference wave is killed.

When there are no more references to a free wave, Igor automatically deletes it. This example illustrates the first three of these scenarios:

```
Function TestFreeWaveDeletion1()
   Wave w = NewFreeWave(2,3)  // Create a free wave with 3 points
   WaveClear w                // w no longer refers to the free wave
   // There are no more references to the free wave so it is deleted

   Wave w = NewFreeWave(2,3)  // Create a free wave with 3 points
   Wave w = root:wave0        // w no longer refers to the free wave
   // There are no more references to the free wave so it is deleted

   Wave w = NewFreeWave(2,3)  // Create a free wave with 3 points
End   // Wave reference w ceases to exist so the free wave is deleted
```

In the preceding example we used NewFreeWave which creates a free wave named '_free_' that is not part of any data folder. Next we will use Make/FREE instead of NewFreeWave. When reading this example, keep in mind that "Make jack" creates an automatic wave reference variable named jack:

```
Function TestFreeWaveDeletion2()
   Make /D /N=3 /FREE jack    // Create a free DP wave with 3 points
   // Make created an automatic wave reference named jack

   Make /D /N=5 /FREE jack    // Create a free DP wave with 5 points
   // Make created an automatic wave reference named jack
   // which refers to the 5-point jack.
   // There are now no references to 3-point jack so it is automatically deleted.

End   // Wave reference jack ceases to exist so free wave jack is deleted
```

In the next example, a subroutine returns a reference to a free wave to the calling routine:

```
Function/WAVE Subroutine1()
   Make /D /N=3 /FREE jack=p  // Create a free DP wave with 3 points
   return jack                // Return reference to calling routine
End

Function MainRoutine1()
   WAVE w= Subroutine1()   // Wave reference w references the free wave jack
   Print w
End   // Wave reference w ceases to exist so free wave jack is deleted
```

In the next example, the wave jack starts as an object in a free data folder (see **Free Data Folders** on page IV-96). It is not free because it is part of a data folder hierarchy even though the data folder is free. We call such a wave a local wave. When the free data folder is deleted, jack becomes a free wave.

When reading this example, keep in mind that the free data folder is automatically deleted when there are no more references to it. Originally, it survives because it is the current data folder and therefore is referenced by Igor internally. When it is no longer the current data folder, there are no more references to it and it is automatically deleted:

```
Function/WAVE Subroutine2()
   DFREF dfSav= GetDataFolderDFR()

   // Create free data folder and set as current data folder
   SetDataFolder NewFreeDataFolder()

   // Create wave jack and an automatic wave reference to it
   Make jack={1,2,3}    // jack is not free - it is an object in a data folder

   SetDataFolder dfSav  // Change the current data folder
   // There are now no references to the free data folder so it is deleted
   // but wave jack remains because there is a reference to it.
   // jack is now a free wave.

   return jack          // Return reference to free wave to calling routine
End

Function MainRoutine2()
   WAVE w= Subroutine2()   // Wave reference w references free wave jack
   Print w
End   // Wave reference w ceases to exist so free wave jack is deleted
```

Not shown in this section is the case of a free wave that persists because a reference to it is stored in a wave reference wave. That situation is illustrated by the **Automatic Parallel Processing with MultiThread** on page IV-323 example.

Free Wave Leaks

A leak occurs when an object is created and is never released. Leaks waste memory. Igor uses wave reference counting to prevent leaks but in the case of free waves there are special considerations.

A free wave must be stored in a wave reference variable in order to be automatically released because Igor does the releasing when a wave reference variable goes out of scope.

To avoid the possibility of leaks, when you call a function that returns a free wave, always store the function result in a wave reference variable.

The following example results in two memory leaks:

```
Function/WAVE GenerateFree()
   return NewFreeWave(2,3)
End

Function Leaks()
   Duplicate/O GenerateFree(),dummy             // This leaks
   Variable maxVal = WaveMax(generateFree())    // So does this
End
```

Both lines leak because the free wave returned by GenerateFree is not stored in any wave reference variable. By contrast, this function does not leak:

```
Function NoLeaks()
   Wave w = GenerateFree()              // w references the free wave
   Duplicate/O w,dummy
   Variable maxVal = WaveMax(w)         // WaveMax is a built-in function
   // The free wave is released when w goes out of scope
End
```

In the Leaks function, there would be no leak if you replaced the call to WaveMax with a call to a user-defined function. This is because Igor automatically creates a wave reference variable when you pass a wave to a user-defined function. Because this distinction is subtle, it is best to always store a free wave in your own explicit wave reference variable.

To a discussion of leak detection and investigation techniques, see **Detecting Wave Leaks** on page IV-206.

Free Wave Names

By default, the name of a free wave is _free_. If you use free waves in programming, you may find that the lack of specific names makes debugging difficult. If you break into the debugger, you see a lot of waves named _free_ and you can't tell which is used for what purpose.

In Igor Pro 9.00 and later, you can override the default and provide specific names for free waves using **Make**/FREE=1 and **NewFreeWave**. This improves debuggability and also helps in investigating leaks using the **WaveTracking** operation.

This example shows how to use Make/FREE=1 to specify the name of a free wave:

```
Function FreeWaveName1()
   // Creates a wave reference named tempw, wave name is _free_
   Make/FREE tempw
   Print NameOfWave(tempw)       // Prints _free_

   // Creates a wave reference named tempw, wave name is also tempw
   Make/FREE=1 tempw2
   Print NameOfWave(tempw2)      // Prints tempw2
End
```

You can also give a name to a free wave using the optional name string input to NewFreeWave:

```
Function FreeWaveName2()
   // Creates a wave reference named tempw, wave name is _free_
   Wave tempw = NewFreeWave(4,2)
   Print NameOfWave(tempw)       // Prints _free_

   // Creates a wave reference named tempw, wave name is myFreeWave
   Wave tempw = NewFreeWave(4,2,"myFreeWave")
   Print NameOfWave(tempw)       // Prints myFreeWave
End
```

Igor does not use the name of a free wave but in user procedure code there could be an assumption that free waves are named '_free_'. In that rare case, specifying the name of a free wave could expose a bug. The fix is to use WaveType to determine if a wave is free instead of the wave name.

Even if you didn't give a name to a free wave explicitly, there are tricky ways for a free wave to have a name other than '_free_'. One such way is to create a wave inside a free data folder, and then kill the free data folder:

```
Function/WAVE WaveInFreeDF()
   DFREF saveDF = GetDataFolderDFR()
   DFREF freeDF = NewFreeDataFolder()
   SetDataFolder freeDF    // Free data folder is current data folder
   Make jack               // Wave in free data folder
   SetDataFolder saveDF    // Free data folder is no longer current data folder
   return jack
   // At this point the free data folder is killed because freeDF
   // goes out of scope and consequently jack becomes a free wave
End

Function Tricky()
   Wave w = WaveInFreeDF()
   Print NameOfWave(w)     // Prints jack
   Print WaveType(w, 3)    // Prints 1 meaning free wave
End
```

Another way is to call an Igor operation that creates an output wave inside a free data folder. This happens if you set a free data folder as the current data folder and then call an operation, such as ColorTab2Wave, that creates an output wave in the current data folder.

Converting a Free Wave to a Global Wave

You can use **MoveWave** to move a free wave into a global data folder, in which case it ceases to be free. If the wave was created by NewFreeWave its name will be '_free_'. You can use MoveWave to provide it with a more descriptive name.

Here is an example illustrating Make/FREE:

```
Function Test()
   Make/FREE/N=(50,50) w
   SetScale x,-5,8,w
   SetScale y,-7,12,w
   w= exp(-(x^2+y^2))
   NewImage w
   if( GetRTError(1) != 0 )
      Print "Can't use a free wave here"
   endif
   MoveWave w,root:wasFree
   NewImage w
End
```

Note that MoveWave requires that the new wave name, wasFree in this case, be unique within the destination data folder.

To determine if a given wave is free or global, use the **WaveType** function with the optional selector = 2.

Free Data Folders

Free data folders are data folders that are not part of any data folder hierarchy. Their principal use is in multithreaded wave assignment using the MultiThread keyword in a function. They can also be used for temporary storage within functions.

Free data folders are recommended for advanced programmers only.

A wave that is stored in a free data folder or its descendants is called a “local” wave to distinguish it from a “global” wave which is stored in the root data folder or its descendants and from a “free” wave which is stored in no data folder.

Note: Free data folders are saved only in packed experiment files. They are not saved in unpacked experiments and are not saved by the SaveData operation or the Data Browser's Save Copy button. In general, they are intended for temporary computation purposes only.

You create a free data folder using the NewFreeDataFolder function. You access it using the data folder reference returned by that function.

After using SetDataFolder with a free data folder, be sure to restore it to the previous value, like this:

```
Function Test()
   DFREF dfrSave = GetDataFolderDFR()

   SetDataFolder NewFreeDataFolder()   // Create new free data folder.

   . . .

   SetDataFolder dfrSave
End
```

This is good programming practice in general but is especially important when using free data folders.

Free data folders can not be used in situations where global persistence is required such as in graphs, tables and controls. In other words, you should use objects in free data folders for short-term computation purposes only.

For a discussion of multithreaded assignment statements, see **Automatic Parallel Processing with MultiThread** on page IV-323. For an example using free data folders, see **Data Folder Reference MultiThread Example** on page IV-325.

Free Data Folder Lifetime

A free data folder is automatically deleted when the last reference to it disappears.

Data folder references can be stored in:

1. Data folder reference variables in user-defined functions
2. Data folder reference fields in structures
3. Elements of a data folder reference wave (created with Make/DF)
4. Igor's internal current data folder reference variable

A data folder reference disappears when:

1. The data folder reference variable containing it is explicitly cleared using KillDataFolder.
2. The data folder reference variable containing it is reassigned to refer to another data folder.
3. The data folder reference variable containing it goes out-of-scope and ceases to exist when the function in which it was created returns.
4. The data folder reference wave element containing it is deleted or the data folder reference wave is killed.
5. The current data folder is changed which causes Igor's internal current data folder reference variable to refer to another data folder.

When there are no more references to a free data folder, Igor automatically deletes it.

In this example, a free data folder reference variable is cleared by KillDataFolder:

```
Function Test1()
   DFREF dfr= NewFreeDataFolder()      // Create new free data folder.
   // The free data folder exists because dfr references it.
   . . .
   KillDataFolder dfr   // dfr no longer refers to the free data folder
End
```

KillDataFolder kills the free data folder only if the given DFREF variable contains the last reference to it.

In the next example, the free data folder is automatically deleted when the DFREF that references it is changed to reference another data folder:

```
Function Test2()
   DFREF dfr= NewFreeDataFolder()      // Create new free data folder.
   // The free data folder exists because dfr references it.
   . . .
   DFREF dfr= root:
   // The free data folder is deleted since there are no references to it.
End
```

In the next example, a free data folder is created and a reference is stored in a local data folder reference variable. When the function ends, the DFREF ceases to exist and the free data folder is automatically deleted:

```
Function Test3()
   DFREF dfr= NewFreeDataFolder()      // Create new free data folder.
   // The free data folder exists because dfr references it.
   . . .
End   // The free data folder is deleted because dfr no longer exists.
```

The fourth case, where a data folder reference is stored in a data folder reference wave, is discussed under **Data Folder Reference Waves** on page IV-82.

In the next example, the free data folder is referenced by Igor's internal current data folder reference variable because it is the current data folder. When the current data folder is changed, there are no more references to the free data folder and it is automatically deleted:

```
Function Test4()
   SetDataFolder NewFreeDataFolder()   // Create new free data folder.
   // The free data folder persists because it is the current data folder
   // and therefore is referenced by Igor's internal
   // current data folder reference variable.
   . . .

   // Change Igor's internal current data folder reference
   SetDataFolder root:
   // The free data folder is deleted since there are no references to it.
End
```

Free Data Folder Objects Lifetime

Next we consider what happens to objects in a free data folder when the free data folder is deleted. In this event, numeric and string variables in the free data folder are unconditionally automatically deleted. A wave is automatically deleted if there are no wave references to it. If there is a wave reference to it, the wave survives and becomes a free wave. Free waves are waves that exists outside of any data folder as explained under **Free Waves** on page IV-91.

For example:

```
Function Test()
   SetDataFolder NewFreeDataFolder()   // Create new free data folder.
   // The free data folder exists because it is the current data folder.

   Make jack      // Make a wave and an automatic wave reference

   . . .

   SetDataFolder root:
   // The free data folder is deleted since there are no references to it.
   // Because there is a reference to the wave jack, it persists
   // and becomes a free wave.

   . . .

End   // The wave reference to jack ceases to exist so jack is deleted
```

When this function ends, the reference to the wave jack ceases to exist, there are no references to jack, and it is automatically deleted.

Next we look at a slight variation. In the following example, Make does not create an automatic wave reference because of the use of $, and we do not create an explicit wave reference:

```
Function Test()
   SetDataFolder NewFreeDataFolder()   // Create new free data folder.
   // The free data folder exists because it is the current data folder.

   Make $"jack"         // Make a wave but no wave reference
   // jack persists because the current data folder references it.
   . . .

   SetDataFolder root:
   // The free data folder is deleted since there are no references to it.
   // jack is also deleted because there are no more references to it.

   . . .

End
```

Converting a Free Data Folder to a Global Data Folder

You can use **MoveDataFolder** to move a free data folder into the global hierarchy. The data folder and all of its contents then become global. The name of a free data folder created by NewFreeDataFolder is 'freeroot'. You should rename it after moving to a global context. For example:

```
Function Test()
   DFREF saveDF = GetDataFolderDFR()
   DFREF dfr = NewFreeDataFolder()      // Create free data folder
   SetDataFolder dfr                    // Set as current data folder
   Make jack=sin(x/8)                   // Create some data in it
   SetDataFolder saveDF                 // Restore original current data folder
   MoveDataFolder dfr, root:            // Free DF becomes root:freeroot
   RenameDataFolder root:freeroot,TestDF  // Rename with a proper name
   Display root:TestDF:jack
End
```

Note that MoveDataFolder requires that the data folder name, freeroot in this case, be unique within the destination data folder.

Structures in Functions

You can define structures in procedure files and use them in functions. Structures can be used only in user-defined functions as local variables and their behavior is defined almost entirely at compile time. Runtime or interactive definition and use of structures is not currently supported; for this purpose, use Data Folders (see Chapter II-8, **Data Folders**), the **StringByKey** function (see page V-997), or the **NumberByKey** function (see page V-714).

Use of structures is an advanced technique. If you are just starting with Igor programming, you may want to skip this section and come back to it later.

Simple Structure Example

Before we get into the details, here is a quick example showing how to define and use a structure.

```
Structure DemoStruct
   double dval
   int32 ival
   char str[100]
EndStructure

Function Subroutine(s)
   STRUCT DemoStruct &s       // Structure parameter

   Printf "dval=%g; ival=%d; str=\"%s\"\r", s.dval, s.ival, s.str
End

Function Routine()
   STRUCT DemoStruct s        // Local structure instance
   s.dval = 1.234
   s.ival = 4321
   s.str = "Hello"
   Subroutine(s)
End
```

As this example shows, you define a structure type using the Structure keyword in a procedure file. You define a local structure variable using the STRUCT keyword in a user-defined function. And you pass a structure from one user-defined function to another as a pass-by-reference parameter, as indicated by the use of & in the subroutine parameter declaration.

Defining Structures

Structures are defined in a procedure file with the following syntax:

```
Structure structureName
   memType memName [arraySize] [, memName [arraySize]]
   …
EndStructure
```

Structure member types (*memType*) can be any of the following Igor objects: Variable, String, WAVE, NVAR, SVAR, DFREF, FUNCREF, or STRUCT.

Igor structures also support additional member types, as given in the next table, for compatibility with C programming structures and disk files.

Igor Member Type	C Equivalent	Byte Size	Note
char	signed char	1	
uchar	unsigned char	1	
int16	16-bit int	2	
uint16	unsigned 16-bit int	2	
int32	32-bit int	4	
uint32	unsigned 32-bit int	4	
int64	64-bit int	8	Igor7 or later
uint64	unsigned 64-bit int	8	Igor7 or later
float	float	4	
double	double	8	

The Variable and double types are identical although Variable can be also specified as complex using the /C flag.

The optional *arraySize* must be specified using an expression involving literal numbers or locally-defined constants enclosed in brackets. The value must be between 1 and 400 for all but STRUCT where the upper limit is 100. The upper limit is a consequence of how memory is allocated on the stack frame.

Structures are two-byte aligned. This means that if an odd number of bytes has been allocated and then a nonchar field is defined, an extra byte of padding is inserted in front of the new field. This is mainly of concern only when reading and writing structures from and to disk files.

The Structure keyword can be preceded with the **Static** keyword (see page V-906) to make the definition apply only to the current procedure window. Without the Static designation, structures defined in one procedure window may be used in any other.

Structure Initialization

When a user-defined function that declares a structure is called, the structure fields are automatically initialized. Numeric fields are initialized to zero. String, WAVE, NVAR, SVAR, DFREF, and FUNCREF fields are initialized to null.

You can further initialize numeric and string fields using normal assignment statements.

You can further initialize WAVE, NVAR, SVAR, DFREF, and FUNCREF fields using the same syntax as used for the corresponding non-structure variables. See **Runtime Lookup of Globals** on page IV-65 and **Function References** on page IV-107.

You can also use the **StructFill** operation to automatically initialize NVAR, SVAR, and WAVE fields.

Using Structures

To use ("instantiate") a structure in a function, you must allocate a STRUCT variable using:

```
STRUCT sName name
```

where *sName* is the name of an existing structure and *name* is the local structure variable name.

To access a member of a structure, specify the STRUCT variable name followed by a "." and the member name:

```
STRUCT Point pt
pt.v= 100
```

When a member is defined as an array:

```
Structure mystruct
   Variable var1
   Variable var2[10]
   …
EndStructure
```

you must specify `[index]` to use a given element in the array:

```
STRUCT mystruct ms
ms.var2[n]= 22
```

Structure and field names must be literal and can not use `$str` notation.

The *index* value can be a variable calculated at runtime or a literal number.

If the field is itself a STRUCT, continue to append "." and field names as needed.

You can define an array of structures as a field in a structure:

```
Structure mystruct
   STRUCT Point pt[100]    // Allowed as a sub-structure
EndStructure
```

However, you can not define an array of structures as a local variable in a function:

```
STRUCT Point pt[100]       // Not allowed as a function local variable
```

Structures can be passed to functions **only** by reference, which allows them to be both input and output parameters (see **Pass-By-Reference** on page IV-59). The syntax is:

```
STRUCT sName  &varName
```

In a user function you define the input parameter:

```
Function myFunc(s)
   STRUCT mystruct &s
   …
End
```

Char and uchar arrays can be treated as zero-terminated strings by leaving off the brackets. Because the Igor compiler knows the size, the entire array can be used with no zero termination. Like normal string variables, concatenation using += is allowed but substring assignment using `[p1,p2]= subStr` is not supported.

Structures, including substructures, can be copied using simple assignment from one structure to the other. The source and destination structures must defined using the same structure name.

The **Print** operation can print individual elements of a structure or can print a summary of the entire STRUCT variable.

Structure Example

Here is a contrived example using structures. Try executing `foo(2)`:

```
Constant kCaSize = 5

Structure substruct
   Variable v1
   Variable v2
EndStructure

Structure mystruct
   Variable var1
   Variable var2[10]
   String s1
   WAVE fred
   NVAR globVar1
   SVAR globStr1
   FUNCREF myDefaultFunc afunc
   STRUCT substruct ss1[3]
   char ca[kCaSize+1]
EndStructure

Function foo(n)
   Variable n

   Make/O/N=20 fred
   Variable/G globVar1 = 111
   String/G aGlobStr="a global string var"

   STRUCT mystruct ms
   ms.var1 = 11
   ms.var2[n] = 22
   ms.s1 = "string s1"
   WAVE ms.fred     // could have =name if want other than waves named fred
   NVAR ms.globVar1
   SVAR ms.globStr1 = aGlobStr
   FUNCREF myDefaultFunc ms.afunc = anotherfunc
   ms.ss1[n].v1 = ms.var1/2
   ms.ss1[0] = ms.ss1[n]
   ms.ca = "0123456789"
   bar(ms,n)
   Print ms.var1,ms.var2[n],ms.s1,ms.globVar1,ms.globStr1,ms.ss1[n].v1
   Print ms.ss 1[n].v2,ms.ca,ms.afunc()
   Print "a whole wave",ms.fred
   Print "the whole ms struct:",ms

   STRUCT substruct ss
   ss = ms.ss1[n]
   Print "copy of substruct",ss
End

Function bar(s,n)
   STRUCT mystruct &s
   Variable n

   s.ss1[n].v2 = 99
   s.fred = sin(x)
   Display s.fred
End

Function myDefaultFunc()
   return 1
End

Function anotherfunc()
   return 2
End
```

Note the use of WAVE, NVAR, SVAR and FUNCREF in the function foo. These keywords are required both in the structure definition and again in the function, when the structure members are initialized.

Built-In Structures

Igor includes a few special purpose, predefined structures for use with certain operations. Some of those structures use these predefined general purpose structures:

```
Structure Rect
   Int16 top,left,bottom,right
EndStructure

Structure Point
   Int16 v,h
EndStructure

Structure RGBColor
   UInt16 red, green, blue
EndStructure
```

A number of operations use built-in structures that the Igor programmer can use. See the command reference information for details about these structures and their members.

Operation	Structure Name
Button	**WMButtonAction**
CheckBox	**WMCheckboxAction**
CustomControl	**WMCustomControlAction**
ListBox	**WMListboxAction**
ModifyFreeAxis	**WMAxisHookStruct**
PopupMenu	**WMPopupAction**
SetVariable	**WMSetVariableAction**
SetWindow	**WMWinHookStruct**
SetWindow	**WMTooltipHookStruct**
Slider	**WMSliderAction**
TabControl	**WMTabControlAction**

Applications of Structures

Structures are useful for reading and writing disk files. The **FBinRead** and the **FBinWrite** understand structure variables and read or write the entire structure from or to a disk file. The individual fields of the structure are byte-swapped if you use the /B flag.

Structures can be used in complex programming projects to reduce the dependency on global objects and to simplify passing data to and getting data from functions. For example, a base function might allocate a local structure variable and then pass that variable on to a large set of lower level routines. Because structure variables are passed by reference, data written into the structure by lower level routines is available to the higher level. Without structures, you would have to pass a large number of individual parameters or use global variables and data folders.

Using Structures with Windows and Controls

Action procedures for controls and window hook functions take parameters that use predefined structure types. These are listed under **Built-In Structures** on page IV-103.

Advanced programmers should also be aware of userdata that can be associated with windows using the **SetWindow** operation (see page V-865). Userdata is binary data that persists with individual windows; it is suitable for storing structures. Storing structures in a window's userdata is very handy in eliminating the need for global variables and reduces the bookkeeping needed to synchronize those globals with the window's life cycle. Userdata is also available for use with controls. See the **ControlInfo**, **GetWindow**, **GetUserData**, and **SetWindow** operations.

Here is an example illustrating built-in and user-defined structures along with userdata in a control. Put the following in the procedure window of a new experiment and run the Panel0 macro. Then click on the buttons. Note that the buttons remember their state even if the experiment is saved and reloaded. To fully understand this example, examine the definition of WMButtonAction in the **Button** operation (see page V-55).

```
#pragma rtGlobals=1        // Use modern global access method.

Structure mystruct
    Int32 nclicks
    double lastTime
EndStructure

Function ButtonProc(bStruct) : ButtonControl
    STRUCT WMButtonAction &bStruct

    if( bStruct.eventCode != 1 )
        return 0         // we only handle mouse down
    endif

    STRUCT mystruct s1
    if( strlen(bStruct.userdata) == 0 )
        Print "first click"
    else
        StructGet/S s1,bStruct.userdata
        String ctime= Secs2Date(s1.lastTime, 1 )+" "+Secs2Time(s1.lastTime,1)
    // Warning: Next command is wrapped to fit on the page.
        Printf "button %s clicked %d time(s), last click =
%s\r",bStruct.ctrlName,s1.nclicks,ctime
    endif
    s1.nclicks += 1
    s1.lastTime= datetime
    StructPut/S s1,bStruct.userdata
    return 0
End

Window Panel0() : Panel
    PauseUpdate; Silent 1         // building window…
    NewPanel /W=(150,50,493,133)
    SetDrawLayer UserBack
    Button b0,pos={12,8},size={50,20},proc=ButtonProc,title="Click"
    Button b1,pos={65,8},size={50,20},proc=ButtonProc,title="Click"
    Button b2,pos={119,8},size={50,20},proc=ButtonProc,title="Click"
    Button b3,pos={172,8},size={50,20},proc=ButtonProc,title="Click"
    Button b4,pos={226,8},size={50,20},proc=ButtonProc,title="Click"
EndMacro
```

Limitations of Structures

Although structures can reduce the need for global variables, they do not eliminate them altogether. A structure variable, like all local variables in functions, disappears when its host function returns. In order to maintain state information, you need to store and retrieve structure information using global variables. You can do this using a global variable for each field or, with certain restrictions, you can store entire structure variables in a single global using the **StructPut** operation (see page V-1004) and the **StructGet** operation (see page V-1003).

As of Igor Pro 5.03, a structure can be passed to an external operation or function. See the Igor XOP Toolkit manual for details.

Using StructFill, StructPut, and StructGet

Igor Pro 8 provides the new convenience operation StructFill, which reads in NVAR, SVAR and WAVE fields, along with relaxed limitations for the StructPut and StructGet operations. The following example illustrates these operations.

```
Structure t
   SVAR gStructData
   WAVE myData
   double someResults
EndStructure

Function AnalyzeThis(DFREF df)
   STRUCT t t
   StructFill/AC=1/SDFR=df t
   if( V_Error )
      print "no data"
      return -1
   endif
   if( strlen(t.gStructData) != 0 )       // this may have been autocreated
      StructGet/S t,t.gStructData
      print "previous result:", t.someResults
   else
      print "no previous result"
   endif
   t.someResults= mean(t.myData)
   print "current result:", t.someResults
   StructPut/S t,t.gStructData
   return 0
End

Function Demo()
   NewDataFolder/O/S root:data1
   Make/O myData= gnoise(1)
   SetDataFolder root:

   AnalyzeThis(root:data1)
   myData= gnoise(2)
   AnalyzeThis(root:data1)
End
```

Running Demo() then prints this:

```
  no previous result
  current result:  -0.0675801
  previous result:  -0.0675801
  current result:  -0.00269252
```

Static Functions

You can create functions that are local to the procedure file in which they appear by inserting the keyword *Static* in front of *Function* (see **Static** on page V-906 for usage details). The main reason for using this technique is to minimize the possibility of your function names conflicting with other names, thereby making the use of common intuitive names practical.

Functions normally have global scope and are available in any part of an Igor experiment, but the static keyword limits the scope of the function to its procedure file and hides it from all other procedure files. Static functions can only be used in the file in which they are defined. They can not be called from the command line and they cannot be accessed from macros.

Because static functions cannot be executed from the command line, you will have write a public test function to test them.

You can break this rule and access a static function using a module name; see **Regular Modules** on page IV-236.

Non-static functions (functions without the static keyword) are sometimes called “public” functions.

ThreadSafe Functions

A ThreadSafe function is one that can operate correctly during simultaneous execution by multiple threads.

ThreadSafe user functions provide support for multiprocessing and can be used for preemptive multitasking background tasks.

Note: Writing a multitasking program is for expert programmers only. Intermediate programmers can write thread-safe curve-fitting functions and multithreaded assignment statements; see **Automatic Parallel Processing with MultiThread** on page IV-323. Beginning programmers should gain experience with regular programming before using multitasking.

You create thread safe functions by inserting the keyword `ThreadSafe` in front of `Function`. For example:

```
ThreadSafe Function myadd(a,b)
   Variable a,b

   return a+b
End
```

Only a subset of Igor's built-in functions and operations can be used in a threadsafe function. Generally, numeric or utility functions can be used but those that access windows can not. To determine if a routine is ThreadSafe, use the Command Help tab of the Help Browser.

Although file operations are listed as threadsafe, they have certain limitations when running in a threadsafe function. If a file load hits a condition that normally would need user assistance, the load is aborted. No printing to history is done.

Threadsafe functions can call other threadsafe functions but may not call non-threadsafe functions. Non-threadsafe functions can call threadsafe functions.

When threadsafe functions execute in the main thread, they have normal access to data folders, waves, and variables. But when running in a preemptive thread, threadsafe functions use their own private data folders, waves, and variables.

When a thread is started, waves can be passed to the function as input parameters. Such waves are flagged as being in use by the thread, which prevents any changes to the size of the wave. When all threads under a given main thread are finished, the waves return to normal. You can pass data folders between the main thread and preemptive threads but such data folders are never shared.

See **ThreadSafe Functions and Multitasking** on page IV-329 for a discussion of programming with preemptive multitasking threads.

Function Overrides

In some very rare cases, you may need to temporarily change an existing function. When that function is part of a package provided by someone else, or by WaveMetrics, it may be undesirable or difficult to edit the original function. By using the keyword "Override" in front of "Function" you can define a new function that will be used in place of another function of the same name that is defined in a *different and later* procedure file.

Although it is difficult to determine the order in which procedure files are compiled, the main procedure window is always first. Therefore, always define override functions in the main procedure file.

Although you can override static functions, you may run into a few difficulties. If there are multiple files with the same static function name, your override will affect all of them, and if the different functions have different parameters then you will get a link error.

Here is an example of the Override keyword. In this example, start with a new experiment and create a new procedure window. Insert the following in the new window (not the main procedure window).

```
Function foo()
   print "this is foo"
```

```
End

Function Test()
   foo()
End
```

Now, on the command line, execute Test(). You will see “this is foo” in the history.

Open the main procedure window and insert the following:

```
Override Function foo()
   print "this is an override version of foo"
End
```

Now execute Test() again. You will see the “this is an override version of foo” in the history.

Function References

Function references provide a way to pass a function to a function. This is a technique for advanced programmers. If you are just starting with Igor programming, you may want to skip this section and come back to it later.

To specify that an input parameter to a function is a function reference, use the following syntax:

```
Function Example(f)
   FUNCREF myprotofunc f
   . . .
End
```

This specifies that the input parameter f is a function reference and that a function named myprotofunc specifies the kind of function that is legal to pass. The calling function passes a reference to a function as the f parameter. The called function can use f just as it would use the prototype function.

If a valid function is not passed then the prototype function is called instead. The prototype function can either be a default function or it can contain error handling code that makes it obvious that a proper function was not passed.

Here is the syntax for creating function reference variables in the body of a function:

```
FUNCREF protoFunc f = funcName
FUNCREF protoFunc f = $"str"
FUNCREF protoFunc f = <FuncRef>
```

As shown, the right hand side can take either a literal function name, a $ expression that evaluates to a function name at runtime, or it can take another FUNCREF variable.

FUNCREF variables can refer to external functions as well as user-defined functions. However, the prototype function must be a user-defined function and it must not be static.

Although you can store a reference to a static function in a FUNCREF variable, you can not then use that variable with Igor operations that take a function as an input. FuncFit is an example of such an operation.

Following are some example functions and FUNCREFs that illustrate several concepts:

```
Function myprotofunc(a)
   Variable a

   print "in myprotofunc with a= ",a
End

Function foo1(var1)
   Variable var1
   print "in foo1 with a= ",var1
End

Function foo2(a)
   Variable a

   print "in foo2 with a= ",a
End

Function foo3(a)
   Variable a

   print "in foo3 with a= ",a
End

Function badfoo(a,b)
   Variable a,b

   print "in badfoo with a= ",a
End

Function bar(a,b,fin)
   Variable a,b
   FUNCREF myprotofunc fin

   if( a==1 )
      FUNCREF myprotofunc f= foo1
   elseif( a==2 )
      FUNCREF myprotofunc f= $"foo2"
   elseif( a==3 )
      FUNCREF myprotofunc f= fin
   endif
   f(b)
End
```

For the above functions, the following table shows the results for various invocations of the bar function executed on the command line:

Command	Result
`bar(1,33,foo3)`	`in foo1 with a= 33`
`bar(2,44,foo3)`	`in foo2 with a= 44`
`bar(3,55,foo3)`	`in foo3 with a= 55`
`bar(4,55,foo3)`	`in myprotofunc with a= 55`

Executing `bar(3,55,badfoo)` generates a syntax error dialog that highlights “badfoo” in the command. This error results because the format of the badfoo function does not match the format of the prototype function, myprotofunc.

Conditional Compilation

Compiler directives can be used to conditionally include or exclude blocks of code. This is especially useful when an XOP may or may not be available. It is also convenient for supporting different versions of Igor with the same code and for testing and debugging code.

For example, to enable a block of procedure code depending on the presence or absence of an XOP, use

```
#if Exists("nameOfAnXopRoutine")
   <code using XOP routines>
#endif
```

To conditionally compile based on the version of Igor, use:

```
#if IgorVersion() < 7.00
   <code for Igor6 or before>
#else
   <code for Igor7 or later>
#endif
```

Conditional Compilation Directives

The conditional compiler directives are modeled after the C/C++ language. Unlike other #keyword directives, these may be indented. For defining symbols, the directives are:

```
#define symbol
```

```
#undef symbol
```

For conditional compilation, the directives are:

```
#ifdef symbol
```

```
#ifndef symbol
```

```
#if expression
```

```
#elif expression
```

```
#else
```

```
#endif
```

Expressions are ordinary Igor expressions, but cannot involve any user-defined objects. They evaluate to TRUE if the absolute value is > 0.5.

Conditionals must be either completely outside or completely inside function definitions; they cannot straddle a function definition. Conditionals cannot be used within macros but the **defined** function can.

Nesting depth is limited to 16 levels. Trailing text other than a comment is illegal.

Conditional Compilation Symbols

#define is used purely for defining symbols (there is nothing like C's preprocessor) and the only use of a symbol is with #if, #ifdef, #ifndef and the defined function.

The defined function allows you to test if a symbol was defined using #define:

```
#if defined(symbol)
```

Symbols exist only in the file where they are defined; the only exception is for symbols defined in the main procedure window, which are available to all other procedure files except independent modules. In addition, you can define global symbols that are available in all procedure windows (including independent modules) using:

```
SetIgorOption poundDefine=symb
```

This adds one symbol to a global list. You can query the global list using:

```
SetIgorOption poundDefine=symb?
```

This sets V_flag to 1 if symb exists or 0 otherwise. To remove a symbol from the global list use:

```
SetIgorOption poundUndefine=symb
```

For non-independent module procedure windows, a symbol is defined if it exists in the global list *or* in the main procedure window's list *or* in the given procedure window.

For independent module procedure windows, a symbol is defined if it exists in the global list *or* in the given procedure window; it does not use the main procedure window list.

A symbol defined in a global list is not undefined by a #undef in a procedure window.

Predefined Global Symbols

These global symbols are predefined if appropriate and available in all procedure windows:

Symbol	Automatically Predefined If
MACINTOSH	The Igor application is a Macintosh application.
WINDOWS	The Igor application is a Windows application.
IGOR64	The Igor application is a 64-bit application.

Conditional Compilation Examples

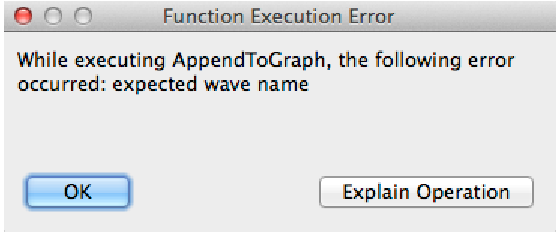
```
#define MYSYMBOL

#ifdef MYSYMBOL

Function foo()
   Print "This is foo when MYSYMBOL is defined"
End

#else

Function foo()
   Print "This is foo when MYSYMBOL is NOT defined"
End

#endif   // MYSYMBOL

// This works in Igor Pro 6.10 or later
#if IgorVersion() >= 7.00
   <code for Igor7 or later>
#else
   <code for Igor6 or before>
#endif

// This works in Igor Pro 6.20 or later
#if defined(MACINTOSH)
   <conditionally compiled code here>
#endif
```

Function Errors

During function compilation, Igor checks and reports syntactic errors and errors in parameter declarations. The normal course of action is to edit the offending function and try to compile again.

Runtime errors in functions are not reported on the spot. Instead, Igor saves information about the error and function execution continues. Igor presents an error dialog only after the last function ceases execution and Igor returns to the idle state. If multiple runtime errors occur, only the first is reported.

When a runtime error occurs, after function execution ends, Igor presents an error dialog:



While executing AppendToGraph, the following error occurred: expected wave name

In this example, we tried to pass to the AppendToGraph function a reference to a wave that did not exist. To find the source of the error, you should use Igor's debugger and set it to break on error (see **Debugging on Error** on page IV-213 for details).

Runtime Errors From Built-in Functions

Normally, when an error occurs in a built-in function, the built-in function does not post an error but instead returns 0, NaN or an empty string as the function result. As a debugging aid, you can use the rtFunctionErrors pragma to force Igor to post an error. See **The rtFunctionErrors Pragma** on page IV-55 for details.

Custom Error Handling

Normally when an error occurs, Igor aborts function execution.

Sophisticated programmers may want to detect and deal with runtime errors on their own. See **Flow Control for Aborts** on page IV-48 for details and examples.

Coercion in Functions

The term "coercion" means the conversion of a value from one numeric precision or numeric type to another. Consider this example:

```
Function foo(awave)
   WAVE/C awave

   Variable/C var1

   var1 = awave[2]*cmplx(2,3)
   return real(var1)
End
```

The parameter declaration specifies that awave is complex. You can pass any kind of wave you like but it will be coerced into complex before use. For example, if you pass a real valued integer wave the value at point index 2 will be converted to double precision and zero will be used for the imaginary part.

Operations in Functions

You can call most operations from user-defined functions. To provide this capability, WaveMetrics had to create special code for each operation. Some operations weren't worth the trouble or could cause problems. If an operation can't be invoked from a function, an error message is displayed when the function is compiled. The operations that can't be called from a function are:

```
AppendToLayout      Layout       Modify              OpenProc
PrintGraphs         Quit         RemoveFromLayout
Stack               Tile
```

If you need to invoke one of these operations from a user-defined function, use the Execute operation. See **The Execute Operation** on page IV-201.

While Modify can not be called from a function, ModifyGraph, ModifyTable and ModifyLayout can.

You can use the **NewLayout** instead of Layout, the **AppendLayoutObject** instead of AppendToLayout, and the **RemoveLayoutObjects** instead of RemoveFromLayout.

External operations implemented by very old XOPs also can not be called directly from user-defined functions. Again, the solution is to use the Execute operation.

Updates During Function Execution

An update is an action that Igor performs which consists of:

- Reexecuting formulas for dependent objects whose antecedents have changed (see Chapter IV-9, **Dependencies**)
- Redrawing graphs, tables and Gizmo plots which display waves that have changed
- Redrawing page layouts containing graphs, tables, Gizmo plots, or annotations that have changed
- Redrawing windows that have been uncovered

When no procedure is executing, Igor continually checks whether an update is needed and does an update if necessary.

When a user-defined function is executing, Igor does no automatic updates at all. You can force an update by calling the **DoUpdate** operation (see page V-168). Call DoUpdate if you don’t want to wait for the next automatic update which will occur when function execution finishes.

Aborting Functions

There are two ways to prematurely stop procedure execution: a user abort or a programmed abort. Both stop execution of all procedures, no matter how deeply nested.

You can abort function execution by pressing the **User Abort Key Combinations** or by clicking the Abort button in the status bar. You may need to press the keys down for a while because Igor looks at the keyboard periodically and if you don’t press the keys long enough, Igor will not see them.

A user abort does not directly return. Instead it sets a flag that stops loops from looping and then returns using the normal calling chain. For this reason some code will still be executed after an abort but execution should stop quickly. This behavior releases any temporary memory allocations made during execution.

A **programmed abort** occurs during procedure execution according to conditions set by the programmer.

The simplest programmed abort occurs when the **Abort** operation (see page V-18) is executed. Here is an example:

```
if (numCells > 10)
   Abort "Too many cells! Quitting."
endif
// code here doesn't execute if numCells > 10
```

Other programmed aborts can be triggered using the AbortOnRTE and AbortOnValue flow control keywords. The try-catch-endtry flow control construct can be used for catching and testing for aborts. See **Flow Control for Aborts** on page IV-48 for more details.

If your code uses Igor pre-emptive threads that can run for a long time, you should detect aborts and make your threads quit. See **Aborting Threads** on page IV-337 for details.

Igor Pro 7 Programming Extensions

The following programming features were added in Igor Pro 7. They bring Igor programming closer to the C language. If you use these features, your procedures will not compile in Igor Pro 6.

Double and Complex Variable Types

You can use `double` and `complex` in functions as aliases for `Variable` and `Variable/C`:

```
Function foo()
   double d = 4
   complex c = cmplx(2,3)
   Print d,c
End
```

Integer Variable Types

In Igor Pro 7 or later you can use the integer types `int`, `int64` and `uint64` for parameters and local variables in user-defined functions. You can also use `int64` and `uint64` in structures. See **Integer Parameters** on page IV-33 for details.

Integer Expressions in User-Defined Functions

Prior to Igor Pro 7, all calculations were performed in double-precision floating point. In Igor Pro 7 or later, Igor uses integer calculations when the destination is an integer type. See **Expression Evaluation** on page IV-38 for details.

Inline Parameters in User-Defined Functions

In Igor Pro 7 or later you can declare user-defined functions parameters inline. See **Inline Parameters** on page IV-33 for details.

Line Continuation in User-Defined Functions

In user-defined functions in Igor Pro 7 or later, you can use arbitrarily long expressions by including a line continuation character at the very end of a line. See **Line Continuation** on page IV-35 for details.

Bit Shift Operators in User-Defined Functions

Igor Pro 7 or later supports the bit shift operators << and >> on local variables. See **Bit Shift Operators** on page IV-43 for details.

Increment and Decrement Operators in User-Defined Functions

In a user-defined function in Igor Pro 7 or later, you can use increment and decrement operators on local variables. See **Increment and Decrement Operators** on page IV-43 for details.

Legacy Code Issues

This section discusses changes that have occurred in Igor programming over the years. If you are writing new code, you don’t need to be concerned with these issues. If you are working with existing code, you may run into some of them.

If you are just starting to learn Igor programming, you have enough to think about already, so it is a good idea to skip this section. Once you are comfortable with the modern techniques described above, come back and learn about these antiquated techniques.

Old-Style Comments and Compatibility Mode

In Igor Pro 1.00 through 3.00, the comment symbol was the vertical bar (|). Starting with Igor Pro 4.00 vertical bar is used as the bitwise OR operator (see Operators) and the comment symbol is //.

In the unlikely event that you encounter an ancient procedure file that used vertical bar as the comment symbol, you need to replace the vertical bars with //.

From Igor Pro 4.00 through Igor Pro 6.37, there was a compatibility mode that told Igor to accept vertical bar as a comment symbol:

```
Silent 100        // Enter compatibility mode
```

Support for this compatibility mode was removed in Igor Pro 7.00.

Text After Flow Control

Prior to Igor Pro 4, Igor ignored any extraneous text after a flow control statement. Such text was an error, but Igor did not detect it.

Igor now checks for extra text after flow control statements. When found, a dialog is presented asking the user if such text should be considered an error or not. The answer lasts for the life of the Igor session.

Because previous versions of Igor ignored this extra text, it may be common for existing procedure files to have this problem. The text may in many cases simply be a typographic error such as an extra closing parenthesis:

```
if( a==1 ))
```

In other cases, the programmer may have thought they were creating an elseif construct:

```
else if( a==2 )
```

even though the “if(a==2)” part was simply ignored. In some cases this may represent a bug in the programmer’s code but most of the time it is asymptomatic.

Global Variables Used by Igor Operations

The section **Local Variables Used by Igor Operations** on page IV-61 explains that certain Igor operations create and set certain special local variables. Very old procedure code expects such variables to be created as global variables and must be rewritten.

Also explained in **Local Variables Used by Igor Operations** on page IV-61 is the fact that some operations, such as CurveFit, look for certain special local variables which modify the behavior of the operations. For historic reasons, operations that look for special variables will look for global variables in the current data folder if the local variable is not found. This behavior is unfortunate and may be removed from Igor some day. New programming should use local variables for this purpose.

Direct Reference to Globals

The section **Accessing Global Variables and Waves** on page IV-65 explains how to access globals from a procedure file. Very old procedure files may attempt to reference globals directly, without using WAVE, NVAR, or SVAR statements. This section explains how to update such procedures.

Here are the steps for converting a procedure file to use the runtime lookup method for accessing globals:

1. Insert the #pragma rtGlobals=3 statement, with no indentation, at or near the top of the procedure in the file.
2. Click the Compile button to compile the procedures.
3. If the procedures use a direct references to access a global, Igor will display an error dialog indicating the line on which the error occurred. Add an NVAR, SVAR or WAVE reference.
4. If you encountered an error in step 3, fix it and return to step 2.

Chapter

IV-4

Macros

Overview

When we first created Igor, some time in the last millennium, it supported automation through macros. The idea of the macro was to allow users to collect commands into conveniently callable routines. Igor interpreted and executed each command in a macro as if it were entered in the command line.

WaveMetrics soon realized the need for a faster, more robust technology that would support full-blown programming. This led to the addition of user-defined functions. Because functions are compiled, they execute much more quickly. Also, compilation allows Igor to catch syntactic errors sooner. Functions have a richer set of flow control capabilities and support many other programming refinements.

Over time, the role of macros has diminished in favor of functions. With rare exceptions, new programming should be done using user-defined functions.

Macros are still supported and there are still a few uses in which they are preferred. When you close a graph, table, page layout, Gizmo plot, or control panel, Igor offers to automatically create a window recreation macro which you can later run to recreate the window. You can also ask Igor to automatically create a window style macro using the Window Control dialog. The vast majority of programming, however, should be done using functions.

The syntax and behavior of macros are similar to the syntax and behavior of functions, but the differences can be a source of confusion for someone first learning Igor programming. If you are just starting, you can safely defer reading the rest of this chapter until you need to know more about macros, if that time ever comes.

Comparing Macros and Functions

Like functions, macros are created by entering text in procedure windows. Each macro has a name, a parameter list, parameter declarations, and a body. Unlike functions, a macro has no return value.

Macros and functions use similar syntax. Here is an example of each. To follow along, open the Procedure window (Windows menu) and type in the macro and function definitions.

```
Macro MacSayHello(name)
   String name

   Print "Hello "+name
End

Function FuncSayHello(name)
   String name

   Print "Hello "+name
End
```

Now click in the command window to bring it to the front.

If you execute the following on the command line

```
MacSayHello("John"); FuncSayHello("Sue")
```

you will see the following output printed in the history area:

```
Hello John
Hello Sue
```

This example may lead you to believe that macros and functions are nearly identical. In fact, there are a lot of differences. The most important differences are:

- Macros automatically appear in the Macros menu. Functions must be explicitly added to a menu, if desired, using a menu definition.
- Most errors in functions are detected when procedures are compiled. Most errors in macros are detected when the macro is executed.

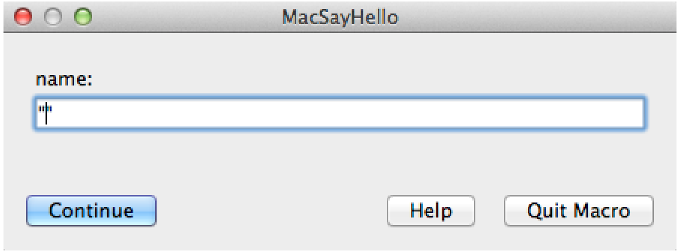
- Functions run a lot faster than macros.
- Functions support wave parameters, for loops, switches, structures, multithreading, and many other features not available in macros.
- Functions have a richer syntax.

If you look in the Macros menu, you will see MacSayHello but not FuncSayHello.

If you execute `FuncSayHello()` on the command line you will see an error dialog. This is because you must supply a parameter. You can execute, for example:

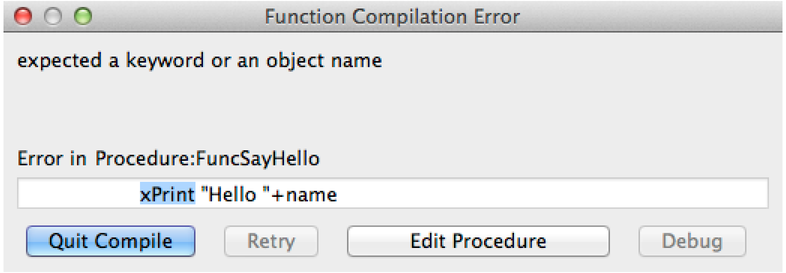
```
FuncSayHello("Sam")
```

On the other hand, if you run MacSayHello from the Macros menu or if you execute `MacSayHello()` on the command line, you will see a dialog that you use to enter the name before continuing:



This is called the “missing parameter dialog”. It is described under **The Missing Parameter Dialog** on page IV-121. Functions can display a similar dialog, called a simple input dialog, with a bit of additional programming.

Now try this: In both procedures, change “Print” to “xPrint”. Then click in the command window. You will see an error dialog complaining about the xPrint in the function:



expected a keyword or an object name

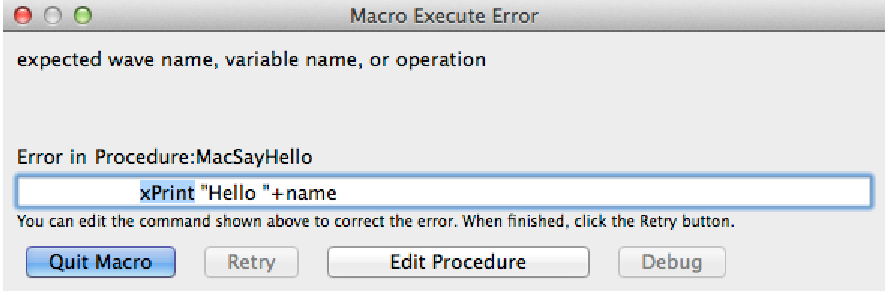
Error in Procedure:FuncSayHello

```
xPrint "Hello "+name
```

Click the Edit Procedure button and change “xPrint” back to “Print” in the function but not in the macro. Then click in the command window.

Notice that no error was reported once you fixed the error in the function. This is because only functions are compiled and thus only functions have their syntax completely checked at compile time. Macros are interpreted and most errors are found only when the line in the procedure window in which they occur is executed.

To see this, run the macro by executing “`MacSayHello("Sam")`” on the command line. Igor displays an error dialog:



expected wave name, variable name, or operation

Error in Procedure:MacSayHello

```
xPrint "Hello "+name
```

You can edit the command shown above to correct the error. When finished, click the Retry button.

Notice the outline around the line containing the error. This outline means you can edit the erroneous command. If you change “xPrint” to “Print” in this dialog, the Retry button becomes enabled. If you click Retry, Igor continues execution of the macro. When the macro finishes, take a look at the Procedure window. You will notice that the correction you made in the dialog was put in the procedure window and your “broken” macro is now fixed.

Macro Syntax

Here is the basic syntax for macros.

```
<Defining keyword> <Name> ( <Input parameter list> ) [:<Subtype>]
   <Input parameter declarations>

   <Local variable declarations>

   <Body code>
End
```

The Defining Keyword

<Defining keyword> is one of the following:

Defining Keyword	Creates Macro In
Window	Windows menu
Macro	Macros menu
Proc	—

The Window keyword is used by Igor when it automatically creates a window recreation macro. Except in rare cases, you will not write window recreation macros but instead will let Igor create them automatically.

The Procedure Name

The names of macros must follow the standard Igor naming conventions. Names can consist of up to 255 characters. Only ASCII characters are allowed. The first character must be alphabetic while the remaining characters can include alphabetic and numeric characters and the underscore character. Names must not conflict with the names of other Igor objects, functions or operations. Names in Igor are case insensitive.

Prior to Igor Pro 8.00, macro names were limited to 31 bytes. If you use long macro names, your procedures will require Igor Pro 8.00 or later.

The Procedure Subtype

You can identify procedures designed for specific purposes by using a subtype. Here is an example:

```
Proc ButtonProc(ctrlName) : ButtonControl
   String ctrlName

   Beep
End
```

Here, “ : ButtonControl” identifies a macro intended to be called when a user-defined button control is clicked. Because of the subtype, this macro is added to the menu of procedures that appears in the Button Control dialog. When Igor automatically generates a procedure it generates the appropriate subtype. See **Procedure Subtypes** on page IV-204 for details.

The Parameter List and Parameter Declarations

The parameter list specifies the name for each input parameter. Macros have a limit of 10 parameters.

The parameter declaration must declare the type of each parameter using the keywords `Variable` or `String`. If a parameter is a complex number, it must be declared `Variable/C`.

Note: There should be no blank lines or other commands until after all the input parameters are defined. There should be one blank line after the parameter declarations, before the rest of the procedure. Igor will report errors if these conditions are not met.

Variable and string parameters in macros are always passed to a subroutine by value.

When macros are invoked with some or all of their input parameters missing, Igor displays a missing parameter dialog to allow the user to enter those parameters. In the past this has been a reason to use macros. However, as of Igor Pro 4, functions can present a similar dialog to fetch input from the user, as explained under **The Simple Input Dialog** on page IV-144.

Local Variable Declarations

The input parameter declarations are followed by the local variable declarations if the macro uses local variables. Local variables exist only during the execution of the macro. They can be numeric or string and are declared using the `Variable` or `String` keywords. They can optionally be initialized. Here is an example:

```
Macro Example(p1)
   Variable p1

   // Here are the local variables
   Variable v1, v2
   Variable v3=0
   Variable/C cv1=cmplx(0,0)
   String s1="test", s2="test2"

   <Body code>
End
```

If you do not supply explicit initialization, Igor automatically initializes local numeric variables with the value zero and local string variables with the value "".

The name of a local variable is allowed to conflict with other names in Igor although they must be unique within the macro. Clearly if you create a local variable named “sin” then you will be unable to use Igor’s built-in sin function within the macro.

You can declare a local variable in any part of a macro with one exception. If you place a variable declaration inside a loop in a macro then the declaration will be executed multiple times and Igor will generate an error since local variable names must be unique.

Body Code

The local variable declarations are followed by the body code. This table shows what can appear in body code of a macro.

What	Allowed in Macros?	Comments
Assignment statements	Yes	Includes wave, variable and string assignments.
Built-in operations	Yes	
External operations	Yes	
External functions	Yes	
Calls to user functions	Yes	
Calls to macros	Yes	
if-else-endif	Yes	
if-elseif-endif	No	
switch-case-endswitch	No	
strswitch-case-endswitch	No	
try-catch-endtry	No	
structures	No	
do-while	Yes	
for-endfor	No	
Comments	Yes	Comments start with //.
break	Yes	Used in loop statements.
continue	No	
default	No	
return	Yes, but with no return value.	

Conditional Statements in Macros

The conditional if-else-endif statement is allowed in macros. It works the same as in functions. See **If-Else-Endif** on page IV-40.

Loops in Macros

The do-while loop is supported in macros. It works the same as in functions. See **Do-While Loop** on page IV-45.

Return Statement in Macros

The return keyword immediately stops executing the current macro. If it was called by another macro, control returns to the calling macro.

A macro has no return value so return is used just to prematurely quit the macro. Most macros will have no return statement.

Invoking Macros

There are several ways to invoke a macro:

- From the command line

- From the Macros, Windows or user-defined menus
- From another macro
- From a button or other user control

The menu in which a macro appears, if any, is determined by the macro's type and subtype.

This table shows how a macro's type determines the menu that Igor puts it in.

Macro Type	Defining Keyword	Menu
Macro	`Macro`	Macros menu
Window Macro	`Window`	Windows menu
Proc	`Proc`	—

If a macro has a subtype, it may appear in a different menu. This is described under **Procedure Subtypes** on page IV-204. You can put macros in other menus as described in Chapter IV-5, **User-Defined Menus**.

You can not directly invoke a macro from a user function. You can invoke it indirectly, using the **Execute** operation (see page V-204).

Using $ in Macros

As shown in the following example, the $ operator can create references to global numeric and string variables as well as to waves.

```
Macro MacroTest(vStr, sStr, wStr)
   String vStr, sStr, wStr

   $vStr += 1
   $sStr += "Hello"
   $wStr += 1
End

Variable/G gVar = 0; String/G gStr = ""; Make/O/N=5 gWave = p
MacroTest("gVar", "gStr", "gWave")
```

See **String Substitution Using $** on page IV-18 for additional examples using $.

Waves as Parameters in Macros

The only way to pass a wave to a macro is to pass the name of the wave in a string parameter. You then use the $ operator to convert the string into a wave reference. For example:

```
Macro PrintWaveStdDev(w)
   String w

   WaveStats/Q $w
   Print V_sdev
End

Make/O/N=100 test=gnoise(1)
Print NamedWaveStdDev("test")
```

The Missing Parameter Dialog

When a macro that is declared to take a set of input parameters is executed with some or all of the parameters missing, it displays a dialog in which the user can enter the missing values. For example:

```
Macro MacCalcDiag(x,y)
   Variable x=10
   Prompt x, "Enter X component: "      // Set prompt for y param
   Variable y=20
   Prompt y, "Enter Y component: "      // Set prompt for x param

   Print "diagonal=",sqrt(x^2+y^2)
End
```

If invoked from the command line or from another macro with parameters missing, like this:

```
MacCalcDiag()
```

Igor displays the Missing Parameter dialog in which the parameter values can be specified.

The Prompt statements are optional. If they are omitted, the variable name is used as the prompt text.

There must be a blank line after the set of input parameter and prompt declarations and there must not be any blank lines within the set.

The missing parameter dialog supports the creation of pop-up menus as described under **Pop-Up Menus in Simple Dialogs** on page IV-145. One difference is that in a missing parameter dialog, the menu item list can be continued using as many lines as you need. For example:

```
Prompt color, "Select Color", popup "red;green;blue;"
"yellow;purple"
```

Macro Errors

Igor can find errors in macros at two times:

- When it scans the macros
- When it executes the macros

After you modify procedure text, scanning occurs when you activate a non-procedure window, click the Compile button or choose Compile from the Macros menu. At this point, Igor is just looking for the names of procedures. The only errors that it detects are name conflicts and ill-formed names. If it finds such an error, it displays a dialog that you use to fix it.

Igor detects other errors when the macro executes. Execution errors may be recoverable or non-recoverable. If a recoverable error occurs, Igor puts up a dialog in which you can edit the erroneous line.

You can fix the error and retry or quit macro execution.

If the error is non-recoverable, you get a similar dialog except that you can’t fix the error and retry. This happens with errors in the parameter declarations and errors related to if-else-endif and do-while structures.

The Silent Option

Normally Igor displays each line of a macro in the command line as it executes the line. This gives you some idea of what is going on. However it also slows macro execution down considerably. You can prevent Igor from showing macro lines as they are executed by using the `Silent 1` command.

You can use `Silent 1` from the command line. It is more common to use it from within a macro. The effect of the `Silent` command ends at the end of the macro in which it occurs. Many macros contain the following line:

```
Silent 1; PauseUpdate
```

The Slow Option

You can observe the lines in a macro as they execute in the command line. However, for debugging purposes, they often whiz by too quickly. The `Slow` operation slows the lines down. It takes a parameter which controls how much the lines are slowed down. Typically, you would execute something like “`Slow 10`” from the command line and then “`Slow 0`” when you are finished debugging.

You can also use the `Slow` operation from within a macro. You must explicitly invoke “`Slow 0`” to revert to normal behavior. It does not automatically revert at the end of the macro from which it was invoked.

We never use this feature. Instead, we generally use print statements for debugging or we use the Igor symbolic debugger, described in Chapter IV-8, **Debugging**.

Accessing Variables Used by Igor Operations

A number of Igor’s operations return results via variables. For example, the WaveStats operation creates a number of variables with names such as V_avg, V_sigma, etc.

When you invoke these operations from the command line, they create global variables in the current data folder.

When you invoke them from a user-defined function, they create local variables.

When you invoke them from a macro, they create local variables unless a global variable already exists. If both a global variable and a local variable exist, Igor uses the local variable.

In addition to creating variables, a few operations, such as CurveFit and FuncFit, check for the existence of specific variables to provide optional behavior. The operations look first for a local variable with a specific name. If the local variable is not found, they then look for a global variable.

Updates During Macro Execution

An update is an action that Igor performs. It consists of:

- Reexecuting assignments for dependent objects whose antecedents have changed (see Chapter IV-9, **Dependencies**);
- Redrawing graphs and tables which display waves that have changed;
- Redrawing page layouts containing graphs, tables, or annotations that have changed;
- Redrawing windows that have been uncovered.

When no procedure is executing, Igor continually checks whether an update is needed and does an update if necessary.

During macro execution, Igor checks if an update is needed after executing each line. You can suspend checking using the PauseUpdate operation. This is useful when you want an update to occur when a macro finishes but not during the course of the macro’s execution.

PauseUpdate has effect only inside a macro. Here is how it is used.

```
Window Graph0() : Graph
   PauseUpdate; Silent 1
   Display /W=(5,42,400,250) w0,w1,w2
   ModifyGraph gFont="Helvetica"
   ModifyGraph rgb(w0)=(0,0,0),rgb(w1)=(0,65535,0),rgb(w2)=(0,0,0)
   <more modifies here...>
End
```

Without the PauseUpdate, Igor would do an update after each modify operation. This would take a long time.

At the end of the macro, Igor automatically reverts the state of update-checking to what it was when this macro was invoked. You can use the ResumeUpdate operation if you want to resume updates before the macro ends or you can call DoUpdate to force an update to occur at a particular point in the program flow. Such explicit updating is rarely needed.

Aborting Macros

There are two ways to prematurely stop macro execution: a user abort or a programmed abort. Both stop execution of all macros, no matter how deeply nested.

You can abort macro execution by pressing the **User Abort Key Combinations** or by clicking the Abort button in the status bar. You may need to press the keys down for a while because Igor looks at the keyboard periodically and if you don't press the keys long enough, Igor will not see them.

A user abort does not directly return. Instead it sets a flag that stops loops from looping and then returns using the normal calling chain. For this reason some code will still be executed after an abort but execution should stop quickly. This behavior releases any temporary memory allocations made during execution.

A **programmed abort** occurs when the Abort operation is executed. Here is an example:

```
if (numCells > 10)
   Abort "Too many cells! Quitting."
endif
// code here doesn't execute if numCells > 10
```

Converting Macros to Functions

If you have old Igor procedures written as macros, as you have occasion to revisit them, you can consider converting them to functions. In most cases, this is a good idea. An exception is if the macros are so complex that there would be a substantial risk of introducing bugs. In this case, it is better to leave things as they are.

If you decide to do the conversion, here is a checklist that you can use in the process.

1. Back up the old version of the procedures.
2. Change the defining keyword from Macro or Proc to Function.
3. If the macro contained Prompt statements, then it was used to generate a missing parameter dialog. Change it to generate a simple input dialog as follows:
   a. Remove the parameters from the parameter list. The old parameter declarations now become local variable declarations.
   b. Make sure that the local variable for each prompt statement is initialized to some value.
   c. Add a DoPrompt statement after all of the Prompt statements.
   d. Add a test on V_Flag after the DoPrompt statement to see if the user canceled.
4. Look for statements that access global variables or strings and create NVAR and SVAR references for them.
5. Look for any waves used in assignment statements and create WAVE references for them.
6. Compile procedures. If you get an error, fix it and repeat step 6.

Chapter

IV-5

User-Defined Menus

Overview

You can add your own menu items to many Igor menus by writing a menu definition in a procedure window. A simple menu definition looks like this:

```
Menu "Macros"
   "Load Data File/1"
   "Do Analysis/2"
   "Print Report"
End
```

This adds three items to the Macros menu. If you choose Load Data File or press Command-1 (*Macintosh*) or Ctrl+1 (*Windows*), Igor executes the procedure LoadDataFile which, presumably, you have written in a procedure window. The command executed when you select a particular item is derived from the text of the item. This is an *implicit* specification of the item's execution text.

You can also *explicitly* specify the execution text:

```
Menu "Macros"
   "Load Data File/1", Beep; LoadWave/G
   "Do Analysis/2"
   "Print Report"
End
```

Now if you choose Load Data File, Igor will execute "`Beep; LoadWave/G`".

When you choose a user menu item, Igor checks to see if there is execution text for that item. If there is, Igor executes it. If not, Igor makes a procedure name from the menu item string. It does this by removing any characters that are not legal characters in a procedure name. Then it executes the procedure. For example, choosing an item that says

```
"Set Sampling Rate..."
```

executes the SetSamplingRate procedure.

If a procedure window is the top window and if Option (*Macintosh*) or Alt (*Windows*) is pressed when you choose a menu item, Igor tries to find the procedure in the window, rather than executing it.

A menu definition can add submenus as well as regular menu items.

```
Menu "Macros"
   Submenu "Load Data File"
      "Text File"
      "Binary File"
   End

   Submenu "Do Analysis"
      "Method A"
      "Method B"
   End

   "Print Report"
End
```

This adds three items to the Macros menu, two submenus and one regular item. You can nest submenus to any depth.

Menu Definition Syntax

The syntax for a menu definition is:

```
Menu <Menu title string> [,<menu options>]
   [<Menu help strings>]
   <Menu item string> [,<menu item flags>] [,<execution text>]
   [<Item help strings>]
   …
   Submenu <Submenu title string>
      [<Submenu help strings>]
      <Submenu item string> [,<execution text>]
      [<Item help strings>]
      …
   End
End
```

<Menu title string> is the title of the menu to which you want to add items. Often this will be Macros but you can also add items to Analysis, Misc and many other built-in Igor menus, including some submenus and the graph marquee and layout marquee menus. If <Menu title string> is not the title of a built-in menu then Igor creates a new main menu on the menu bar.

<Menu options> are optional comma-separated keywords that change the behavior of the menu. The allowed keywords are dynamic, hideable, and contextualmenu. For usage, see **Dynamic Menu Items** (see page IV-129), **HideIgorMenus** (see page V-346), and **PopupContextualMenu** (see page V-756) respectively.

<Menu help strings> specifies the help for the menu title. This is optional. As of Igor7, menu help is not supported and the menu help specification, if present, is ignored.

<Menu item string> is the text to appear for a single menu item, a semicolon-separated string list to define **Multiple Menu Items** (see page IV-131), or **Specialized Menu Item Definitions** (see page IV-132) such as a color, line style, or font menu.

<Menu item flags> are optional flags that modify the behavior of the menu item. The only flag currently supported is /Q, which prevents Igor from storing the executed command in the history area. This is useful for menu commands that are executed over and over through a keyboard shortcut. This feature was introduced in Igor Pro 5. Using it will cause errors in earlier versions of Igor. Menus defined with the contextualmenu keyword implicitly set all the menu item flags in the entire menu to /Q; it doesn't matter whether /Q is explicitly set or not, the executed command is not stored in the history area.

<Execution text> is an Igor command to execute for the menu item. If omitted, Igor makes a procedure name from the menu item string and executes that procedure. Use "" to prevent command execution (useful only with PopupContextualMenu/N).

<Item help strings> specifies the help for the menu item. This is optional. As of Igor7, menu help is not supported and the menu item help specification, if present, is ignored.

The Submenu keyword introduces a submenu with <Submenu title string> as its title. The submenu continues until the next End keyword.

<Submenu item string> acts just like <Menu item string>.

Built-in Menus That Can Be Extended

The titles of the built-in menus that you can extend using user-defined menus are listed below.

These menu titles must appear in double quotes when used in a menu definition.

Use these menu titles to identify the menu to which you want to append items even if you are working with a version of Igor translated into a language other than English.

All other Igor menus, including menus added by XOPs, can not accept user-defined items.

Main Menu Bar Menus That You Can Extend

You can extend the following main menu bar menus using user-defined menus:

- Add Controls
- Analysis
- Append to Graph
- Control
- Data
- Edit
- File
- Gizmo
- Graph
- Help
- Layout
- Load Waves
- Macros
- Misc
- Statistics
- New
- Notebook
- Open File
- Panel
- Procedure
- Save Waves
- Table

Contexual Menus That You Can Extend

Contextual menus are also called pop-up menus. You can extend the following contextual menus using user-defined menus:

- AllTracesPopup
- GraphMarquee
- DataBrowserObjectsPopup
- GraphPopup
- LayoutMarquee
- TablePopup
- TracePopup
- WindowBrowserWindowsPopup

See **Marquee Menus** on page IV-137, **Trace Menus** on page IV-137, **TablePopup Menu**, **DataBrowserObjectsPopup Menu** on page IV-138, and xxx for more information about these items.

Hiding User-Defined Menu Extensions

The **HideIgorMenus** operation (see page V-346) and the **ShowIgorMenus** operation (see page V-869) hide or show most of the built-in main menus (but not the Marquee and Popup menus). User-defined menus that add items to built-in menus are normally not hidden or shown by these operations. When a built-in menu is hidden, the user-defined menu items create a user-defined menu with only user-defined items. For example, this user-defined menu:

```
Menu "Table"
   "Append Columns to Table...", DoIgorMenu "Table", "Append Columns to Table"
End
```

will create a Table menu with only one item in it after the HideIgorMenus "Table" command is executed.

To have your user-defined menu items hidden along with the built-in menu items, add the hideable keyword after the Menu definition:

```
Menu "Table", hideable
   "Append Columns to Table...", DoIgorMenu "Table", "Append Columns to Table"
End
```

Adding a New Main Menu

You can add an entirely new menu to the main menu bar by using a menu title that is not used by Igor. For example:

```
Menu "Test"
   "Load Data File"
   "Do Analysis"
   "Print Report"
End
```

Dynamic Menu Items

In the examples shown so far all of the user-defined menu items are static. Once defined, they never change. This is sufficient for the vast majority of cases and is by far the easiest way to define menu items.

Igor also provides support for dynamic user-defined menu items. A dynamic menu item changes depending on circumstances. The item might be enabled under some circumstances and disabled under others. It might be checked or deselected. Its text may toggle between two states (e.g. “Show Tools” and “Hide Tools”).

Because dynamic menus are much more difficult to program than static menus and also slow down Igor’s response to a menu-click, we recommend that you keep your use of dynamic menus to a minimum. The effort you expend to make your menu items dynamic may not be worth the time you spend to do it.

For a menu item to be dynamic, you must define it using a string expression instead of the literal strings used so far. Here is an example.

```
Function DoAnalysis()
   Print "Analysis Done"
End

Function ToggleTurboMode()
   Variable prevMode = NumVarOrDefault("root:gTurboMode", 0)
   Variable/G root:gTurboMode = !prevMode
End

Function/S MacrosMenuItem(itemNumber)
   Variable itemNumber

   Variable turbo = NumVarOrDefault("root:gTurboMode", 0)

   if (itemNumber == 1)
      if (strlen(WaveList("*", ";", ""))==0) // any waves exist?
         return "(Do Analysis"   // disabled state
      else
         return "Do Analysis"    // enabled state
      endif
   endif

   if (itemNumber == 2)
      if (turbo)
         return "!"+num2char(18)+"Turbo"  // Turbo with a check
      else
         return "Turbo"
      endif
   endif
End

Menu "Macros", dynamic
   MacrosMenuItem(1)
   MacrosMenuItem(2), /Q, ToggleTurboMode()
End
```

In this example, the text for the menu item is computed by the MacrosMenuItem function. It computes text for item 1 and for item 2 of the menu. Item 1 can be enabled or disabled. Item 2 can be checked or unchecked.

The dynamic keyword specifies that the menu definition contains a string expression that needs to be reevaluated each time the menu item is drawn. This rebuilds the user-defined menu each time the user clicks in the menu bar. Under the current implementation, it rebuilds *all* user menus each time the user clicks in the menu bar if *any* user-defined menu is declared dynamic. If you use a large number of user-defined items, the time to rebuild the menu items may be noticeable.

There is another technique for making menu items change. You define a menu item using a string expression rather than a literal string but you do not declare the menu dynamic. Instead, you call the BuildMenu operation whenever you need the menu item to be rebuilt. Here is an example:

```
Function ToggleItem1()
   String item1Str = StrVarOrDefault("root:MacrosItem1Str","On")
   if (CmpStr(item1Str,"On") == 0)      // Item is now "On"?
      String/G root:MacrosItem1Str = "Off"
   else
      String/G root:MacrosItem1Str = "On"
   endif
   BuildMenu "Macros"
End

Menu "Macros"
   StrVarOrDefault("root:MacrosItem1Str","On"), /Q, ToggleItem1()
End
```

Here, the menu item is controlled by the global string variable MacrosItem1Str. When the user chooses the menu item, the ToggleItem1 function runs. This function changes the MacrosItem1Str string and then calls BuildMenu, which rebuilds the user-defined menu the next time the user clicks in the menu bar. Under the current implementation, it rebuilds *all* user-defined menus if BuildMenu is called for *any* user-defined menu.

Optional Menu Items

A dynamic user-defined menu item *disappears* from the menu if the menu item string expression evaluates to ""; the remainder of the menu definition line is then ignored. This makes possible a variable number of items in a user-defined menu list. This example adds a menu listing the names of up to 8 waves in the current data folder. If the current data folder contains less than 8 waves, then only those that exist are shown in the menu:

```
Menu "Waves", dynamic
   WaveName("",0,4), DoSomething($WaveName("",0,4))
   WaveName("",1,4), DoSomething($WaveName("",1,4))
   WaveName("",2,4), DoSomething($WaveName("",2,4))
   WaveName("",3,4), DoSomething($WaveName("",3,4))
   WaveName("",4,4), DoSomething($WaveName("",4,4))
   WaveName("",5,4), DoSomething($WaveName("",5,4))
   WaveName("",6,4), DoSomething($WaveName("",6,4))
   WaveName("",7,4), DoSomething($WaveName("",7,4))
End

Function DoSomething(w)
   Wave/Z w

   if( WaveExists(w) )
      Print "DoSomething: wave's name is "+NameOfWave(w)
   endif
End
```

This works because WaveName returns "" if the indexed wave doesn’t exist.

Note that each potential item must have a menu definition line that either appears or disappears.

Multiple Menu Items

A menu item string that contains a semicolon-separated “string list” (see **StringFromList** on page V-998) generates a menu item for each item in the list. For example:

```
Menu "Multi-Menu"
   "first item;second item;", DoItem()
End
```

Multi-Menu is a two-item menu. When either item is selected the DoItem procedure is called.

This begs the question: How does the DoItem procedure know which item was selected? The answer is that DoItem must call the **GetLastUserMenuInfo** operation (see page V-306) and examine the appropriate returned variables, usually V_value (the selected item’s one-based item number) or S_Value (the selected item’s text).

The string list can be dynamic, too. The above "Waves" example can be rewritten to handle an arbitrary number of waves using this definition:

```
Menu "Waves", dynamic
   WaveList("*",";",""), DoItem()
End

Function DoItem()
   GetLastUserMenuInfo           // Sets S_value, V_value, etc.
   WAVE/Z w= $S_value
   if( WaveExists(w) )
      Print "The wave's name is "+NameOfWave(w)
   endif
End
```

Consolidating Menu Items Into a Submenu

It is common to have many utility procedure files open at the same time. Each procedure file could add menu items which would clutter Igor’s menus. When you create a utility procedure file that adds multiple menu items, it is usually a good idea to consolidate all of the menu items into one submenu. Here is an example.

Let’s say we have a procedure file full of utilities for doing frequency-domain data analysis and that it contains the following:

```
Function ParzenDialog()
   …
End

Function WelchDialog()
   …
End

Function KaiserDialog()
   …
End
```

We can consolidate all of the menu items into a single submenu in the Analysis menu:

```
Menu "Analysis"
   Submenu "Windows"
      "Parzen…", ParzenDialog()
      "Welch…", WelchDialog()
      "Kaiser…", KaiserDialog()
   End
End
```

Specialized Menu Item Definitions

A menu item string that contains certain special values adds a specialized menu such as a color menu.

Only one specialized menu item string is allowed in each menu or submenu, it must be the first item, and it must be the only item.

Menu Item String	Result
"*CHARACTER*	"Character menu, no character is initially selected, font is Geneva, font size is 12.
"*CHARACTER*(Arial)	Character menu shown using Arial font at default size.
"*CHARACTER*(Arial,36)	"Character menu of Arial font in 36 point size.
"*CHARACTER*(,36)	"Character menu of Geneva font in 36 point size.
"*CHARACTER*(Arial,36,m)	"Character menu of Arial font in 36 point size, initial character is m.
"*COLORTABLEPOP*	"Color table menu, initial table is Grays.
"*COLORTABLEPOP*(YellowHot)	"Color table menu, initial table is YellowHot. See **CTabList** on page V-118 for a list of color tables.
"*COLORTABLEPOP*(YellowHot,1)	"Color table menu with the colors drawn reversed.
"*COLORPOP*	"Color menu, initial color is black.
"*COLORPOP*(0,65535,0)	"Color menu, initial color is green.
"*COLORPOP*(0,65535,0,49151)	"Color menu, initial color is green at 75% opacity.
"*FONT*	"Font menu, no font is initially selected, does not include “default” as a font choice.
"*FONT*(Arial)	"Font menu, Arial is initially selected.
"*FONT*(Arial,default)	"Font menu with Arial initially selected and including “default” as a font choice.
"*LINESTYLEPOP*	"Line style menu, no line style is initially selected.
"*LINESTYLEPOP*(3)	"Line style menu, initial line style is style=3 (coarse dashed line).
"*MARKERPOP*	"Marker menu, no marker is initially selected.
"*MARKERPOP*(8)	"Marker menu, initial marker is 8 (empty circle).
"*PATTERNPOP*	"Pattern menu, no pattern is initially selected.
"*PATTERNPOP*(1)	"Pattern menu, initial pattern is 1 (SW-NE light diagonal).

To retrieve the selected color, line style, etc., the execution text must be a procedure that calls the **GetLastUserMenuInfo** operation (see page V-306). Here’s an example of a color submenu implementation:

```
Menu "Main", dynamic
   "First Item", /Q, Print "First Item"
   Submenu "Color"
      CurrentColor(), /Q, SetSelectedColor() // Must be first submenu item
      // No items allowed here
   End
End

Function InitializeColors()
   NVAR/Z red= root:red
   if(!NVAR_Exists(red))
      Variable/G root:red=65535, root:green=0, root:blue=1, root:alpha=65535
   endif
End

Function/S CurrentColor()
   InitializeColors()
   NVAR red = root:red
   NVAR green = root:green
   NVAR blue = root:blue
   NVAR alpha = root:alpha
   String menuText
   sprintf menuText, "*COLORPOP*(%d,%d,%d,%d)", red, green, blue, alpha
   return menuText
End

Function SetSelectedColor()
   GetLastUserMenuInfo// Sets V_Red, V_Green, V_Blue, V_Alpha, S_value, V_value
   NVAR red = root:red
   NVAR green = root:green
   NVAR blue = root:blue
   NVAR alpha = root:alpha
   red = V_Red
   green = V_Green
   blue = V_Blue
   alpha = V_Alpha

   Make/O/N=(2,2,4) root:colorSpot
   Wave colorSpot = root:colorSpot
   colorSpot[][][0] = V_Red
   colorSpot[][][1] = V_Green
   colorSpot[][][2] = V_Blue
   colorSpot[][][3] = V_Alpha

   CheckDisplayed/A colorSpot
   if (V_Flag == 0)
      NewImage colorSpot
   endif
End
```

Special Characters in Menu Item Strings

You can control some aspects of a menu item using special characters. These special characters are based on the behavior of the Macintosh menu manager and are only partially supported on Windows (see **Special Menu Characters on Windows** on page IV-134). They affect user-defined menus in the main menu bar. On Macintosh, but not on Windows, they also affect user-defined pop-up menus in control panels, graphs and simple input dialogs.

By default, special character interpretation is enabled in user-defined menu bar menus and is disabled in user-defined control panel, graph and simple input dialog pop-up menus. This is almost always what you would want. In some cases, you might want to override the default behavior. This is discussed under **Enabling and Disabling Special Character Interpretation** on page IV-135.

This table shows the special characters and their effect if special character interpretation is enabled. See **Special Menu Characters on Windows** on page IV-134 for Windows-specific considerations.

Character	Behavior
/	Creates a keyboard shortcut for the menu item.
	The character after the slash defines the item's keyboard shortcut. For example, `"Low Pass/1"` makes the item "Low Pass" with a keyboard shortcut for Command-1 (*Macintosh*) or Ctrl-1 (*Windows*). You can also use function keys. To avoid conflicts with Igor, use the numeric keys and the function keys only. See **Keyboard Shortcuts** on page IV-136 and **Function Keys** on page IV-136 for further details.
	Keyboard shortcuts are not supported in the graph marquee and layout marquee menus.
-	Creates a divider between menu items.
	If a hyphen (minus sign) is the first character in the item then the item will be a disabled divider. This can be a problem when trying to put negative numbers in a menu. Use a leading space character to prevent this. The string "(-" also disables the corresponding divider.
(	Disables the menu item.
	If the first character of the item text is a left parenthesis then the item will be disabled.
!	Adds a mark to the menu item.
	If an exclamation point appears in the item, any character after the exclamation point adds a checkmark to the left of the menu item. For example:
	`"Low Pass!*"`
	makes an item "Low Pass" with an checkmark to the left.
	For compatibility with Igor6, use:
	`"Low Pass!" + num2char(18)`
;	Separates one menu item from the next.
	Example: `"Item 1;Item 2"`

Whereas it is standard practice to use a semicolon to separate items in a pop-up menu in a control panel, graph or simple input dialog, you should avoid using the semicolon in user-defined main-menu-bar menus. It is clearer if you use one item per line. It is also necessary in some cases (see **Menu Definition Syntax** on page IV-126).

If special character interpretation is disabled, these characters will appear in the menu item instead of having their special effect. The semicolon character is treated as a separator of menu items even if special character interpretation is disabled.

Special Menu Characters on Windows

On Windows, these characters are treated as special in menu bar menus but not in pop-up menus in graphs, control panels, and simple input dialogs. The following table shows which special characters are supported.

Character	Meaning
/	Defines accelerator
-	Divider
(	Disables item
!	Adds mark to item
;	Separates items

In general, Windows does not allow using Ctrl+<punctuation> as an accelerator. Therefore, in the following example, the accelerator will not do anything:

```
Menu "Macros"
   "Test/["       // "/[" will not work on Windows.
End
```

On Windows, you can designate a character in a menu item as a mnemonic keystroke by preceding the character with an ampersand:

```
Menu "Macros"
   "&Test", Print "This is a test"
End
```

This designates “T” as the mnemonic keystroke for Test. To invoke this menu item, press Alt and release it, press the “M” key to highlight the Macros menu, and press the “T” key to invoke the Test item. If you hold the Alt key pressed while pressing the “M” and “T” keys and if the active window is a procedure window, Igor will not execute the item’s command but rather will bring up the procedure window and display the menu definition. This is a programmer’s shortcut.

Note: The mnemonic keystroke is not supported on Macintosh. For this reason, if you care about cross-platform compatibility, you should not use ampersands in your menu items.

On Macintosh, if you include a single ampersand in a menu item, it does not appear in the menu item. If you use a double ampersand, it appears as a single ampersand.

Enabling and Disabling Special Character Interpretation

The interpretation of special characters in menu items can sometimes get in the way. For example, you may want a menu item to say “m/s” or “A<B”. With special character interpretation enabled, the first of these would become “m” with “s” as the keyboard shortcut and the second would become “A” in a bold typeface.

Igor provides WaveMetrics-defined escape sequences that allow you to override the default state of special character interpretation. These escape sequences are case sensitive and must appear at the very start of the menu item:

Escape Code	Effect	Example
`"\\M0"`	Turns special character interpretation off.	`"\\M0m/s"`
`"\\M1"`	Turns special character interpretation on.	`"\\M1m/s"`

The most common use for this will be in a user-defined menu bar menu in which the default state is on and you want to display a special character in the menu item text itself. That is what you can do with the “`\\M0`” escape sequence.

Another possible use on Macintosh is to create a disabled menu item in a control panel, graph or simple input dialog pop-up menu. The default state of special character interpretation in pop-up menus is off. To disable the item, you either need to turn it on, using “\\M1” or to use the technique described in the next paragraph.

What if we want to include a special character in the menu item itself *and* have a keyboard shortcut for that item? The first desire requires that we turn special character interpretation off and the second requires that we turn it on. The WaveMetrics-defined escape sequence can be extended to handle this. For example:

```
"\\M0:/1:m/s"
```

The initial “\\M0” turns normal special character interpretation off. The first colon specifies that one or more special characters are coming. The /1 makes Command-1 (*Macintosh*) or Ctrl+1 (*Windows*) the keyboard shortcut for this item. The second colon marks the end of the menu commands and starts the regular menu text which is displayed in the menu without special character interpretation. The final result is as shown above.

Any of the special characters can appear between the first colon and the second. For example:

```
Menu "Macros"
   "\\M0:/1:(Cmd-1 on Macintosh, Ctrl+1 on Windows)"
   "\\M0:(:(Disabled item)"
   "\\M0:!*:(Checked)"
   "\\M0:/2!*:(Cmd-2 on Macintosh, Ctrl+2 on Windows and checked)"
End
```

The \\M escape code affects just the menu item currently being defined. In the following example, special character interpretation is enabled for the first item but not for the second:

```
"\\M1(First item;(Second item"
```

To enable special character interpretation for both items, we need to write:

```
"\\M1(First item;\\M1(Second item"
```

Keyboard Shortcuts

A keyboard shortcut is a set of one or more keys which invoke a menu item. In a menu item string, a keyboard shortcut is introduced by the / special character. For example:

```
Menu "Macros"
   "Test/1"       // The keyboard shortcut is Cmd-1 (Macintosh)
End               // or Ctrl+1 (Windows).
```

All of the plain alphabetic keyboard shortcuts (/A through /Z) are used by Igor.

Numeric keyboard shortcuts (/0 through /9) are available for use in user menu definitions as are Function Keys, described below.

You can define a numeric keyboard shortcut that includes one or more modifier keys. The modifier keys are:

Macintosh:	Shift (S), Option (O), Control (L)
Windows:	Shift (S), Alt (O), Meta (L) (Meta is often called the “Windows key”.

For example:

```
Menu "Macros"
   "Test/1"       // Cmd-1, Ctrl+1
   "Test/S1"      // Shift-Cmd-1, Ctrl+Shift+1.
   "Test/O1"      // Option-Cmd-1, Ctrl+Alt+1
   "Test/OS1"     // Option-Shift-Cmd-1, Ctrl+Shift+Alt+1
End
```

Function Keys

Most keyboards have function keys labeled F1 through F12. In Igor, you can treat a function key as a keyboard shortcut that invokes a menu item.

Note: Mac OS X reserves nearly all function keys for itself. In order to use function keys for an application, you must check a checkbox in the Keyboard control panel. Even then the OS will intercept some function keys.

Note: On Windows, Igor uses F1 for help-related operations. F1 will not work as a keyboard shortcut on Windows. Also, the Windows OS reserves Ctrl-F4 and Ctrl-F6 for closing and reordering windows. On some systems, F12 is reserved for debugging.

Here is a simple function key example:

```
Menu "Macros"
   "Test/F5"      // The keyboard shortcut is F5.
End
```

As with other keyboard shortcuts, you can specify that one or more modifier keys must be pressed along with the function key. By including the “C” modifier character, you can specify that Command (*Macintosh*) or Ctrl (*Windows*) must also be pressed:

```
Menu "Macros"
   // Function keys without modifiers
   "Test/F5"      // F5

   // Function keys with Shift and/or Option/Alt modifiers
   "Test/SF5"     // Shift-F5 (Macintosh), Shift+F5 (Windows)
   "Test/OF5"     // Option-F5, Alt+F5
   "Test/SOF5"    // Shift-Option-F5, Shift+Alt+F5

   // Function keys with Command (Macintosh) or Ctrl (Windows) modifiers
   "Test/CF5"     // Cmd-F5, Ctrl+F5
   "Test/SCF5"    // Shift-Cmd-F5, Shift-Ctrl-F5
   "Test/OCF5"    // Option-Cmd-F5, Alt-Ctrl-F5
   "Test/OSCF5"   // Option-Shift-Cmd-F5, Alt-Shift-Ctrl-F5

   // Function keys with Ctrl (Macintosh) or Meta (Windows) modifiers
   "Test/LOSCF5"  // Control-Option-Shift-Cmd-F5, Meta+Alt+Shift+Ctrl+F5
End
```

Although some keyboards have function keys labeled F13 and higher, they do not behave consistently and are not supported.

Marquee Menus

Igor has two menus called “marquee menus”. In graphs and page layouts you create a marquee when you drag diagonally. Igor displays a dashed-line box indicating the area of the graph or layout that you have selected. If you click inside the marquee, you get a marquee menu.

You can add your own menu items to a marquee menu by creating a GraphMarquee or LayoutMarquee menu definition. For example:

```
Menu "GraphMarquee"
   "Print Marquee Coordinates", GetMarquee bottom; Print V_left, V_right
End
```

The use of keyboard shortcuts is not supported in marquee menus.

See **Marquee Menu as Input Device** on page IV-163 for details.

Trace Menus

Igor has two “trace” menus named “TracePopup” and “AllTracesPopup”. When you control-click or right-click in a graph on a trace you get the TracesPopup menu. If you press Shift while clicking, you get the AllTracesPopup (standard menu items in that menu operated on all the traces in the graph). You can append menu items to these menus with Menu “TracePopup” and Menu “AllTracesPopup” definitions.

For example, this code adds an Identify Trace item to the TracePopup contextual menu but not to the AllTracesPopup menu:

```
Menu "TracePopup"
   "Identify Trace", /Q, IdentifyTrace()
End

Function IdentifyTrace()
   GetLastUserMenuInfo
   Print S_graphName, S_traceName
End
```

GraphPopup Menu

Igor has a contextual menu named "GraphPopup". When you control-click or right-click in a graph away from any trace while in operate mode, you get the GraphPopup menu. You can append menu items to this menu with a GraphPopup menu definition.

For example, the following code adds an "Identify Graph" item to the GraphPopup contextual menu:

```
Menu "GraphPopup"
   "Identify Graph", /Q, IdentifyGraph()
End

Function IdentifyGraph()
   Print WinName(0,1)
End
```

TablePopup Menu

Igor Pro 9.00 and later have a contextual menu named "TablePopup". When you right-click in a table, Igor displays the TablePopup menu. You can append menu items to this menu with a TablePopup menu definition.

For example, the following code adds a "Print Path To Wave" item to the TablePopup contextual menu:

```
Menu "TablePopup", dynamic
   ContextualTableMenuItem(), /Q, PrintPathToSelWave()
End

Function/S ContextualTableMenuItem()
   GetLastUserMenuInfo
   WAVE/Z selWave = $S_firstColumnPath
   if (WaveExists(selWave))
      return "Print Path To Wave"
   endif

   // Here if the user clicked the Point column or an unused column
   return ""         // No menu item is added
End

Function PrintPathToSelWave()
   GetLastUserMenuInfo
   WAVE/Z selWave = $S_firstColumnPath
   if (WaveExists(selWave))
      Print GetWavesDataFolder(selWave,2)
   endif
End
```

DataBrowserObjectsPopup Menu

Igor Pro 9.00 and later have a contextual menu named "DataBrowserObjectsPopup". When you right click the list of objects in the Data Browser, Igor displays the DataBrowserObjectsPopup menu.

You can append menu items to this menu with a DataBrowserObjectsPopup menu definition. If necessary, you can use **GetLastUserMenuInfo** to get information about which menu was selected by the user and **GetBrowserSelection** to determine which objects, if any, are currently selected in the Data Browser.

For example, the following code adds a menu item to the DataBrowserObjectsPopup contextual menu if two numeric waves are selected. The text of the menu item is different depending on whether or not the shift key is pressed when the menu is shown.

```
Menu "DataBrowserObjectsPopup", dynamic
   // This menu item is displayed if the shift key is not pressed
   Display1vs2MenuItemString(0), /Q, DisplayWave1vsWave2(0)

   // This menu item is displayed if the shift key is pressed
   Display1vs2MenuItemString(1), /Q, DisplayWave1vsWave2(1)
End

// If at least two items are selected in the Data Browser object list and the
// first two selected items are numeric waves, this function returns the first
// selected wave via w1 and the second selected wave via w1 unless reverse
// is non-zero in which case the waves are reversed.
// The function result is 1 if the first two selected objects are numeric waves
// and 0 otherwise.
static Function GetWave1AndWave2(WAVE/Z &w1, WAVE/Z &w2, int reverse)
   if (strlen(GetBrowserSelection(-1)) == 0)
      return 0       // Data Browser is not open
   endif

   WAVE/Z w1 = $(GetBrowserSelection(reverse ? 1 : 0))// May be null
   WAVE/Z w2 = $(GetBrowserSelection(reverse ? 0 : 1))// May be null

   if (!WaveExists(w1) || !WaveExists(w2))
      return 0       // Fewer than two waves are selected
   endif

   if (WaveType(w1,1)!=1 || WaveType(w2,1)!=1)
      return 0       // Waves are not numeric
   endif

   return 1
End

Function/S Display1vs2MenuItemString(reverse)
   int reverse    // True (1) if caller wants the reverse menu item string

   int shiftKeyPressed = GetKeyState(0) & 4  // User is asking for reverse?
   if (shiftKeyPressed && !reverse)
      // User is asking for reverse so hide unreversed menu item
      return ""
   endif
   if (!shiftKeyPressed && reverse)
      // User is not asking for reverse so hide reversed menu item
      return ""
   endif

   WAVE/Z w1, w2
   int twoNumericWavesSelected = GetWave1AndWave2(w1, w2, reverse)
   if (!twoNumericWavesSelected)
      return ""
   endif

   String menuText
   sprintf menuText, "Display %s vs %s", NameOfWave(w1), NameOfWave(w2)
   return menuText
End

// If reverse is false, execute Display w1 vs w2
// If reverse is true, execute Display w2 vs w1
Function DisplayWave1vsWave2(int reverse)
   WAVE/Z w1, w2
   int twoNumericWavesSelected = GetWave1AndWave2(w1, w2, reverse)
   if (twoNumericWavesSelected)
      Display w1 vs w2
   endif
End
```

WindowBrowserWindowsPopup Menu

Igor Pro 9.00 and later have a contextual menu named "WindowBrowserWindowsPopup". When you right click the list of windows in the Window Browser, Igor displays the WindowBrowserWindowsPopup menu.

You can add items to this menu with a WindowBrowserWindowsPopup menu definition. If necessary, you can use **GetLastUserMenuInfo** to get information about which menu was selected by the user and **GetWindowBrowserSelection** to determine which windows, if any, are currently selected in the Window Browser.

For example, the following code adds a menu item to the WindowBrowserWindowsPopup contextual menu if the Window Browser has one graph window selected. The text of the menu item is different depending on whether or not a modifier key is pressed when the menu is shown. Clicking on the menu item copies the selected graph to the clipboard as either PNG or SVG format, depending on whether or not a modifier key was pressed.

```
Menu "WindowBrowserWindowsPopup", dynamic
   // If a single graph is selected in the Window Browser, if the shift
   // key is pressed, this adds "Copy <graph> as SVG".
   // If a single graph is selected in the Window Browser, if the shift
   // key is not pressed, this adds "Copy <graph> as PNG".
   // Otherwise it adds no menu items.
   CopySelectedGraphToClipMenuString(0),/Q,CopySelectedGraphToClip(0)
   CopySelectedGraphToClipMenuString(1),/Q,CopySelectedGraphToClip(1)
End

// If the Window Browser is open, and exactly one item is selected, and that
// item is a graph, the name of the graph is returned. Otherwise an empty string
// is returned.
Function/S GetSingleSelectedGraphName()
   Variable dummy

   // GetWindowBrowserSelection generates an error if the Window Browser is not
   // open. If the Debugger's Stop on Error setting is enabled, the error causes
   // the debugger to break on this statement. Calling GetRTError(1) on the same
   // line clears the error and prevents the debugger from breaking here.
   String selectedList = GetWindowBrowserSelection(0); dummy = GetRTError(1)

   if (ItemsInList(selectedList,";") == 1)
      String name = StringFromList(0, selectedList, ";")
      if (WinType(name) == 1)
         return name          // It is a graph
      endif
   endif

   return ""                  // Nothing selected or not a graph
End

// If doSVG is false and the shift key is not key pressed, return
// "Copy Graph as PNG". If doSVG is true and the shift key is pressed,
// return "Copy Graph as SVG".
Function/S CopySelectedGraphToClipMenuString(Variable doSVG)
   int shiftKeyPressed = (GetKeyState(0) & 4) != 0
   if (doSVG && !shiftKeyPressed)
      return ""   // Caller wants SVG menu item but shift key is not pressed
   endif
   if (!doSVG && shiftKeyPressed)
      return ""   // Caller wants PNG menu item but shift key is pressed
   endif

   String selectedGraphName = GetSingleSelectedGraphName()
   if (strlen(selectedGraphName) == 0)
      return ""   // The selection is other than a single graph window
   endif

   String format = "PNG"
   if (doSVG)
      format = "SVG"
   endif
   String menuText = ""
   sprintf menuText, "Copy %s as %s", selectedGraphName, format
   return menuText
End

// If doSVG is true, copy the graph as SVG. Otherwise copy it as PNG.
Function CopySelectedGraphToClip(Variable doSVG)
   String selectedGraphName = GetSingleSelectedGraphName()
   if (strlen(selectedGraphName) > 0)
      Variable formatCode = -5   // PNG
      if (doSVG)
         formatCode = -9         // SVG
      endif
      SavePict/E=(formatCode)/WIN=$(selectedGraphName) as "Clipboard"
   endif
End
```

Popup Contextual Menus

You can create a custom pop-up contextual menu to respond to a control-click or right-click. For an example, see **Creating a Contextual Menu** on page IV-162.

Chapter

IV-6

Interacting with the User

Overview

The following sections describe the various programming techniques available for getting input from and for interacting with a user during the execution of your procedures. These techniques include:

- The simple input dialog
- Control panels
- Cursors
- Marquee menus

The simple input dialog provides a bare bones but functional user interfaces with just a little programming. In situations where more elegance is required, control panels provide a better solution.

Modal and Modeless User Interface Techniques

Before the rise of the graphical user interface, computer programs worked something like this:

1. The program prompts the user for input.
2. The user enters the input.
3. The program does some processing.
4. Return to step 1.

In this model, the program is in charge and the user must respond with specific input at specific points of program execution. This is called a “modal” user interface because the program has one mode in which it will only accept specific input and another mode in which it will only do processing.

The Macintosh changed all this with the idea of event-driven programming. In this model, the computer waits for an event such as a mouse click or a key press and then acts on that event. The user is in charge and the program responds. This is called a “modeless” user interface because the program will accept any user action at any time.

You can use both techniques in Igor. Your program can put up a modal dialog asking for input and then do its processing or you can use control panels to build a sophisticated modeless event-driven system.

Event-driven programming is quite a bit more work than dialog-driven programming. You have to be able to handle user actions in any order rather than progressing through a predefined sequence of steps. In real life, a combination of these two methods is often used.

The Simple Input Dialog

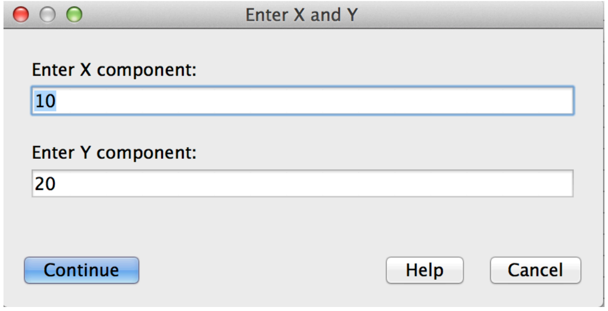
The simple input dialog is a way by which a function can get input from the user in a modal fashion. It is very simple to program and is also simple in appearance.

A simple input dialog is presented to the user when a DoPrompt statement is executed in a function. Parameters to DoPrompt specify the title for the dialog and a list of local variables. For each variable, you must include a Prompt statement that provides the text label for the variable.

Generally, the simple input dialog is used in conjunction with routines that run when the user chooses an item from a menu. This is illustrated in the following example which you can type into the procedure window of a new experiment:

```
Menu "Macros"
   "Calculate Diagonal...", CalcDiagDialog()
End

Function CalcDiagDialog()
   Variable x=10,y=20
   Prompt x, "Enter X component: "  // Set prompt for x param
   Prompt y, "Enter Y component: "  // Set prompt for y param
   DoPrompt "Enter X and Y", x, y
   if (V_Flag)
      return -1                     // User canceled
   endif

   Print "Diagonal=",sqrt(x^2+y^2)
End
```

If you run the CalcDiagDialog function, you see the following dialog:



Enter X and Y

Enter X component: 10

Enter Y component: 20

If the user presses Continue without changing the default values, “Diagonal= 22.3607” is printed in the history area of the command window. If the user presses Cancel, nothing is printed because DoPrompt sets the V_Flag variable to 1 in this case.

The simple input dialog allows for up to 10 numeric or string variables. When more than 5 items are used, the dialog uses two columns and you may have to limit the length of your Prompt text.

The simple input dialog is unique in that you can enter not only literal numbers or strings but also numeric expressions or string expressions. Any literal strings that you enter must be quoted.

If the user presses the Help button, Igor searches for a help topic with a name derived from the dialog title. If such a help topic is not found, then generic help about the simple input dialog is presented. In both cases, the input dialog remains until the user presses either Continue or Cancel.

Pop-Up Menus in Simple Dialogs

The simple input dialog supports pop-up menus as well as text items. The pop-up menus can contain an arbitrary list of items such as a list of wave names. To use a pop-up menu in place of the normal text entry item in the dialog, you use the following syntax in the prompt declaration:

```
Prompt <variable name>, <title string>, popup <menu item list>
```

The popup keyword indicates that you want a pop-up menu instead of the normal text entry item. The menu list specifies the items in the pop-up menu separated by semicolons. For example:

```
Prompt color, "Select Color", popup "red;green;blue;"
```

If the menu item list is too long to fit on one line, you can compose the list in a string variable like so:

```
String stmp= "red;green;blue;"
stmp += "yellow;purple"
Prompt color, "Select Color", popup stmp
```

The pop-up menu items support the same special characters as the user-defined menu definition (see **Special Characters in Menu Item Strings** on page IV-133) except that items in pop-up menus are limited to 50 characters, keyboard shortcuts are not supported, and special characters are disabled by default.

You can use pop-up menus with both numeric and string parameters. When used with numeric parameters the number of the item chosen is placed in the variable. Numbering starts from one. When used with string parameters the text of the chosen item is placed in the string variable.

There are a number of functions, such as the **WaveList** function (see page V-1075) and the **TraceNameList** function (see page V-1044), that are useful in creating pop-up menus.

To obtain a menu item list of all waves in the current data folder, use:

```
WaveList("*", ";", "")
```

To obtain a menu item list of all waves in the current data folder whose names end in “_X”, use:

```
WaveList("*_X", ";", "")
```

To obtain a menu item list of all traces in the top graph, use:

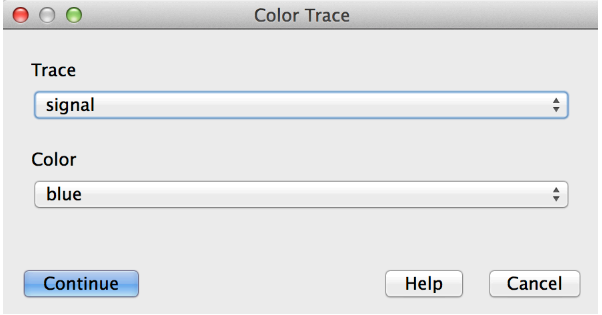
```
TraceNameList("", ";", 1)
```

For a list of all contours in the top graph, use ContourNameList. For a list of all images, use ImageNameList. For a list of waves in a table, use WaveList.

This next example creates two pop-up menus in the simple input dialog.

```
Menu "Macros"
   "Color Trace...", ColorTraceDialog()
End

Function ColorTraceDialog()
   String traceName
   Variable color=3
   Prompt traceName,"Trace",popup,TraceNameList("",";",1)
   Prompt color,"Color",popup,"red;green;blue"
   DoPrompt "Color Trace",traceName,color
   if( V_Flag )
      return 0          // user canceled
   endif

   if (color == 1)
      ModifyGraph rgb($traceName)=(65000, 0, 0)
   elseif(color == 2)
      ModifyGraph rgb($traceName)=(0, 65000, 0)
   elseif(color == 3)
      ModifyGraph rgb($traceName)=(0, 0, 65000)
   endif
End
```

If you choose Color Trace from the Macros menu, Igor displays the simple input dialog with two pop-up menus. The first menu contains a list of all traces in the target window which is assumed to be a graph. The second menu contains the items red, green and blue with blue (item number 3) initially chosen.


Color Trace

Trace: signal

Color: blue

After you choose the desired trace and color from the pop-up menus and click the Continue button, the function continues execution. The string parameter traceName will contain the name of the trace chosen from the first pop-up menu. The numeric parameter color will have a value of 1, 2 or 3, corresponding to red, green and blue.

In the preceding example, we needed a trace name to pass to the ModifyGraph operation. In another common situation, we need a wave reference to operate on. For example:

```
Menu "Macros"
   "Smooth Wave In Graph...",SmoothWaveInGraphDialog()
End

Function SmoothWaveInGraphDialog()
   String traceName
   Prompt traceName,"Wave",popup,TraceNameList("",";",1)
   DoPrompt "Smooth Wave In Graph",traceName

   WAVE w = TraceNameToWaveRef("", traceName)
   Smooth 5, w
End
```

The traceName parameter alone is not sufficient to specify which wave we want to smooth because it does not identify in which data folder the wave resides. The TraceNameToWaveRef function returns a wave reference which solves this problem.

Saving Parameters for Reuse

It is possible to write a procedure that presents a simple input dialog with default values for the parameters saved from the last time it was invoked. To accomplish this, we use global variables to store the values between calls to the procedure. Here is an example that saves one numeric and one string variable.

```
Function TestDialog()
   String saveDF = GetDataFolder(1)
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S :TestDialog

   Variable num = NumVarOrDefault("gNum", 42)
   Prompt num, "Enter a number"
   String str = StrVarOrDefault("gStr", "Hello")
   Prompt str, "Enter a string"
   DoPrompt "test",num,str

   Variable/G gNum = num      // Save for next time
   String/G gStr = str

   // Put function body here
   Print num,str

   SetDataFolder saveDF
End
```

This example illustrates the NumVarOrDefault and StrVarOrDefault functions. These functions return the value of a global variable or a default value if the global variable does not exist. 42 is the default value for gNum. NumVarOrDefault returns 42 if gNum does not exist. If gNum does exist, it returns the value of gNum. Similarly, “Hello” is the default value for gStr. StrVarOrDefault returns “Hello” if gStr does not exist. If gStr does exist, it returns the value of gStr.

Multiple Simple Input Dialogs

Prompt statements can be located anywhere within the body of a function and they do not need to be grouped together, although it will aid code readability if associated Prompt and DoPrompt code is kept together. Functions may contain multiple DoPrompt statements, and Prompt statements can be reused or redefined.

The following example illustrates multiple simple input dialogs and prompt reuse:

```
Function Example()
   Variable a= 123
   Variable/C ca= cmplx(3,4)
   String s

   Prompt a,"Enter a value"
   Prompt ca,"Enter complex value"
   Prompt s,"Enter a string", popup "red;green;blue"
   DoPrompt "Enter Values",a,s,ca
   if(V_Flag)
      Abort "The user pressed Cancel"
   endif

   Print "a= ",a,"s= ",s,"ca=",ca

   Prompt a,"Enter a again please"
   Prompt s,"Type a string"
   DoPrompt "Enter Values Again", a,s

   if(V_Flag)
      Abort "The user pressed Cancel"
   endif

   Print "Now a=",a," and s=",s
End
```

When this function is executed, it produces two simple input dialogs, one after the other after the user clicks Continue.

Help For Simple Input Dialogs

You can create, for each simple input dialog, custom help that appears when the user clicks the Help button. You do so by providing a custom help file with topics that correspond to the titles of your dialogs as specified in the DoPrompt commands.

If there is no exactly matching help topic or subtopic for a given dialog title, Igor munges the presumptive topic by replacing underscore characters with spaces and inserting spaces before capital letters in the interior of the topic. For example, if the dialog title is “ReallyCoolFunction”, and there is no matching help topic or subtopic, Igor looks for a help topic or subtopic named “Really Cool Function”.

See **Creating Your Own Help File** on page IV-255 for information on creating custom help files.

Displaying an Open File Dialog

You can display an Open File dialog to allow the user to choose a file to be used with a subsequent command. For example, the user can choose a file which you will then use in a **LoadWave** command. The Open File dialog is displayed using an **Open**/D/R command. Here is an example:

```
Function/S DoOpenFileDialog()
   Variable refNum
   String message = "Select a file"
   String outputPath
   String fileFilters = "Data Files (*.txt,*.dat,*.csv):.txt,.dat,.csv;"
   fileFilters += "All Files:.*;"

   Open /D /R /F=fileFilters /M=message refNum
   outputPath = S_fileName

   return outputPath    // Will be empty if user canceled
End
```

Here the Open operation does not actually open a file but instead displays an Open File dialog. If the user chooses a file and clicks the Open button, the Open operation returns the full path to the file in the S_fileName output string variable. If the user cancels, Open sets S_fileName to "".

The /M flag is used to set the prompt message. As of OS X 10.11, Apple no longer shows the prompt message in the Open File dialog. It continues to work on Windows.

The /F flag is used to control the file filter which determines what kinds of files the user can select. This is explained further under **Open File Dialog File Filters**.

Displaying a Multi-Selection Open File Dialog

You can display an Open File dialog to allow the user to choose multiple files to be used with subsequent commands. The multi-selection Open File dialog is displayed using an **Open**/D/R/MULT=1 command. The list of files selected is returned via S_fileName in the form of a carriage-return-delimited list of full paths.

Here is an example:

```
Function/S DoOpenMultiFileDialog()
   Variable refNum
   String message = "Select one or more files"
   String outputPaths
   String fileFilters = "Data Files (*.txt,*.dat,*.csv):.txt,.dat,.csv;"
   fileFilters += "All Files:.*;"

   Open /D /R /MULT=1 /F=fileFilters /M=message refNum
   outputPaths = S_fileName

   if (strlen(outputPaths) == 0)
      Print "Cancelled"
   else
      Variable numFilesSelected = ItemsInList(outputPaths, "\r")
      Variable i
      for(i=0; i<numFilesSelected; i+=1)
         String path = StringFromList(i, outputPaths, "\r")
         Printf "%d: %s\r", i, path
      endfor
   endif

   return outputPaths   // Will be empty if user canceled
End
```

Here the Open operation does not actually open a file but instead displays an Open File dialog. Because /MULT=1 was used, if the user chooses one or more files and clicks the Open button, the Open operation returns the list of full paths to files in the S_fileName output string variable. If the user cancels, Open sets S_fileName to "".

The list of full paths is delimited with a carriage return character, represented by "\r" in the example above. We use carriage return as the delimiter because the customary delimiter, semicolon, is a legal character in a Macintosh file name.

The /M flag is used to set the prompt message. As of OS X 10.11, Apple no longer shows the prompt message in the Open File dialog. It continues to work on Windows.

The /F flag is used to control the file filter which determines what kinds of files the user can select. This is explained further under **Open File Dialog File Filters**.

Open File Dialog File Filters

The **Open** operation displays the open file dialog if you use the /D/R flags or if the file to be opened is not fully specified using the pathName and fileNameStr parameters. The Open File dialog includes a file filter menu that allows the user to choose the type of file to be opened. By default this menus contain "Plain Text Files" and "All Files". You can use the /T and /F flags to override the default filter behavior.

The /T flag uses obsolescent Macintosh file types or file name extensions consisting of a dot plus three characters. The /F flag, added in Igor Pro 6.10, supports file name extensions only (not Macintosh file types) and extensions can be from one to 31 characters. Procedures written for Igor Pro 6.10 or later should use the /F flag in most cases but can use /T or both /T and /F. Procedures that must run with Igor Pro 6.0x and earlier must use the /T flag.

Using the /T=typeStr flag, you specify acceptable Macintosh-style file types represented by four-character codes (e.g., "TEXT") or acceptable three-character file name extensions (e.g., ".txt"). The pattern "????" means "any type of file" and is represented by "All Files" in the filter menu.

typeStr may contain multiple file types or extensions (e.g., "TEXTEPSF????" or ".txt.eps????"). Each file type or extension must be exactly four characters in length. Consequently the /T flag can accommodate only three-character file name extensions. Each file type or extension creates one entry in the Open File dialog filter menu.

If you use the /T flag, the Open operation automatically adds a filter for All Files ("????") if you do not add one explicitly.

Igor maps Macintosh file types to extensions. For example, if you specify /T="TEXT", you can open files with the extension ".txt" as well as any file whose Macintosh file type property is 'TEXT'. Igor does similar mappings for other extensions. See **File Types and Extensions** on page III-455 for details.

Using the /F=fileFilterStr flag, you specify a filter menu string plus acceptable file name extensions for each filter. fileFilterStr specifies one or more filters in a semicolon-separated list. For example, this specifies three filters:

```
String fileFilters = "Data Files (*.txt,*.dat,*.csv):.txt,.dat,.csv;"
fileFilters += "HTML Files (*.htm,*.html):.htm,.html;"
fileFilters += "All Files:.*;"
Open /F=fileFilters . . .
```

Each file filter consists of a filter menu string (e.g., "Data Files") followed by a colon, followed by one or more file name extensions (e.g., ".txt,.dat,.csv") followed by a semicolon. The syntax is rigid - no extra characters are allowed and the semicolons shown above are required. In this example the filter menu would contain "Data Files" and would accept any file with a ".txt", ".dat", or ".csv" extension. ".*" creates a filter that accepts any file.

If you use the /F flag, it is up to you to add a filter for All Files as shown above. It is recommended that you do this.

Displaying a Save File Dialog

You can display a Save File dialog to allow the user to choose a file to be created or overwritten by a subsequent command. For example, the user can choose a file which you will then create or overwrite via a Save command. The Save File dialog is displayed using an **Open**/D command. Here is an example:

```
Function/S DoSaveFileDialog()
   Variable refNum
   String message = "Save a file"
   String outputPath
   String fileFilters = "Data Files (*.txt):.txt;"
   fileFilters += "All Files:.*;"

   Open /D /F=fileFilters /M=message refNum
   outputPath = S_fileName

   return outputPath    // Will be empty if user canceled
End
```

Here the Open operation does not actually open a file but instead displays a Save File dialog. If the user chooses a file and clicks the Save button, the Open operation returns the full path to the file in the S_fileName output string variable. If the user cancels, Open sets S_fileName to "".

The /M flag is used to set the prompt message. As of OS X 10.11, Apple no longer shows the prompt message in the Save File dialog. It continues to work on Windows.

The /F flag is used to control the file filter which determines what kinds of files the user can create. This is explained further under **Save File Dialog File Filters**.

Save File Dialog File Filters

The Save File dialog includes a file filter menu that allows the user to choose the type of file to be saved. By default this menus contain "Plain Text File" and, on Windows only, "All Files". You can use the /T and /F flags to override the default filter behavior.

The /T and /F flags work as explained under Open File Dialog File Filters. Using the /F flag for a Save File dialog, you would typically specify just one filter plus All Files, like this:

String fileFilters = "Data File (*.dat):.dat;"

fileFilters += "All Files:.*;"

Open /F=fileFilters . . .

The file filter chosen in the Save File dialog determines the extension for the file being saved. For example, if the "Plain Text Files" filter is selected, the ".txt" extension is added if you don't explicitly enter it in the File Name edit box. However if you select the "All Files" filter then no extension is automatically added and the final file name is whatever you enter in the File Name edit box. You should include the "All Files" filter if you want the user to be able to specify a file name with any extension. If you want to force the file name extension to an extension of your choice rather than the user's, omit the "All Files" filter.

Using Open in a Utility Routine

To be as general and useful as possible, a utility routine that acts on a file should have a pathName parameter and a fileName parameter, like this:

```
Function ShowFileInfo(pathName, fileName)
   String pathName   // Name of symbolic path or "" for dialog.
   String fileName   // File name or "" for dialog.

   <Show file info here>
End
```

This provides flexibility to the calling function. The caller can supply a valid symbolic path name and a simple leaf name in fileName, a valid symbolic path name and a partial path in fileName, or a full path in fileName in which case pathName is irrelevant.

If pathName and fileName fully specify the file of interest, you want to just open the file and perform the requested action. However, if pathName and fileName do not fully specify the file of interest, you want to display an Open File dialog so the user can choose the file. This is accomplished by using the **Open** operation's /D=2 flag.

With /D=2, if pathName and fileName fully specify the file, the Open operation merely sets the S_fileName output string variable to the full path to the file. If pathName and fileName do not fully specify the file, Open displays an Open File dialog and then sets the S_fileName output string variable to the full path to the file. If the user cancels the Open File dialog, Open sets S_fileName to "". In all cases, Open/D=2 just sets S_fileName and does not actually open the file.

If pathName and fileName specify an alias (*Macintosh*) or shortcut (*Windows*), Open/D=2 returns the file referenced by the alias or shortcut.

Here is how you would use Open /D=2.

```
Function ShowFileInfo(pathName, fileName)
   String pathName      // Name of symbolic path or "" for dialog.
   String fileName      // File name or "" for dialog.

   Variable refNum

   Open /D=2 /R /P=$pathName refNum as fileName    // Sets S_fileName

   if (strlen(S_fileName) == 0)
      Print "ShowFileInfo was canceled"
   else
      String fullPath = S_fileName
      Print fullPath
      Open /R refNum as fullPath
      FStatus refNum    // Sets S_info
      Print S_info
      Close refNum
   endif
End
```

In this case, we wanted to open the file for reading. To create a file and open it for writing, omit /R from both calls to Open.

Pause For User

The **PauseForUser** operation (see page V-738) allows an advanced programmer to create a more sophisticated semi-modal user interface. When you invoke it from a procedure, Igor suspends procedure execution and the user can interact with graph, table or control panel windows using the mouse or keyboard. Execution continues when the user kills the main window specified in the PauseForUser command.

PauseForUser is fragile because it attempts to create a semi-modal mode which is not supported by the operating system. Before using it, consider using an **Alternative to PauseForUser**.

Uses of PauseForUser

Pausing execution can serve two purposes. First, the programmer can pause function execution so that the user can, for example, adjust cursors in a graph window before continuing with a curve fit. In this application, the programmer creates a control panel with a continue button that the user presses after adjusting the cursors in the target graph. Pressing the continue button kills the host control panel (see example below).

In the second application, the programmer may wish to obtain input from the user in a more sophisticated manner than can be done using DoPrompt commands. This method uses a control panel as the main window with no optional target window. It is similar to the control panel technique shown above, except that it is modal.

Following are some examples of how you can use the **PauseForUser** operation (see page V-738) in your own user functions.

PauseForUser Simple Cursor Example

This example shows how to allow the user to adjust cursors on a graph while a procedure is executing. Most of the work is done by the UserCursorAdjust function. UserCursorAdjust is called by the Demo function which first creates a graph and shows the cursor info panel.

This example illustrates two modes of PauseForUser. When called with autoAbortSecs=0, UserCursorAdjust calls PauseForUser without the /C flag in which case PauseForUser retains control until the user clicks the Continue button.

When called with autoAbortSecs>0, UserCursorAdjust calls PauseForUser/C. This causes PauseForUser to handle any pending events and then return to the calling procedure. The procedure checks the V_flag variable, set by PauseForUser, to determine when the user has finished interacting with the graph. PauseForUser/C, which requires Igor Pro 6.1 or later, is for situations where you want to do something while the user interacts with the graph.

To try this yourself, copy and paste all three routines below into the procedure window of a new experiment and then run the Demo function with a value of 0 and again with a value such as 30.

```
Function UserCursorAdjust(graphName,autoAbortSecs)
   String graphName
   Variable autoAbortSecs

   DoWindow/F $graphName                  // Bring graph to front
   if (V_Flag == 0)                       // Verify that graph exists
      Abort "UserCursorAdjust: No such graph."
      return -1
   endif

   NewPanel /K=2 /W=(187,368,437,531) as "Pause for Cursor"
   DoWindow/C tmp_PauseforCursor          // Set to an unlikely name
   AutoPositionWindow/E/M=1/R=$graphName  // Put panel near the graph

   DrawText 21,20,"Adjust the cursors and then"
   DrawText 21,40,"Click Continue."
   Button button0,pos={80,58},size={92,20},title="Continue"
   Button button0,proc=UserCursorAdjust_ContButtonProc
   Variable didAbort= 0
   if( autoAbortSecs == 0 )
      PauseForUser tmp_PauseforCursor,$graphName
   else
      SetDrawEnv textyjust= 1
      DrawText 162,103,"sec"
      SetVariable sv0,pos={48,97},size={107,15},title="Aborting in "
      SetVariable sv0,limits={-inf,inf,0},value= _NUM:10
      Variable td= 10,newTd
      Variable t0= ticks
      Do
         newTd= autoAbortSecs - round((ticks-t0)/60)
         if( td != newTd )
            td= newTd
            SetVariable sv0,value= _NUM:newTd,win=tmp_PauseforCursor
            if( td <= 10 )
               SetVariable sv0,valueColor= (65535,0,0),win=tmp_PauseforCursor
            endif
         endif
         if( td <= 0 )
            KillWindow tmp_PauseforCursor
            didAbort= 1
            break
         endif

         PauseForUser/C tmp_PauseforCursor,$graphName
      while(V_flag)
   endif
   return didAbort
End

Function UserCursorAdjust_ContButtonProc(ctrlName) : ButtonControl
   String ctrlName

   KillWindow/Z tmp_PauseforCursor           // Kill panel
End

Function Demo(autoAbortSecs)
```

```
   Variable autoAbortSecs

   Make/O jack;SetScale x,-5,5,jack
   jack= exp(-x^2)+gnoise(0.1)
   DoWindow Graph0
   if( V_Flag==0 )
      Display jack
      ShowInfo
   endif

   if (UserCursorAdjust("Graph0",autoAbortSecs) != 0)
      return -1
   endif

   if (strlen(CsrWave(A))>0 && strlen(CsrWave(B))>0)// Cursors are on trace?
      CurveFit gauss,jack[pcsr(A),pcsr(B)] /D
   endif
End
```

Alternative to PauseForUser

In normal non-modal operation, when the user invokes a user-defined function, the function executes, terminates, and returns control to the user. The user is in complete control of what happens next. With PauseForUser, the user-defined function is running the whole time and the user's control is restricted control to what PauseForUser allows.

For example, with PauseForUser, the user can not use the help system, search for text in a notebook, activate another window for copy and paste into a table, use the Data Browser, and so on. The user can do the few things PauseForUser permits.

The alternative to PauseForUser is non-modal operation. To facilitate this in a situation where you might consider using PauseForUser, you need to break the task into two or more steps, each implemented by a separate function, and provide the user with the means to invoke each function as he sees fit. The user initiates the first step which runs to completion. The user is then free to do what he wishes before invoking the second step.

To see the non-modal approach in action, execute this in Igor and follow the instructions for running the example:

```
DisplayHelpTopic "Alternatives to PauseForUser"
```

The main difference between the non-modal alternative approach and the PauseForUser approach is that, in the non-modal case, the user is free to do whatever he wants between step one (creating the graph) and step two (performing the fit).

A hidden difference is that the semi-modal operation of PauseForUser relies on tricky and potentially fragile code inside Igor while the non-modal approach uses Igor in a natural way.

PauseForUser Advanced Cursor Example

Now for something a bit more complex. Here we modify the preceding example to include a Cancel button. For this, we need to return information about which button was pressed. Although we could do this by creating a single global variable in the root data folder, we use a slightly more complex technique using a temporary data folder. This technique is especially useful for more complex panels with multiple output variables because it eliminates name conflict issues. It also allows much easier clean up because we can kill the entire data folder and everything in it with just one operation.

```
Function UserCursorAdjust(graphName)
   String graphName

   DoWindow/F $graphName   // Bring graph to front
   if (V_Flag == 0)        // Verify that graph exists
```

```
      Abort "UserCursorAdjust: No such graph."
      return -1
   endif

   NewDataFolder/O root:tmp_PauseforCursorDF
   Variable/G root:tmp_PauseforCursorDF:canceled= 0

   NewPanel/K=2 /W=(139,341,382,450) as "Pause for Cursor"
   DoWindow/C tmp_PauseforCursor          // Set to an unlikely name
   AutoPositionWindow/E/M=1/R=$graphName  // Put panel near the graph

   DrawText 21,20,"Adjust the cursors and then"
   DrawText 21,40,"Click Continue."
   Button button0,pos={80,58},size={92,20},title="Continue"
   Button button0,proc=UserCursorAdjust_ContButtonProc
   Button button1,pos={80,80},size={92,20}
   Button button1,proc=UserCursorAdjust_CancelBProc,title="Cancel"

   PauseForUser tmp_PauseforCursor,$graphName

   NVAR gCaneled= root:tmp_PauseforCursorDF:canceled
   Variable canceled= gCaneled   // Copy from global to local
                                 // before global is killed
   KillDataFolder root:tmp_PauseforCursorDF

   return canceled
End

Function UserCursorAdjust_ContButtonProc(ctrlName) : ButtonControl
   String ctrlName

   KillWindow/Z tmp_PauseforCursor  // Kill self
End

Function UserCursorAdjust_CancelBProc(ctrlName) : ButtonControl
   String ctrlName

   Variable/G root:tmp_PauseforCursorDF:canceled= 1
   KillWindow/Z tmp_PauseforCursor  // Kill self
End

Function Demo()
   Make/O jack;SetScale x,-5,5,jack
   jack= exp(-x^2)+gnoise(0.1)
   DoWindow Graph0
   if (V_Flag==0)
      Display jack
      ShowInfo
   endif
   Variable rval= UserCursorAdjust("Graph0")
   if (rval == -1)            // Graph name error?
      return -1;
   endif
   if (rval == 1)             // User canceled?
      DoAlert 0,"Canceled"
      return -1;
   endif
   CurveFit gauss,jack[pcsr(A),pcsr(B)] /D
End
```

PauseForUser Control Panel Example

This example illustrates using a control panel as a modal dialog via PauseForUser. This technique is useful when you need a more sophisticated modal user interface than provided by the simple input dialog.

We started by manually creating a control panel. When the panel design was finished, we closed it to create a recreation macro. We then used code copied from the recreation macro in the DoMyInputPanel function and deleted the recreation macro.

```
Function UserGetInputPanel_ContButton(ctrlName) : ButtonControl
   String ctrlName

   KillWindow/Z tmp_GetInputPanel      // Kill self
End

// Call with these variables already created and initialized:
//    root:tmp_PauseForUserDemo:numvar
//    root:tmp_PauseForUserDemo:strvar
Function DoMyInputPanel()
   NewPanel /W=(150,50,358,239)
   DoWindow/C tmp_GetInputPanel        // Set to an unlikely name
   DrawText 33,23,"Enter some data"
   SetVariable setvar0,pos={27,49},size={126,17},limits={-Inf,Inf,1}
   SetVariable setvar0,value= root:tmp_PauseForUserDemo:numvar
   SetVariable setvar1,pos={24,77},size={131,17},limits={-Inf,Inf,1}
   SetVariable setvar1,value= root:tmp_PauseForUserDemo:strvar
   Button button0,pos={52,120},size={92,20}
   Button button0,proc=UserGetInputPanel_ContButton,title="Continue"

   PauseForUser tmp_GetInputPanel
End

Function Demo1()
   NewDataFolder/O root:tmp_PauseForUserDemo
   Variable/G root:tmp_PauseForUserDemo:numvar= 12
   String/G root:tmp_PauseForUserDemo:strvar= "hello"

   DoMyInputPanel()

   NVAR numvar= root:tmp_PauseForUserDemo:numvar
   SVAR strvar= root:tmp_PauseForUserDemo:strvar

   printf "You entered %g and %s\r",numvar,strvar

   KillDataFolder root:tmp_PauseForUserDemo
End
```

Progress Windows

Sometimes when performing a long calculation, you may want to display an indication that the calculation is in progress, perhaps showing how far along it is, and perhaps providing an abort button. As of Igor Pro 6.1, you can use a control panel window for this task using the **DoUpdate** /E and /W flags and the mode=4 setting for **ValDisplay**.

DoUpdate /W=win /E=1 marks the specified window as a progress window that can accept mouse events while user code is executing. The /E flag need be used only once to mark the panel but it does not hurt to use it in every call. This special state of the control panel is automatically cleared when procedure execution finishes and Igor's outer loop again runs.

For a window marked as a progress window, DoUpdate sets V_Flag to 2 if a mouse up happened in a button since the last call. When this occurs, the full path to the subwindow containing the button is stored in S_path and the name of the control is stored in S_name.

Here is a simple example that puts up a progress window with a progress bar and a Stop button. Try each of the four input flag combinations.

```
//    ProgressDemo1(0,0)
//    ProgressDemo1(1,0)
//    ProgressDemo1(0,1)
//    ProgressDemo1(1,1)
Function ProgressDemo1(indefinite, useIgorDraw)
   Variable indefinite
   Variable useIgorDraw// True to use Igor's own draw method rather than native

   NewPanel /N=ProgressPanel /W=(285,111,739,193)
   ValDisplay valdisp0,pos={18,32},size={342,18}
   ValDisplay valdisp0,limits={0,100,0},barmisc={0,0}
   ValDisplay valdisp0,value= _NUM:0
   if( indefinite )
      ValDisplay valdisp0,mode= 4// candy stripe
   else
      ValDisplay valdisp0,mode= 3// bar with no fractional part
   endif
   if( useIgorDraw )
      ValDisplay valdisp0,highColor=(0,65535,0)
   endif
   Button bStop,pos={375,32},size={50,20},title="Stop"
   DoUpdate /W=ProgressPanel /E=1// mark this as our progress window

   Variable i,imax= indefinite ? 10000 : 100
   for(i=0;i<imax;i+=1)
      Variable t0= ticks
      do
      while( ticks < (t0+3) )
      if( indefinite )
         ValDisplay valdisp0,value= _NUM:1,win=ProgressPanel
      else
         ValDisplay valdisp0,value= _NUM:i+1,win=ProgressPanel
      endif
      DoUpdate /W=ProgressPanel
      if( V_Flag == 2 )// we only have one button and that means stop
         break
      endif
   endfor
   KillWindow ProgressPanel
End
```

When performing complex calculations, it is often difficult to insert DoUpdate calls in the code. In this case, you can use a window hook that responds to event #23, spinUpdate. This is called at the same time that the beachball icon in the status bar spins. The hook can then update the window's control state and then call DoUpdate/W on the window. If the window hook returns non-zero, then an abort is performed. If you desire a more controlled quit, you might set a global variable that your calculation code can test

The following example provides an indefinite indicator and an abort button. Note that if the abort button is pressed, the window hook kills the progress window since otherwise the abort would cause the window to remain.

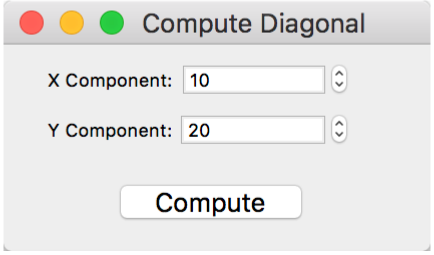
```
// Example: ProgressDemo2(100)
Function ProgressDemo2(nloops)
   Variable nloops

   Variable useIgorDraw=0  // set true for Igor draw method rather than native

   NewPanel/FLT /N=myProgress/W=(285,111,739,193) as "Calculating..."
   ValDisplay valdisp0,pos={18,32},size={342,18}
   ValDisplay valdisp0,limits={0,100,0},barmisc={0,0}
   ValDisplay valdisp0,value= _NUM:0
   ValDisplay valdisp0,mode=4    // candy stripe
   if( useIgorDraw )
      ValDisplay valdisp0,highColor=(0,65535,0)
   endif
   Button bStop,pos={375,32},size={50,20},title="Abort"
   SetActiveSubwindow _endfloat_
   DoUpdate/W=myProgress/E=1     // mark this as our progress window

   SetWindow myProgress,hook(spinner)=MySpinnHook

   Variable t0= ticks,i
   for(i=0;i<nloops;i+=1)
      PerformLongCalc(1e6)
   endfor
   Variable timeperloop= (ticks-t0)/(60*nloops)

   KillWindow myProgress

   print "time per loop=",timeperloop
End

Function MySpinnHook(s)
   STRUCT WMWinHookStruct &s

   if( s.eventCode == 23 )
      ValDisplay valdisp0,value= _NUM:1,win=$s.winName
      DoUpdate/W=$s.winName
      if( V_Flag == 2 )    // we only have one button and that means abort
         KillWindow $s.winName
         return 1
      endif
   endif
   return 0
End

Function PerformLongCalc(nmax)
   Variable nmax

   Variable i,s
   for(i=0;i<nmax;i+=1)
      s+= sin(i/nmax)
   endfor
End
```

Control Panels and Event-Driven Programming

The CalcDiagDialog function shown under **The Simple Input Dialog** on page IV-144 creates a modal dialog. “Modal” means that the function retains complete control until the user clicks Cancel or Continue. The user can not activate another window or choose a menu item until the dialog is dismissed.

This section shows how to implement the same functionality using a control panel as a modeless dialog. “Modeless” means that the user can activate another window or choose a menu item at any time. The modeless window accepts input whenever the user wants to enter it but does not block the user from accessing other windows.

The control panel looks like this:



X Component: 10

Y Component: 20

The code implementing this control panel is given below. Before we look at the code, here is some explanation of the thinking behind it.

The X Component and Y Component controls are SetVariable controls. We attach each SetVariable control to a global variable so that, if we kill and later recreate the panel, the SetVariable control is restored to its previous state. In other words, we use global variables to remember settings across invocations of the panel. To keep the global variables from cluttering the user's space, we bury them in a data folder located at root:Packages:DiagonalControlPanel.

The DisplayDiagonalControlPanel routine creates the data folder and the global variables if they do not already exist. DisplayDiagonalControlPanel creates the control panel or, if it already exists, just brings it to the front.

We added a menu item to the Macros menu so the user can easily invoke DisplayDiagonalControlPanel.

We built the control panel manually using techniques explained in Chapter III-14, **Controls and Control Panels**. Then we closed it so Igor would create a display recreation macro which we named DiagonalControlPanel. We then manually tweaked the macro to attach the SetVariable controls to the desired globals and to set the panel's behavior when the user clicks the close button by adding the /K=1 flag.

Here are the procedures.

```
// Add a menu item to display the control panel.
Menu "Macros"
   "Display Diagonal Control Panel", DisplayDiagonalControlPanel()
End

// This is the display recreation macro, created by Igor
// and then manually tweaked. The parts that were tweaked
// are shown in bold. NOTE: Some lines are wrapped to fit on the page.
Window DiagonalControlPanel() : Panel
   PauseUpdate; Silent 1   // building window...

   NewPanel/W=(162,95,375,198)/K=1 as "Compute Diagonal"

   SetVariable XSetVar,pos={22,11},size={150,15},title="X Component:"
   SetVariable XSetVar,limits={-Inf,Inf,1},value=
               root:Packages:DiagonalControlPanel:gXComponent

   SetVariable YSetVar,pos={22,36},size={150,15},title="Y Component:"
   SetVariable YSetVar,limits={-Inf,Inf,1},value=
               root:Packages:DiagonalControlPanel:gYComponent

   Button ComputeButton,pos={59,69},size={90,20},
                    proc=ComputeDiagonalProc,title="Compute"
EndMacro

// This is the action procedure for the Compute button.
// We created it using the Button dialog.
Function ComputeDiagonalProc(ctrlName) : ButtonControl
   String ctrlName

   DFREF dfr = root:Packages:DiagonalControlPanel
   NVAR gXComponent = dfr:gXComponent
   NVAR gYComponent = dfr:gYComponent
   Variable diagonal
   diagonal = sqrt(gXComponent^2 + gYComponent^2)
   Printf "Diagonal=%g\r", diagonal
End

// This is the top level routine which makes sure that the globals
// and their enclosing data folders exist and then makes sure that
// the control panel is displayed.
Function DisplayDiagonalControlPanel()
   // If the panel is already created, just bring it to the front.
   DoWindow/F DiagonalControlPanel
   if (V_Flag != 0)
      return 0
   endif

   String dfSave = GetDataFolder(1)

   // Create a data folder in Packages to store globals.
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S root:Packages:DiagonalControlPanel

   // Create global variables used by the control panel.
   Variable xComponent = NumVarOrDefault(":gXComponent", 10)
   Variable/G gXComponent = xComponent
   Variable yComponent = NumVarOrDefault(":gYComponent", 20)
   Variable/G gYComponent = yComponent

   // Create the control panel.
   Execute "DiagonalControlPanel()"

   SetDataFolder dfSave
End
```

To try this example, copy all of the procedures and paste them into the procedure window of a new experiment. Close the procedure window to compile it and then choose Display Diagonal Control Panel from the Macros menu. Next enter values in the text entry items and click the Compute button. Close the control panel and then reopen it using the Display Diagonal Control Panel menu item. Notice that the values that you entered were remembered. Use the Data Browser to inspect the root:Packages:DiagonalControlPanel data folder.

Although this example is very simple, it illustrates the process of creating a control panel that functions as a modeless dialog. There are many more examples of this in the Examples folder. You can access them via the File→Example Experiments submenu.

See Chapter III-14, **Controls and Control Panels**, for more information on building control panels.

Detecting a User Abort

If you have written a user-defined function that takes a long time to execute, you may want to provide a way for the user to abort it. One solution is to display a progress window as discussed under **Progress Windows** on page IV-156.

Here is a simple alternative using the escape key:

```
Function PressEscapeToAbort(phase, title, message)
   Variable phase    // 0: Display control panel with message.
                     // 1: Test if Escape key is pressed.
                     // 2: Close control panel.
   String title      // Title for control panel.
   String message    // Tells user what you are doing.
   if (phase == 0)   // Create panel
      DoWindow/F PressEscapePanel
      if (V_flag == 0)
         NewPanel/K=1 /W=(100,100,350,200)
         DoWindow/C PressEscapePanel
         DoWindow/T PressEscapePanel, title
      endif
      TitleBox Message,pos={7,8},size={69,20},title=message
      String abortStr = "Press escape to abort"
      TitleBox Press,pos={6,59},size={106,20},title=abortStr
      DoUpdate
   endif

   if (phase == 1)   // Test for Escape key
      Variable doAbort = 0
      if (GetKeyState(0) & 32)      // Is Escape key pressed now?
         doAbort = 1
      else
         if (strlen(message) != 0)  // Want to change message?
            TitleBox Message,title=message
            DoUpdate
         endif
      endif
      return doAbort
   endif

   if (phase == 2)   // Kill panel
      KillWindow/Z PressEscapePanel
   endif

   return 0
End

Function Demo()
   // Create panel
   PressEscapeToAbort(0, "Demonstration", "This is a demo")

   Variable startTicks = ticks
   Variable endTicks = startTicks + 10*60
   Variable lastMessageUpdate = startTicks

   do
      String message
      message = ""
      if (ticks>=lastMessageUpdate+60) // Time to update message?
         Variable remaining = (endTicks - ticks) / 60
         sprintf message, "Time remaining: %.1f seconds", remaining
         lastMessageUpdate = ticks
      endif

      if (PressEscapeToAbort(1, "", message))
         Print "Test aborted by Escape key."
         break
      endif
   while(ticks < endTicks)

   PressEscapeToAbort(2, "", "")       // Kill panel.
End
```

Creating a Contextual Menu

You can use the **PopupContextualMenu** operation to create a pop-up menu in response to a contextual click (control-click (*Macintosh*) or right-click). You would do this from a window hook function or from the action procedure for a control in a control panel.

In this example, we create a control panel with a list. When the user right-clicks on the list, Igor sends a mouse-down event to the listbox procedure, TickerListProc in this case. The listbox procedure uses the eventMod field of the WMListboxAction structure to determine if the click is a right-click. If so, it calls HandleTickerListRightClick which calls PopupContextualMenu to display the contextual menu.

```
Menu "Macros"

   "Show Demo Panel", ShowDemoPanel()
End

static Function HandleTickerListRightClick()
   String popupItems = ""
   popupItems += "Refresh;"

   PopupContextualMenu popupItems
   strswitch (S_selection)
      case "Refresh":
         DoAlert 0, "Here is where you would refresh the ticker list."
         break
   endswitch
End

Function TickerListProc(lba) : ListBoxControl
   STRUCT WMListboxAction &lba

   switch (lba.eventCode)
      case 1:                    // Mouse down
         if (lba.eventMod & 0x10)// Right-click?
            HandleTickerListRightClick()
         endif
         break
   endswitch

   return 0
End

Function ShowDemoPanel()
   DoWindow/F DemoPanel
   if (V_flag != 0)
      return 0 // Panel already exists.
   endif

   // Create panel data.
   Make/O/T ticketListWave = {{"AAPL","IBM","MSFT"}, {"90.25","86.40","17.17"}}

   // Create panel.
   NewPanel /N=DemoPanel /W=(321,121,621,321) /K=1
   ListBox TickerList,pos={48,16},size={200,100},fSize=12
   ListBox TickerList,listWave=root:ticketListWave
   ListBox TickerList,mode= 1,selRow= 0, proc=TickerListProc
End
```

Cursors as Input Device

You can use the cursors on a trace in a graph to identify the data to be processed.

The examples shown above using PauseForUser are modal - the user adjusts the cursors in the middle of procedure execution and can do nothing else. This technique is non-modal — the user is expected to adjust the cursors before invoking the procedure.

This function does a straight-line curve fit through the data between cursor A (the round cursor) and cursor B (the square cursor). This example is written to handle both waveform and XY data.

```
Function FitLineBetweenCursors()
   Variable isXY

   // Make sure both cursors are on the same wave.
   WAVE wA = CsrWaveRef(A)
   WAVE wB = CsrWaveRef(B)
   String dfA = GetWavesDataFolder(wA, 2)
   String dfB = GetWavesDataFolder(wB, 2)
   if (CmpStr(dfA, dfB) != 0)
      Abort "Both cursors must be on the same wave."
      return -1
   endif

   // Find the wave that the cursors are on.
   WAVE yWave = CsrWaveRef(A)

   // Decide if this is an XY pair.
   WAVE xWave = CsrXWaveRef(A)
   isXY = WaveExists(xWave)

   if (isXY)
      CurveFit line yWave(xcsr(A),xcsr(B)) /X=xWave /D
   else
      CurveFit line yWave(xcsr(A),xcsr(B)) /D
   endif
End
```

This technique is demonstrated in the **Fit Line Between Cursors** example experiment in the “Examples:Curve Fitting” folder.

Advanced programmers can set things up so that a hook function is called whenever the user adjusts the position of a cursor. For details, see **Cursors — Moving Cursor Calls Function** on page IV-339.

Marquee Menu as Input Device

A marquee is the dashed-line rectangle that you get when you click and drag diagonally in a graph or page layout. It is used for expanding and shrinking the range of axes, for selecting a rectangular section of an image, and for specifying an area of a layout. You can use the marquee as an input device for your procedures. This is a relatively advanced technique.

This menu definition adds a user-defined item to the graph marquee menu:

```
Menu "GraphMarquee"
   "Print Marquee Coordinates", PrintMarqueeCoords()
End
```

To add an item to the layout marquee menu, use LayoutMarquee instead of GraphMarquee.

When the user chooses Print Marquee Coordinates, the following function runs. It prints the coordinates of the marquee in the history area. It assumes that the graph has left and bottom axes.

```
Function PrintMarqueeCoords()
   String format
   GetMarquee/K left, bottom
   format = "flag: %g; left: %g; top: %g; right: %g; bottom: %g\r"
   printf format, V_flag, V_left, V_top, V_right, V_bottom
End
```

The use of the marquee menu as in input device is demonstrated in the **Marquee Demo** and **Delete Points from Wave** example experiments.

Polygon as Input Device

This technique is similar to the marquee technique except that you can identify a nonrectangular area. It is implemented using **FindPointsInPoly** operation (see page V-248).

Chapter

IV-7

Programming Techniques

Overview

This chapter discusses programming techniques and issues that go beyond the basics of the Igor programming language and which all Igor programmers should understand. Techniques for advanced programmers are discussed in Chapter IV-10, **Advanced Topics**.

The Include Statement

The include statement is a compiler directive that you can put in your procedure file to open another procedure file. A typical include statement looks like this:

```
#include <Split Axis>
```

This statement opens the file "Split Axis.ipf" in the WaveMetrics Procedures folder. Once this statement has been compiled, you can call routines from that file. This is the recommended way to access procedure files that contain utility routines.

The # character at the beginning specifies that a compiler directive follows. Compiler directives, except for conditional compilation directives, must start at the far left edge of the procedure window with no leading tabs or spaces. Include statements can appear anywhere in the file but it is conventional to put them near the top.

It is possible to include a file which in turn includes other files. Igor opens all of the included files.

Igor opens included files when it compiles procedures. If you remove an include statement from a procedure file, Igor automatically closes the file on the next compile.

There are four forms of the include statement:

1. Igor searches for the named file in "Igor Pro Folder/WaveMetrics Procedures" and in any subfolders:
   `#include <`*`fileName`*`>`
   Example: `#include <Split Axis>`
2. Igor searches for the named file in "Igor Pro Folder/User Procedures" and "Igor Pro User Files/User Procedures" and in any subfolders:
   `#include "`*`fileName`*`"`
   Example: `#include "Utility Procs"`
3. Igor looks for the file only in the exact location specified:
   `#include "`*`full file path`*`"`
   Example: `#include "Hard Disk:Procedures:Utility Procs"`
4. Igor looks for the file relative to the Igor Pro folder and the Igor Pro User Files folder and the folder containing the procedure file that contains the #include statement:
   `#include "`*`:partial file path`*`"`
   Example: `#include ":Spectroscopy Procedures:Voigt Procs"`

   Igor looks first relative to the Igor Pro Folder. If that fails, it looks relative to the **Igor Pro User Files** folder. If that fails it looks relative to the procedure file containing the #include statement or, if the #include statement is in the built-in procedure window, relative to the experiment file.

   The name of the file being included must end with the standard ".ipf" extension but the extension is not used in the include statement.

Procedure File Version Information

If you create a procedure file to be used by other Igor users, it's a good idea to add version information to the file. You do this by putting a #pragma version statement in the procedure file. For example:

```
#pragma version = 1.10
```

This statement must appear with no indentation and must appear in the first 50 lines of the file. If Igor finds no version pragma, it treats the file as version 1.00.

Igor looks for the version information when the user invokes the File Information dialog from the Procedure menu. If the file has version information, Igor displays the version next to the file name in the dialog.

Igor also looks for version information when it opens an included file. An include statement can require a certain version of the included file using the following syntax:

```
#include <Bivariate Histogram> version>=1.02
```

If the required version of the procedure file is not available, Igor displays a warning to inform the user that the procedure file needs to be updated.

In Igor Pro 9.01 and later, you can programmatically determine the procedure file's version using the **ProcedureVersion** function.

Turning the Included File's Menus Off

Normally an included procedure file's menus and menu items are displayed. However, if you are including a file merely to call one of its procedures, you may not want that file's menu items to appear. To suppress the file's menus and menu items, use:

```
#include <Decimation> menus=0
```

To use both the menus and version options, you must separate them with a comma:

```
#include <Decimation> menus=0, version>=1.1
```

Optionally Including Files

Compilation usually ceases and returns an error if the included file isn't found. On occasion it is advantageous to allow compilation to proceed if an included file isn't present or is the wrong version. Optionally including a procedure file is appropriate only if the file truly isn't needed to make procedures compile or operate.

Use the "optional" keyword to optionally include a procedure file:

```
#include <Decimation> version>=1.1, optional
```

Writing General-Purpose Procedures

Procedures can be placed on a scale ranging from general to specific. Usually, the high-level procedures of a program are specific to the task at hand and call on more general, lower-level procedures. The most general procedures are often called "utility" procedures.

You can achieve a high degree of productivity by building a library of utility procedures that you reuse in different programs. In Igor, you can think of the routines in an experiment's built-in Procedure window as a program. It can call utility routines which should be stored in a separate procedure file so that they are available to multiple experiments.

The files stored in the WaveMetrics Procedures folder contain general-purpose procedures on which your high-level procedures can build. Use the include statement (see **The Include Statement** on page IV-166) to access these and other utility files.

When you write utility routines, you should keep them as general as possible so that they can be reused as much as possible. Here are some guidelines.

- Avoid the use of global variables as inputs or outputs. Using globals hard-wires the routine to specific names which makes it difficult to use and limits its generality.
- If you use globals to store state information between invocations of a routine, package the globals in data folders.

  WaveMetrics packages usually create data folders inside "root:Packages". We use the prefix "WM" for data folder names to avoid conflict with other packages. Follow this lead and should pick your own data folder names so as to minimize the chance of conflict with WaveMetrics packages or with others. See **Managing Package Data** on page IV-249 for details.

- Choose a clear and specific name for your utility routine.

  By choosing a name that says precisely what your utility routine does, you minimize the likelihood of collision with the name of another procedure. You also increase the readability of your program.
- Make functions which are used only internally by your procedures static.

  By making internal functions static (i.e., private), you minimize the likelihood of collision with the name of another procedure.

Programming with Liberal Names

Standard names in Igor can contain letters, numbers and the underscore character only.

It is possible to use wave names and data folder names that contain almost any character. Such names are called “liberal names” (see **Liberal Object Names** on page III-501).

As this section explains, programmers need to use special techniques for their procedures to work with liberal names. Consequently, if you do use liberal names, some existing Igor procedures may break.

Whenever a liberal name is used in a command or expression, the name must be enclosed in single quotes. For example:

```
Make 'Wave 1', wave2, 'miles/hour'
'Wave 1' = wave2 + 'miles/hour'
```

Without the single quotes, Igor has no way to know where a particular name ends. This is a problem whenever Igor parses a command or statement. Igor parses commands at the following times:

- When it compiles a user-defined function.
- When it compiles the right-hand side of an assignment statement, including a formula (:= dependency expression).
- When it interprets a macro.
- When it interprets a command that you enter in the command line or via an Igor Text file.
- When it interprets a command you submit for execution via the Execute operation.
- When it interprets a command that an XOP submits for execution via the XOPCommand or XOPSilentCommand callback routines.

When you use an Igor dialog to generate a command, Igor automatically uses quotes where necessary.

Programmers need to be concerned about liberal names whenever they create a command and then execute it, via an Igor Text file, the Execute operation or the XOPCommand and XOPSilentCommand callback routines, and when creating a formula. In short, when you create something that Igor has to parse, names must be quoted if they are liberal. Names that are not liberal can be quoted or unquoted.

If you have a procedure that builds up a command in a string variable and then executes it via the Execute operation, you must use the PossiblyQuoteName function to provide the quotes if needed.

Here is a trivial example showing the liberal-name unaware and liberal-name aware techniques:

```
Function AddOneToWave(w)
   WAVE w

   String cmd
   String name = NameOfWave(w)

   // Liberal-name unaware - generates error with liberal name
   sprintf cmd, "%s += 1", name
   Execute cmd

   // Liberal-name aware way
   sprintf cmd, "%s += 1", PossiblyQuoteName(name)
   Execute cmd
End
```

Imagine that you pass a wave named *wave 1* to this function. Using the old technique, the Execute operation would see the following text:

```
wave 1 += 1
```

Igor would generate an error because it would try to find an operation, macro, function, wave or variable named *wave*.

Using the new technique, the Execute operation would see the following text:

```
'wave 1' += 1
```

This works correctly because Igor sees 'wave 1' as a single name.

When you pass a string expression containing a simple name to the $ operator, the name must not be quoted because Igor does no parsing:

```
String name = "wave 1"; Display $name                 // Right
String name = "'wave 1'"; Display $name               // Wrong
```

However, when you pass a *path* in a string to $, any liberal names in the path must be quoted:

```
String path = "root:'My Data':'wave 1'"; Display $path// Right
String path = "root:My Data:wave 1"; Display $path    // Wrong
```

For further explanation of liberal name quoting rules, see **Accessing Global Variables and Waves Using Liberal Names** on page IV-68.

Some built-in string functions return a list of waves. WaveList is the most common example. If liberal wave names are used, you will have to be extra careful when using the list of names. For example, you can't just append the list to the name of an operation that takes a list of names and then execute the resulting string. Rather, you will have to extract each name in turn, possibly quote it and then append it to the operation.

For example, the following will work if all wave names are standard but not if any wave names are liberal:

```
Execute "Display " + WaveList("*", ",", "")
```

Now you will need a string function that extracts out each name, possibly quotes it, and creates a new string using the new names separated by commas. The PossiblyQuoteList function in the WaveMetrics procedure file Strings as Lists does this. The preceding command becomes:

```
Execute "Display " + PossiblyQuoteList(WaveList("*", ",", ""), ",")
```

To include the Strings as Lists file in your procedures, see **The Include Statement** on page IV-166.

For details on using liberal names in user-defined functions, see **Accessing Global Variables and Waves Using Liberal Names** on page IV-68.

These functions are useful for programmatically generating object names: **CreateDataObjectName**, **CheckName**, **CleanupName**, **UniqueName**.

Programming with Data Folders

For general information on data folders, including syntax for using data folders in command line operations, see Chapter II-8, **Data Folders**.

Data folders provide a powerful way to organize data and reduce clutter. However, using data folders introduces some complexity in Igor programming. The name of a variable or wave is no longer sufficient to uniquely identify it because the name alone does not indicate in which data folder it resides.

If a procedure is designed to work on the current data folder, then it is the user's responsibility to set the current data folder correctly before running the procedure. When writing such a procedure, you can use

WaveExists, **NVAR_Exists**, and **SVAR_Exists** to check the existence of expected objects and fail gracefully if they do not exist.

Data Folder Applications

There are two main uses for data folders:

- To store globals used by procedures to keep track of their state
- To store runs of data with identical structures

The following sections explore these applications.

Storing Procedure Globals

If you create packages of procedures, for example data acquisition, data analysis, or a specialized plot type, you typically need to store global variables, strings and waves to maintain the state of your package. You should keep all such items in a data folder that you create with a unique name to reduce clutter and avoid conflicts with other packages.

If your package is inherently global in nature, data acquisition for example, create your data folder within a data folder named Packages that you create in the root data folder. If your package is named MyDataAcq, you would store your globals in root:Packages:MyDataAcq. See **Managing Package Data** on page IV-249 for details and examples.

On the other hand, if your package needs to create a group of variables and waves as a result of an operation on a single input wave, it makes sense to create your data folder in the one that holds the input wave (see the **GetWavesDataFolder** function on page V-317). Similarly, if you create a package that takes an entire data folder as input (via a DFREF parameter pointing to the data folder) and needs to create a data folder containing output, it should create the output data folder in the one, or possibly at the same level as the one containing the input.

If your package creates and maintains windows, it should create a master data folder in root:Packages and then create individual data folders within the master that correspond to each instance of a window. For example, a Smith Chart package would create a master data folder as root:Packages:SmithCharts: and then create individual data folders inside the master with names that correspond to the individual graphs.

A package designed to operate on traces within graphs would go one step further and create a data folder for each trace that it has worked on, inside the data folder for the graph. This technique is illustrated by the Smooth Control Panel procedure file. For a demonstration, choose File→Examples→Feature Demos→Smoothing Control Panel.

Storing Runs of Data

If you acquire data using a well-established experimental protocol, your data will have a well-defined structure. Each time you run your protocol, you produce a new data set with the same structure. Storing each data set in its own data folder avoids name conflicts between corresponding items of different data sets. It also simplifies the writing of procedures to analyze and compare data set.

To use a trivial example, your data might have a structure like this:

```
<Run of data>
   String: conditions
   Variable: temperature
   Wave: appliedVoltage
   Wave: luminosity
```

Having defined this structure, you could then write procedures to:

1. Load your data into a data folder
2. Create a graph showing the data from one run
3. Create a graph comparing the data from many runs

Writing these procedures is greatly simplified by the fact that the names of the items in a run are fixed. For example, step 2 could be written as:

```
Function GraphOneRun()  // Graphs data run in the current data folder.
   SVAR conditions
   NVAR temperature
   WAVE appliedVoltage, luminosity

   Display luminosity vs appliedVoltage

   String text
   sprintf text, "Temperature: %g.\rExperimental conditions: %s.",
         temperature, conditions
   Textbox text
End
```

To create a graph, you would first set the current data folder to point to a run of data and then you would invoke the GraphOneRun function.

The Data Folder Tutorial experiment shows in detail how to accomplish the three steps listed above. Choose File→Example Experiments→Tutorials→Data Folder Tutuorial.

Setting and Restoring the Current Data Folder

There are many reasons why you might want to save and restore the current data folder. In this example, we have a function that does a curve fit to a wave passed in as a parameter. The CurveFit operation creates two waves, W_coef and W_sigma, in the current data folder. If you use the /D option, it also creates a destination wave in the current data folder. In this function, we make sure that the W_coef, W_sigma and destination waves are all created in the same data folder as the source wave.

```
Function DoLineFit(w)
   WAVE w

   String saveDF = GetDataFolder(1)
   SetDataFolder GetWavesDataFolder(w,1)
   CurveFit line w /D
   SetDataFolder saveDF
End
```

Many other operations create output waves in the current data folder. Depending on what your goal is, you may want to use the technique shown here to control where the output waves are created.

A function should always save and restore the current data folder unless it is designed explicitly to change the current data folder.

Determining a Function’s Target Data Folder

There are three common methods for determining the data folder that a function works on or in:

1. Passing a wave in the target data folder as a parameter
2. Having the function work on the current data folder
3. Passing a DFREF parameter that points to the target data folder

For functions that operate on a specific wave, method 1 is appropriate.

For functions that operation on a large number of variables within a single data folder, methods 2 or 3 are appropriate. In method 2, the calling routine sets the data folder of interest as the current data folder. In method 3, the called function does this, and restores the original current data folder before it returns.

Clearing a Data Folder

There are times when you might want to clear a data folder before running a procedure, to remove things left over from a preceding run. If the data folder contains no child data folders, you can achieve this with:

```
KillWaves/A/Z; KillVariables/A/Z
```

If the data folder does contain child data folders, you could use the KillDataFolder operation. This operation kills a data folder and its contents, including any child data folders. You could kill the main data folder and then recreate it. A problem with this is that, if the data folder or its children contain a wave that is in use, you will generate an error which will cause your function to abort.

Here is a handy function that kills the contents of a data folder and the contents of its children without killing any data folders and without attempting to kill any waves that may be in use.

```
Function ZapDataInFolderTree(path)
   String path

   String saveDF = GetDataFolder(1)
   SetDataFolder path

   KillWaves/A/Z
   KillVariables/A/Z
   KillStrings/A/Z

   Variable i
   Variable numDataFolders = CountObjects(":", 4)
   for(i=0; i<numDataFolders; i+=1)
      String nextPath = GetIndexedObjName(":", 4, i)
      ZapDataInFolderTree(nextPath)
   endfor

   SetDataFolder saveDF
End
```

Using Strings

This section explains some common ways in which Igor procedures use strings. The most common techniques use built-in functions such as StringFromList and FindListItem. In addition to the built-in functions, there are a number of handy Igor procedure files in the WaveMetrics Procedures:Utilities:String Utilities folder.

Using Strings as Lists

Procedures often need to deal with lists of items. Such lists are usually represented as semicolon-separated text strings. The StringFromList function is used to extract each item, often in a loop. For example:

```
Function Test()
   Make jack,sam,fred,sue
   String list = WaveList("*",";","")
   Print list

   Variable numItems = ItemsInList(list), i
   for(i=0; i<numItems; i+=1)
      Print StringFromList(i,list)      // Print ith item
   endfor
End
```

For lists with a large number of items, using StringFromList as shown above is slow. This is because it must search from the start of the list to the desired item each time it is called. In Igor7 or later, you can iterate quickly using the optional StringFromList offset parameter:

```
Function Test()
   Make jack,sam,fred,sue
   String list = WaveList("*",";","")
   Print list

   Variable numItems = ItemsInList(list), i
   Variable offset = 0
   for(i=0; i<numItems; i+=1)
      String item = StringFromList(0, list, ";", offset) // Get ith item
      Print item
      offset += strlen(item) + 1
   endfor
End
```

See **ListToTextWave** for another fast way to iterate through the items in a large string list.

Using Keyword-Value Packed Strings

A collection of disparate values is often stored in a list of keyword-value pairs. For example, the TraceInfo and AxisInfo functions return such a list. The information in these strings follows this form

```
KEYWORD:value;KEYWORD:value; ...
```

The **keyword** is always upper case and followed by a colon. Then comes the **value** and a semicolon. When parsing such a string, you should avoid reliance on the specific ordering of keywords — the order is likely to change in future releases of Igor.

Two functions will extract information from keyword-value strings. NumberByKey is used when the data is numeric, and StringByKey is used when the information is in the form of text.

Using Strings with Extractable Commands

The **AxisInfo** and **TraceInfo** readback functions provide much of their information in a form that resembles the commands that you use to make the settings in the first place. For example, to set an axis to log mode, you would execute something like this:

```
ModifyGraph log(left)=1
```

If we now look at the characters of the string returned by AxisInfo we see:

```
AXTYPE:left;CWAVE:jack; ... RECREATION:grid(x)=0;log(x)=1 ...
```

To use this information, we need to extract the value following “log(x)=” and then use it in a ModifyGraph command with the extracted value with the desired graph as the target. Here is a user function that sets the log mode of the bottom axis to the same value as the left axis:

```
Function CopyLogMode()
   String info,text
   Variable pos
   info=AxisInfo("","left")
   pos= StrSearch(info,"RECREATION:",0) // skip to start of "RECREATION:"
   pos += strlen("RECREATION:")         // skip "RECREATION:", too
                                        // get text after "log(x)="
   text= StringByKey("log(x)",info[pos,inf],"=",";")

   String cmd = "ModifyGraph log(bottom)=" + text
   Execute cmd
End
```

Character-by-Character Operations

Igor functions like **strlen** and **strsearch**, as well as Igor string indexing (see **String Indexing** on page IV-16), are byte-oriented. That is, strlen returns the number of bytes in a string. Strsearch takes a byte position as a parameter. Expressions like `str[<index>]` return a single byte.

In Igor6, for western languages, each character is represented by a single byte, so the distinction between bytes and characters is insignificant, and byte-by-byte operations are sufficient.

In Igor7 and later, which use the UTF-8 text encoding to represent text in strings, all ASCII characters are represented by one byte but all non-ASCII characters are represented by multiple bytes (see **Text Encodings** on page III-459 for details). Consequently, byte-by-byte operations are insufficient for stepping through characters. Character-by-character operations are needed.

There is no straightforward way to step through text containing non-ASCII characters. In the rare cases where this is necessary, you can use the user-defined functions shown in this section.

The following functions show how to step through UTF-8 text character-by-character rather than byte-by-byte.

```
// NumBytesInUTF8Character(str, byteOffset)
// Returns the number of bytes in the UTF-8 character that starts byteOffset
// bytes from the start of str.
// NOTE: If byteOffset is invalid this routine returns 0.
//       Also, if str is not valid UTF-8 text, this routine return 1.
Function NumBytesInUTF8Character(str, byteOffset)
   String str
   Variable byteOffset

   Variable numBytesInString = strlen(str)
   if (byteOffset<0 || byteOffset>=numBytesInString)
      return 0          // Programmer error
   endif

   Variable firstByte = char2num(str[byteOffset]) & 0x00FF

   if (firstByte < 0x80)
      return 1
   endif

   if (firstByte>=0xC2 && firstByte<=0xDF)
      return 2
   endif

   if (firstByte>=0xE0 && firstByte<=0xEF)
      return 3
   endif

   if (firstByte>=0xF0 && firstByte<=0xF4)
      return 4
   endif

   // If we are here, str is not valid UTF-8.
   // Treat the first byte as a 1-byte character.
   // This prevents infinite loops.
   return 1
End

Function UTF8CharactersInString(str)
   String str

   Variable numCharacters = 0

   Variable length = strlen(str)
   Variable byteOffset = 0
   do
      if (byteOffset >= length)
         break
      endif
      Variable numBytesInCharacter = NumBytesInUTF8Character(str, byteOffset)
      if (numBytesInCharacter <= 0)
         Abort "Bug in CharactersInUTF8String"     // Should not happen
      endif
      numCharacters += 1
      byteOffset += numBytesInCharacter
   while(1)

   return numCharacters
End

Function/S UTF8CharacterAtPosition(str, charPos)
   String str
   Variable charPos

   if (charPos < 0)
      // Print "charPos is invalid"             // For debugging only
      return ""
   endif

   Variable length = strlen(str)
   Variable byteOffset = 0
   do
      if (byteOffset >= length)
         // Print "charPos is out-of-range"     // For debugging only
         return ""
      endif
      if (charPos == 0)
         break
      endif
      Variable numBytesInCharacter = NumBytesInUTF8Character(str, byteOffset)
      byteOffset += numBytesInCharacter
      charPos -= 1
   while(1)

   numBytesInCharacter = NumBytesInUTF8Character(str, byteOffset)
   String result = str[byteOffset, byteOffset+numBytesInCharacter-1]
   return result
End

Function DemoCharacterByCharacter()
   String str = "<string containing non-ASCII text>"

   Variable numCharacters = UTF8CharactersInString(str)
   Variable charPos

   for(charPos=0; charPos<numCharacters; charPos+=1)
      String character = UTF8CharacterAtPosition(str, charPos);
      Printf "CharPos=%d, char=\"%s\"\r", charPos, character
   endfor
End
```

See Also: **Text Encodings** on page III-459, **String Indexing** on page IV-16

Working With Binary String Data

Although Igor strings were originally conceived to store human-readable text such (e.g., "Hello, World"), advanced users have found them useful as containers for binary data. Such binary data is usually read from binary files, obtained from the Internet (e.g., via **FetchURL** or **URLRequest**), or obtained via data acquisition.

It is sometimes useful to store the bytes contained in a string into a byte wave so commands that work on waves can be used to analyze the contents. One example is determining the frequency of each letter in a piece of text for a linguistic analysis or analysis of DNA/RNA/protein sequences. Another use is to convert the bytes of a downloaded string into a numeric wave. Igor provides the **StringToUnsignedByteWave** and **WaveDataToString** functions to make facilitate such analyses. These functions were added in Igor Pro 9.00.

Examples can be found at https://www.wavemetrics.com/code-snippet/working-binary-string-data-examples.

Regular Expressions

A regular expression is a pattern that is matched against a subject string from left to right. Regular expressions are used to identify lines of text containing a particular pattern and to extracts substrings matching a particular pattern.

A regular expression can contain regular characters that match the same character in the subject and special characters, called "metacharacters", that match any character, a list of specific characters, or otherwise identify patterns.

The regular expression syntax is based on PCRE — the Perl-Compatible Regular Expression Library.

Igor syntax is similar to regular expressions supported by various UNIX and POSIX `egrep(1)` commands.

See **Regular Expressions References** on page IV-194 for details on PCRE.

Regular Expression Operations and Functions

Here are the Igor operations and functions that work with regular expressions:

Grep

The **Grep** operation identifies lines of text that match a pattern.

The subject is each line of a file or each row of a text wave or each line of the text in the clipboard.

Output is stored in a file or in a text wave or in the clipboard.

Grep Example

```
Function DemoGrep()
   Make/T source={"Monday","Tuesday","Wednesday","Thursday","Friday"}
   Make/T/N=0 dest
   Grep/E="sday" source as dest           // Find rows containing "sday"
   Print dest
End
```

The output from Print is:

```
   dest[0]= {"Tuesday","Wednesday","Thursday"}
```

GrepList

The **GrepList** function identifies items that match a pattern in a string containing a delimited list.

The subject is each item in the input list.

The output is a delimited list returned as the function result.

GrepList Example

```
Function DemoGrepList()
   String source = "Monday;Tuesday;Wednesday;Thursday;Friday"
   String dest = GrepList(source, "sday") // Find items containing "sday"
   Print dest
End
```

The output from Print is:

```
   Tuesday;Wednesday;Thursday;
```

GrepString

The **GrepString** function determines if a particular pattern exists in the input string.

The subject is the input string.

The output is 1 if the input string contains the pattern or 0 if not.

GrepString Example

```
Function DemoGrepString()
   String subject = "123.45"
   String regExp = "[0-9]+"             // Match one or more digits
   Print GrepString(subject,regExp)     // True if subject contains digit(s)
End
```

The output from Print is: 1

SplitString

The **Demo** operation identifies subpatterns in the input string.

The subject is the input string.

Output is stored in one or more string output variables.

SplitString Example

```
Function DemoSplitString()
   String subject = "Thursday, May 7, 2009"
   String regExp = "([[:alpha:]]+), ([[:alpha:]]+) ([[:digit:]]+), ([[:digit:]]+)"
   String dayOfWeek, month, dayOfMonth, year
   SplitString /E=(regExp) subject, dayOfWeek, month, dayOfMonth, year
   Print dayOfWeek, month, dayOfMonth, year
End
```

The output from Print is:

```
   Thursday  May  7  2009
```

Basic Regular Expressions

Here is a **Grep** command that uses "Fred" as the regular expression. "Fred" contains no metacharacters so this command identifies lines of text containing the literal string "Fred". It examines each line from the input file, afile.txt. All lines containing the pattern "Fred" are written to the output file, "FredFile.txt":

```
Grep/P=myPath/E="Fred" "afile.txt" as "FredFile.txt"
```

Character matching is case-sensitive by default. Prepending the Perl 5 modifier (?i) gives a case-insensitive pattern that matches upper and lower-case versions of “Fred”:

```
// Copy lines that contain "Fred", "fred", "FRED", "fREd", etc
Grep/P=myPath/E="(?i)fred" "afile.txt" as "AnyFredFile.txt"
```

To copy lines that do not match the regular expression, use the Grep /E flag with the optional *reverse* parameter set to 1 to reverse the sense of the match:

```
// Copy lines that do NOT contain "Fred", "fred", "fREd", etc
Grep/P=myPath/E={"(?i)fred",1} "afile.txt" as "NotFredFile.txt"
```

Note: Igor doesn't use the Perl opening and closing regular expression delimiter character which is the forward slash. In Perl you would use `"/Fred/"` and `"/(?i)fred/"`.

Regular Expression Metacharacters

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of “metacharacters”, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within brackets, and those that are recognized in brackets. Outside brackets, the metacharacters are as follows:

`\`	General escape character with several uses
`^`	Match start of string
`$`	Match end of string
`.`	Match any character except newline (by default)
	(To match newline, see **Matching Newlines** on page IV-184)
`[`	Start character class definition (for matching one of a set of characters)
`|`	Start of alternative branch (for matching one or the other of two patterns)
`(`	Start subpattern
`)`	End subpattern
`?`	0 or 1 quantifier (for matching 0 or 1 occurrence of a pattern)
	Also extends the meaning of (
	Also quantifier minimizer
`*`	0 or more quantifier (for matching 0 or more occurrence of a pattern)
`+`	1 or more quantifier (for matching 1 or more occurrence of a pattern)
	Also possessive quantifier
`{`	Start min/max quantifier (for matching a number or range of occurrences)

Character Classes in Regular Expressions

A character class is a set of characters and is used to specify that one character of the set should be matched. Character classes are introduced by a left-bracket and terminated by a right-bracket. For example:

```
[abc]       Matches a or b or c

[A-Z]       Matches any character from A to Z

[A-Za-z]    Matches any character from A to Z or a to z.
```

POSIX character classes specify the characters to be matched symbolically. For example:

```
[[:alpha:]] Matches any alphabetic character (A to Z or a to z).

[[:digit:]] Matches 0 to 9.
```

In a character class the only metacharacters are:

`\`	General escape character
`^`	Negate the class, but only if ^ is the first character
`-`	Indicates character range
`[`	POSIX character class (only if followed by POSIX syntax)

`]`	Terminates the character class

Backslash in Regular Expressions

The backslash character has several uses. First, if it is followed by a nonalphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, the * character normally means "match zero or more of the preceding subpattern". If you want to match a `*` character, you write `\*` in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a metacharacter, so it is always safe to precede a nonalphanumeric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write `\\`.

Note: Because Igor also has special uses for backslash (see **Escape Sequences in Strings** on page IV-14), you must double the number of backslashes you would normally use for a Perl or grep pattern. Each pair of backslashes sends one backslash to, say, the **Grep** command.

For example, to copy lines that contain a backslash followed by a *z* character, the Perl pattern would be `"\\z"`, but the equivalent Igor Grep expression would be `/E="\\\\z"`.

Igor's input string parser converts "`\\`" to "`\`" so, when you write `/E="\\\\z"`, the regular expression engine sees `/E="\\z"`.

This difference is important enough that the PCRE and Igor Patterns (using Grep /E syntax) are both shown below when they differ.

Only ASCII numbers and letters have any special meaning after a backslash. All other characters are treated as literals.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between `\Q` and `\E`. This is different from Perl in that `$` and `@` are handled as literals in `\Q`…`\E` sequences in PCRE, whereas in Perl, `$` and `@` cause variable interpolation. Note the following examples:

Igor Pattern	PCRE Pattern	PCRE Matches	Perl Matches
`\\Qabc$xyz\\E`	`\Qabc$xyz\E`	abc\$xyz	abc followed by the contents of \$xyz
`\\Qabc\\$xyz\\E`	`\Qabc\$xyz\E`	abc\\$xyz	abc\\$xyz
`\\Qabc\\E\\$\\Qxyz\\E`	`\Qabc\E\$\Qxyz\E`	abc\$xyz	abc\$xyz

The `\Q`…`\E` sequence is recognized both inside and outside character classes.

Backslash and Nonprinting Characters

A second use of backslash provides a way of encoding nonprinting characters in patterns in a visible manner. There is no restriction on where nonprinting characters can occur, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

Igor Pattern	PCRE Pattern	Character Matched
`\\a`	`\a`	Alarm, that is, the BEL character (hex 07)
`\\cx`	`\cx`	“Control-x”, where x is any character
`\\e`	`\e`	Escape (hex 1B)
`\\f`	`\f`	Formfeed (hex 0C)
`\\n`	`\n`	Newline (hex 0A)
`\\r`	`\r`	Carriage return (hex 0D)

Igor Pattern	PCRE Pattern	Character Matched
`\\t`	`\t`	Tab (hex 09)
`\\ddd`	`\ddd`	Character with octal code ddd, or backreference
`\\xhh`	`\xhh`	Character with hex code hh

Backslash and Nonprinting Characters Arcania

The material in this section is arcane and rarely needed. We recommend that you skip it.

The precise effect of `\cx` is if `x` is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus `\cz` becomes hex 1A, but `\c{` becomes hex 3B, while `\c;` becomes hex 7B.

After `\x`, from zero to two hexadecimal digits are read (letters can be in upper or lower case). If characters other than hexadecimal digits appear between `\x{` and `}`, or if there is no terminating `}`, this form of escape is not recognized. Instead, the initial `\x` will be interpreted as a basic hexadecimal escape, with no following digits, giving a character whose value is zero.

After `\0` up to two further octal digits are read. In both cases, if there are fewer than two digits, just those that are present are used. Thus the sequence `\0\x\07` specifies two binary zeros followed by a BEL character (code value `7`). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a back reference. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE rereads up to three octal digits following the backslash, and generates a single byte from the least significant 8 bits of the value. Any subsequent digits stand for themselves. For example:

Igor Pattern	PCRE Pattern	Character(s) Matched
`\\040`	`\040`	Another way of writing a space
`\\40`	`\40`	A space, provided there are fewer than 40 previous capturing subpatterns
`\\7`	`\7`	Always a back reference
`\\11`	`\11`	Might be a back reference, or another way of writing a tab
`\\011`	`\011`	Always a tab
`\\0113`	`\0113`	A tab followed by the character “3”
`\\113`	`\113`	Might be a back reference, otherwise the character with octal code 113
`\\377`	`\377`	Might be a back reference, otherwise the byte consisting entirely of 1 bits
`\\81`	`\81`	Either a back reference or a binary zero followed by the two characters “8” and “1”

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single byte value can be used both inside and outside character classes. In addition, inside a character class, the sequence `\b` is interpreted as the backspace character (hex 08), and the sequence `\X` is interpreted as the character *X*. Outside a character class, these sequences have different meanings (see **Backslash and Nonprinting Characters** on page IV-179).

Backslash and Generic Character Types

The third use of backslash is for specifying generic character types. The following are always recognized:

Igor Pattern	PCRE Pattern	Character(s) Matched
`\\d`	`\d`	Any decimal digit
`\\D`	`\D`	Any character that is not a decimal digit
`\\s`	`\s`	Any whitespace character
`\\S`	`\S`	Any character that is not a whitespace character
`\\w`	`\w`	Any “word” character
`\\W`	`\W`	Any “nonword” character

Each pair of escape sequences, such as `\d` and `\D`, partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

The `\s` whitespace characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). In Igor Pro 6 `\s` did not match the VT character (code 11).

A “word” character is an underscore or any character that is a letter or digit.

Backslash and Simple Assertions

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described in **Assertions** on page IV-190. The backslashed assertions are:

Igor Pattern	PCRE Pattern	Character(s) Matched
`\\b`	`\b`	At a word boundary
`\\B`	`\B`	Not at a word boundary
`\\A`	`\A`	At start of subject
`\\Z`	`\Z`	At end of subject or before newline at end
`\\z`	`\z`	At end of subject
`\\G`	`\G`	At first matching position in subject

These assertions may not appear in character classes (but note that `\b` has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match `\w` or `\W` (i.e. one matches `\w` and the other matches `\W`), or the start or end of the string if the first or last character matches `\w`, respectively.

While PCRE defines additional simple assertions (`\A`, `\Z`, `\z` and `\G`), they are not any more useful to Igor’s regular expression commands than the `^` and `$` characters.

Circumflex and Dollar

Outside a character class, in the default matching mode, the circumflex character `^` is an assertion that is true only if the current matching point is at the start of the subject string. Inside a character class, circumflex has an entirely different meaning (see **Character Classes and Brackets** on page IV-182).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an “anchored” pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character `$` is an assertion that is true only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

Dot, Period, or Full Stop

Outside a character class, a dot in the pattern matches any one character in the subject, including a nonprinting character, but not (by default) newline. Dot has no special meaning in a character class.

The match option setting (?s) changes the default behavior of dot so that it matches any character including newline. The setting can appear anywhere before the dot in the pattern. See **Matching Newlines** on page IV-184 for details.

Character Classes and Brackets

An opening bracket introduces a character class which is terminated by a closing bracket. A closing bracket on its own is not special. If a closing bracket is required as a member of the class, it must be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class `[aeiou]` matches any English lower case vowel, whereas `[^aeiou]` matches any character that is not an English lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not.

When caseless matching is set, using the `(?i)` match option setting, any letters in a class represent both their upper case and lower case versions, so for example, the caseless pattern `(?i)[aeiou]` matches *A* as well as *a*, and the caseless pattern `(?i)[^aeiou]` does not match *A*.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, `[d-m]` matches any letter between *d* and *m*, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

To include a right-bracket in a range you must use `\]`. As usual, this would be represented in a literal Igor string as `\\]`.

Though it is rarely needed, you can specify a range using octal numbers, for example `[\000-\037]`.

The character types `\d`, `\D`, `\p`, `\P`, `\s`, `\S`, `\w`, and `\W` may also appear in a character class, and add the characters that they match to the class. For example, `[\dABCDEF]` matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class `[^\W_]` matches any letter or digit, but not underscore. The corresponding **Grep** command would begin with

```
Grep/E="[^\\W_]"…
```

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening bracket (only when it can be interpreted as introducing a POSIX class name — see **POSIX Character Classes** on page IV-183), and the terminating closing bracket. However, escaping other nonalphanumeric characters does no harm.

POSIX Character Classes

Perl supports the POSIX notation for character classes. This uses names enclosed by `[:` and `:]` within the enclosing brackets. PCRE also supports this notation. For example,

```
[01[:alpha:]%]
```

matches “0”, “1”, any alphabetic character, or “%”.

The supported class names, all of which must appear between `[:` and `:]` inside a character class specification, are

`alnum`	Letters and digits.
`alpha`	Letters.
`ascii`	Character codes 0 - 127.
`blank`	Space or tab only.
`cntrl`	Control characters.
`digit`	Decimal digits (same as `\d`).
`graph`	Printing characters, excluding space. [:graph:] matches all characters with the Unicode L, M, N, P, S, or Cf properties with a few arcane exceptions.
`lower`	Lower case letters.
`print`	Printing characters, including space. [:print:] matches the same characters as [:graph:] plus space characters that are not controls, that is, characters with the Unicode Zs property.
`punct`	Printing characters, excluding letters and digits. [:punct:] matches all characters that have the Unicode P (punctuation) property, plus those characters whose code points are less than 128 that have the S (Symbol) property.
`space`	White space (not quite the same as `\s`).
`upper`	Upper case letters.
`word`	“Word” characters (same as `\w`).
`xdigit`	Hexadecimal digits.

The “space” characters are horizontal tab (`HT-9`), linefeed (`LF-10`), vertical tab (`VT-11`), formfeed (`FF-12`), carriage-return (`CR-13`), and space (`32`).

The class name `word` is a Perl extension, and `blank` is a GNU extension from Perl 5.8. Another Perl extension is negation that is indicated by a `^` character after the colon. For example,

```
[12[:^digit:]]
```

matches “1”, “2”, or any nondigit. PCRE (and Perl) also recognize the POSIX syntax `[.ch.]` and `[=ch=]` where “ch” is a “collating element”, but these are not supported, and an error is given if they are encountered.

Alternation

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either “gilbert” or “sullivan”. Any number of alternative patterns may be specified, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined in **Subpatterns** on page IV-186), “succeeds” means matching the rest of the main pattern as well as the alternative in the subpattern.

Match Option Settings

Character matching options can be changed from within the pattern by a sequence of Perl option letters enclosed between `(?` and `)`. The option letters are:

Option	PCRE Name	Characters Matched
i	PCRE_CASELESS	Upper and lower case.
m	PCRE_MULTILINE	^ matches start of string and just after a new line (\n). $ matches end of string and just before a new line. Without (?m), ^ and $ match only the start and end of the entire string.
s	PCRE_DOTALL	. matches all characters including newline. Without (?s), . does not match newlines. See **Matching Newlines** on page IV-184.
U	PCRE_UNGREEDY	Reverses the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by ?.

For example, `(?i)` sets caseless matching.

You can unset these options by preceding the letter with a hyphen: (?-i) turns off caseless matching.

You can combine a setting and unsetting such as `(?i-U)`, which sets PCRE_CASELESS while unsetting PCRE_UNGREEDY.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options.

An option change within a subpattern affects only that part of the current pattern that follows it, so

```
(a(?i)b)c
```

matches abc and aBc and no other strings.

Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
(a(?i)b|c)
```

matches “ab”, “aB”, “c”, and “C”, even though when matching “C” the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time.

The other PCRE 5-specific options, such as (?e), (?A), (?D), (?S), (?x) and (?X), are implemented but are not very useful with Igor's regular expression commands.

Matching Newlines

Igor generally uses carriage-return (ASCII code 13) return to indicate a new line in text. This is represented in a literal string by "\r". For example:

```
Print "Hello\rGoodbye"
```

prints two lines of text to the history area.

However, in regular expressions, the newline character is linefeed (ASCII code 10). This is represented in a literal string by "\n". Carriage-return has no special status in regular expressions.

The match option setting `(?s)` changes the default behavior of dot so that it matches any character including newline. The setting can appear anywhere before the dot in the pattern. For example:

```
Function DemoNewline(includeNewline)
   Variable includeNewline

   String subject = "Hello\nGoodbye"   // \n represents newline

   String regExp
   if (includeNewline)
      regExp = "(?s)(.+)"      // One or more of any character
   else
      regExp = "(.+)"          // One or more of any character except newline
   endif

   String result
   SplitString /E=(regExp) subject, result
   Print result
End
```

The output from DemoNewline(0) is:

```
   Hello
```

The output from DemoNewline(1) is:

```
   Hello\nGoodbye
```

Here is a more realistic example:

```
Function DemoDotAll()
   String theString = ExampleHTMLString()

   // This regular expression attempts to extract items from
   // HTML code for an ordered list. It fails because the HTML
   // code contains newlines and, by default, . does not match newlines.
   String regExpFails="<li>(.*?)</li>.*<li>(.*?)</li>"
   String str1, str2
   SplitString/E=regExpFails theString, str1, str2
   Print V_flag, " subpatterns were matched using regExpFails"
   Printf "\"%s\", \"%s\"\r", str1, str2

   // This regular expression works because the "(?s)" match
   // option setting causes . to match newlines.
   String regExpWorks="(?s)<li>(.*?)</li>.*<li>(.*?)</li>"
   SplitString/E=regExpWorks theString, str1, str2
   Print V_flag, " subpatterns were matched using regExpWorks"
   Printf "\"%s\", \"%s\"\r", str1, str2
End
Function/S ExampleHTMLString()   // Returns HTML code containing newlines
   String theString = ""
   theString += "<ol>\n"
   theString += "\t<li>item 1</li>\n"
   theString += "\t<li>item 2</li>\n"
   theString += "<\ol>\n"
   return theString
End
```

```
•DemoDotAll()
  0   subpatterns were matched using regExpFails
  "", ""
  2 subpatterns were matched using regExpWorks
  "item 1", "item 2"
```

Subpatterns

Subpatterns are used to group alternatives, to match a previously-matched pattern again, and to extract match text using **Demo**.

Subpatterns are delimited by parentheses, which can be nested. Turning part of a pattern into a subpattern localizes a set of alternatives. For example, this pattern (which includes two vertical bars signifying alternation):

```
cat(aract|erpillar|)
```

matches one of the words "cat", "cataract", or "caterpillar". Without the parentheses, it would match "cataract", "erpillar" or the empty string.

Named Subpatterns

Specifying subpatterns by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with this difficulty, PCRE supports the naming of subpatterns, something that Perl does not provide. The Python syntax (`?P<`*`name`*`>`...) is used. For example:

```
My (?P<catdog>cat|dog) is cooler than your (?P=catdog)
```

Here catdog is the name of the first and only subpattern. `?P<catdog>` names the subpattern and `(?P=catdog)` matches the previous match for that subpattern.

Names consist of alphanumeric characters and underscores, and must be unique within a pattern.

Named capturing parentheses are still allocated numbers as well as names. This has the same effect as the previous example:

```
My (?P<catdog>cat|dog) is cooler than your \\1
```

Repetition

Repetition is specified by quantifiers:

?	0 or 1 quantifier Example: `[abc]?` - Matches 0 or 1 occurrences of a or b or c
*	0 or more quantifier Example: `[abc]*` - Matches 0 or more occurrences of a or b or c
+	1 or more quantifier Example: `[abc]+` - Matches 1 or more occurrences of a or b or c
{n}	n times quantifier Example: `[abc]{3}` - Matches 3 occurrences of a or b or c
{n,m}	n to m times quantifier Example: `[abc]{3,5}` - Matches 3 to 5 occurrences of a or b or c

Quantifiers can follow any of the following items:

- A literal data character
- The . metacharacter
- The \C escape sequence
- An escape such as \d that matches a single character
- A character class

- A back reference (see **Back References** on page IV-189)
- A parenthesized subpattern (unless it is an assertion)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in braces, separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
z{2,4}
```

matches “zz”, “zzz”, or “zzzz”.

If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
\d{8}
```

matches exactly 8 digits. A left brace that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, `{,6}` is not a quantifier, but a literal string of four characters.

Quantifiers apply to characters rather than to individual data units. Thus, for example, \x{100}{2} matches two characters, each of which is represented by a two-byte sequence in a UTF-8 string. Similarly, \X{3} matches three Unicode extended grapheme clusters, each of which may be several data units long (and they may be of different lengths).

The quantifier `{0}` is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility) the three most common quantifiers have single-character abbreviations:

`*`	Equivalent to `{0,}`
`+`	Equivalent to `{1,}`
`?`	Equivalent to `{0,1}`

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

Quantifier Greediness

By default, the quantifiers are “greedy”, that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between `/*` and `*/` and within the comment, individual `*` and `/` characters may appear. An attempt to match C comments by applying the pattern

```
/\*.*\*/        or    Grep/E="/\\*.*\\*/"
```

to the string

```
/* first comment */  not comment  /* second comment */
```

fails because it matches the entire string owing to the greediness of the `.*` item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*.*?\*/        or    Grep/E="/\\*.*?\\*/"
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches.

Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d           or    Grep/E="\\d??\\d"
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the PCRE_UNGREEDY option `(?U)` is set, the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behavior.

Quantifiers With Subpatterns

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more memory is required for the compiled pattern, in proportion to the size of the minimum or maximum.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[dume]{3}\s*)+   or    Grep/E="(tweedle[dume]{3}\\s*)+"
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
/(a|(b))+/
```

matches "aba" the value of the second captured substring is "b".

Atomic Grouping and Possessive Quantifiers

With both maximizing and minimizing repetition, failure of what follows normally reevaluates the repeated item to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern `\d+foo` when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the `\d+` item, and then with 4, and so on, before ultimately failing. "Atomic grouping" provides the means for specifying that once a subpattern has matched, it is not to be reevaluated in this way.

If we use atomic grouping for the previous example, the matcher would give up immediately on failing to match "foo" the first time. The notation is a kind of special parenthesis, starting with `(?>` as in this example:

```
(?>\d+)foo       or    Grep/E="(?>\\d+)foo"
```

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both `\d+` and `\d+?`

are prepared to adjust the number of digits they match in order to make the rest of the pattern match, `(?>\d+)` can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a “possessive quantifier” can be used. This consists of an additional + character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++foo         or    Grep/E="\\d++foo"
```

Possessive quantifiers are always greedy; the setting of the PCRE_UNGREEDY option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning or processing of a possessive quantifier and the equivalent atomic group.

The possessive quantifier syntax is an extension to the Perl syntax. It originates in Sun’s Java package.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
(\D+|<\d+>)*[!?]  or    Grep/E="(\\D+|<\\d+>)*[!?]"
```

matches an unlimited number of substrings that either consist of nondigits, or digits enclosed in <>, followed by either `!` or `?`. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the internal `\D+` repeat and the external `*` repeat in a large number of ways, and all have to be tried. (The example uses `[!?]` rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed so that it uses an atomic group, like this:

```
((?>\D+)|<\d+>)*[!?]     or    Grep/E="((?>\\D+)|<\\d+>)*[!?]"
```

sequences of nondigits cannot be broken, and failure happens quickly.

Back References

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. See **Backslash and Nonprinting Characters** on page IV-179 for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see **Subpatterns as Subroutines** on page IV-194 for a way of doing that). So the pattern

```
(sens|respons)e and \1ibility or /E="(sens|respons)e and \\1ibility"
```

matches “sense and sensibility” and “response and responsibility”, but not “sense and responsibility”. If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1    or    Grep/E="((?i)rah)\\s+\\1"
```

matches “rah rah” and “RAH RAH”, but not “RAH rah”, even though the original capturing subpattern is matched caselessly.

Back references to named subpatterns use the Python syntax (?P<name>). We could rewrite the above example as follows:

```
(?P<p1>(?i)rah)\s+(?P=p1)   or     Grep/E="(?P<p1>(?i)rah)\\s+(?P=p1)"
```

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match "a" rather than "bc". Because there may be many capturing parentheses in a pattern, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. An empty comment (see **Regular Expression Comments** on page IV-193) can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

```
(a|b\1)+          or     Grep/E="(a|b\\1)+"
```

matches any number of *a*'s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

Assertions

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as \b, \B, ^ and $ are described in **Backslash and Simple Assertions** on page IV-181.

More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it. An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed.

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Lookahead Assertions

Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example,

```
\w+(?=;)          or     Grep/E="\\w+(?=;)"
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

```
(?!foo)bar
```

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve the other effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with (?!) because an empty string always matches, so an assertion that requires there not to be an empty string must always fail.

Lookbehind Assertions

Lookbehind assertions start with `(?<=` for positive assertions and `(?<!` for negative assertions. For example,

```
(?<!foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several alternatives, they do not all have to have the same fixed length. Thus

```
(?<=bullock|donkey)
```

is permitted, but

```
(?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl (at least for 5.8), which requires all branches to match the same length of string. An assertion such as

```
(?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

```
(?<=abc|abde)
```

In some cases, the escape sequence \K (see above) can be used instead of a lookbehind assertion to get round the fixed-length restriction.

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail.

PCRE does not allow the \C escape (which matches a single data unit) to appear in lookbehind assertions, because it makes it impossible to calculate the length of the lookbehind. The \X and \R escapes, which can match different numbers of data units, are also not permitted.

Atomic groups can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```
abcd$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each *a* in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
^.*abcd$
```

the initial `.*` matches the entire string at first, but when this fails (because there is no following *a*), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for *a* covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^(?>.*)(?<=abcd)
```

or, equivalently, using the possessive quantifier syntax,

```
^.*+(?<=abcd)
```

there can be no backtracking for the `.*` item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Using Multiple Assertions

Several assertions (of any sort) may occur in succession. For example,

```
(?<=\d{3})(?<!999)foo    or    Grep/E="(?<=\\d{3})(?<!999)foo"
```

matches “foo” preceded by three digits that are not “999”. Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not “999”. This pattern does not match “foo” preceded by six characters, the first of which are digits and the last three of which are not “999”. For example, it doesn’t match “123abcfoo”. A pattern to do that is

```
(?<=\d{3}...)(?<!999)fooor    Grep/E="(?<=\\d{3}...)(?<!999)foo"
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not “999”.

Assertions can be nested in any combination. For example,

```
(?<=(?<!foo)bar)baz
```

matches an occurrence of “baz” that is preceded by “bar” which in turn is not preceded by “foo”, while

```
(?<=\d{3}(?!999)...)foo or Grep/E=" (?<=\\d{3}(?!999)...)foo"
```

is another pattern that matches “foo” preceded by three digits and any three characters that are not “999”.

Conditional Subpatterns

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are three kinds of condition. If the text between the parentheses consists of a sequence of digits, the condition is satisfied if the capturing subpattern of that number has previously matched. The number must be greater than zero. Consider the following pattern, which contains nonsignificant white space to make it more readable and to divide it into three parts for ease of discussion:

```
( \( )?    [^()]+    (?(1) \) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of nonparentheses, optionally enclosed in parentheses.

If the condition is the string `(R)`, it is satisfied if a recursive call to the pattern or subpattern has been made. At “top level”, the condition is false. This is a PCRE extension. Recursive patterns are described in the next section.

If the condition is not a sequence of digits or `(R)`, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing nonsignificant white space, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
\d{2}-[a-z]{3}-\d{2}  |  \d{2}-\d{2}-\d{2})
```

The condition is a positive lookahead assertion that matches an optional sequence of nonletters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms `dd-aaa-dd` or `dd-dd-dd`, where `aaa` are letters and `dd` are digits.

Regular Expression Comments

The sequence `(?#` marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

Recursive Patterns

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth. Perl provides a facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at runtime, and the code can refer to the expression itself. A Perl pattern to solve the parentheses problem can be created like this:

```
$re = qr{\( (?: (?>[^()]+) | (?p{$re}) )* \)}x;
```

The `(?p{…})` item interpolates Perl code at runtime, and in this case refers recursively to the pattern in which it appears. Obviously, PCRE cannot support the interpolation of Perl code. Instead, it supports some special syntax for recursion of the entire pattern, and also for individual subpattern recursion.

The special item that consists of `(?` followed by a number greater than zero and a closing parenthesis is a recursive call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a “subroutine” call, which is described in **Subpatterns as Subroutines** on page IV-194.) The special item `(?R)` is a recursive call of the entire regular expression.

For example, this PCRE pattern solves the nested parentheses problem (additional nonfunction whitespace has been added to separate the expression into parts):

```
\( ( (?>[^()]+) | (?R) )* \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of nonparentheses, or a recursive match of the pattern itself (that is a correctly parenthesized substring). Finally there is a closing parenthesis.

If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

```
( \( ( (?>[^()]+) | (?1) )* \))
```

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern. In a larger pattern, keeping track of parenthesis numbers can be tricky. It may be more convenient to use named parentheses instead. For this, PCRE uses `(?P>name)`, which is an extension to the Python syntax that PCRE uses for named parentheses (Perl does not provide named parentheses). We could rewrite the above example as follows:

```
(?P<pn> \( ( (?>[^()]+) | (?P>pn) )* \))
```

This particular example pattern contains nested unlimited repeats, and so the use of atomic grouping for matching strings of nonparentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it yields “no match” quickly. However, if atomic grouping is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

At the end of a match, the values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If you want to obtain intermediate values, a callout function can be used (see **Subpatterns as Subroutines** on page IV-194). If the pattern above is matched against

```
(ab(cd)ef)
```

the value for the capturing parentheses is “ef”, which is the last value taken on at the top level. If additional parentheses are added, giving

```
\(( ( (?>[^()]+) | (?R) )* ) \)
    ↑                       ↑
```

the string they capture is “ab(cd)ef”, the contents of the top level parentheses. If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion, which it does by using pcre_malloc, freeing it via pcre_free afterward. If no memory can be obtained, the match fails with the PCRE_ERROR_NOMEMORY error.

Do not confuse the `(?R)` item with the condition `(R)`, which tests for recursion. Consider this pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas any characters are permitted at the outer level.

```
< (?: (?(R) \d++  | [^<>]*+) | (?R)) * >
```

In this pattern, `(?(R)` is the start of a conditional subpattern, with two different alternatives for the recursive and nonrecursive cases. The `(?R)` item is the actual recursive call.

Subpatterns as Subroutines

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. An earlier example pointed out that the pattern

```
(sens|respons)e and \1ibility
```

matches “sense and sensibility” and “response and responsibility”, but not “sense and responsibility”. If instead the pattern

```
(sens|respons)e and (?1)ibility
```

is used, it does match “sense and responsibility” as well as the other two strings. Such references must, however, follow the subpattern to which they refer.

Regular Expressions References

The regular expression syntax supported by **Grep**, **GrepString**, **GrepList**, and **Demo** is based on the *PCRE — Perl-Compatible Regular Expression Library* by Philip Hazel, University of Cambridge, Cambridge, England. The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5.

Visit <`http://pcre.org/`> for more information about the PCRE library. The description of regular expressions above is taken from the PCRE documentation.

A good introductory book on regular expressions is: Forta, Ben, *Regular Expressions in 10 Minutes*, Sams Publishing, 2004.

A good comprehensive book on regular expressions is: Friedl, Jeffrey E. F., *Mastering Regular Expressions*, 2nd ed., 492 pp., O’Reilly Media, 2002.

A good web site is: http://www.regular-expressions.info

Working with Files

Here are the built-in operations that you can use to read from or write to files:

Operation	What It Does
Open	Opens an existing file for reading or writing. Can also create a new file. Can also append to an existing file. Returns a file reference number that you must pass to the other file operations. Use Open/D to present a dialog that allows the user to choose a file without actually opening the file.
fprintf	Writes formatted text to an open file.

Operation	What It Does
wfprintf	Writes wave data as formatted text to an open file.
FGetPos	Gets the position at which the next file read or write will be done.
FSetPos	Sets the position at which the next file read or write will be done.
FStatus	Given a file reference number, returns miscellaneous information about the file.
FBinRead	Reads binary data from a file into an Igor string, variable or wave. This is used mostly to read nonstandard file formats.
FBinWrite	Writes binary data from an Igor string, variable or wave. This is used mostly to write nonstandard file formats.
FReadLine	Reads a line of text from a text file into an Igor string variable. This can be used to parse an arbitrary format text file.
Close	Closes a file opened by the Open operation.

Before working with a file, you must use the Open operation to obtain a file reference number that you use with all the remaining commands. The Open operation creates new files, appends data to an existing file, or reads data from an existing file. There is no facility to deal with the resource fork of a Macintosh file.

Sometimes you may write a procedure that uses the Open operation and the Close operation but, because of an error, the Close operation never gets to execute. You then correct the procedure and want to rerun it. The file will still be open because the Close operation never got a chance to run. In this case, execute:

```
Close/A
```

from the command line to close all open files.

Finding Files

Two functions are provided to help you to determine what files exist in a particular folder:

- **TextFile**
- **IndexedFile**

IndexedFile is a more general version of TextFile.

You can also use the Open operation with the /D flag to present an open dialog.

Other File– and Folder–Related Operations and Functions

Igor Pro also supports a number of operations and functions for file or folder manipulation:

CopyFile	CopyFolder
DeleteFile	DeleteFolder
GetFileFolderInfo	SetFileFolderInfo
MoveFile	MoveFolder
CreateAliasShortcut	
NewPath	PathInfo
ParseFilePath	

Warning: Use caution when writing code that deletes or moves files or folders. These actions are not undoable.

Because the DeleteFolder, CopyFolder and MoveFolder operations have the potential to do a lot of damage if used incorrectly, they require user permission before overwriting or deleting a folder. The user controls the permission process using the Miscellaneous Settings dialog (Misc menu).

Writing to a Text File

You can generate output text files from Igor procedures or from the command line in formats acceptable to other programs. To do this, you need to use the **Open** and **Close** operations and the **fprintf** operation (page V-260) and the **wfprintf** operation (page V-1093).

The following commands illustrate how you could use these operations to create an output text file. Each operation is described in detail in following sections:

```
Function WriteFileDemo(pathName, fileName)
   String pathName   // Name of Igor symbolic path (see Symbolic Paths)
   String fileName   // File name, partial path or full path

   Variable refNum
   Open refNum as "A New File"                // Create and open file
   fprintf refNum, "wave1\twave2\twave3\r"    // Write column headers
   Wave wave1, wave2, wave3
   wfprintf refNum, "" wave1, wave2, wave3    // Write wave data
   Close refNum                               // Close file
End
```

Open and Close Operations

You use the **Open** operation (page V-716) to open a file. For our purposes, the syntax of the Open operation is:

```
Open [/R/A/P=pathName/M=messageStr] variableName [as "filename"]
```

variableName is the name of a numeric variable. The Open operation puts a file reference number in that variable. You use the reference number to access the file after you've opened it.

A file specifications consists of a path (directions for finding a folder) and a file name. In the Open operation, you can specify the path in three ways:

Using a full path as the filename parameter:

```
Open refNum as “hd:Data:Run123.dat”
```

Or using a symbolic path name and a file name:

```
Open/P=DataPath refNum as “Run123.dat”
```

Or using a symbolic path name and a partial path including the file name:

```
Open/P=HDPath refNum as “:Data:Run123.dat”
```

A symbolic path is a short name that refers to a folder on disk. See **Symbolic Paths** on page II-22.

If you do not provide enough information to find the folder and file of interest, Igor displays a dialog which lets you select the file to open. If you supply sufficient information, Igor open the file without displaying the dialog.

To open an existing file for reading, use the /R flag. To append new data to an existing file, use the /A flag. If you omit both of these flags and if the file already exists, you overwrite any data that is in the file.

If you open a file for writing (you don't use /R) then, if there exists a file with the specified name, Igor opens the file and overwrites the existing data in the file. If there is no file with the specified name, Igor creates a new file.

Warning: If you're not careful you can inadvertently lose data using the Open operation by opening for writing without using the /A flag. To avoid this, use the /R (open for read) or /A (append) flags.

Wave Reference Functions

It is common to write a user-defined function that operates on all of the waves in a data folder, on the waves displayed in a graph or table, or on a wave identified by a cursor in a graph. For these purposes, you need to use wave reference functions.

Wave reference functions are built-in Igor functions that return a reference that can be used in a user-defined function. Here is an example that works on the top graph. Cursors are assumed to be placed on a region of a trace in the graph.

```
Function WaveAverageBetweenCursors()
   WAVE/Z w = CsrWaveRef(A)
   if (!WaveExists(w))         // Cursor is not on any wave.
      return NaN
   endif

   Variable xA = xcsr(A)
   Variable xB = xcsr(B)
   Variable avg = mean(w, xA, xB)

   return avg
End
```

CsrWaveRef returns a wave reference that identifies the wave a cursor is on.

An older function, CsrWave, returns the *name* of the wave the cursor is on. It would be tempting to use this to determine the wave the cursor is on, but it would be incorrect. The name of a wave by itself does not uniquely identify a wave because it does not specify the data folder in which the wave resides. For this reason, we usually need to use the wave reference function CsrWaveRef instead of CsrWave.

This example uses a wave reference function to operate on the waves displayed in a graph:

```
Function SmoothWavesInGraph()
   String list = TraceNameList("", ";", 1)
   String traceName
   Variable index = 0
   do
      traceName = StringFromList(index, list)
      if (strlen(traceName) == 0)
         break          // No more traces.
      endif
      WAVE w = TraceNameToWaveRef("", traceName)
      Smooth 5, w
      index += 1
   while(1)
End
```

Use **WaveRefIndexedDFR** to iterate over the waves in a given data folder.

Here are the wave reference functions. See Chapter V-1, **Igor Reference**, for details on each of them.

Function	Comment
CsrWaveRef	Returns a reference to the Y wave to which a cursor is attached.
CsrXWaveRef	Returns a reference to the X wave when a cursor is attached to an XY pair.
WaveRefIndexedDFR	Returns a reference to a wave in the specified data folder.
WaveRefIndexed	Returns a reference to a wave in a graph or table or to a wave in the current data folder.
XWaveRefFromTrace	Returns a reference to an X wave in a graph. Used with the output of TraceNameList.

Function	Comment
ContourNameToWaveRef	Returns a reference to a wave displayed as a contour plot. Used with the output of ContourNameList.
ImageNameToWaveRef	Returns a reference to a wave displayed as an image. Used with the output of ImageNameList.
TraceNameToWaveRef	Returns a reference to a wave displayed as a waveform or as the Y wave of an XY pair in a graph. Used with the output of TraceNameList.
TagWaveRef	Returns a reference to a wave to which a tag is attached in a graph. Used in creating a smart tag.
WaveRefsEqual	Returns the truth two wave references are the same.
WaveRefWaveToList	Returns a semicolon-separated string list containing the full or partial path to the wave referenced by each element of the input wave reference wave.
ListToWaveRefWave	Returns a free wave containing a wave reference for each of the waves in the semicolon-separated input string list.

This simple example illustrates the use of the CsrWaveRef wave reference function:

```
Function/WAVE CursorAWave()
   WAVE/Z w = CsrWaveRef(A)
   return w
End

Function DemoCursorAWave()
   WAVE/Z w = $CursorAWave()
   if (WaveExists(w)==0)
      Print "oops: no wave"
   else
      Printf "Cursor A is on the wave '%s'\r", NameOfWave(w)
   endif
End
```

Processing Lists of Waves

Igor users often want to use a string list of waves in places where Igor is looking for just the name of a single wave. For example, they would like to do this:

```
Display "wave0;wave1;wave2"
```

or, more generally:

```
Function DisplayListOfWaves(list)
   String list                // e.g., "wave0;wave1;wave2"

   Display $list
End
```

Unfortunately, Igor can't handle this. However, there are techniques for achieving the same result.

Graphing a List of Waves

This example illustrates the basic technique for processing a list of waves.

```
Function DisplayWaveList(list)
   String list             // A semicolon-separated list.

   Variable index = 0
   do
      // Get the next wave name
      String name = StringFromList(index, list)
      if (strlen(name) == 0)
```

```
         break                       // Ran out of waves
      endif

      Wave w = $name
      if (index == 0)                // Is this the first wave?
         Display w
      else
         AppendToGraph w
      endif
      index += 1
   while (1)         // Loop until break above
End
```

To make a graph of all of the waves in the current data folder, you could execute

```
DisplayWaveList(WaveList("*", ";", ""))
```

Operating on the Traces in a Graph

In a previous section, we showed an example that operates on the waves displayed in a graph. It used a wave reference function, TraceNameToWaveRef. If you want to write a function that operates on *traces* in a graph, you would *not* use wave reference functions. That's because Igor operations that operate on *traces* expect *trace names*, not wave references. For example:

```
Function GrayOutTracesInGraph()
   String list = TraceNameList("", ";", 1)
   Variable index = 0
   do
      String traceName = StringFromList(index, list)
      if (strlen(traceName) == 0)
         break                       // No more traces.
      endif

      // WRONG: ModifyGraph expects a trace name and w is not a trace name
      WAVE w = TraceNameToWaveRef("", traceName)
      ModifyGraph rgb(w)=(50000,50000,50000)

      // RIGHT
      ModifyGraph rgb($traceName)=(50000,50000,50000)

      index += 1
   while(1)
End
```

Using a Fixed-Length List

In the previous examples, the number of waves in the list was unimportant and all of the waves in the list served the same purpose. In this example, the list has a fixed number of waves and each wave has a different purpose.

```
Function DoLineFit(list)
   String list          // List of waves names: source, weight

   // Pick out the expected wave names

   String sourceStr = StringFromList(0, list)
   Wave source = $sourceStr

   String weightStr = StringFromList(2, list)
   Wave weight = $weightStr

   CurveFit line source /D /W=weight
End
```

You could invoke this function as follows:

```
DoLineFit("wave0;wave1;")
```

For most purposes, it is better to design the function to take wave reference parameters rather than a string list.

Operating on Qualified Waves

This example illustrates how to operate on waves that match a certain criterion. It is broken into two functions - one that creates the list of qualified waves and a second that operates on them. This organization gives us a general purpose routine (ListOfMatrices) that we would not have if we wrote the whole thing as one function.

```
Function/S ListOfMatrices()
   String list = ""
   Variable index=0
   do
      WAVE/Z w=WaveRefIndexedDFR(:,index)     // Get next wave.
      if (WaveExists(w) == 0)
         break                                // No more waves.
      endif
      if (WaveDims(w) == 2)
         // Found matrix. Add to list with separator.
         list += NameOfWave(w) + ";"
      endif
      index += 1
   while(1)             // Loop till break above.
   return list
End

Function ChooseAndDisplayMatrix()
   String theList = ListOfMatrices()

   String theMatrix
   Prompt theMatrix, "Matrix to display:", popup theList
   DoPrompt "Display Matrix", theMatrix
   if (V_Flag != 0)
      return -1
   endif

   WAVE m = $theMatrix
   NewImage m
End
```

In the preceding example, we needed a list of wave names in a string to use in a Prompt statement. More often we want a list of wave references on which to operate. The next example illustrates how to do this using a general purpose routine that returns a list of wave references in a free wave:

```
// Returns a free wave containing wave references
// for each 2D wave in the current data folder
Function/WAVE GetMatrixWavesInCDF()
   Variable numWavesInCDF = CountObjects(":", 1)
   Make/FREE/WAVE/N=(numWavesInCDF) list

   Variable numMatrixWaves = 0
   Variable i
   for(i=0; i<numWavesInCDF; i+=1)
      WAVE w = WaveRefIndexedDFR(:,i)
      Variable numDimensions = WaveDims(w)
      if (numDimensions == 2)
         list[numMatrixWaves] = w
         numMatrixWaves += 1
      endif
   endfor

   Redimension/N=(numMatrixWaves) list
   return list       // Ownership of the wave is passed to the calling routine
End

Function DemoMatrixWaveList()
   Wave/WAVE freeList = GetMatrixWavesInCDF()   // We now own the freeList wave
   Variable numMatrixWaves = numpnts(freeList)
   Printf "Number of matrix waves in current data folder: %d\r", numMatrixWaves

   Variable i
   for(i=0; i<numMatrixWaves; i+=1)
      Wave w = freeList[i]
      String name = NameOfWave(w)
      name = PossiblyQuoteName(name)
      Printf "Wave %d: %s\r", i, name
   endfor

   // freeList is automatically killed when the function returns
End
```

The ExecuteCmdOnList Function

The ExecuteCmdOnList function is implemented by a WaveMetrics procedure file and executes any command for each wave in the list. For example, the following commands do a WaveStats operation on each wave.

```
Make wave0=gnoise(1), wave1=gnoise(10), wave2=gnoise(100)
ExecuteCmdOnList("WaveStats %s", "wave0;wave1;wave2;")
```

The ExecuteCmdOnList function is supplied in the “Execute Cmd On List” procedure file. See **The Include Statement** on page IV-166 for instructions on including a procedure file.

This technique is on the kludgy side. If you can achieve your goal in a more straightforward fashion without heroic efforts, you should use regular programming techniques.

The Execute Operation

Execute is a built-in operation that executes a string expression as if you had typed it into the command line. The main purpose of Execute is to get around the restrictions on calling macros and external operations from user functions.

We try to avoid using Execute because it makes code obscure and difficult to debug. If you can write a procedure without Execute, do it. However, there are some cases where using Execute can save pages of code or achieve something that would otherwise be impossible. For example, the TileWindow operation can not be directly called from a user-defined function and therefore must be called using Execute.

It is a good idea to compose the command to be executed in a local string variable and then pass that string to the Execute operation. Use this to print the string to the history for debugging. For example:

```
Function DemoExecute(list)
   String list    // Semicolon-separated list of names of windows to tile
   list = ReplaceString(";", list, ",")    // We need commas for TileWindows
   list = RemoveEnding(list, ",")          // Remove trailing comma, if any
   String cmd
   sprintf cmd, "TileWindows %s", list
   Print cmd                               // For debugging only
   Execute cmd
End
```

When you use Execute, you must be especially careful in the handling of wave names. See **Programming with Liberal Names** on page IV-168 for details.

When Execute runs, it is as if you typed a command in the command line. Local variables in macros and functions are not accessible. The example in **Calling an External Operation From a User-Defined Function** on page IV-202 shows how to use the sprintf operation to solve this problem.

Using a Macro From a User-Defined Function

A macro can not be called directly from a user function. To do so, we must use Execute. This is a trivial example for which we would normally not resort to Execute but which clearly illustrates the technique.

```
Function Example()
   Make wave0=enoise(1)
   Variable/G V_avg              // Create a global
   Execute "MyMacro(\"wave0\")"  // Invokes MyMacro("wave0")
   return V_avg
End

Macro MyMacro(wv)
   String wv
   WaveStats $wv     // Sets global V_avg and 9 other local vars
End
```

Execute does not supply good error messages. If the macro generates an error, you may get a cryptic message. Therefore, debug the macro *before* you call it with the Execute operation.

Calling an External Operation From a User-Defined Function

Prior to Igor Pro 5, external operations could not be called directly from user-defined functions and had to be called via Execute. Now it is possible to write an external operation so that it can be called directly. However, very old XOPs that have not been updated still need to be called through Execute. This example shows how to do it.

If you attempt to directly use an external operation which does not support it, Igor displays an error dialog telling you to use Execute for that operation.

The external operation in this case is VDTWrite which sends text to the serial port. It is implemented by the VDT XOP (no longer shipped as of Igor7).

```
Function SetupVoltmeter(range)
   Variable range    // .1, .2, .5, 1, 2, 5 or 10 volts

   String voltmeterCmd
   sprintf voltmeterCmd, "DVM volts=%g", range
   String vdtCmd
   sprintf vdtCmd "VDTWrite \"%s\"\r\n", voltmeterCmd
   Execute vdtCmd
End
```

In this case, we are sending the command to a voltmeter that expects something like:

```
DVM volts=.2<CR><LF>
```

to set the voltmeter to the 0.2 volt range.

The parameter that we send to the Execute operation is:

```
VDTWrite "DVM volts=.2\r\n"
```

The backslashes used in the second sprintf call insert two quotation marks, a carriage return, and a linefeed in the command about to be executed.

A newer VDT2 XOP exists which includes external operations that *can* be directly called from user-functions. Thus, new programming should use the VDT2 XOP and will not need to use Execute.

Other Uses of Execute

Execute can also accept as an argument a string variable containing commands that are algorithmically constructed. Here is a simple example:

```
Function Fit(w, fitType)
   WAVE w               // Source wave
   String fitType       // One of the built-in Igor fit types

   String name = NameOfWave(w)
   name = PossiblyQuoteName(name)   // Liberal names need quotes

   String cmd
   sprintf cmd, "CurveFit %s %s", fitType, name
   Execute cmd
End
```

Use this function to do any of the built-in fits on the specified wave. Without using the Execute operation, we would have to write it as follows:

```
Function Fit(w, fitType)
   WAVE w                  // Source wave
   String fitType          // One of the built-in Igor fit types

   strswitch(fitType)
      case "line":
         CurveFit line w
         break

      case "exp":
         CurveFit exp w
         break

      <and so on for each fit type>
   endswitch
End
```

Note the use of sprintf to prepare the string containing the command to be executed. The following would not work because when Execute runs, local variables and parameters are not accessible:

```
Execute "CurveFit fitType name"
```

Deferred Execution Using the Operation Queue

It is sometimes necessary to execute a command after the current function has finished. This is done using the Execute/P operation. For details, see **Operation Queue** on page IV-278.

Procedures and Preferences

As explained under **Preferences** on page III-515, Igor allows you to set many preferences. Most of the preferences control the style of new objects, including new graphs, traces and axes added to graphs, new tables, columns added to tables, and so on.

Preferences are usually used only for manual “point-and-click” operation. We usually don’t want preferences to affect the behavior of procedures. The reason for this is that we want a procedure to do the same thing no matter who runs it. Also, we want it to do the same thing tomorrow as it does today. If we allowed preferences to take effect during procedure execution, a change in preferences could change the effect of a procedure, making it unpredictable.

By default, preferences do not take effect during procedure execution. If you want to override the default behavior, you can use the Preferences operation. From within a procedure, you can execute `Preferences 1` to turn preferences on. This affects the procedure and any subroutines that it calls. It stays in effect until you execute `Preferences 0`.

When a *macro* ends, the state of preferences reverts to what it was when that macro started. If you change the preference setting within a *function*, the preferences state does *not* revert when that function ends. You must turn preferences on, save the old preferences state, execute Igor operations, and then restore the preferences state. For example:

```
Function DisplayWithPreferences(w)
   Wave w

   Variable oldPrefState
   Preferences 1                     // Turn preferences on
   oldPrefState = V_Flag             // Save the old state

   Display w                         // Create graph using preferences

   Preferences oldPrefState          // Restore old prefs state
End
```

Experiment Initialization Procedures

When Igor loads an experiment, it checks to see if there are any commands in the procedure window before the first macro, function or menu declaration. If there are such commands Igor executes them. This provides a way for you to initialize things. These initialization commands can invoke procedures that are declared later in the procedure window.

Also see **BeforeFileOpenHook** on page IV-287 and **IgorStartOrNewHook** on page IV-292 for other initialization methods.

Procedure Subtypes

A procedure subtype identifies the purpose for which a particular procedure is intended and the appropriate menu from which it can be chosen. For example, the Graph subtype puts a procedure in the Graph Macros submenu of the Windows menu.

```
Window Graph0() : Graph
   PauseUpdate; Silent 1      // building window...
   Display/W=(5,42,400,250) wave0,wave1,wave2
   <more commands>
End
```

When Igor automatically creates a procedure, for example when you close and save a graph, it uses the appropriate subtype. When you create a curve fitting function using the Curve Fitting dialog, the dialog automatically uses the FitFunc subtype. You usually don't need to use subtypes for procedures that you write.

This table shows the available subtypes and how they are used.

Subtype	Effect	Available for
Graph	Displayed in Graph Macros submenu.	Macros
GraphStyle	Displayed in Graph Macros submenu and in Style pop-up menu in New Graph dialog.	Macros
GraphMarquee	Displayed in graph marquee. This keyword is no longer recommended. See **Marquee Menu as Input Device** on page IV-163 for details.	Macros and functions
CursorStyle	Displayed in Style Function submenu of cursor pop-up menu in graph info pane.	Functions
DrawUserShape	Marks a function as suitable for drawing a user-defined drawing object. See **DrawUserShape**.	Functions
GridStyle	Displayed in Grid-Style submenu of mover pop-up menu in drawing tool palette.	Functions

Subtype	Effect	Available for
`Table`	Displayed in Table Macros submenu.	Macros
`TableStyle`	Displayed in Table Macros submenu and in Style pop-up menu in New Table dialog.	Macros
`Layout`	Displayed in Layout Macros submenu.	Macros
`LayoutStyle`	Displayed in Layout Macros submenu and in Style pop-up menu in New Layout dialog.	Macros
`LayoutMarquee`	Displayed in layout marquee. This keyword is no longer recommended. See **Marquee Menu as Input Device** on page IV-163 for details.	Macros and functions
`ListBoxControl`	Displayed in Procedure pop-up menu in ListBox Control dialog.	Macros and functions
`Panel`	Displayed in Panel Macros submenu.	Macros
`GizmoPlot`	Displayed in Other Macros submenu.	Macros
`CameraWindow`	Displayed in Other Macros submenu.	Macros
`FitFunc`	Displayed in Function pop-up menu in Curve Fitting dialog.	Functions
`ButtonControl`	Displayed in Procedure pop-up menu in Button Control dialog.	Macros and functions
`CheckBoxControl`	Displayed in Procedure pop-up menu in Checkbox Control dialog.	Macros and functions
`PopupMenuControl`	Displayed in Procedure pop-up menu in PopupMenu Control dialog.	Macros and functions
`SetVariableControl`	Displayed in Procedure pop-up menu in SetVariable Control dialog.	Macros and functions
`SliderControl`	Displayed in Procedure pop-up menu in Slider Control dialog.	Macros and functions
`TabControl`	Displayed in Procedure pop-up menu in Tab Control dialog.	Macros and functions
`CDFFunc`	Displayed in the Kolmogorov-Smirnov Test dialog. See **StatsKSTest** for details.	Functions

Memory Considerations

Running out of memory is usually not an issue unless you load gigabytes of data into memory at one time. If this is true in your case, make sure to run IGOR64 (the 64-bit version of Igor) rather than IGOR32 (the 32-bit version). IGOR32 is provided only for users who rely on 32-bit XOPs that have not yet been ported to 64 bits and is available on Windows only.

On most systems, IGOR32 can access 4 GB of virtual memory. The limits for IGOR64 are much higher and depend on your operating system.

If memory becomes fragmented, you may get unexpected out-of-memory errors. This is much more likely in IGOR32 than IGOR64.

See **Memory Management** on page III-512 for further information.

Wave Reference Counting

Igor uses reference counting to determine when a wave is no longer referenced anywhere and memory can be safely deallocated.

Because memory is not deallocated until all references to a given wave are gone, memory may not be freed when you think. Consider the function:

```
Function myfunc()
   Make/N=10E6 bigwave
   // do stuff
   FunctionThatKillsBigwave()
   // do more stuff
End
```

The memory allocated for bigwave is not deallocated until the function returns because Make creates an automatic wave reference variable which, by default, exists until the function returns. To free memory before the function returns, use the **WAVEClear**:

```
Function myfunc()
   Make/N=10E6 bigwave
   // do stuff
   FunctionThatKillsBigwave()
   WAVEClear bigwave
   // do more stuff
End
```

The WAVEClear command takes a list of WAVE reference variables and stores NULL into them. It is the same as:

```
WAVE/Z wavref= $""
```

Detecting Wave Leaks

Igor Pro 9.00 introduced two ways to detect wave leaks: **IgorInfo**(16) and **Wave Tracking**. These are tools for advanced Igor programmers. Wave tracking is discussed in the next section.

IgorInfo(16) is a quick and lightweight way to detect possible wave leaks. Unlike wave tracking, IgorInfo(16) can be used independently by different procedure packages without interference among them.

IgorInfo(16) returns the total number of waves of all kinds in existence at any given time. You can use it like this:

```
Function DetectWaveLeaks()
   int originalNumberOfWaves = str2num(IgorInfo(16))

   <Do a lot of processing that you suspect might leak waves>

   int newNumberOfWaves = str2num(IgorInfo(16))
   int change = newNumberOfWaves - originalNumberOfWaves
   if (change > 0)
      Beep
      Printf "Possible leak: Number of waves changed by %d\r", change
   endif
End
```

IgorInfo(16) may sometimes create false positives. For example, the Igor debugger and the Igor Data Browser sometimes create hidden waves for their internal use. An example is the debugger if you choose Show Waves in Graph from the Local Waves, Strings, and SVARs popup menu. This can cause a discrepancy in the number of waves at the start and end of a function but it is not a real leak as Igor will eventually reuse or kill the wave.

Because of the possibility of false positives, you should consider a discrepancy such as detected by the DetectWaveLeaks function above evidence of a possible leak only. To confirm the leak, rerun the function without the use of the debugger or any other user-interface interactions. If it still appears to be a leak, use Wave Tracking to investigate the cause.

Wave Tracking

The **WaveTracking** operation is a tool for detecting and investigating wave leaks. WaveTracking was introduced in Igor Pro 9.00.

A fast alternative and lightweight alternative to wave tracking is **IgorInfo**(16). For details, see **Detecting Wave Leaks** above.

Igor waves fall into three categories and each category can be tracked separately: global, free, and local.

Global waves are waves stored in the main data folder hierarchy, i.e., in the root data folder or one of its descendants. These are the usual waves that you work with on a regular basis. They are global in the sense that they are persistent (you must explicitly kill them) and are accessible from any context (any function or from the command line) at any time.

Free waves are waves that are not contained in any data folder. You create them in user-defined functions via the **NewFreeWave** function, the **Make**/FREE operation, or other operations that support the /FREE flag. You use a wave reference variable to access a free wave (see **Wave References** on page IV-71). When the last wave reference for a given free wave has gone out of scope or is cleared by **WAVEClear**, Igor automatically kills the wave.

Local waves are waves that are stored in a free data folder or a descendant of a free data folder. They are local in the sense that they are automatically killed and are accessible only if you hold a data folder reference for the containing data folder or its ancestor.

Local waves become free waves when the containing data folder is destroyed if a wave reference still remains. See **Free Wave Lifetime** on page IV-92 and **Free Data Folder Lifetime** on page IV-97.

The topic **Free Wave Leaks** on page IV-94 illustrates how you can permanently leak a free wave. After they are leaked, there is no way to access them, so the only way to recover the memory is to restart Igor. You can use the WaveTracking operation to detect and investigate such leaks.

Sometimes you may have a backlog of global waves that were created for temporary use but not deleted. These waves are stored in memory and on disk when you save an experiment. You can use the WaveTracking operation to identify and clean up the backlog.

The **WaveTracking** operation allows you to keep track of the number of waves created but not killed over a given period of time and to get information on those waves. When Igor starts up, tracking is turned off. You must turn it on, as illustrated below, before you run code that you suspect is causing a problem.

WaveTracking provides two modes. Counter mode tells you the number waves of a given category created minus the number killed since counting started. Tracker mode keeps a list of every wave created but not yet been killed since tracking started. Tracker mode adds to the time it takes to create and kill waves, so counter mode is useful for detecting leaks while tracker mode is useful for investigating leaks once they are detected.

To illustrate, we will use the Leaks function defined here. It demonstrates how free wave leaks can occur and in the process leaks two free waves.

```
Function/WAVE GenerateFree()
   return NewFreeWave(2,3)
End

Function Leaks()
   Duplicate/O GenerateFree(),dummy             // This leaks
   Variable maxVal = WaveMax(GenerateFree())    // So does this
End
```

If you want to follow along with the example, copy these functions to your procedure window and then close it.

First we turn on the free wave counter:

```
WaveTracking/FREE counter
```

Turning wave tracking on resets the counter even if it was already on.

This command demonstrates that no free waves were leaked since counting was turned on:

```
WaveTracking/FREE count; Print V_numWaves
  0
```

Now we run the Leaks function and check the count again:

```
Leaks(); WaveTracking/FREE count; Print V_numWaves
  2
```

After running the Leaks function, there are two free waves that did not exist before. There is no way at this point to get rid of them, but at least now you know about it, and perhaps you can figure out what leaked them.

The counter tracking mode does a simple increment when a wave of a given category is created, and a decrement when a wave is killed. If you kill a wave that was created before the tracking counter was started, the count can go negative.

Naturally the Leaks function is much less complex than typical code. It can be quite difficult to figure out where the leaked waves are created. To help with that, you can use the tracker mode and give your free waves names. If you want to follow along with the example, copy these functions to your procedure window and then close it:

```
Function/WAVE GenerateFreeNamed()
   return NewFreeWave(2,3, "MadeByGenerateFree")
End

Function LeaksNamed()
   Duplicate/O GenerateFreeNamed(),dummy           // This leaks
   Variable maxVal = WaveMax(GenerateFreeNamed())  // So does this
End
```

We now run LeaksNamed with WaveTracking in tracker mode:

```
WaveTracking/FREE tracker
LeaksNamed()
WaveTracking/FREE dump
Since tracking began, 2 free wave(s) have been created and not killed.
Wave 'MadeByGenerateFree'; refcount: 0
Wave 'MadeByGenerateFree'; refcount: 0
```

This time, instead of printing the count of free waves, we used "dump" to get a list of the free waves. The dump keyword prints a list in the history area showing the name of each free wave and its reference count. The names tell you that those were waves made by the GenerateFreeNamed function.

In the example in **Wave Reference MultiThread Example** on page IV-327, a WAVE wave, named ww, is created at the start of the Demo function. ww is filled with free waves by the MultiThread call. At the end of the function, KillWaves is called to kill ww which automatically kills the free waves. If you omitted the KillWaves call at the end, this would create a large leak. You could investigate it using `WaveTracking/FREE tracker` and `WaveTracking/FREE dump`.

Creating Igor Extensions

Igor includes a feature that allows a C or C++ programmer to extend its capabilities. Using Apple's Xcode or Microsoft Visual C++, a programmer can add command line operations and functions to Igor. These are called external operations and external functions. Experienced programmers can also add menu items, dialogs and windows.

A file containing external operations or external functions is called an “XOP file” or “XOP” (pronounced “ex-op”) for short.

Here are some things for which you might want to write an XOP:

- To do number-crunching on waves.
- To import data from a file format that Igor does not support.
- To acquire data directly into Igor waves.
- To communicate with a remote computer, instrument or database.

The main reasons for writing something as an XOP instead of writing it as an Igor procedure are:

- It needs to be blazing fast.
- You already have a lot of C code that you want to reuse.
- You need to call drivers for data acquisition.
- It requires programming techniques that Igor doesn’t support.
- You want to add your own dialogs or windows.

Writing an XOP that does number-crunching is considerably easier than writing a stand-alone program. You don’t need to know anything about Macintosh or Windows APIs.

If you are a C or C++ programmer and would like to extend Igor with your own XOP, you need to purchase the Igor External Operations Toolkit from WaveMetrics. This toolkit contains documentation on writing XOPs as well as the source code for several WaveMetrics sample XOPs. It supplies a large library of routines that enable communication between Igor and the external code. You will also need the a recent version of Xcode, or Microsoft Visual C++.

Chapter IV-8

Debugging

Debugging Procedures

There are two techniques for debugging procedures in Igor:

- Using print statements
- Using the symbolic debugger

For most situations, the symbolic debugger is the most effective tool. In some cases, a strategically placed print statement is sufficient.

Debugging With Print Statements

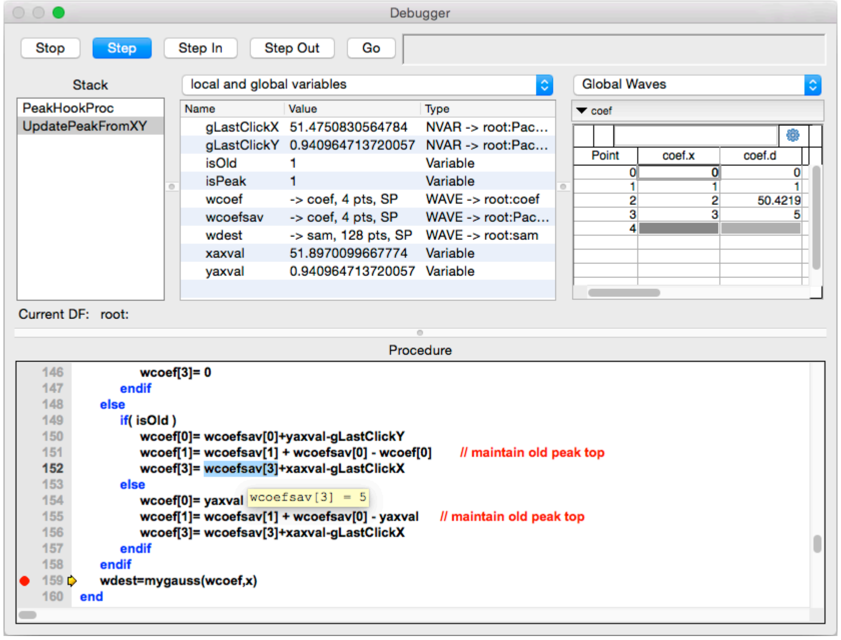
This technique involves putting print statements at a certain point in a procedure to display debugging messages in Igor's history area. In this example, we use Printf to display the value of parameters to a function and then Print to display the function result.

```
Function Test(w, num, str)
   Wave w
   Variable num
   String str

   Printf "Wave=%s, num=%g, str=%s\r", NameOfWave(w), num, str

   <body of function>

   Print result
   return result
End
```

See **Creating Formatted Text** on page IV-259 for details on the Printf operation.

The Debugger

When a procedure doesn't produce the results you want, you can use Igor's built-in debugger to observe the execution of macros and user-defined functions while single-stepping through the lines of code.



Stack:
PeakHookProc
UpdatePeakFromXY

Name	Value	Type
gLastClickX	51.4750830564784	NVAR -> root:Pac…
gLastClickY	0.940964713720057	NVAR -> root:Pac…
isOld	1	Variable
isPeak	1	Variable
wcoef	-> coef, 4 pts, SP	WAVE -> root:coef
wcoefsav	-> coef, 4 pts, SP	WAVE -> root:Pac…
wdest	-> sam, 128 pts, SP	WAVE -> root:sam
xaxval	51.8970099667774	Variable
yaxval	0.940964713720057	Variable

Point	coef.x	coef.d
0	0	0
1	1	1
2	2	50.4219
3	3	5
4		

Current DF: root:

```
            wcoef[3]= 0
         endif
      else
         if( isOld )
            wcoef[0]= wcoefsav[0]+yaxval-gLastClickY
            wcoef[1]= wcoefsav[1] + wcoefsav[0] - wcoef[0]      // maintain old peak top
            wcoef[3]= wcoefsav[3]+xaxval-gLastClickX
         else
            wcoef[0]= yaxval
            wcoef[1]= wcoefsav[1] + wcoefsav[0] - yaxval      // maintain old peak top
            wcoef[3]= wcoefsav[3]+xaxval-gLastClickX
         endif
      endif
      wdest=mygauss(wcoef,x)
   end
```

The debugger is normally disabled. Select Enable Debugger in either the Procedure menu or the contextual menu shown by control-clicking (Macintosh) or by right-clicking (Windows) in any procedure window.

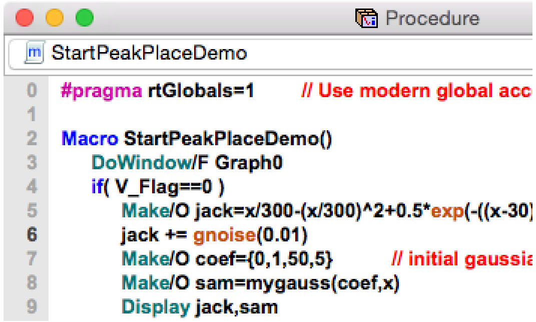
Igor displays the debugger window when one of the following events occurs:

1. A breakpoint that you previously set is hit
2. An error occurs, and you have enabled debugging on that kind of error
3. An error dialog is presented, and you click the Debug button
4. The **Debugger** command is executed

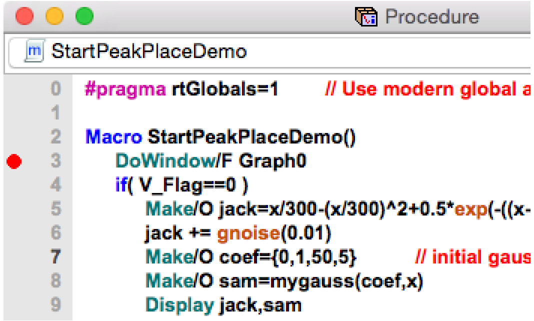
The debugger can not be used with threadsafe code. See **Debugging ThreadSafe Code** on page IV-225 for details.

Setting Breakpoints

When you want to observe a particular routine in action, set a breakpoint on the line where you want the debugger to appear. To do this, open the procedure window which contains the routine, and click in the left “breakpoint margin”. The breakpoint margin appears only if the debugger has been enabled. These graphics show the procedure windowwith the debugger disabled (left) and enabled (right):



```
#pragma rtGlobals=1		// Use modern global acc

Macro StartPeakPlaceDemo()
	DoWindow/F Graph0
	if( V_Flag==0 )
		Make/O jack=x/300-(x/300)^2+0.5*exp(-((x-30)
		jack += gnoise(0.01)
		Make/O coef={0,1,50,5}		// initial gaussia
		Make/O sam=mygauss(coef,x)
		Display jack,sam
```



```
#pragma rtGlobals=1		// Use modern global a

Macro StartPeakPlaceDemo()
	DoWindow/F Graph0
	if( V_Flag==0 )
		Make/O jack=x/300-(x/300)^2+0.5*exp(-((x-
		jack += gnoise(0.01)
		Make/O coef={0,1,50,5}		// initial gaus
		Make/O sam=mygauss(coef,x)
		Display jack,sam
```

The red dot denotes a breakpoint that you have set.

When a line of code marked with a breakpoint is about to execute, Igor displays the debugger window.

Click the red dot again to clear the breakpoint. Control-click (*Macintosh*) or right-click (*Windows*) and use the pop-up menu to clear all breakpoints or disable a breakpoint on the currently selected line of the procedure window.

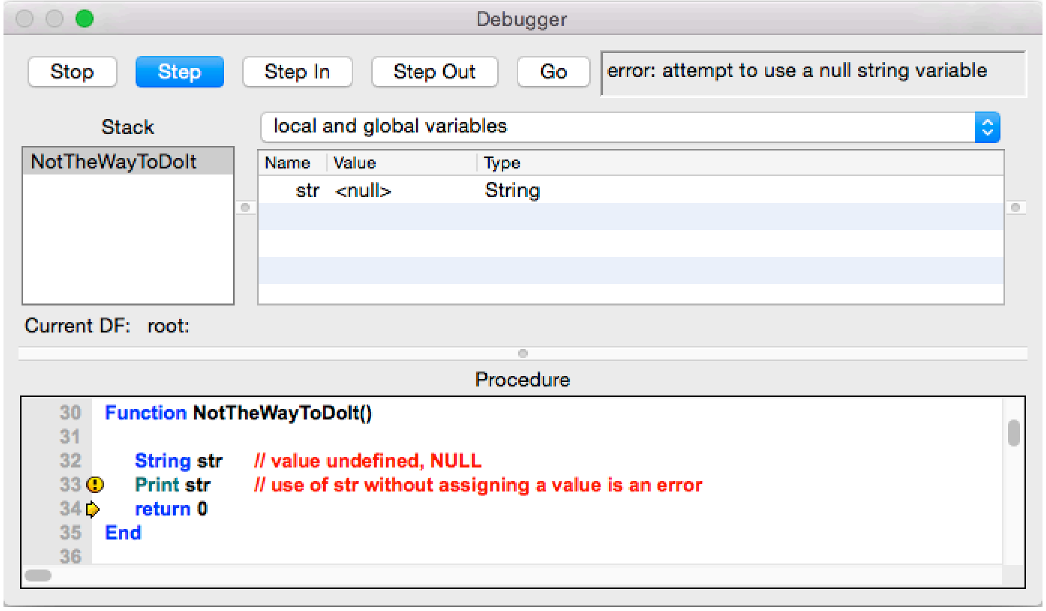
Debugging on Error

You can automatically open the debugger window when an error occurs. There are two categories of errors to choose from:

Debug On Error	Any runtime error except failed NVAR, SVAR, or WAVE references.
NVAR SVAR WAVE Checking	Failed NVAR, SVAR, or WAVE references.

We recommend that Igor programmers turn both of these options on to get timely information about errors.

Use the Procedure or contextual menus to enable or disable both error categories. If the selected error occurs, Igor displays the debugger with an error message in its status area. The error message was generated by the command indicated by a round yellow icon, in this example the Print str command:



error: attempt to use a null string variable

NotTheWayToDoIt

Name	Value	Type
str	<null>	String

Current DF: root:

```
Function NotTheWayToDoIt()

    String str    // value undefined, NULL
    Print str     // use of str without assigning a value is an error
    return 0
End
```

Sometimes you do something that you know may cause an error and you want to handle the error yourself, without breaking into the debugger. One such case is attempting to access a wave or variable that may or may not exist. You want to test its existence without breaking into the debugger.

You can use the /Z flag to prevent the Debug on Error feature from kicking in when an NVAR, SVAR, or WAVE reference fails. For example:

```
WAVE/Z w = <path to possibly missing wave>
if (WaveExists(w))
   <do something with w>
endif
```

In other cases where an error may occur and you want to handle it yourself, you need to temporarily disable the debugger and use **GetRTError** to get and clear the error. For example:

```
Function DemoDisablingDebugger()
   DebuggerOptions              // Sets V_enable to 1 if debugger is enabled
   Variable debuggerEnabled=V_enable
   DebuggerOptions enable=0     // Disable debugger

   String name = ";"            // This is an illegal wave name
   Make/O $name                 // So this will generate and error

   DebuggerOptions enable=debuggerEnabled    // Restore

   Variable err = GetRTError(1)              // Clear error
   if (err != 0)
      Printf "Error %d\r", err
   else
      Print "No error"
   endif
End
```

Debugging on Abort

You can tell Igor to automatically open the debugger window if you interrupt command execution by enabling Debug On Abort. This is useful for stopping code that is taking much longer than expected. It pauses execution and opens the debugger so you can see what is going on.

To enable or disable this feature, choose Procedure→Debug On Abort or right-click in a procedure window and choose Debug On Abort from the pop-up menu.

When Debug On Abort is enabled, the Abort button in the status bar is labeled "Debug", instead of "Abort". Click Debug to enter the debugger after the currently executing line finishes.

When Debug On Abort is disabled, clicking the Abort button stops all command execution and returns Igor to normal interactive operation.

Debug on Abort does not affect programmed aborts using the Abort, AbortOnRTE, or AbortOnValue operations.

Normally clicking the Abort button interrupts whatever command is running. For example, you can abort a wave assignment statement that is taking a long time.

To avoid entering the debugger when a command is half-finished, with Debug on Abort enabled, when you click the Debug button, Igor waits for the current command to finish before breaking into the debugger. If a very length operation or assignment statement is running, you may have to wait a long time for the debugger to activate.

You can interrupt the currently running command and enter the debugger more quickly using the **User Abort Key Combinations** instead of clicking the Debug button. This may enter the debugger at a time when an assignment is only partially finished.

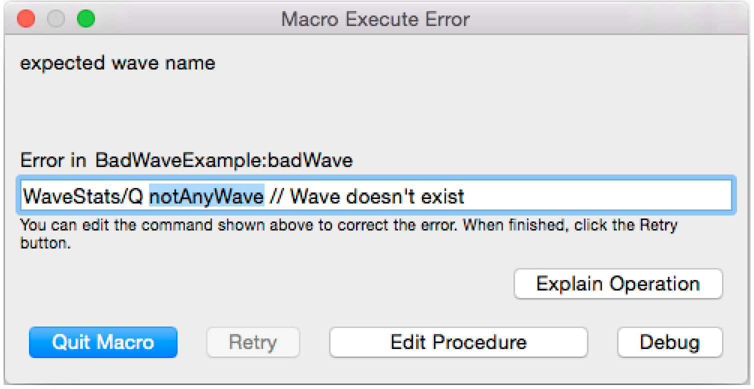
Some Igor code cannot be debugged. This includes thread-safe function code, hidden code, or independent module code (but see **SetIgorOption IndependentModuleDev=1** on page IV-239). If you click the Debug button or use the Abort key combinations while running code that cannot be debugged, code execution is stopped just as if Debug on Abort was not set. For instance, this:

```
ThreadSafe Function Mistake()
   Variable n=1
   do
      // forgot to increment n
   while (n<10)
End
```

cannot be debugged. Clicking Debug or using the Abort key combinations aborts execution. To debug this code, temporarily remove the Threadsafe keyword from the function declaration.

Macro Execute Error: The Debug Button

When the debugger is enabled and an error occurs in a macro, Igor presents an error dialog with, in most cases, a Debug button. Click the Debug button to open the debugger window.



expected wave name

Error in BadWaveExample:badWave

WaveStats/Q notAnyWave // Wave doesn't exist

You can edit the command shown above to correct the error. When finished, click the Retry button.

Errors in macros and procs are reported immediately after they occur.

When an error occurs in a user-defined function, Igor displays an error dialog long after the error actually occurred. The Debug On Error option is for programmers and displays errors in functions when they occur.

Stepping Through Your Code

Single-stepping through code is useful when you are not sure what path it is taking or how variables wound up containing their values.

Begin by enabling the debugger and setting a breakpoint on the line of code you are interested in, or begin when the debugger automatically opens because of an error. Use the buttons at the top of the debugger window to step through your code:

The Stop Button

The Stop button ceases execution of the running function or macro before it completes. This is equivalent to clicking Igor's Abort button while the procedure is running. If you have enabled Debug on Abort, the Stop button still causes execution to cease.

Keyboard shortcuts: Command-period (*Macintosh*), Ctrl+Break (*Windows*)

Pressing Command-period on a Macintosh while the debugger window is showing is equivalent to clicking the *Go* button, not the Stop button.

The Step Button

The Step button executes the next line. If the line contains a call to one or more subroutines, execution continues until the subroutines return or until an error or breakpoint is encountered. Upon return, execution halts until you click a different button.

Keyboard shortcuts: Enter, keypad Enter, or Return

The Step Into Button

The Step Into button executes the next line. If that line contains a call to one or more subroutines, execution halts when the first subroutine is entered. The Stack list of currently executing routines shows the most recently entered routine as the last item in the list.

Keyboard shortcuts: +, =, or keyPad +

The Step Out Button

The Step Out button executes until the current subroutine is exited, or an error or breakpoint is encountered.

Keyboard shortcuts: -, _ (underscore) or keypad -

The Go Button

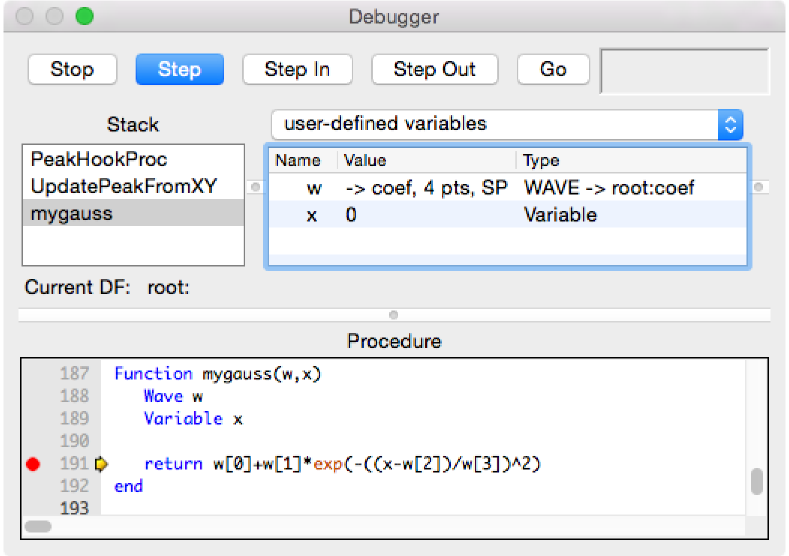
The Go button resumes program execution. The debugger window remains open until execution completes or an error or breakpoint is encountered.

If you press the Option (*Macintosh*) or Alt (*Windows*) key while clicking the Go button, the debugger window is closed until execution completes or an error or breakpoint is encountered.

Keyboard shortcuts: Esc

The Stack and Variables Lists

The Stack List shows the routine that is currently executing and the chain of routines that called it. The top item in the list is the routine that began execution and the bottom item is the routine which is currently executing.



user-defined variables

PeakHookProc
UpdatePeakFromXY
mygauss

Name	Value	Type
w	-> coef, 4 pts, SP	WAVE -> root:coef
x	0	Variable

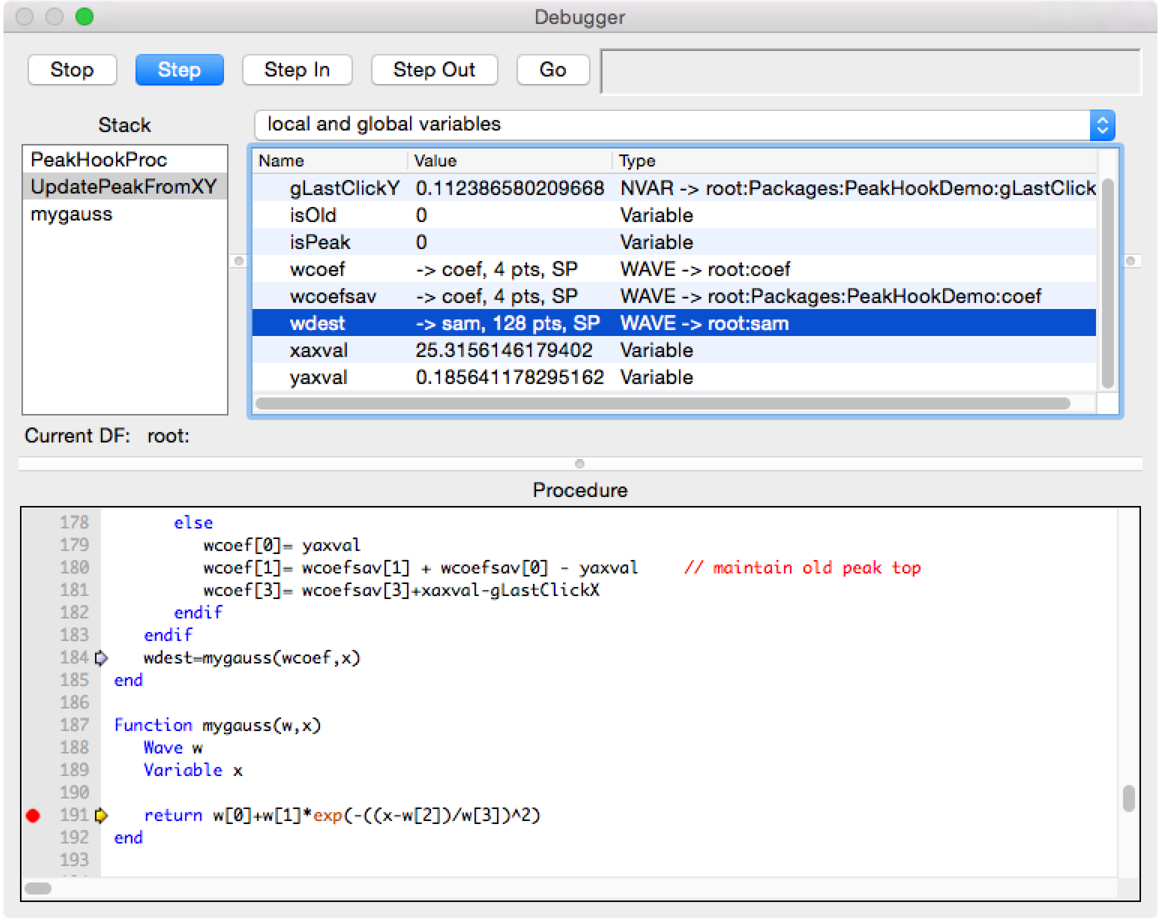
Current DF: root:

```
Function mygauss(w,x)
	Wave w
	Variable x

	return w[0]+w[1]*exp(-((x-w[2])/w[3])^2)
end

```

In this example, the routine that started execution is PeakHookProc, which most recently called UpdatePeakFromXY, which then called the currently executing mygauss user function.

The Variables List, to the right of the Stack List, shows that the function parameters w and x have the values coef (a wave) and 0 (a number). The pop-up menu controls which variables are displayed in the list; the example shows only user-defined local variables.

You can examine the variables associated with any routine in the Stack List by simply selecting the routine:



local and global variables

PeakHookProc
UpdatePeakFromXY
mygauss

Name	Value	Type
gLastClickY	0.112386580209668	NVAR -> root:Packages:PeakHookDemo:gLastClick
isOld	0	Variable
isPeak	0	Variable
wcoef	-> coef, 4 pts, SP	WAVE -> root:coef
wcoefsav	-> coef, 4 pts, SP	WAVE -> root:Packages:PeakHookDemo:coef
wdest	-> sam, 128 pts, SP	WAVE -> root:sam
xaxval	25.3156146179402	Variable
yaxval	0.185641178295162	Variable

Current DF: root:

```
		else
			wcoef[0]= yaxval
			wcoef[1]= wcoefsav[1] + wcoefsav[0] - yaxval		// maintain old peak top
			wcoef[3]= wcoefsav[3]+xaxval-gLastClickX
		endif
	endif
	wdest=mygauss(wcoef,x)
end

Function mygauss(w,x)
	Wave w
	Variable x

	return w[0]+w[1]*exp(-((x-w[2])/w[3])^2)
end

```

Here we’ve selected UpdatePeakFromXY, the routine that called mygauss (see the light blue arrow). Notice that the Variables List is showing the variables that are local to UpdatePeakFromXY.

For illustration purposes, the Variables List has been resized by dragging the dividing line, and the pop-up menu has been set to show local and global variables and type information.

The Variables List Columns

The Variables List shows either two or three columns, depending on whether the "show variable types" item in the Variable pop-up menu is checked.

Double-clicking a column's header resizes the column to fit the contents. Double-clicking again resizes the column to a default width.

The first column is the name of the local variable. The name of an NVAR, SVAR, or WAVE reference is a name local to the macro or function that refers to a global object in a data folder.

The second column is the value of the local variable. Double-click the second column to edit numbers in-place, double-click anywhere on the row to "inspect" waves, strings, SVARS, or char arrays in structures in the appropriate Inspector.

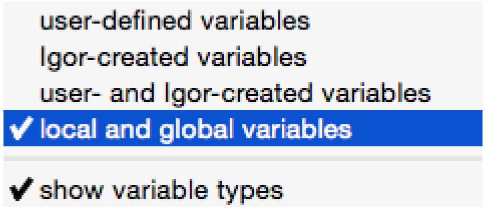
In the case of a wave, the size and precision of the wave are shown here. The "->" characters mean "refers to". In our example wcoef is a local name that refers to a (global) wave named coef, which is one-dimensional, has 4 points, and is single precision.

To determine the value of a particular wave element, use an inspector as described under **Inspecting Waves**.

The optional third column shows what the type of the variable is, whether Variable, String, NVAR, SVAR, WAVE, etc. For global references, the full data folder path to the global is shown.

Variables Pop-Up Menu

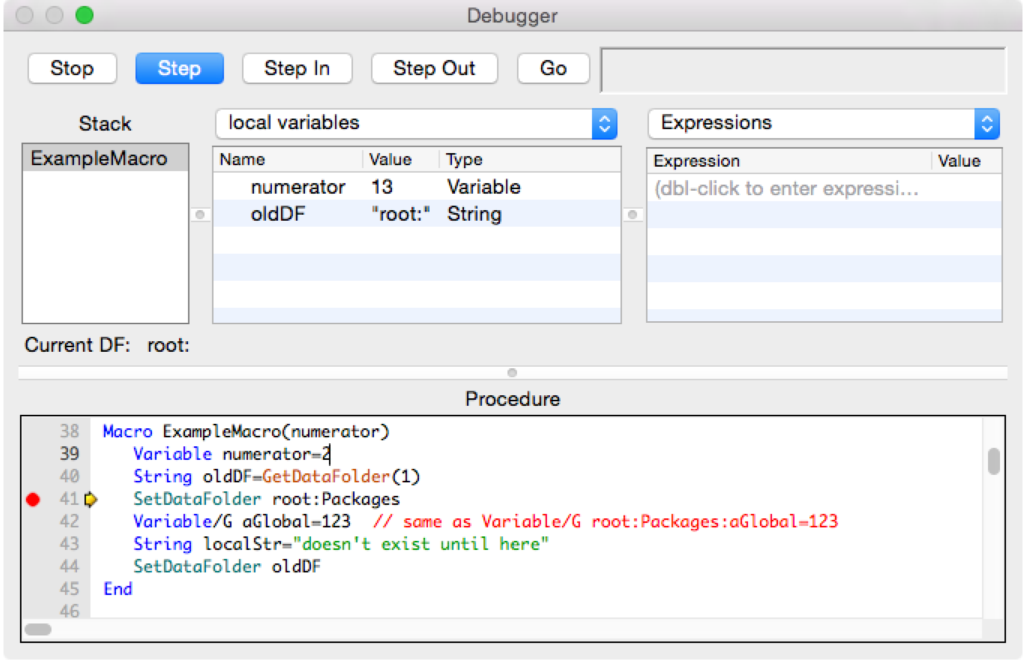
The Variables pop-up menu controls which information is displayed in the Variables List. When debugging a function, it looks like this:



When debugging a macro, proc or window macro, the first two items in the popup menu are unavailable.

Macro Variables

The ExampleMacro below illustrates how variables in Macros, Procs or Window procedures are classified as locals or globals:



Name	Value	Type
numerator	13	Variable
oldDF	"root:"	String

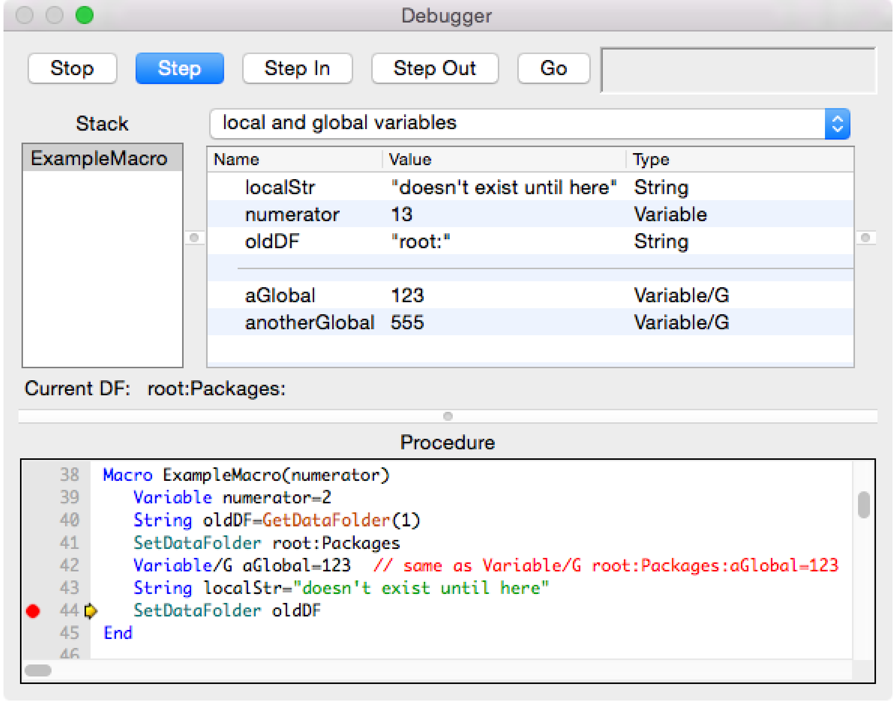
Expression	Value
(dbl-click to enter expressi…	

Current DF: root:

```
Macro ExampleMacro(numerator)
   Variable numerator=2
   String oldDF=GetDataFolder(1)
   SetDataFolder root:Packages
   Variable/G aGlobal=123  // same as Variable/G root:Packages:aGlobal=123
   String localStr="doesn't exist until here"
   SetDataFolder oldDF
End
```

Local variables in macros include all items passed as parameters (numerator in this example) and local variables and local strings (oldDF) whose definitions have been executed, and Igor-created local variables created by operations such as WaveStats after the operation has been executed. Note that localStr isn't listed, because the command has not yet executed.

Global variables in macros include all items in the current data folder, whether they are used in the macro or not. If the data folder changes because of a SetDataFolder operation, the list of global variables also changes. Note that there are no NVAR, SVAR, WAVE or STRUCT references in a macro.



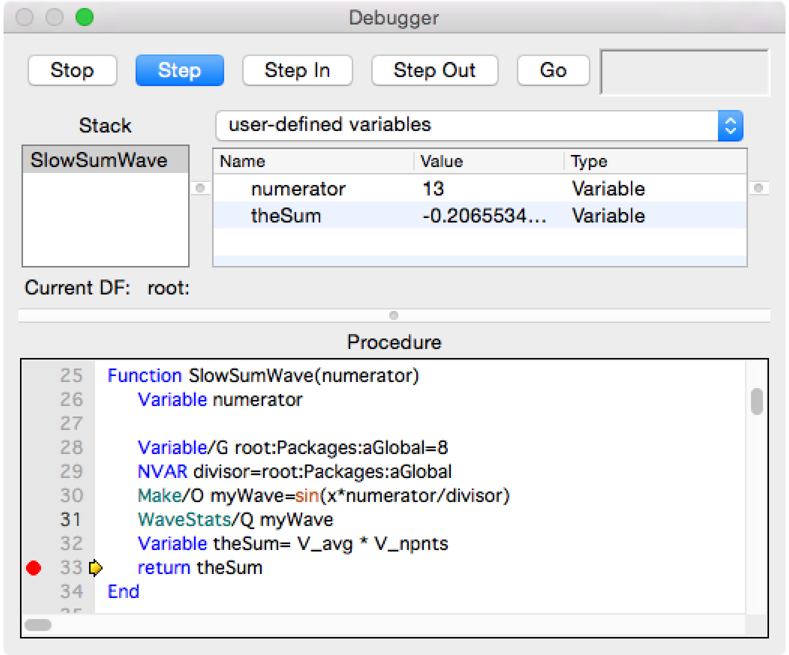
Name	Value	Type
localStr	"doesn't exist until here"	String
numerator	13	Variable
oldDF	"root:"	String
aGlobal	123	Variable/G
anotherGlobal	555	Variable/G

Current DF: root:Packages:

```
Macro ExampleMacro(numerator)
   Variable numerator=2
   String oldDF=GetDataFolder(1)
   SetDataFolder root:Packages
   Variable/G aGlobal=123  // same as Variable/G root:Packages:aGlobal=123
   String localStr="doesn't exist until here"
   SetDataFolder oldDF
End
```

Function Variables

The SlowSumWaveFunction example below illustrates how different kinds of variables in functions are classified:



Name	Value	Type
numerator	13	Variable
theSum	-0.2065534...	Variable

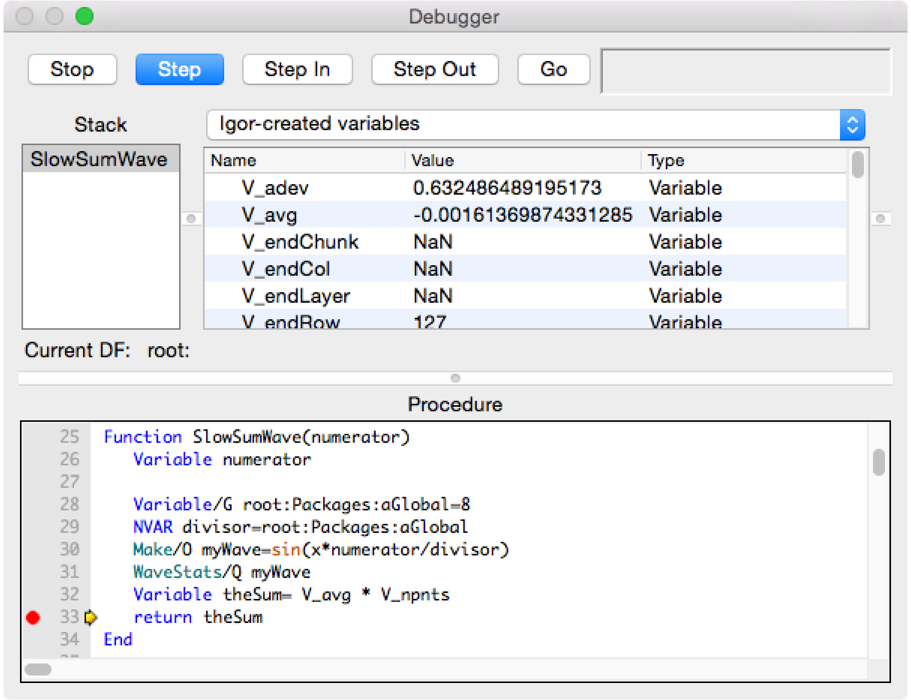
```
Function SlowSumWave(numerator)
    Variable numerator

    Variable/G root:Packages:aGlobal=8
    NVAR divisor=root:Packages:aGlobal
    Make/O myWave=sin(x*numerator/divisor)
    WaveStats/Q myWave
    Variable theSum= V_avg * V_npnts
    return theSum
End
```

User-defined variables in functions include all items passed as parameters (numerator in this example) and any local strings and variables.

Local variables exist while a procedure is running, and cease to exist when the procedure returns; they never exist in a data folder like globals do.

NVAR, SVAR, WAVE, Variable/G and String/G references point to global variables, and therefore, aren't listed as user-defined (local) variables.

Use "Igor-created variables" to show local variables that Igor creates for functions when they call an operation or function that returns results in specially-named variables. The **WaveStats** operation (see page V-1082), for example, defines V_adev, V_avg, and other variables to contain the statistics results:

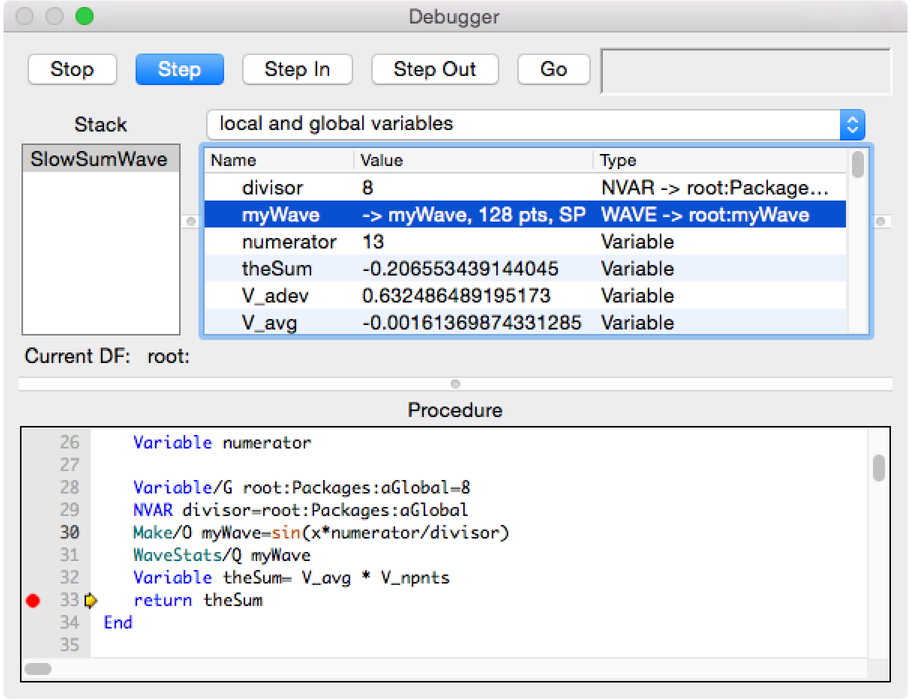


Name	Value	Type
V_adev	0.632486489195173	Variable
V_avg	-0.00161369874331285	Variable
V_endChunk	NaN	Variable
V_endCol	NaN	Variable
V_endLayer	NaN	Variable
V_endRow	127	Variable

```
Function SlowSumWave(numerator)
   Variable numerator

   Variable/G root:Packages:aGlobal=8
   NVAR divisor=root:Packages:aGlobal
   Make/O myWave=sin(x*numerator/divisor)
   WaveStats/Q myWave
   Variable theSum= V_avg * V_npnts
   return theSum
End
```

The “user- and Igor-created” menu item shows both kinds of local variables.

The “local and global variables” item shows user-created local variables, most Igor-created local variables, and references to global variables and waves through NVAR, SVAR, and WAVE references:

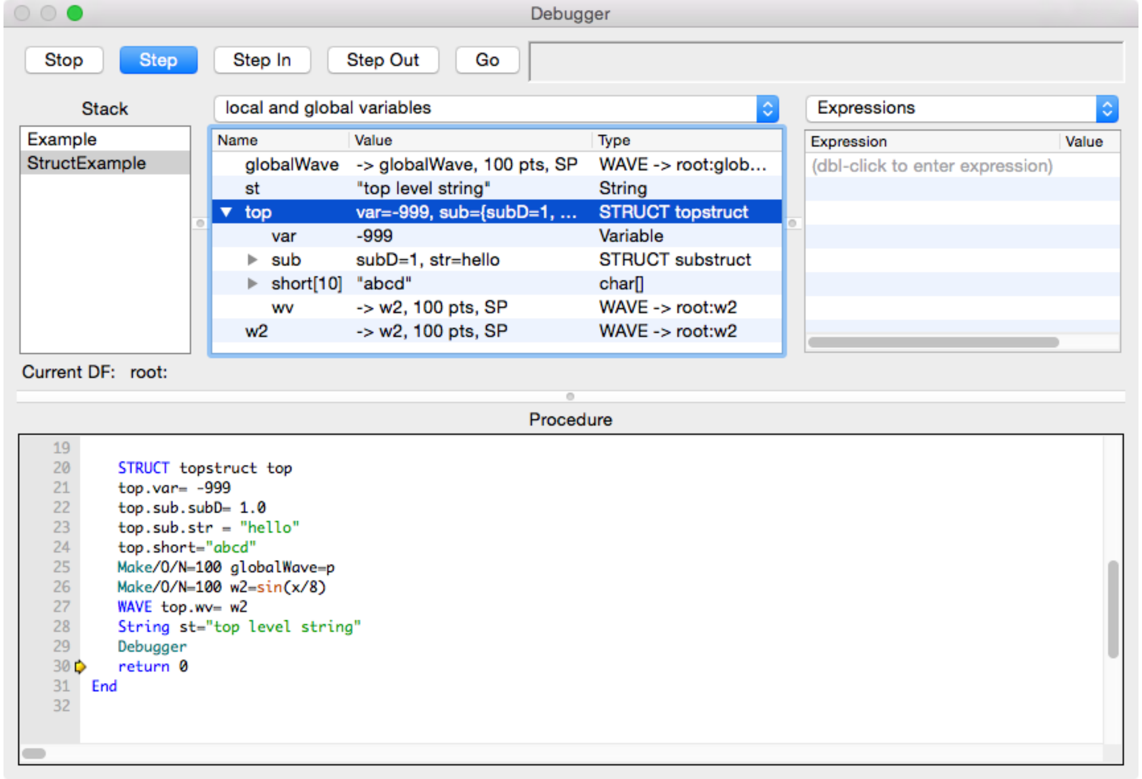


Name	Value	Type
divisor	8	NVAR -> root:Package...
myWave	-> myWave, 128 pts, SP	WAVE -> root:myWave
numerator	13	Variable
theSum	-0.206553439144045	Variable
V_adev	0.632486489195173	Variable
V_avg	-0.00161369874331285	Variable

```
   Variable numerator

   Variable/G root:Packages:aGlobal=8
   NVAR divisor=root:Packages:aGlobal
   Make/O myWave=sin(x*numerator/divisor)
   WaveStats/Q myWave
   Variable theSum= V_avg * V_npnts
   return theSum
End

```

Function Structures

The elements of a structure (see **Structures in Functions** on page IV-99) are displayed in tree form in the Variables "list". Click the triangles to expand or collapse a node in the structure, or double-click the row:



Stack
Example
StructExample

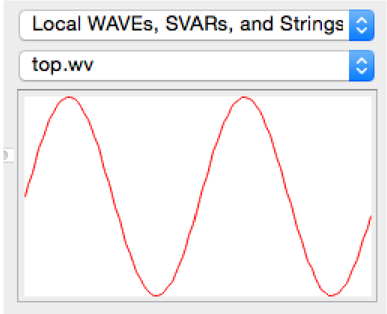
Current DF: root:

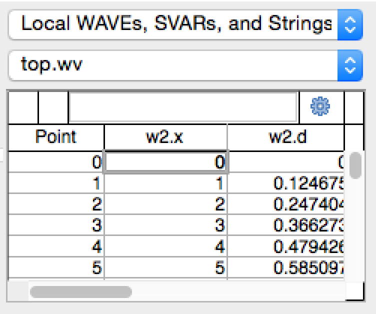
Name	Value	Type
globalWave	-> globalWave, 100 pts, SP	WAVE -> root:glob…
st	"top level string"	String
▼ top	var=-999, sub={subD=1, …	STRUCT topstruct
var	-999	Variable
▶ sub	subD=1, str=hello	STRUCT substruct
▶ short[10]	"abcd"	char[]
wv	-> w2, 100 pts, SP	WAVE -> root:w2
w2	-> w2, 100 pts, SP	WAVE -> root:w2

Expression	Value
(dbl-click to enter expression)	

```

	STRUCT topstruct top
	top.var= -999
	top.sub.subD= 1.0
	top.sub.str = "hello"
	top.short="abcd"
	Make/O/N=100 globalWave=p
	Make/O/N=100 w2=sin(x/8)
	WAVE top.wv= w2
	String st="top level string"
	Debugger
	return 0
End

```

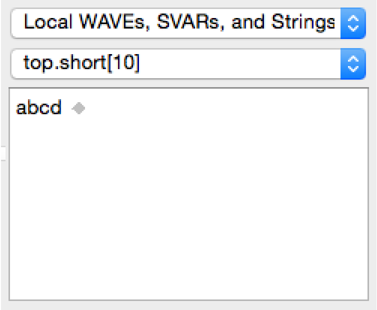
Double-clicking a WAVE element (such as top.wv) will send it to the Wave Inspector (either a table or graph depending on what is checked in the Inspector popup):





Point	w2.x	w2.d
0	0	0
1	1	0.124675
2	2	0.247404
3	3	0.366273
4	4	0.479426
5	5	0.585097

Double-clicking a String element or char array (such as top.short) will send it to the String Inspector:



abcd

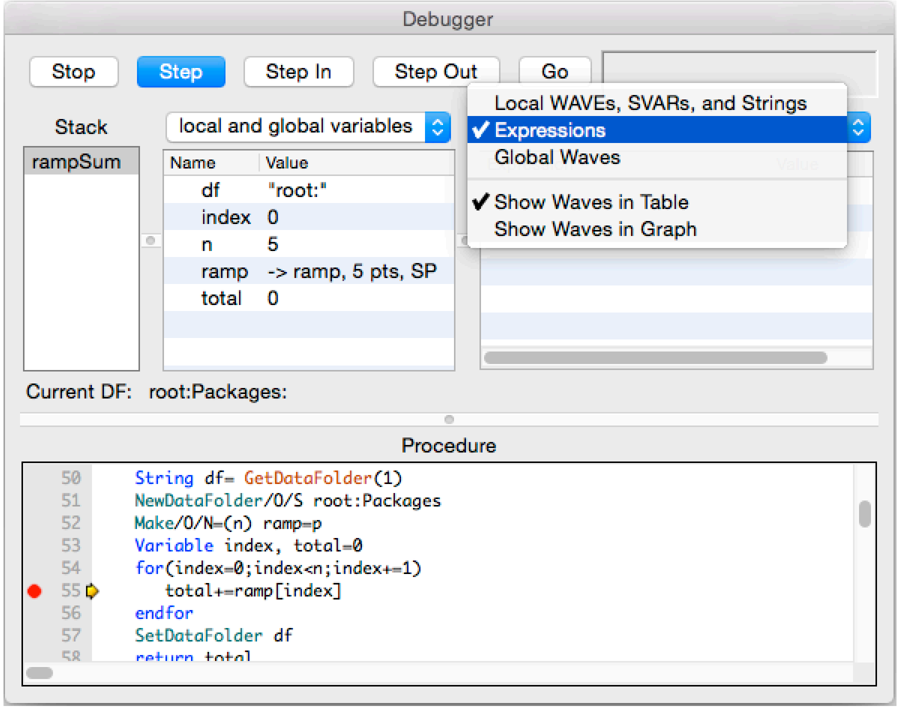
The Current Data Folder

The "Current DF" displays the path to the current data folder. You can select and copy the data folder path. See **Data Folders** on page II-107 for more information about data folders.

Graph, Table, String, and Expressions Inspectors

One of the Wave, String, or Expressions Inspectors is visible to right side of the variables list. This pane is hidden when the divider between the pane and the variables list is dragged all the way to the right. Drag the divider to the left to show the pane. You may need to widen the window to make room.

Use the popup to select the Inspector you want.



Name	Value
df	"root:"
index	0
n	5
ramp	-> ramp, 5 pts, SP
total	0

Current DF: root:Packages:

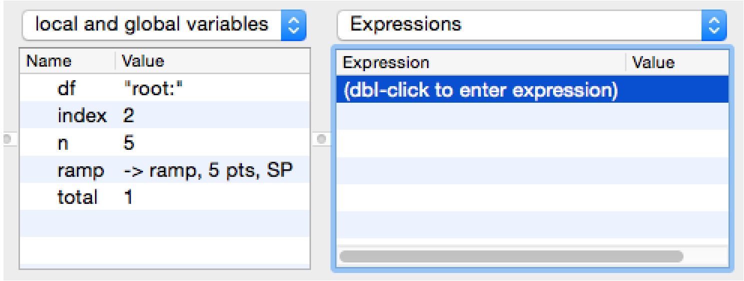
```
	String df= GetDataFolder(1)
	NewDataFolder/O/S root:Packages
	Make/O/N=(n) ramp=p
	Variable index, total=0
	for(index=0;index<n;index+=1)
		total+=ramp[index]
	endfor
	SetDataFolder df
	return total
```

The pop-up menu controls what is shown in the pane:

Pop-up Menu Selection	Pane Contents
Local WAVEs, SVARs, and Strings	A list of these things in the selected function. References to free waves are also listed here, as are elements in a STRUCT that are "inspectable", including char arrays which can be viewed as if they were Strings.
Local Strings	A list of Strings local to the selected function, macro or proc.
Expressions	Numeric or string expressions which are evaluated in the context of the selected function or macro.
Global Waves	A popup wave selector to display any wave in a global data folder. Free waves are not listed here.
Show Waves in Table Show Waves in Graph	Waves will be displayed in a table or graph, depending on which one of these two is checked.

Expressions Inspector

Selecting "Expressions" from the inspector popup shows a list of Expressions and their values:



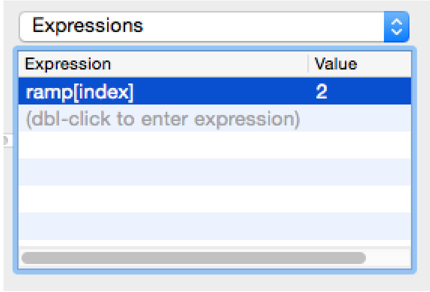
local and global variables

Name	Value
df	"root:"
index	2
n	5
ramp	-> ramp, 5 pts, SP
total	1

Expressions

Expression	Value
(dbl-click to enter expression)	

Replace the "(dbl-click to enter expression)" invitation by clicking it, typing a numeric or string expression, and pressing Return.



Expressions

Expression	Value
ramp[index]	2
(dbl-click to enter expression)	

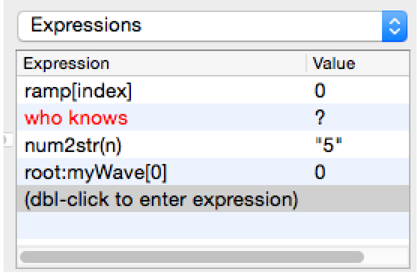
Adding an expression adds a blank row at the end of the list that can be double-clicked to enter another expression. You can edit any of the expressions by double-clicking and typing.

The expression can be removed by selecting it and pressing Delete or Backspace.

The result of the expression is recomputed when stepping through procedures. The expressions are evaluated in the context of the currently selected procedure.

Global expressions are evaluated in the context of the current data folder, though you can specify the data folder explicitly as in the example below.

If an expression is invalid the result is shown as “?” and the expression is changed to red:



Expressions

Expression	Value
ramp[index]	0
who knows	?
num2str(n)	"5"
root:myWave[0]	0
(dbl-click to enter expression)	

The expressions are discarded when a new Igor experiment is opened or when Igor quits.

Inspecting Waves

You can "inspect" (view) the contents of a wave in either a table or a graph. They aren't full-featured tables or graphs, as there are no supporting dialogs for them. You can change their properties using contextual menus.

Select the Wave to be inspected by one of three methods:

1. Choose "Global Waves", and then choose a wave from the popup wave browser.
2. Choose "Local WAVEs, SVARs, and Strings", and then choose a wave from among the objects listed.
3. Double-click any WAVE reference in the Variables list.

Inspecting Waves in a Table

You can edit the values in a wave using the table, just like a regular table. With the contextual menu you can alter the column format, among other things.

Inspecting Waves in a Graph

You can view waves in a Graph. With the contextual menu, you can choose to show or hide the axes and change display modes.

Two-dimensional waves are displayed as an image plot.

Inspecting Strings

Select a String or char array to be inspected by two methods:

1. Choose "Local WAVEs, SVARs, and Strings", and then choose a String, SVAR or char array from among the objects listed.
2. Double-click any String, SVAR or char array in the Variables list.

The Procedure Pane

The procedure pane contains a copy of the procedure window of the routine selected in the Stack List. You can set and clear breakpoints in this pane just as you do in a procedure window, using the breakpoint margin and the Control-click (*Macintosh*) or right-click (*Windows*) menu.

A very useful feature of the debugger is the automatic text expression evaluator that shows the value of a variable or expression under the cursor. The value is displayed as a tool tip. This is often faster than scrolling through the Variables List or entering an expression in the Expressions List to determine the value of a variable, wave, or structure member reference.

The value of a variable can be displayed whether or not the variable name is selected. To evaluate an expression such as "wave[ii]+3", the expression must be selected and the cursor must be over the selection.

The debugger won't evaluate expressions that include calls to user-defined functions; this prevents unintended side effects (a function could overwrite a wave's contents, for example). You can remove this limitation by creating the global variable root:V_debugDangerously and setting it to 1.

After You Find a Bug

Editing in the debugger window is disabled because the code is currently executing. Tracking down the routine after you've exited the debugger is easy if you follow these steps:

1. Scroll the debugger text pane back to the name of the routine you want to modify, and select it.
2. Control-click (Macintosh) or Right-click (Windows) the name, and choose "Go to <routineName>" from the pop-up menu.
3. Exit the debugger by clicking the "Go" button or by pressing Escape.

Now the selected routine will be visible in the top procedure window, where you can edit it.

Debugging ThreadSafe Code

The Igor debugger can not be used with threadsafe functions.

The debugger does not break on breakpoints in threadsafe functions.

The debugger does not allow you to step into a threadsafe function.

These restrictions apply even if you call a threadsafe function from a non-threadsafe function.

The main technique for debugging threadsafe code is the use of print statements. See Debugging With Print Statements.

You can use the debugger on functions marked as threadsafe by temporarily disabling multithreading by executing:

```
SetIgorOption DisableThreadSafe = 1    // Disable multithreading
```

This causes Igor to recompile all procedures and to ignore the Threadsafe and MultiThread keywords. You can then debug procedures using the debugger. When you are finished, re-enable multithreading by executing:

```
SetIgorOption DisableThreadSafe = 0    // Re-enable multithreading
```

Debugger Shortcuts

Action	Shortcut
To enable debugger	Choose Enable Debugger from the Procedure menu or choose Enable Debugger from the procedure window's pop-up menu after Control-clicking (*Macintosh*) or right-clicking (*Windows*).
To automatically enter the debugger when an error occurs	Choose Debug on Error from the Procedure menu or choose Enable Debugger from a procedure window's pop-up menu after Control-clicking (*Macintosh*) or right-clicking (*Windows*).
To set or clear a breakpoint	Click in the left margin of the procedure window or click anywhere on the procedure window line where you want to set or clear the breakpoint and choose Set Breakpoint or Clear Breakpoint from a procedure window's pop-up menu after Control-clicking (*Macintosh*) or right-clicking (*Windows*).
To enable or disable a breakpoint	Shift-click a breakpoint in the left margin of the procedure window. Click anywhere on the procedure window line where you want to enable or disable the breakpoint and choose Enable Breakpoint or Disable Breakpoint procedure window's pop-up menu after Control-clicking (*Macintosh*) or right-clicking (*Windows*).
To execute the next command	On *Macintosh* press Enter, keypad Enter, or Return. For *Windows*, if no button has the focus, press Enter or Return. Otherwise, click the yellow arrow button.
To step into a subroutine	Press the +, =, or keypad + keys, or click the blue descending arrow button.
To step out of a subroutine to the calling routine	Press the -, _ (underscore) or keypad - keys, or click the blue ascending arrow button.
To resume executing normally	Press Escape (Esc), or click the green arrow button.
To cancel execution	Click the red stop sign button.
To edit the value of a macro or function variable	Double-click the second column of the variables list, edit the value, and press Return or Enter.
To set the value of a function's string to null	Double-click the second column of the variables list, type "<null>" (without the quotes), and press Return or Enter.
To view the current value of a macro or function variable	Move the cursor to the procedure text of the variable name and wait. On *Macintosh*, the value appears to the right of the debugger buttons. On *Windows*, the value appears in a tooltip window.

Action	Shortcut
To view the current value of an expression	Select the expression text with the cursor, position the cursor over the selection, and wait. (Expressions involving user-defined functions will not be evaluated unless V_debugDangerously is set to 1.)
To view global values in the current data folder	Choose “local and global variables” from the debugger pop-up menu.
To view type information about variables	Choose “show variable types” from the debugger pop-up menu.
To resize the columns in the variables list	Drag a divider in the list to the left or right.
To show or hide the Waves, Structs, and Expressions pane	Drag the divider on the right side of the Variables list left or right.

Chapter

IV-9

Dependencies

Dependency Formulas

Igor Pro supports “dependent objects”. A dependent object can be a wave, a global numeric variable or a global string variable that has been linked to an expression. The expression to which an object is linked is called the object’s “dependency formula” or “formula” for short.

The value of a dependent object is updated whenever any other global object involved in the formula is modified, even if its value stays the same. We say that the dependent object *depends* on these other global objects *through* the formula.

You might expect that an assignment such as:

```
Variable var0 = 1
Make wave0
wave0 = sin(var0*x/16)
```

meant that wave0 would be updated whenever var0 changed. It doesn’t. Values are computed for wave0 only once, and the relationship between wave0 and var0 is forgotten.

However, if the equal sign in the above assignment is replaced by a colon followed by an equal sign:

```
wave0 := sin(var0*x/16)
```

then Igor *does* create just such a dependency. Now whenever var0 changes, Igor reevaluates the assignment statement and updates wave0. In this example, wave0 is a dependent object. It depends on var0, and “sin(var0*x/16)” is wave0’s dependency formula.

You can also establish a dependency using the SetFormula operation, like this:

```
SetFormula wave0, "sin(var0*x/16)"
```

Wave1 depends on var0 because var0 is a changeable variable. Wave1 *also* depends on the function x which returns the X scaling of the destination wave (wave0). When the X scaling of wave0 changes, the values that the x function returns change, so this dependency assignment is reevaluated. The remaining terms (sin and 16) are not changeable, so wave0 does not depend on them.

From the command line, you can use either a dependency assignment statement or SetFormula to establish a dependency. In a user-defined function, you must use SetFormula.

Like other assignment statements, the data folder context for the right-hand side is that of the destination object. Therefore, in the following example, wave2 and var1 must be in the data folder named foo, var2 must be in root, and var3 must be in root:bar.

```
root:foo:wave0 := wave2*var1 + ::var2 + root:bar:var3
```

Data Folders are described in Chapter II-8, **Data Folders**.

A dependency assignment is often used in conjunction with SetVariable controls and ValDisplay controls.

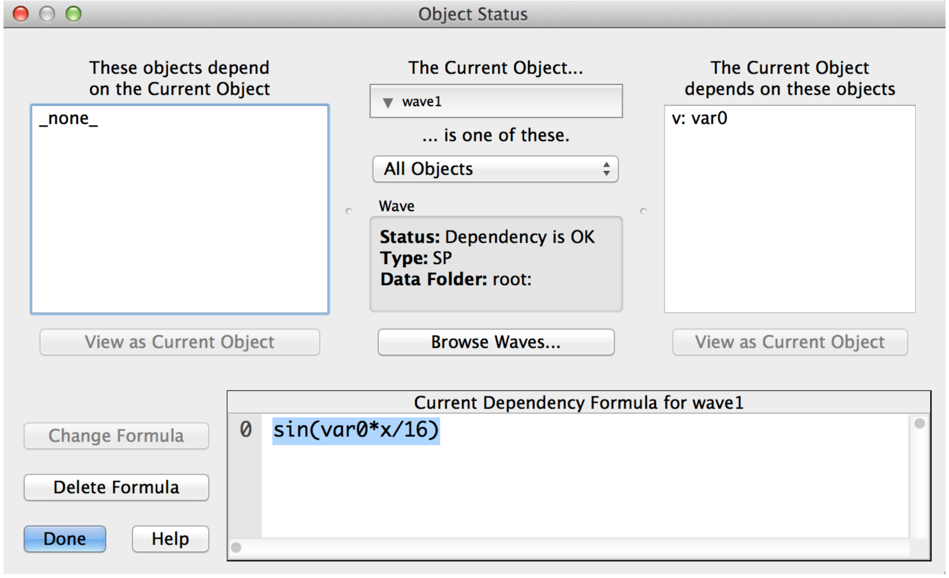
Here’s a simple example. Execute these commands on the command line:

```
Variable/G var0 = 1
Make/O wave0:=sin(var0*x/16)
Display /W=(4,53,399,261) wave0
ControlBar 23
SetVariable setvar0,size={60,15},value=var0
```

Click the SetVariable control’s up and down arrows to adjust var0 and observe that wave0 is automatically updated.

Dependencies and the Object Status Dialog

You can use the Object Status dialog, which you can access via the the Misc menu, to check dependencies. After executing the commands in the preceding section, the dialog looks like this:



Status: Dependency is OK
Type: SP
Data Folder: root:

```
0  sin(var0*x/16)
```

The Status area indicates any dependency status:

- “No dependency” means that the current object does not depend on anything.
- “Dependency is OK” means that the dependency formula successfully updated the current object.
- “Update failed” means that the dependency formula used to compute the current object’s value failed.

An update may fail because there is a syntax error in the formula or one of the objects referenced in the formula does not exist or has been renamed. If the formula includes a call to a user-defined function then the update will fail if the function does not exist or if procedures are not compiled.

If an update fails, then the objects that depend on that update are broken. See **Broken Dependent Objects** on page IV-233 for details.

You can create a new dependency formula with the New Formula button, which appears only if the current object is not the target of a dependency formula.

You can delete a dependency formula using the Delete Formula button.

You can change an existing dependency formula by typing in the Dependency Formula window, and clicking the Change Formula button.

For further details on the Object Status dialog, click the Help button in the dialog.

Numeric and String Variable Dependencies

Dependencies can also be created for global user-defined numeric and string variables. You can not create a dependency that uses a local variable on either side of the dependency assignment statement.

Here is a user-defined function that creates a dependency. The global variable `recalculateThis` is made to depend on the global variable `dependsOnThis`:

```
Function TestRecalculation()
   Variable/G recalculateThis
   Variable/G dependsOnThis = 1

   // Create dependency on global variable
   SetFormula recalculateThis, "dependsOnThis"

   Print recalculateThis       // Prints original value

   dependsOnThis = 2           // Changes something recalculateThis
   DoUpdate                    // Make Igor recalculate formulae

   Print recalculateThis       // Prints updated value
End
```

Running this function prints the following to the history area:

```
•TestRecalculation()
   1
   2
```

The call to DoUpdate is needed because Igor recalculates dependency formulas only when no user-defined functions are running or when DoUpdate is called.

This function uses SetFormula to create the dependency because the := operator is not allowed in user-defined functions.

Wave Dependencies

The assignment statement:

```
dependentWaveName := formula
```

creates a dependency and links the dependency formula to the dependent wave. Whenever any change is made to any object in the formula, the contents of the dependent wave are updated.

The command

```
SetFormula dependentWaveName, "formula"
```

establishes the same dependency.

From the command line, you can use either a dependency assignment statement or SetFormula to establish a dependency. In a user-defined function, you must use SetFormula.

Cascading Dependencies

“Cascading dependencies” refers to the situation that arises when an object depends on a second object, which in turn depends on a third object, etc. When an object changes, all objects that directly depend on that object are updated, *and* objects that depend directly on those updated objects are updated until no more updates are needed.

You can use the Object Status dialog to investigate cascading dependencies.

Deleting a Dependency

A dependency is deleted when the dependent object is assigned a value using the = operator:

```
recalculateThis := dependsOnThis     // Creates a dependency
recalculateThis = 0                  // Deletes the dependency
```

This method of deleting a dependency does not work in user-defined functions. You must use the SetFormula operation.

For example:

```
Execute "recalculateThis = 0"
```

will delete the dependency even in a user-defined function.

You can also delete this dependency using the SetFormula operation.

```
SetFormula recalculateThis, ""
```

Wave dependencies are also deleted by operations that overwrite the values of their wave parameters. Some of these operations are:

```
FFT      Convolve        Correlate   Smooth   GraphWaveEdit
Hanning  Differentiate   Integrate   UnWrap
```

Dependencies can also be deleted using the Object Status dialog.

Broken Dependent Objects

Igor compiles the text of a dependency formula to low-level code and stores both the original text and the low-level code with the dependent object. At various times, Igor may need to recompile the dependency formula text. At that time, a compilation error will occur if:

- The dependency formula contains an error
- An object used in the dependency formula has been deleted, renamed, or moved
- The dependency formula references a user-defined function that is missing
- The dependency formula references a user-defined function and procedures are not compiled

When this happens, the dependent object will no longer update but will retain its last value. We call such an object “broken”.

To inspect broken objects, invoke the Object Status dialog. Choose Broken Objects from the pop-up menu above the status area. If there are any broken objects, they will appear in the Current Object pop-up. Select an object from the Current Object pop-up to inspect it.

When Dependencies are Updated

Dependency updates take place at the same time that graphs are updated. This happens after each line in a macro is executed, or when DoUpdate is called from a macro or user function, or continuously if a macro or function is not running.

Dependency formulas used as input to the SetBackground and ValDisplay operations, and in some other contexts, can alternately be specified as a literal string of characters using the following syntax:

```
#"text_of_the_dependency_expression"
```

Note that what follows the # char must be a literal string — not a string expression.

This sets the dependency formula *without* compiling it or checking it for validity. If you need to set the dependency formula of an object to something that is not currently valid but will be in the future, then use this alternate method.

Programming with Dependencies

You cannot use := to create dependencies in user-defined functions. Instead you must use the **SetFormula** operation (see page V-847).

```
Function TestFunc()
   Variable/G varNum=-666
   Make wave0
   SetFormula wave0, "varNum"    // Equivalent to wave0 := varNum
End
```

Using Operations in Dependency Formulas

The dependency formula must be a single expression — and you can not use operations, such as FFT's, or other command type operations. However, you can invoke user-defined functions which invoke operations. For example:

```
Function MakeDependencyUsingOperation()
   Make/O/N=128 data = p         // A ramp from 0 to 127
   Variable/G power

   SetFormula power, "RMS(data)" // Dependency on function and wave
   Print power

   data = p * 2                  // Changes something power depends
   DoUpdate                      // Make Igor recalc formulae
   Print power
EndMacro

Function RMS(w)
   Wave w

   WaveStats/Q w           // An operation! One output is V_rms
   return V_rms
End
```

When `MakeDependencyUsingOperation` is executed, it prints the following in the history area:

```
•MakeDependencyUsingOperation()
   73.4677
   146.935
```

Dependency Caveats

The extensive use of dependencies can create a confusing tangle that can be difficult to manage. Although you can use the Object Status dialog to explore the dependency hierarchy, you can still become very confused very quickly, especially when the dependencies are highly cascaded. You should use dependencies only where they are needed. Use conventional assignments for the majority of your calculations.

Dependency formulas are generally not recalculated when a user-defined function is running unless you explicitly call the DoUpdate operation. However, they can run at other unpredictable times so you should not make any assumptions as to the timing or the current data folder when they run.

The text of the dependency formula that is saved for a dependent object is the original literal text. The dependency formula needs to be recompiled from time to time, for example when procedures are compiled. Therefore, any objects used in the formula must persist until the formula is deleted.

We recommend that you never use $ expressions in a dependency formula.

Chapter

IV-10

Advanced Topics

This chapter contains material on topics of interest to advanced Igor users.

Procedure Modules

Igor supports grouping procedure files into modules to prevent name conflicts and to isolate procedure packages from other packages. This feature is for intermediate to advance Igor programmers.

There are three types of modules:

- ProcGlobal
- Regular Modules
- Independent Modules

You set a procedure file's module using the **ModuleName** or **IndependentModule** pragmas. A procedure file that does not contain one of these pragmas is in the ProcGlobal module by default.

Igor compiles independent modules separately from other procedures. This allows independent modules to continue to work even if there are errors in other procedure files. Independent module programming is demanding and is intended for use by advance Igor programmers. See **Independent Modules** on page IV-238 for details.

Regular modules provide protection against name conflicts and are easier to work with than independent modules. Regular modules are intended for use by intermediate to advanced Igor programmers who are creating procedure packages. See **Regular Modules** for details.

Regular Modules

Regular modules provide a way to avoid name conflicts between procedure files. Regular modules are distinct from "independent modules" which are discussed in the next section.

Igor's module concept provides a way to group related procedure files and to prevent name conflicts between procedure packages. You specify that a procedure file is part of a regular module using the #pragma ModuleName statement. The statement is allowed in any procedure file but not in the built-in main procedure window.

A procedure file that does not contain a #pragma ModuleName statement (or a #pragma IndependentModule statement) defines public procedure names in the default ProcGlobal module. All functions in the procedure file can be called from any other procedure file in ProcGlobal or in any regular module by simply specifying the name without any name qualifications (explained below).

When you execute a function from the command line or use the **Execute** operation, you are operating in the ProcGlobal context.

Functions are public by default and private if declared using the static keyword. For example

```
// In a procedure file with no #pragma ModuleName or #pragma IndependentModule

static Function Test()           // "static" means private to procedure file
   Print "Test in ProcGlobal"
End

Function TestInProcGlobal()      // Public
   Print "TestInProcGlobal in ProcGlobal"
End
```

Because it is declared static, the Test function is private to its procedure file. Each procedure file can have its own static Test function without causing a name conflict.

The TestInProcGlobal function is public so there can be only one public function with this name in ProcGlobal.

In this example the static Test function is accessible only from the procedure file in which it is defined.

Sometimes you have a need to avoid name conflicts but still want to be able to call functions from other procedure files, from control action procedures or from the command line. This is where a regular module is useful.

You specify that a procedure file is in a regular module using the ModuleName pragma so that even static functions can be called from other procedure files. For example:

```
#pragma ModuleName = ModuleA      // The following procedures are in ModuleA

static Function Test()            // Semi-private
   Print "Test in ModuleA"
End

Function TestModuleA()            // Public
   Print "Test in ModuleA"
End
```

Because it is declared static, this Test function does not conflict with Test functions in other procedure files. But because it is in a regular module (ModuleA), it can be called from other procedure files using a qualified name:

```
ModuleA#Test()                    // Call Test in ModuleA
```

This qualified name syntax overrides the static nature of Test and tells Igor that you want to execute the Test function defined in ModuleA. The only way to access a static function from another procedure file is to put it in a regular module and use a qualified name.

If you are writing a non-trivial set of procedures, it is a good idea to use a module and to declare your functions static, especially if other people will be using your code. This prevents name conflicts with other procedures that you or other programmers write.

Make sure to choose a distinctive module name. Module names must be unique.

Regular Modules in Action Procedures and Hook Functions

Control action procedures and hook functions are called by Igor at certain times. They are executed in the ProcGlobal context, as if they were called from the command line. This means that you must use a qualified name to call a static function as an action procedure or a hook function. For example:

```
#pragma ModuleName = RegularModuleA

static Function ButtonProc(ba) : ButtonControl
   STRUCT WMButtonAction &ba

   switch (ba.eventCode)
      case 2: // mouse up
         Print "Running RegularModuleA#ButtonProc"
         break
   endswitch

   return 0
End

static Function CreatePanel()
   NewPanel /W=(375,148,677,228)
   Button button0,pos={106,23},size={98,20},title="Click Me"
   Button button0,proc=RegularModuleA#ButtonProc
End
```

RegularModuleA is the name we have chosen for the regular module for demonstration purposes. You should choose a more descriptive module name.

The use of a qualified name, RegularModuleA#ButtonProc, allows Igor to find and execute the static ButtonProc function in the RegularModuleA module even though ButtonProc is running in the ProcGlobal context.

To protect the CreatePanel function from name conflicts we also made it static. To create the panel, execute:

```
RegularModuleA#CreatePanel()
```

Regular Modules and User-Defined Menus

Menu item execution text also runs in the ProcGlobal context. If you want to call a routine in a regular module you must use a qualified name.

Continuing the example from the preceding section, here is how you would write a menu definition:

```
#pragma ModuleName = RegularModuleA

Menu "Macros"
   "Create Panel", RegularModuleA#CreatePanel()
End
```

See also **Procedure Modules** on page IV-236, **Independent Modules** below, **Controls and Control Panels** on page III-413, **User-Defined Hook Functions** on page IV-280 and **User-Defined Menus** on page IV-125.

Independent Modules

An independent module is a set of procedure files that are compiled separately from all other procedures. Because it is compiled separately, an independent module can run when other procedures are in an uncompiled state because the user is editing them or because an error occurred in the last compile. This allows the independent module's control panels and menus to continue to work regardless of user programming errors.

Creating an independent module adds complexity and requires a solid understanding of Igor programming. You should use an independent module if it is important that your procedures be runnable at all times. For example, if you have created a data acquisition package that must run regardless of what the user is doing, that would be a good candidate for an independent module.

A file is designated as being part of an independent module using the IndependentModule pragma:

```
#pragma IndependentModule = imName
```

Make sure to use a distinctive name for your independent module.

The IndependentModule pragma is not allowed in the built-in procedure window which is always in the ProcGlobal module.

It is normal for multiple procedure files that are part of the same package to be in the same independent module.

An independent module creates an independent namespace. Function names in an independent module do not conflict with the same names used in other modules. To call an independent module function from the ProcGlobal module or from a regular module the function must be public (non-static) and you must use a qualified name as illustrated in the next section.

An independent module can call only procedures in that independent module.

Independent Modules - A Simple Example

Here is a simple example using an independent module. This code must be in its own procedure file and not in the built-in procedure file:

```
#pragma IndependentModule = IndependentModuleA

static Function Test()              // static means private to file
   Print "Test in IndependentModuleA"
End

// This must be non-static to call from command line (ProcGlobal context)
Function CallTestInIndependentModuleA()
   Test()
End
```

From the command line (the ProcGlobal context):

```
CallTestInIndependentModuleA()                     // Error

IndependentModuleA#CallTestInIndependentModuleA()  // OK

IndependentModuleA#Test()                          // Error
```

The first command does not work because the functions in the independent module are accessible only using a qualified name. The second command does work because it uses a qualified name and because the function is public (non-static). The third command does not work because the function is private (static) and therefore is accessible only from the file in which it is defined. A static function in an independent module is not accessible from outside the procedure file in which it is defined unless it is in an enclosed regular module as described under **Regular Modules Within Independent Modules** on page IV-242.

SetIgorOption IndependentModuleDev=1

By default, the debugger is disabled for independent modules. It can be enabled using:

```
SetIgorOption IndependentModuleDev=1
```

Also by default, independent module procedure windows are not listed in the Windows→Procedure Windows submenu unless you use `SetIgorOption IndependentModuleDev=1`.

When `SetIgorOption IndependentModuleDev=1` is in effect, the Windows→Procedure Windows submenu shows all procedure windows, and those that belong to an independent module are listed with the independent module name in brackets. For example:

```
DemoLoader.ipf [WMDemoLoader]
```

This bracket syntax is used in the **WinList**, **FunctionList**, **DisplayProcedure**, and **ProcedureText** functions and operations.

To get the user experience, as opposed to the programmer experience, return to normal operation by executing:

```
SetIgorOption IndependentModuleDev=0
```

Independent Module Development Tips

Development of an independent module may be easier if it is first done as for normal code. Add the module declaration

```
#pragma IndependentModule = moduleName
```

only after the code has been fully debugged and is working properly.

A procedure file that is designed to be #included should ideally work inside or outside of an independent module. Read the sections on independent modules below to learn what the issues are.

When developing an independent module, you will usually want to execute:

```
SetIgorOption IndependentModuleDev=1
```

Independent Modules and #include

If you #include a procedure file from an independent module, Igor copies the #included file into memory and makes it part of the independent module by inserting a #pragma IndependentModule statement at the start of the copy. If the same file is included several times, there will be several copies, each with a different independent module name.

Warning: Do not edit procedure windows created by #including into an independent module because they are temporary and your changes will not be saved. You would not want to save them anyway because Igor has modified them.

Warning: Do not #include files that already contain a #pragma IndependentModule statement unless the independent module name is the same.

Limitations of Independent Modules

Independent modules are not for every-day programming and are more difficult to create than normal modules because of the following limitations:

1. Macros and Procs are not supported.
2. Button and control dialogs do not list functions in an independent module.
3. Functions in an independent module can not call functions in other modules except through the Execute operation.

Independent Modules in Action Procedures and Hook Functions

Normally you must use a qualified name to invoke a function defined in an independent module from the ProcGlobal context. Control action procedures and hook functions execute in the ProcGlobal context. But, as a convenience and to make #include files more useful, Igor eliminates this requirement when you create controls and specify hook functions from a user-defined function in an independent module.

When you execute an operation that creates a control or specifies a hook function while running in an independent module, Igor examines the specified control action function name or hook function name. If the named function is defined in the same independent module, Igor automatically inserts the independent module name. This means you can write something like:

```
#pragma IndependentModule = IndependentModuleA
Function SetProcAndHook()
   Button b0, proc=ButtonProc
   SetWindow hook(Hook1)=HookFunc
End
```

You don't have to write:

```
#pragma IndependentModule = IndependentModuleA
Function SetProcAndHook()
   Button b0, proc=IndependentModuleA#ButtonProc
   SetWindow hook(Hook1)=IndependentModuleA#HookFunc
End
```

Such independent module name insertion is only done when an operation is called from a function defined in an independent module. It is not done if the operation is executed from the command line or via **Execute**.

The control action function or hook function must be public (non-static) except as described under **Regular Modules Within Independent Modules** on page IV-242.

Here is a working example:

```
#pragma IndependentModule = IndependentModuleA

Function ButtonProc(ba) : ButtonControl   // Must not be static
   STRUCT WMButtonAction &ba

   switch (ba.eventCode)
      case 2: // mouse up
         Print "Running IndependentModuleA#ButtonProc"
         break
   endswitch

   return 0
End

Function CreatePanel()
   NewPanel /W=(375,148,677,228)
   Button button0,pos={106,23},size={98,20},title="Click Me"
   Button button0,proc=ButtonProc
End
```

Independent Modules and User-Defined Menus

Independent modules can contain user-defined menus. When you choose a user-defined menu item, Igor determines if the menu item was defined in an independent module. If so, and if the menu item's execution text starts with a call to a function defined in the independent module, then Igor prepends the independent module name before executing the text. This means that the second and third menu items in the following example both call IndependentModuleA#DoAnalysis:

```
#pragma IndependentModule = IndependentModuleA

Menu "Macros"
   "Load Data File/1", Beep; LoadWave/G
   "Do Analysis/2", DoAnalysis() // Igor automatically prepends IndependentModuleA#
   "Do Analysis/3", IndependentModuleA#DoAnalysis()
End

Function DoAnalysis()
   Print "DoAnalysis in IndependentModuleA"
End
```

This behavior on Igor's part makes it possible to #include a procedure file that creates menu items into an independent module and have the menu items work. However, in many cases you will not want a #included file's menu items to appear. You can suppress them using `menus=0` option in the #include statement. See **Turning the Included File's Menus Off** on page IV-167.

Note: If a procedure file with menu definitions is included into multiple independent modules, the menus are repeatedly defined (see **Independent Modules and #include** on page IV-240). Judicious use of the menus=0 option in the #include statements helps prevent this. See **Turning the Included File's Menus Off** on page IV-167.

When the execution text doesn't start with a user-defined function name, as for the first menu item in this example, Igor executes the text without altering it.

Independent Modules and Popup Menus

In an independent module, implementing a popup menu whose items are determined by a function call at click time requires special care. For example, outside of an independent module, this works:

```
Function/S MyPopMenuList()
   return "Item 1;Item2;"
```

```
End
...
PopupMenu pop0 value=#"MyPopMenuList()"       // Note the quotation marks
```

But inside an independent module you need this:

```
#pragma IndependentModule=MyIM
Function/S MyPopMenuList()
   return "Item 1;Item2;"
End
...
String cmd= GetIndependentModuleName()+"#MyPopMenuList()"
PopupMenu pop0 value=#cmd                     // No enclosing quotation marks
```

GetIndependentModuleName returns the name of the independent module to which the currently-running function belongs or "ProcGlobal" if the currently-running function is not part of an independent module.

You could change the command string to:

```
PopupMenu pop0 value=#"MyIM#MyPopMenuList()"
```

but using GetIndependentModuleName allows you to disable the IndependentModule pragma by commenting it out and have the code still work, which can be useful during development. With the pragma commented out you are running in ProcGlobal context and GetIndependentModuleName returns "ProcGlobal".

When the user clicks the popup menu, Igor generates the menu items by evaluating the text specified by the PopupMenu value keyword as an Igor expression. The expression ("MyIM#MyPopMenuList()" in this case) is evaluated in the ProcGlobal context. In order for Igor to find the function in the independent module, it must be public (non-static), except as describe under **Regular Modules Within Independent Modules** on page IV-242, and you must use a qualified name.

Note that #cmd is not the same as #"cmd". The #cmd form was introduced with Igor Pro 6. The string variable cmd is evaluated when PopupMenu runs which occurs in the context of the independent module. The contents of cmd ("MyIM#MyPopMenuList()" in this case) are stored in the popup menu's internal data structure. When the popup menu is clicked, Igor evaluates the stored text as an Igor expression in the ProcGlobal context. This causes the function MyIM#MyPopMenuList to run.

With the older #"cmd" syntax, the stored text is evaluated only when the popup menu is clicked, not when the PopupMenu operation runs, and this evaluation occurs in the ProcGlobal context. It is too late to capture the independent module in which the text should be evaluated.

Regular Modules Within Independent Modules

It is usually not necessary but you can create a regular module within an independent module. For example:

```
#pragma IndependentModule = IndependentModuleA
#pragma ModuleName = RegularModuleA
Function Test()
   Print "Test in RegularModuleA within IndependentModuleA"
End
```

Here RegularModuleA is a regular module within IndependentModuleA.

To call the function Test from outside of the independent module you must qualify the call like this:

```
IndependentModuleA#RegularModuleA#Test()
```

This illustrates that the independent module establishes its own namespace (IndependentModuleA) which can host one level of sub-namespace (RegularModuleA). By contrast, a regular module creates a namespace within the global namespace (called ProcGlobal) and can not host additional sub-namespaces.

This nesting of modules is useful to prevent name conflicts in a large independent module project comprising multiple procedure files. Otherwise it is not necessary.

Because all procedure files in a given independent module are compiled separately from all other files, function names never conflict with those outside the group and there is little or no need to use the static designation on functions in an independent module. However, if need be, you can call static functions in a regular module inside an independent module from outside the independent module using a triple-qualified name:

```
IndependentModuleName#RegularModuleName#FunctionName()
```

Calling Routines From Other Modules

Code in an independent module can not directly call routines in other modules and usually should not need to. If you must call a routine from another module, you can do it using the **Execute** operation. You must use a qualified name. For example:

```
Execute "ProcGlobal#foo()"
```

To call a function in a regular module, you must prepend ProcGlobal and the regular module name to the function name:

```
Execute "ProcGlobal#MyRegularModule#foo()"
```

Calling a nonstatic function in a different independent modules requires prepending just the other independent module name:

```
Execute "OtherIndependentModule#bar()"
```

Calling static functions in other independent modules requires prepending the independent module name and a regular module name:

```
Execute "OtherIndependentModule#RegularModuleName#staticbar()"
```

Using Execute Within an Independent Module

If you need to call a function in the current independent module using Execute, you can compose the name using the **GetIndependentModuleName** function. For example, outside of an independent module the commands would be:

```
String cmd = "WS_UpdateWaveSelectorWidget(\"Panel0\", \"selectorWidgetName\")"
Execute cmd
```

But inside an independent module the commands are:

```
#pragma IndependentModule=MyIM
String cmd="WS_UpdateWaveSelectorWidget(\"Panel0\", \"selectorWidgetName\")"
cmd = GetIndependentModuleName() + "#" + cmd    // Make qualified name
Execute cmd
```

You could change the command string to:

```
cmd = "MyIM#" + cmd
```

but using GetIndependentModuleName allows you to disable the IndependentModule pragma by commenting it out and have the code still work, which can be useful during development. With the pragma commented out you are running in ProcGlobal context and GetIndependentModuleName returns "ProcGlobal".

Independent Modules and Dependencies

GetIndependentModuleName is also useful for defining dependencies using functions in the current independent module. Dependencies are evaluated in the global procedure context (ProcGlobal). In order for dependencies to evaluate correctly, the dependency must use GetIndependentModuleName to create a formula to pass to the **SetFormula** operation. For example, outside of an independent module, this works:

```
String formula = "foo(root:wave0)"
SetFormula root:aVariable $formula
```

But inside an independent module you need this:

```
#pragma IndependentModule=MyIM
String formula = GetIndependentModuleName() + "#foo(root:wave0)"
SetFormula root:aVariable $formula
```

Independent Modules and Pictures

To allow DrawPICT to use a picture in the picture gallery, you must prepend GalleryGlobal# to the picture name:

```
DrawPICT 0,0,1,1,GalleryGlobal#PICT_0
```

Without GalleryGlobal, only proc pictures can be used in an independent module.

Making Regular Procedures Independent-Module-Compatible

You may want to make an existing set of procedures into an independent module. Alternatively, you may want to make an existing procedure independent-module-compatible so that it can be #included into an independent module. This section outlines the necessary steps.

1. If you are creating an independent module, add the IndependentModule pragma:

   ```
   #pragma IndependentModule=<NameOfIndependentModule>
   ```
2. Change any Macro or Proc procedures to functions.
3. Make Execute commands suitable for running in the ProcGlobal context or in an independent module using GetIndependentModuleName. See **Using Execute Within an Independent Module** on page IV-243.
4. Make PopupMenu controls that call a string function to populate the menu work in the ProcGlobal context or in an independent module using GetIndependentModuleName. See **Independent Modules and Popup Menus** on page IV-241.
5. Make any dependencies work in the ProcGlobal context or in an independent module using GetIndependentModuleName. See **Independent Modules and Dependencies** on page IV-243.

See also **Procedure Modules** on page IV-236, **Regular Modules** on page IV-236, **Controls and Control Panels** on page III-413, **User-Defined Hook Functions** on page IV-280, **User-Defined Menus** on page IV-125, and **GetIndependentModuleName** on page V-303.

Public and Static Functions

The static keyword marks a user-defined function as private to the file in which it is defined. It can be called within that file only, with an exception explained below for "regular modules".

If you are a beginning Igor programmer and you are not familiar with the advanced concepts of Regular Modules and Independent Modules, it is sufficient to think of this like this:

- A static function can be called only from the procedure file in which it is defined.
- A public function can be called from any procedure file.

The rest of this section is for advanced programmers with an understanding of **Regular Modules** and **Independent Modules**.

- A public function defined in the default ProcGlobal module can be called from another file in the ProcGlobal module or from a regular module without using a qualified name.
- A public function defined in a regular module can be called from a procedure file in ProcGlobal or

from a regular module without using a qualified name.

- A public function defined in an independent module can be called from a procedure file in ProcGlobal or from a regular module using a qualified name such as IndependentModuleA#Test.
- A public function defined in an independent module can not be called from another independent module.
- A static function defined in ProcGlobal can be called only from the file in which it is defined.
- A static function defined in a regular module can be called from a procedure file in ProcGlobal or from another regular module using a qualified name such as RegularModuleA#Test.
- A static function defined in a regular module can not be called from an independent module.
- A static function defined in an independent module can be called only from the procedure file in which it is defined.
- An independent module can call only functions defined in that independent module.

Sound

Two operations are provided for playing of sound through the computer speakers:

- PlaySound
- PlaySnd (*Macintosh*)

The **PlaySound** operation takes the sound data from a wave.

The obsolete **PlaySnd** operation gets its data from a Macintosh 'snd ' resource stored in a file.

A number of sound input operations are provided: **SoundInStatus** (page V-888) , **SoundInSet** (page V-887) , **SoundInRecord** (page V-887) , **SoundInStartChart** (page V-888) and **SoundInStopChart** (page V-889) . Several example experiments that use these routines can be found in your Igor Pro Folder in the Examples folder.

The **SoundLoadWave** operation loads various sound file formats into waves and **SoundSaveWave** saves wave data to sound files. These operations replace SndLoadWave, SoundSaveAIFF and SoundSaveWAV from the obsolete SndLoadSaveWave XOP.

Movies

You can create movies, optionally with a soundtrack, and extract frames from movies for analysis.

Playing Movies

Use the PlayMovie operation to play a movie in your default movie viewing program.

Creating Movies

You can create a movie from a graph, page layout, or Gizmo window. To do this, you write a procedure that modifies the window and adds a frame to the movie in a loop. On Windows, you can include audio.

Here are the operations used to create and play a movie:

- **NewMovie**
- **AddMovieFrame**
- **AddMovieAudio**
- **CloseMovie**
- **PlayMovie**
- **PlayMovieAction**

The NewMovie operation creates a movie file and also defines the movie frame rate and optional audio track specifications.

Before calling NewMovie, you need to prepare the first frame of your movie as the target graph, page layout, or Gizmo window.

If you will be using audio you also need to prepare a sound wave. The sound wave can be of any time duration but usually will either be the entire length of the movie or will be the length of one video frame. As of Igor Pro 7, sound is not supported on Macintosh.

After creating the file and the first video frame and optional audio, you use AddMovieFrame to add as many video frames as you wish. You may also add more audio using the AddMovieAudio operation. Finally you use the CloseMovie and PlayMovie operations.

When you write a procedure to generate a movie, you need to call the DoUpdate operation after all modifications to the graph, page layout, or Gizmo window and before calling AddMovieFrame. This allows Igor to process any changes you have made to the window.

In addition to creating a movie from a window, you can also create movies from pictures in the picture gallery (see **Pictures** on page III-509) using the /PICT flag with NewMovie and AddMovieFrame. You can put pictures of Igor graphs, tables, page layouts, and Gizmo plots in the gallery using **SavePICT**.

Extracting Movie Frames

You can extract individual frames from a movie and can control movie playback using **PlayMovieAction**.

Movie Programming Examples

For examples of programming with movies, choose File→Example Experiments→Movies & Audio.

Timing

There are two methods you can use when you want to measure elapsed time:

- The ticks counter using the ticks function
- The microsecond timer using **StartMSTimer** and **StopMSTimer**

Ticks Counter

You can easily measure elapsed time with a precision of 1/60th of a second using the ticks function. It returns the tick count which starts at zero when you first start your computer and is incremented at a rate of approximately 60 Hz rate from then on.

Here is an example of typical use:

```
…
Variable t0
…
t0= ticks
<operations you wish to time>
printf "Elapsed time was %g seconds\r",(ticks-t0)/60
…
```

Microsecond Timer

You can measure elapsed time to microsecond accuracy for durations up to 35 minutes using the microsecond timer. See the **StartMSTimer** function (page V-906) for details and an example.

Packages

A package is a set of files that adds significant functionality to Igor. Packages consist of procedure files and may also include XOPs, help files and other supporting files.

A package usually adds one or more items to Igor's menus that allow the user to interactively load the package, access its functionality, and unload the package.

A package typically provides some level of user-interface, such as a menu item and a control panel, for accessing the added functionality. It may store settings in experiments or in global preferences.

A package is typically loaded into memory and unloaded at the user's request.

Igor comes pre-configured with numerous WaveMetrics packages accessed through the Data→Packages, Analysis→Packages, Misc→Packages, Windows→New→Packages and Graph→Packages submenus as well as others. Take a peek at these submenus to see what packages are supplied with Igor.

Menu items for WaveMetrics packages are added to Igor's menus by the WMMenus.ipf procedure file which is shipped in the Igor Procedures folder. WMMenus.ipf is hidden unless you enable independent module development. See **Independent Modules** on page IV-238.

Creating a Package

This section shows how to create a package through a simple example. The package is called "Sample Package". It adds a Load Sample Package item to the Macros menu. When the user chooses Load Sample Package, the package's procedure file is loaded. This adds two additional items to the Macros menu: Hello From Sample Package and Unload Sample Package.

The package consists of two procedure files stored in a folder in the Igor Pro User Files folder. If you are not familiar with Igor Pro User Files, take a short detour and read **Special Folders** on page II-30 and **Igor Pro User Files** on page II-31.

The sample package is installed as follows:

```
Igor Pro User Files
   Sample Package
      Sample Package Loader.ipf
      Sample Package.ipf
   Igor Procedures
      Alias or shortcut pointing to the "Sample Package Loader.ipf" file
   User Procedures
      Alias or shortcut pointing to the "Sample Package" folder
```

Putting the alias/shortcut for the "Sample Package Loader.ipf" in Igor Procedures causes Igor to load that file at launch time. The file adds the "Load Sample Package" item to the Macros menu. See **Global Procedure Files** on page III-399 for details.

Putting the alias/shortcut for the "Sample Package" folder in User Procedures causes Igor to search that folder when a #include is invoked. See **Shared Procedure Files** on page III-400 for details.

A real package might include other procedure files and a help file in the "Sample Package" folder.

To try this out yourself, follow these steps:

1. Create the "Sample Package" folder in your Igor Pro User Files folder.

   You can locate your Igor Pro User Files folder using the Help menu.

2. Create a new procedure file named "Sample Package Loader.ipf" in the "Sample Package" folder and enter the following contents in the file:

```
Menu "Macros"
   "Load Sample Package", /Q, LoadSamplePackage()
End

Function LoadSamplePackage()
   Execute/P/Q/Z "INSERTINCLUDE \"Sample Package\""
   Execute/P/Q/Z "COMPILEPROCEDURES "// Note the space before final quote
End
```

Save the procedure file.

3. Create a new procedure file named "Sample Package.ipf" in the "Sample Package" folder and enter the following contents in the file:

```
Menu "Macros"
   "Hello From Sample Package", HelloFromSamplePackage()
   "Unload Sample Package", UnloadSamplePackage()
End

Function HelloFromSamplePackage()
   DoAlert /T="Sample Package Wants to Say" 0, "Hello!"
End

Function UnloadSamplePackage()
   Execute /P /Q /Z "DELETEINCLUDE \"Sample Package\""
   Execute /P /Q /Z "COMPILEPROCEDURES "// Note the space before final quote
End
```

   Save the procedure file.

4. In the desktop, make an alias or shortcut for "Sample Package Loader.ipf" file and put it in the Igor Procedures folder in the Igor Pro User Files folder.

   This causes Igor to load the "Sample Package Loader.ipf" file at launch time. This is how the Load Sample Package menu item gets into the Macros menu.

5. In the desktop, make an alias or shortcut for the "Sample Package" folder and put it in the User Procedures folder in the Igor Pro User Files folder.

   This causes Igor to search the "Sample Package" folder when a #include is invoked. This allows Igor to find the "Sample Package.ipf" file when it is #included.

6. Quit and restart Igor so that Igor will load the "Sample Package Loader.ipf" file.

   If you prefer you can just manually make sure that "Sample Package Loader.ipf" is open and "Sample Package.ipf" is closed. This simulates the state of affairs after restarting Igor.

7. Choose Windows→Procedure Windows and verify that Igor has loaded the "Sample Package Loader.ipf" file.

8. Click the Macros menu and verify that the "Load Sample Package" item is present.

9. Choose Macros→Load Sample Package.

   The LoadSamplePackage function runs, adds a #include statement to the built-in procedure window, and forces procedures to be recompiled. This cause Igor to load the "Sample Package.ipf" procedure file which contains the bulk of the package's procedures and adds items to the Macros menu.

10. Click the Macros menu and notice that the "Hello From Sample Package" and "Unload Sample Package" items have been added.

11. Choose Macros→Hello From Sample Package.

    The package displays an alert. A real package would do something more exciting.

12. Choose Macros→Unload Sample Package.

    The UnloadSamplePackage function runs, removes the #include statement from the built-in procedure window, and forces procedures to be recompiled. This cause Igor to unload the "Sample Package.ipf" procedure.

13. Click the Macros menu and notice that the "Hello From Sample Package" and "Unload Sample Package" items have been removed.

Most real packages do not create Unload menu items. Instead they provide an Unload Package button in a control panel or automatically unload when a control panel is closed. Or they might not support unloading.

A real package typically does not include "Package" in its name or in its menu items.

Lightweight Packages

A lightweight package is one that consists of at most a few procedure files and does not create clutter in the current experiment unless it is actually used.

If your package is lightweight you might prefer to dispense with loading and unload it and just keep it loaded all the time. To do this you would organize your files like this:

```
Igor Pro User Files
   Your Package
      Your Package Part 1.ipf
      Your Package Part 2.ipf
   Igor Procedures
      Alias or shortcut pointing to the "Your Package" folder
```

Here both of your package procedure files are global, meaning that Igor loads them at launch time and never unloads them. You do not need procedures for loading and unloading your package.

If you have an ultra-light package, consisting of just a single procedure file, you can dispense with the "Your Package" folder and put the procedure file directly in the Igor Procedures folder.

Managing Package Data

When you create a package of procedures, you need some place to store private data used by the package to keep track of its state. It's important to keep this data separate from the user's data to avoid clutter and to protect your data from inadvertent changes.

Private data should be stored in a data folder named after the package inside a generic data folder named Packages. For example, if your package is named My Package you would store your private data in `root:Packages:My Package`.

There are two general types of private data that you might need to store: overall package data and per-instance data. For example, for a data acquisition package, you may need to store data describing the state of the acquisition as a whole and other data on a per-channel basis.

Creating and Accessing the Package Data Folder

This section demonstrates the recommended way to create and access a package data folder. We use a bottleneck function that returns a **DFREF** for the package data folder. If the package data folder does not yet exist, the bottleneck function creates and initializes it. This way calling functions don't need to worry about whether the package data folder has been created.

First we write a function to create and initialize the package data folder:

```
Function/DF CreatePackageData()     // Called only from GetPackageDFREF
   // Create the package data folder
   NewDataFolder/O root:Packages
   NewDataFolder/O root:Packages:'My Package'

   // Create a data folder reference variable
   DFREF dfr = root:Packages:'My Package'

   // Create and initialize package data
   Variable/G dfr:gVar1 = 1.0
   String/G dfr:gStr1 = “hello”
   Make/O dfr:wave1
   WAVE wave1 = dfr:wave1
   wave1= x^2

   return dfr
End
```

Now we can write the bottleneck function:

```
Function/DF GetPackageDFREF()
   DFREF dfr = root:Packages:'My Package'
   if (DataFolderRefStatus(dfr) != 1)      // Data folder does not exist?
      DFREF dfr = CreatePackageData()      // Create package data folder
   endif
   return dfr
End
```

GetPackageDFREF would be used like this:

```
Function/DF DemoPackageDFREF()
   DFREF dfr = GetPackageDFREF()

   // Read a package variable
   NVAR gVar1 = dfr:gVar1
   Printf "On entry gVar1=%g\r", gVar1

   // Write to a package variable
   gVar1 += 1
   Printf "Now gVar1=%g\r", gVar1
End
```

All functions that access the package data folder should do so through GetPackageDFREF. The calling functions do not need to worry about whether the data folder has been created and initialized because GetPackageDFREF does this for them.

Creating and Accessing the Package Per-Instance Data Folders

Here we extend the technique of the preceding section to handle per-instance data. This example shows how you might handle per-channel data in a data acquisition package. If your package does not use per-instance data then you can skip this section.

First we write a function to create and initialize the per-instance package data folder:

```
Function/DF CreatePackageChannelData(channel)    // Called only from
   Variable channel     // 0 to 3                // GetPackageChannelDFREF

   DFREF dfr = GetPackageDFREF()    // Access main package data folder

   String dfName = "Channel" + num2istr(channel)    // Channel0, Channel1, ...

   // Create the package channel data folder
   NewDataFolder/O dfr:$dfName

   // Create a data folder reference variable
   DFREF channelDFR = dfr:$dfName

   // Initialize per-instance data
   Variable/G channelDFR:gGain = 5.0
   Variable/G channelDFR:gOffset = 0.0

   return channelDFR
End
```

Now we can write the bottleneck function:

```
Function/DF GetPackageChannelDFREF(channel)
   Variable channel     // 0 to 3

   DFREF dfr = GetPackageDFREF()    // Access main package data folder
   String dfName = "Channel" + num2istr(channel)    // Channel0, Channel1, ...
   DFREF channelDFR = dfr:$dfName
   if (DataFolderRefStatus(channelDFR) != 1)     // Data folder does not exist?
      DFREF channelDFR = CreatePackageChannelData(channel)  // Create it
   endif
   return channelDFR
End
```

GetPackageChannelDFREF would be used like this:

```
Function/DF DemoPackageChannelDFREF(channel)
   Variable channel     // 0 to 3

   DFREF channelDFR = GetPackageChannelDFREF(channel)

   // Read a package variables
   NVAR gGain = channelDFR:gGain
   NVAR gOffset = channelDFR:gOffset
   Printf "Channel %d: Gain=%g, offset=%g\r", channel, gGain, gOffset
End
```

All functions that access a package channel data folder should do so through GetPackageChannelDFREF. The calling functions do not need to worry about whether the data folder has been created and initialized because GetPackageChannelDFREF does this for them.

Saving Package Preferences

If you are writing a sophisticated package of Igor procedures you may want to save preferences for your package. For example, if your package creates a control panel that can be opened in any experiment, you may want it to remember its position on screen between invocations. Or you may want to remember various settings in the panel from one invocation to the next.

Such “state” information can be stored either separately in each experiment or it can be stored just once for all experiments in preferences. These two approaches both have their place, depending on circumstances. But, if your package creates a control panel that is intended to be present at all times and used in any experiment, then the preferences approach is usually the best fit.

If you choose the preferences approach, you will store your package preference file in a directory created for your package. Your package directory will be in the Packages directory, inside Igor’s own preferences directory.

The location of Igor’s Packages directory depends on the operating system and the particular user’s configuration. You can find where it is on a particular system by executing:

```
Print SpecialDirPath("Packages", 0, 0, 0)
```

Important:You must choose a very distinctive name for your package because that is the only thing that prevents some other package from overwriting yours. All package names starting with "WM" are reserved for WaveMetrics.

A package name is limited to 255 bytes and must be a legal name for a directory on disk.

If you use a name longer than 31 bytes, your package will require Igor Pro 8.00 or later.

There are two ways to store package preference data:

- In a special-format binary file stored in your package directory
- As Igor waves and variables in an Igor experiment file stored in your package directory

The special-format binary file approach is relatively simple to implement but is not suitable for storing very large amounts of data. In most cases it is not necessary to store very large amounts of data so this is the way to go.

The use of the Igor experiment file supports storing a large amount of preference data but creates a problem of synchronizing your preference data stored in memory and your preference data stored on disk. It also leads to a proliferation of preference data stored in various experiments. You should avoid using this technique if possible.

Saving Package Preferences in a Special-Format Binary File

This approach supports preference data consisting of a collection of numeric and string data. You define a structure encapsulating your package preference data. You use the **LoadPackagePreferences** operation (page V-505) to load your data from disk and the **SavePackagePreferences** operation (page V-825) to save it to disk.

SavePackagePreferences stores data from your package's preferences data structure in memory. LoadPackagePreferences returns that data to you via the same structure.

SavePackagePreferences also creates a directory for your package preferences and stores your data in a file in that directory. Your package directory is located in the Packages directory in Igor's preferences directory. The job of storing the preferences data in the file is handled transparently which, by default, automatically flushes your data to the file when the current experiment is saved or closed and when Igor quits.

You would call LoadPackagePreferences every time you need to access your package preference data and SavePackagePreferences every time you want to change your package preference data. You pass to these operations an instance of a structure that you define.

Here are example functions from the Package Preferences Demo experiment that use the LoadPackagePreferences and SavePackagePreferences operations to implement preferences for a particular package:

```
// NOTE: The package name you choose must be distinctive!
static StrConstant kPackageName = "Acme Data Acquisition"
static StrConstant kPrefsFileName = "PanelPreferences.bin"
static Constant kPrefsVersion = 100
static Constant kPrefsRecordID = 0

Structure AcmeDataAcqPrefs
   uint32version    // Preferences structure version number. 100 means 1.00.
   double panelCoords[4]      // left, top, right, bottom
   uchar phaseLock
   uchar triggerMode
   double ampGain
   uint32 reserved[100]       // Reserved for future use
EndStructure

// DefaultPackagePrefsStruct(prefs)
// Sets prefs structure to default values.
static Function DefaultPackagePrefsStruct(prefs)
   STRUCT AcmeDataAcqPrefs &prefs

   prefs.version = kPrefsVersion

   prefs.panelCoords[0] = 5          // Left
   prefs.panelCoords[1] = 40         // Top
   prefs.panelCoords[2] = 5+190      // Right
   prefs.panelCoords[3] = 40+125     // Bottom
   prefs.phaseLock = 1
   prefs.triggerMode = 1
   prefs.ampGain = 1.0

   Variable i
   for(i=0; i<100; i+=1)
      prefs.reserved[i] = 0
   endfor
End

// SyncPackagePrefsStruct(prefs)
// Syncs package prefs structures to match state of panel.
// Call this only if the panel exists.
static Function SyncPackagePrefsStruct(prefs)
   STRUCT AcmeDataAcqPrefs &prefs

   // Panel does exists. Set prefs to match panel settings.
   prefs.version = kPrefsVersion
```

```
    GetWindow AcmeDataAcqPanel wsize
    // NewPanel uses device coordinates. We therefore need to scale from
    // points (returned by GetWindow) to device units for windows created
    // by NewPanel.
    Variable scale = ScreenResolution / 72
    prefs.panelCoords[0] = V_left * scale
    prefs.panelCoords[1] = V_top * scale
    prefs.panelCoords[2] = V_right * scale
    prefs.panelCoords[3] = V_bottom * scale

    ControlInfo /W=AcmeDataAcqPanel PhaseLock
    prefs.phaseLock = V_Value        // 0=unchecked; 1=checked

    ControlInfo /W=AcmeDataAcqPanel TriggerMode
    prefs.triggerMode = V_Value      // Menu item number starting from on

    ControlInfo /W=AcmeDataAcqPanel AmpGain
    prefs.ampGain = str2num(S_value)  // 1, 2, 5 or 10
End

// InitPackagePrefsStruct(prefs)
// Sets prefs structures to match state of panel or
// to default values if panel does not exist.
static Function InitPackagePrefsStruct(prefs)
    STRUCT AcmeDataAcqPrefs &prefs

    DoWindow AcmeDataAcqPanel
    if (V_flag == 0)
        // Panel does not exist. Set prefs struct to default.
        DefaultPackagePrefsStruct(prefs)
    else
        // Panel does exists. Sync prefs struct to match panel state.
        SyncPackagePrefsStruct(prefs)
    endif
End

static Function LoadPackagePrefs(prefs)
    STRUCT AcmeDataAcqPrefs &prefs

    // This loads preferences from disk if they exist on disk.
    LoadPackagePreferences kPackageName, kPrefsFileName, kPrefsRecordID, prefs

    // If error or prefs not found or not valid, initialize them.
    if (V_flag!=0 || V_bytesRead==0 || prefs.version!=kPrefsVersion)
        InitPackagePrefsStruct(prefs) // Set from panel if it exists or to default values.
        SavePackagePrefs(prefs)       // Create initial prefs record.
    endif
End

static Function SavePackagePrefs(prefs)
    STRUCT AcmeDataAcqPrefs &prefs

    SavePackagePreferences kPackageName, kPrefsFileName, kPrefsRecordID, prefs
End
```

NOTE: The package preferences structure, AcmeDataAcqPrefs in this case, must not use fields of type Variable, String, WAVE, NVAR, SVAR or FUNCREF because these fields refer to data that may not exist when LoadPackagePreferences is called.

The structure can use fields of type char, uchar, int16, uint16, int32, uint32, int64, uint64, float and double as well as fixed-size arrays of these types and substructures with fields of these types.

Use the reserved field to add fields to the structure in a backward-compatible fashion. For example, a subsequent version of the structure might look like this:

```
Structure AcmeDataAcqPrefs
    uint32   // Preferences structure version number. 100 means 1.00.
    double panelCoords[4]       // left, top, right, bottom
    uchar phaseLock
    uchar triggerMode
    double ampGain
```

```
   uint32 triggerDelay
   uint32 reserved[99]        // Reserved for future use
EndStructure
```

Here the triggerDelay field was added and size of the reserved field was reduced to keep the overall size of the structure the same. The AcmeDataAcqLoadPackagePrefs function would also need to be changed to set the default value of the triggerDelay field.

If you need to change the structure such that its size changes or its fields are changed in an incompatible manner then you must change your structure version, which will overwrite old preferences with new preferences.

A functioning example using this technique can be found in:

“Igor Pro Folder:Examples:Programming:Package Preferences Demo.pxp”

In the example above we store just one structure in the preference file. However LoadPackagePreferences and SavePackagePreferences allow storing any number of structures of the same or different types in the preference file. You can store either multiple instances of the same structure or multiple different structures. You must assign a unique nonnegative integer as a record ID for each structure stored and pass this record ID to LoadPackagePreferences and SavePackagePreferences. You could use this feature, for example, to store a different structure for each type of control panel that your package presents. Since all data is cached in memory you should not attempt to store hundreds or thousands of structures.

In almost all cases a particular package will need just one preference file. For the rare cases where this is inconvenient, LoadPackagePreferences and SavePackagePreferences allow each package to create any number of preference files, each with a distinct file name. All of the preference files for a particular package are stored in the same directory, the package’s preferences directory. Each file can store a different set of structure. However, the code that implements this feature is not tuned to handle large numbers of files so you should not use this feature indiscriminately.

Saving Package Preferences in an Experiment File

This approach supports package preference data consisting of waves, numeric variables and string variables. It is more difficult to implement than the special-format binary file approach and is not recommended except for expert programmers and then only if the previously described approach is not suitable.

You use the SaveData operation to store your waves and variables in a packed experiment file in your package directory on disk. You can later use the LoadData operation to load the waves and variables into a new experiment.

You must create your package directory as illustrated by the SavePackagePrefs function below.

The following example functions save and load package preferences. These functions assume that the package preferences consist of all waves and variables at the top level of the package’s data folder. You may need to customize these functions for your situation.

```
// SavePackagePrefs(packageName)
// Saves the top-level waves, numeric variables and string variables
// from the data folder for the named package into a file in the Igor
// preferences hierarchy on disk.
Function SavePackagePrefs(packageName)
   String packageName             // NOTE: Use a distinctive package name.

   // Get path to Packages preferences directory on disk.
   String fullPath = SpecialDirPath("Packages", 0, 0, 0)
   fullPath += packageName

   // Create a directory in the Packages directory for this package
   NewPath/O/C/Q tempPackagePrefsPath, fullPath

   fullPath += ":Preferences.pxp"

   DFREF saveDF = GetDataFolderDFR()
   SetDataFolder root:Packages:$packageName
   SaveData/O/Q fullPath          // Save the preference file
   SetDataFolder saveDF
```

```
    // Kill symbolic path but leave directory on disk.
    KillPath/Z tempPackagePrefsPath
End

// LoadPackagePrefs(packageName)
// Loads the data from the previously-saved package preference file,
// if it exist, into the package's data folder.
// Returns 0 if the preference file existed, -1 if it did not exist.
// In either case, this function creates the package's data folder if it
// does not already exist.
// LoadPackagePrefs does not affect any other data already in the
// package's data folder.
Function LoadPackagePrefs(packageName)
    String packageName              // NOTE: Use a distinctive package name.

    Variable result = -1
    DFREF saveDF = GetDataFolderDFR()

    NewDataFolder/O/S root:Packages     // Ensure root:Packages exists
    NewDataFolder/O/S $packageName      // Ensure package data folder exists

    // Find the disk directory in the Packages directory for this package
    String fullPath = SpecialDirPath("Packages", 0, 0, 0)
    fullPath += packageName
    GetFileFolderInfo/Q/Z fullPath
    if (V_Flag == 0)                    // Disk directory exists?
        fullPath += ":Preferences.pxp"
        GetFileFolderInfo/Q/Z fullPath
        if (V_Flag == 0)                // Preference file exist?
            LoadData/O/R/Q fullPath     // Load the preference file.
            result = 0
        endif
    endif

    SetDataFolder saveDF
    return result
End
```

The hard part of using the experiment file for saving package preferences is not in saving or loading the package preference data but in choosing when to save and load it so that the latest preferences are always used. There is no ideal solution to this problem but here is one strategy:

1. When package preference data is needed (e.g., you are about to create your control panel and need to know the preferred coordinates), check if it exists in memory. If not load it from disk.
2. When the user does a New Experiment or quits Igor, if package preference data exists in memory, save it to disk. This requires that you create an IgorNewExperimentHook function and an IgorQuitHook function.
3. When the user opens an experiment file, if it contains package preference data, delete it and reload from disk. This requires that you create an AfterFileOpenHook function. This is necessary because the package preference data in the just opened experiment is likely to be older than the data in the package preference file.

Creating Your Own Help File

If you are an advanced user, you can create an Igor help file that extends the Igor help system. This is something you might want to do if you write a set of Igor procedures or extensions for use by your colleagues. If your procedures or extensions are generally useful, you might want to make them available to all Igor users. In either case, you can provide documentation in the form of an Igor help file.

Here are the steps for creating an Igor help file.

1. Create a formatted-text notebook.

   A good way to do this is to open the Igor Help File Template provided by WaveMetrics in the More Help Files folder. Alternatively, you can start by duplicating another WaveMetrics-supplied help file and then open it as a notebook using File→Open File→Notebook. Either way, you are starting with a notebook that contains the rulers used to format an Igor help file.
2. Choose Save Notebook As from the File menu to create a new file. Use a “.ihf” extension so that Igor will recognize it as a help file.

3. Enter your help text in the new file.
4. Save and kill the notebook.
5. Open the file as a help file using File→Open File→Help File.

When you open the file as a help file, it needs to be compiled. When Igor compiles a help file, it scans through it to find out where the topics start and end and makes a note of subtopics. When the compilation is finished, it saves the help file which now includes the help compiler information.

Once Igor has successfully compiled the help file, it acts like any other Igor help file. That is, when opened it appears in the Help Windows submenu, its topics will appear in the Help Browser, and you can click links to jump around.

Here are the steps for modifying a help file.

1. If the help file is open, kill it by pressing Option (*Macintosh*) or Alt (*Windows*) and clicking the close button.
2. Open it as a notebook, using File→Open File→Notebook.

   Alternatively, you can press Shift while choosing the file from File→Recent Files. Then, in the resulting dialog, specify that you want to open the file as a formatted notebook.
3. Modify it using normal editing techniques.
4. Choose Save Notebook from the File menu.
5. Click the close button and kill the notebook.
6. Reopen it as a help file using File→Open File→Help File.

   Alternatively, you can press Shift while choosing the file from File→Recent Files. Then, in the resulting dialog, specify that you want to open the file as a help file.

Syntax of a Help File

Igor needs to be able to identify topics, subtopics, related topics declarations, and links in Igor help files. To do this it looks for certain rulers, text patterns and text formats described in **Creating Links** on page IV-257. You can get most of the required text formats by using the appropriate ruler from the Igor Help File Template file.

Igor considers a paragraph to be a help topic declaration if it starts with a bullet character followed by a tab and if the paragraph's ruler is named Topic. By convention, the Topic ruler's font is Geneva on Macintosh or Arial on Windows, its text size is 12 and its text style is bold-underlined. The bullet and tab characters should be plain, not bold or underlined.

The easiest way to create a new topic with the right formatting is to copy an existing topic and then modify it.

Once Igor finds a topic declaration, it scans the body of the topic. The body is all of the text until the next topic declaration, a related-topics declaration, or the end of the file. While scanning, it notes any subtopics.

Igor considers a paragraph to be a subtopic declaration if the name of the ruler governing the paragraph starts with "Subtopic". Thus if the ruler is named Subtopic or Subtopic+ or Subtopic2, the paragraph is a subtopic declaration. By convention, the Subtopic ruler's font is Geneva on Macintosh or Arial on Windows, its text size is 10 and its text style is bold and underlined. Text following the subtopic name that is not bold and underlined is not part of the subtopic name.

The easiest way to create a new subtopic with the right formatting is to copy an existing subtopic and then modify it.

Igor considers a paragraph to be a related-topics declaration if the ruler governing the paragraph is named RelatedTopics and if the paragraph starts with the text pattern "Related Topics:". When Igor sees this pattern it knows that this is the end of the current topic. The related-topics declaration is optional. Prior to Igor Pro 4, Igor displayed a list of related topics in the Igor Help Browser. Igor Pro no longer displays this list. The user can still click the links in the related topics paragraph to jump to the referenced topics.

Igor knows that it has hit the end of the current topic when it finds the related-topics declaration or when it finds a new topic declaration. In either case, it proceeds to compile the next topic. It continues compiling until it hits the end of the file.

When compiling the help file, Igor may encounter syntax that it can't understand. For example, if you have a related-topics declaration paragraph, Igor will expect the next paragraph to be a topic declaration. If it is not, Igor will stop the compilation and display an error dialog. You need to open the file as a notebook, fix the error, save and kill it and then reopen it as a help file.

Another error that is easy to make is to fail to use the plain text format for syntactic elements like bullet-tab, "Related Topics:" or the comma and space between related topics. If you run into a non-obvious compile error in a topic, subtopic or related topics declaration, recreate the declaration by copying from a working help file.

The help files supplied by WaveMetrics contain a large number of rulers to define various types of paragraphs such as topic paragraphs, subtopic paragraphs, related topic paragraphs, topic body paragraphs and so on. The Igor Help File Template contains many but not all of these rulers. If you find that you need to use a ruler that exists in a WaveMetrics help file but not in your help file then copy a paragraph governed by that ruler from the WaveMetrics help file and paste it into your file. This transfers the ruler to your file.

Creating Links

A link is text in an Igor help file that, when clicked, takes the user to some other place in the help. Igor considers any pure blue, underlined text to be a link. Pure blue means that the RGB value is (0, 0, 65535). By convention links use the Geneva font on Macintosh and the Arial font on Windows.

To create a link, select the text in the notebook that you are preparing to be a help file. Then choose Make Help Link from the Notebook menu. This sets the text format for the selected text to pure blue and underlined.

The link text refers to another place in the help using one of these forms:

- The name of a help topic (e.g., Command Window)
- The name of a help subtopic (e.g., History Area)
- A combined topic and subtopic (e.g., Command Window[History Area])

Use the combined form if there is a chance that the help topic or subtopic name by itself may be ambiguous. For example, to refer to the Preferences operation, use Operations[Preferences] rather than Preferences by itself.

When the user double-clicks a link, Igor performs the following search:

1. If the link is a topic name, Igor goes to that topic.
2. If the link is in topic[subtopic] form, Igor goes to that subtopic.
3. If steps 1 and 2 fail, Igor searches for a subtopic with the same name as the link. First, it searches for a subtopic in the current topic. If that fails, it searches for a subtopic in the current help file. If that fails, it searches for a subtopic in all help files.
4. If step 3 fails, Igor searches all help files in the Igor Pro folder. If it finds the topic in a closed help file, it opens and displays it.
5. If step 4 fails, Igor searches all help files in the Igor Pro User Files folder. If it finds the topic in a closed help file, it opens and displays it.
6. If all of the above fail, Igor displays a dialog saying that the required help file is not available.

You can create a link in a help file that will open a Web page or FTP site in the user's Web or FTP browser. You do this by entering the Web or FTP URL in the help file while you are editing it as a notebook. The URL must appear in this format:

```
<http://www.wavemetrics.com>
<ftp://ftp.wavemetrics.com>
```

The URL must include the angle brackets and the "http://", "https://" or "ftp://" protocol specifier. Support for https was added in Igor Pro 7.02.

After entering the URL, select the entire URL, including the angle brackets, and choose Make Help Link from the notebook menu. Once the file is compiled and opened as a help file, clicking the link will open the user's Web or FTP browser and display the specified URL.

For any other kind of URL, such as sftp or mailto, use a notebook action that calls BrowseURL instead of a help link.

It is currently not possible make ordinary text into a Web or FTP link. The text must be an actual URL in the format shown above or you can insert a notebook action which brings up a web page using the **BrowseURL** operation on page V-54. See **Notebook Action Special Characters** on page III-14 for details.

Checking Links

You can tell Igor to check your help links as follows:

1. Open your Igor help file and compile it as a help file if necessary.
2. Activate your help window.
3. Right-click in the body of the help file and choose Check Help Links. Igor will check your links from where you clicked to the end of the file and note any problems by writing diagnostics to the history area of the command window.
4. When Igor finishes checking, if it found bad links, kill the help file and open it as a notebook.
5. Use the diagnostics that Igor wrote in the history to find and fix any link errors.
6. Save the notebook and kill it.
7. Open the notebook as a help file. Igor will compile it.
8. Repeat the check by going back to Step 1 until you have no bad links.

During this process, Igor searches for linked topics and subtopics in open and closed help files and opens any closed help file to which a link refers. If a link is not satisfied by an already open help file, Igor searches closed help files in:

- The Igor Pro Folder and subfolders
- The Igor Pro User Files folder and subfolders
- Files and folders referenced by aliases or shortcuts in one of those folders

You can abort the check by pressing the **User Abort Key Combinations**.

The diagnostic that Igor writes to the history in case of a bad link is in the form:

```
Notebook $nb selection={(33,292), (33,334)} …
```

This is set up so that you can execute it to find the bad link. At this point, you have opened the help file as a notebook. Assuming that it is named Notebook0, execute

```
String/G nb = "Notebook0"
```

Now, you can execute the diagnostic commands to find the bad link and activate the notebook. Fix the bad link and then proceed to the next diagnostic. It is best to do this in reverse order, starting with the last diagnostic and cutting it from the history after fixing the problem.

If you press the Shift key while right-clicking a help window, you can choose Check Help Links in All Open Help Files. Then Igor checks all help links all help files open at that time. While checking a help file, Igor may open a previously unopened help file. Such newly opened help files are not checked. Only those help files open when you chose Check Help Links in All Open Help Files are checked. However, if you repeat the process, help files opened during the previous iteration are checked.

When fixing a bad link, check the following:

- A link is the name of a topic or subtopic in a currently open help file. Check spelling.

- There are no extraneous blue/underlined characters, such as tabs or spaces, before or after the link. (You can not identify the text format of spaces and tabs by looking at them. Check them by selecting them and then using the Set Text Format dialog.)
- There are no duplicate topics. If you specify a link in topic[subtopic] form and there are two topics with the same topic name, Igor may not find the subtopic.

Help for User-Defined Functions

You can provide help for user-defined functions in your package by including a topic similar to the built-in Functions topic in your help file. In Igor Pro 9.00 or later, the user can go to the help for your user-defined function by selecting it in a procedure or notebook window, right-clicking, and choosing Help For <topic name>.

Here is how add help for your user-defined functions:

1. Display the built-in Functions topic.
2. Copy that topic paragraph and the first subtopic to the clipboard.
3. Paste into your help file.
4. Change the topic name to a distinctive name such as My Package Functions.
6. Edit the subtopic to provide help for one of your user-defined function.
7. Add additional subtopics by copying and pasting the original and editing as needed.

Make sure that your package name and user-defined function names are distinctive to avoid collisions with other packages.

The Insert Template For item in the contextual menu gets information from the procedure file and does not depend on help.

Creating Formatted Text

The printf, sprintf, and fprintf operations print formatted text to Igor's history area, to a string variable or to a file respectively. The wfprintf operation prints formatted text based on data in waves to a file.

All of these operations are based on the C printf function which prints the contents of a variable number of string and numeric variables based on the contents of a format string. The format string can contain literal text and conversion specifications. Conversion specifications define how a variable is to be printed.

Here is a simple example:

```
printf "The minimum is %g and the maximum is %g\r", V_min, V_max
```

In this example, the format string is `"The minimum is %g and the maximum is %g\r"` which contains some literal text along with two conversion specifications — both of which are "`%g`"— and an escape code ("`\r`") indicating "carriage-return". If we assume that the Igor variable V_min = .123 and V_max = .567, this would print the following to Igor's history area:

```
The minimum is .123 and the maximum is .567
```

We could print this output to an Igor string variable or to a file instead of to the history using the **sprintf** (see page V-902) or **fprintf** (see page V-260) operations.

Printf Operation

The syntax of the printf operation is:

```
printf format [, parameter [, parameter ]. . .]
```

where *format* is the format string containing literal text or format specifications. The number and type of parameters depends on the number and type of format specifications in the format string. The parameters, if any, can be literal numbers, numeric variables, numeric expressions, literal strings, string variables or string expressions.

The conversion specifications are very flexible and make printf a powerful tool. They can also be quite involved. The simplest specifications are:

Specification	What It Does
%g	Converts a number to text using integer, floating point or exponential notation depending on the number's magnitude.
%e	Converts a number to text using exponential notation.
%f	Converts a number to text using floating point notation.
%d	Converts a number to text using integer notation.
%s	Converts a string to text.

Here are some examples:

```
printf "%g, %g, %g\r", PI, 6.022e23, 1.602e-19
```

prints:

```
3.14159, 6.022e+23, 1.602e-19
```

```
printf "%e, %e, %e\r", PI, 6.022e23, 1.602e-19
```

prints:

```
3.141593e+00, 6.022000e+23, 1.602000e-19
```

```
printf "%f, %f, %f\r", PI, 6.022e23, 1.602e-19
```

prints:

```
3.141593, 602200000000000027200000.000000, 0.000000
```

```
printf "%d, %d, %d\r", PI, 6.022e23, 1.602e-19
```

prints:

```
3, 9223372036854775807, 0
```

```
printf "%s, %s\r", "Hello, world", "The time is " + Time()
```

prints:

```
Hello, world, The time is 11:43:40 AM
```

Note that the output for 6.022e23 when printed using the %d conversion specification is wrong. This is because 6.022e23 is too big a number to represent as an 64-bit integer.

If you want better control of the output format, you need to know more about conversion specifications. It gets quite involved. See the **printf** operation on page V-770.

sprintf Operation

The sprintf operation is very similar to printf except that it prints to a string variable instead of to Igor's history. The syntax of the sprintf operation is:

```
sprintf stringVariable, format [, parameter [, parameter ]. . .]
```

where *stringVariable* is the name of the string variable to print to and the remaining parameters are as for printf. sprintf is useful for generating text to use as prompts in macros, in axis labels and in annotations.

fprintf Operation

The fprintf operation is very similar to printf except that it prints to a file instead of to Igor's history. The syntax of the fprintf operation is:

```
fprintf variable, format [, parameter [, parameter ]. . .]
```

where *variable* is the name of a numeric variable containing the file reference number for the file to print to and the remaining parameters are as for printf. You get the file reference number using the Open operation, described under **Open and Close Operations** on page IV-196.

For debugging purposes, if you specify 1 for the file reference number, Igor prints to the history area instead of to a file, as if you used printf instead of fprintf.

wfprintf Operation

The wfprintf operation is similar to printf except that it prints the contents of one to 100 waves to a file. The syntax of the wfprintf operation is:

```
wfprintf variable, format [/R=(start,end)] wavelist
```

variable is the name of a numeric variable containing the file reference number for the file to print to.

Unlike printf, which rounds, wfprintf converts floating point values to integers by truncating, if you use an integer conversion specification such as “%d”.

Example Using fprintf and wfprintf

Here is an example of a command sequence that creates some waves and put values into them and then writes them to an output file with column headers.

```
Make/N=25 wave1, wave2, wave3
wave1 = 100+x; wave2 = 200+x; wave3 = 300+x
Variable f1
Open f1
fprintf f1, "wave1, wave2, wave3\r"
wfprintf f1, "%g, %g, %g\r" wave1, wave2, wave3
Close f1
```

This generates a comma delimited file. To generate a tab delimited file, use:

```
fprintf f1, "wave1\twave2\twave3\r"
wfprintf f1, "%g\t%g\t%g\r" wave1, wave2, wave3
```

Since tab-delimited is the default format for wfprintf, this last command is equivalent to:

```
wfprintf f1, "" wave1, wave2, wave3
```

Client/Server Overview

An application can interact with other software as a server or as a client.

A server accepts commands and data from a client and returns results to the client.

A client sends commands and data to a server program and receives results from the server.

For the Macintosh, see **Apple Events** on page IV-261 for server information and **AppleScript** on page IV-263 for client capabilities.

For Windows, see **ActiveX Automation** on page IV-265. Igor can play the role of an Automation server but not an Automation client. However it is possible to generate script files that allow Igor to indirectly play the role of client.

Apple Events

This topic is of interest to *Macintosh* programmers. For Windows, see **ActiveX Automation** on page IV-265.

This topic contains information for Igor users who wish to control Igor from other programs (e.g., AppleScript). It also contains information useful to people who are writing their own programs and wish to use Igor Pro as a computing or graphing engine.

There is also a mechanism that allows Igor to act like a controller and initiate Apple event communication with other programs. See **AppleScript** on page IV-263.

Apple Event Capabilities

Igor Pro supports the following Apple events:

Event	Suite	Action
Open Application	Required	Basically a nop; don't use.
Open Document	Required	Loads an experiment.
Print Document	Required	NA; don't use.
Quit Application	Required	Quits.
Close	Core	Acts on experiment, window or PPC port.
Save	Core	Acts on experiment only.
Open	Core	NA; don't use.
Do Script	Misc	Executes commands; can return ASCII results.
Eval Expression	Misc	Same as Do Script; obsolete but included for compatibility.

Apple Events — Basic Scenario

You use the Open Document event to cause Igor to load an experiment with whatever goodies you find useful. You then use the Do Script or Eval Expression events to send commands to Igor for execution and to retrieve results. To get data into Igor you write files and then send commands to Igor to load the data. To get data waves from Igor you do the reverse. To get graphics, you send commands to Igor that cause it to write a graphics file that you can read. You may then close the experiment and start over with a new one. You will not likely use the Save event.

Apple Events — Obtaining Results from Igor

To return information from Igor, you will need to embed special commands in the script you send to Igor for execution. When Igor encounters these commands, it appends results to a packet that is returned to your application after script execution ends. The special commands are variations on the standard Igor commands FBinWrite and fprintf. Both of these commands take a file reference number as a parameter. If the magic value zero is used rather than a real file reference number, then the data that would normally be written to a file is appended to the result packet.

As far as Igor is concerned, there is no difference between the Do Script and Eval Expression events. However, old applications may expect results from Eval Expression and not from Do Script.

To use waves and graphics with Apple events, you will need to write or read the data via standard Igor files. For example, you might include

```
SavePICT /E=-8 /P=MyPath as "Graphics.pdf"
```

in a script that you send to Igor for execution. You could then read the file in your application.

Apple Event Details

This information is intended for programmers familiar with Apple events terminology.

Some of the following events can act on experiments or windows.

To specify an experiment, use object class cDocument (`'docu'`) and specify either formAbsolutePosition with index=1 or formName with name=*name of experiment*.

To specify a window, use object class cWindow ('cwin') and either formAbsolutePosition or formName with name=*title of window*.

Event	Class	Code	Action
Open Application	'aevt'	'oapp'	Basically a nop; don’t use.
Open Document	'aevt'	'odoc'	Loads an experiment. Direct object is assumed to be coercible to a File System Spec record.
Print Document	'aevt'	'pdoc'	NA; don’t use.
Quit Application	'aevt'	'quit'	Quits the program. If the experiment was modified, then Igor attempts to interact with the user to get save/no save directions. If interaction is not allowed, then an error is returned and nothing is done. To prevent errors, send the close event with appropriate save options prior to sending quit.
Close	'core'	'clos'	Acts on an experiment or window. For a window, the save/no save/ask optional parameter (keyAESaveOptions) is allowed and refers to making/replacing a recreation macro. For a document (experiment), keyAESaveOptions is allowed and an additional optional parameter keyAEDestination may be used to specify where to save (must be coercible to a FSS). If this is not given and the experiment is untitled and if an attempt to interact with the user fails then the experiment is not saved and an error (such as errAENoUserInteraction) is returned. Note that if the optional destination is given then the save options are ignored (why give a destination and then say no save?).
Save	'core'	'save'	Acts on experiment only. Takes same optional destination parameters as Close. A save with a destination is the same as a Save as.
Do Script	'misc'	'dosc'	Same as Eval Expression.
Eval Expression	'aevt'	'eval'	Executes commands. Acts just as if commands had been typed into the command line except the individual command lines are preceded by a percent symbol rather than the usual bullet symbol. Also, errors are returned in the error reply parameter of the event rather than putting up a dialog. **Note**: You can suppress history logging by executing the command, “Silent 2”, and you can turn it back on by executing “Silent 3”. Direct parameter must be text and not a file. Text can be of any length. You can return a string containing results by using the fprintf command with a file reference number of zero.

AppleScript

This topic is of interest to *Macintosh* programmers. For Windows, see **ActiveX Automation** on page IV-265.

Igor supports the creation and execution of simple AppleScripts in order to send commands to other programs.

To execute an AppleScript program, you first compose it in a string and then pass it to the ExecuteScriptText operation, which in turn passes the text to Apple's scripting module for compilation and execution. The result, which might be an error message, is placed in a string variable named S_value. Igor does not save the compiled script so every time you call ExecuteScriptText your script will have to be recompiled. See the **ExecuteScriptText** operation on page V-205 for additional details.

The documentation for the **ExecuteScriptText** operation (page V-205) includes an example that shows how to execute a Unix command.

Because there is no easy way to edit a script or to see where errors occur, you should first test your script using Apple's Script Editor application.

You can use "Silent 2" to prevent commands your script sends to Igor from being placed in the history area.

You can send commands to Igor without using the tell keyword.

You should check your quoting carefully. Your text must be quoted both for Igor and for Apple's scripting system. For example,

```
ExecuteScriptText "Do Script \"Print \\\"hello\\\"\""
```

You should compose scripts in string variables one line at a time to improve readability.

If an error occurs that you can't figure out, print the string, copy from the history and paste into a Script Editor for debugging.

If the script returns a text return value, it may be quoted within the S_value string. See the discussion of quoting in the **ExecuteScriptText** documentation for details.

Don't forget to include the carriage return escape code, `\r`, at the end of each line of a multiline script.

The first time you call this routine, it may take an extra long time while the Mac OS loads the scripting modules.

Executing Unix Commands on Mac OS X

On Mac OS X, you can use AppleScript to send a command to the Unix shell. Here is a function that illustrates this:

```
Function/S ExecuteUnixShellCommand(uCommand, printCommandInHistory,
                                        printResultInHistory [, asAdmin])
    String uCommand                     // Unix command to execute
    Variable printCommandInHistory
    Variable printResultInHistory
    Variable asAdmin                    // Optional - defaults to 0

    if (ParamIsDefault(asAdmin))
        asAdmin = 0
    endif

    if (printCommandInHistory)
        Printf "Unix command: %s\r", uCommand
    endif

    String cmd
    sprintf cmd, "do shell script \"%s\"", uCommand
    if (asAdmin)
        cmd += " with administrator privileges"
    endif
    ExecuteScriptText/UNQ/Z cmd // /UNQ removes quotes surrounding reply

    if (printResultInHistory)
        Print S_value
    endif

    return S_value
End
```

You can test the function with this command:

```
ExecuteUnixShellCommand("ls", 1, 1)
```

Life is a bit more complicated if the command that you want to execute contains spaces or other nonstandard Unix command characters. For example, imagine that you want to execute this:

```
ls /System/Library/Image Capture
```

These commands will not work because of the space in the command's file path:

```
String unixCmd = "ls /System/Library/Image Capture"
ExecuteUnixShellCommand(unixCmd, 1, 1)
```

You need to quote the path part of the command so that the UNIX parses:

```
/System/Library/Image Capture
```

as one argument, not two (/System/Library/Image and ./Capture).

There are two ways to do this:

1. Use single-quote characters around the parts of the command where needed:

   ```
   String unixCmd = "ls '/System/Library/Image Capture'"
   ExecuteUnixShellCommand(unixCmd, 1, 1)
   ```

2. Use double-quote characters with the backslashes needed to make it through Igor's parser and AppleScript's parser:

   ```
   String unixCmd = "ls \\\"/System/Library/Image Capture\\\""
   ExecuteUnixShellCommand(unixCmd, 1, 1)
   ```

   Igor's parser converts \\ to \ and \" to ", so AppleScript sees this:

   ```
   "ls \"/System/Library/Image Capture\""
   ```

   AppleScript's parser converts \" to " so Unix sees this:

   ```
   ls "/System/Library/Image Capture"
   ```

ActiveX Automation

ActiveX Automation, often called just Automation, is Microsoft's technology for allowing one program to control another. The program that does the controlling is called the Automation client. The program that is controlled is called the Automation Server. The client initiates things by making calls to the server which carries out the requested actions and returns results.

Automation client programs are most often written in Visual Basic or C#. They can also be written in C++ and other programming languages, and in various scripting languages such as VBScript, JavaScript, Perl, Python and so on.

Igor can play the role of Automation Server. If you want to write an client program to drive Igor, see "Automation Server Overview" in the "Automation Server" help file in "\Igor Pro Folder\Miscellaneous\Windows Automation".

Igor Pro does not directly support playing the role of Automation client. However, it is possible to write an Igor program which generates a script file which can act like an Automation client. For an example, choose File→Example Experiments→Programming→CallMicrosoftWord.

Calling Igor from Scripts

You can call Igor from shell scripts, batch files, Apple Script, and the Macintosh terminal window using an operation-like syntax. You can also use this feature to register an Igor license.

The syntax for calling Igor is:

```
<IGOR> [/I /Q /X /N /Automation]  [pathToFileOrCommands] [pathToFile] ...

<IGOR> [/I /N /Automation]  [pathToFileOrCommands] [pathToFile] ...

<IGOR> [/I /Q /X /Automation] "commands"

<IGOR> /SN=num /KEY="key" /NAME="name" [/ORG="org" /QUIT]
```

where `<IGOR>` is the full path to the Igor executable file.

On Windows, the Igor executable file resides in a folder within the Igor Pro folder. The full path will be something like:

```
"C:\Program Files\WaveMetrics\IgorBinaries_x64\Igor Pro Folder\Igor64.exe"
```

On Macintosh, the Igor application is an "application bundle" and the actual executable file is inside the bundle. The full path will be something like:

```
'/Applications/Igor Pro Folder/Igor64.app/Contents/MacOS/Igor64'
```

In the following discussions, `<IGOR>` means "the full path to the Igor executable file".

Parameters

All parameters are optional. If you omit all parameters, including just the full path to the Igor executable, a new instance of Igor is launched.

The usual parameter is a file for Igor to open. It is recommended that both the path and the path to the file parameter be enclosed in quotes.

You can open multiple files by using a space between one quoted file path and the next.

With the /X flag, only one parameter is allowed and it is interpreted as an Igor command.

Flags

When you specify a flag, you can use a - instead of /. For example, you can write /Q or -Q.

/Automation — This flag is supported on Windows only. The Windows OS uses it when launching Igor Pro as an ActiveX Automation server. It is not intended for use in batch files.

/I — Launches a new instance of Igor if one would otherwise not be launched. See ***Details*** for a discussion of instances.

/KEY="*key*" — Specifies the license activation key when registering a license. For example:

```
/KEY="ABCD-EFGH-IJKL-MNOP-QRST-UVWX-Y"
```

Do not omit the quotes, or it will fail.

/N — Forces the current experiment to be closed without saving if any of the file parameters are an experiment file.

To save a currently open experiment, use:

```
<IGOR> /X "SaveExperiment"
```

/NAME="*name*" — Specifies the name of the licensed user when registering a license. A name is required when registering.

/ORG="*org*" — Specifies the optional name of the licensed organization when registering a license. /ORG is optional and defaults to "".

/Q — Prevents the command from being displayed in Igor's command line as it is executing.

/QUIT — Quits Igor Pro after entering license information when used with /SN, /KEY, and /NAME. Otherwise /QUIT is ignored.

To quit Igor Pro, use:

```
<IGOR> /X "Quit/N"
```

/SN=*num* — Specifies the license serial number when registering a license.

/START=*x* Controls logging of diagnostic information for use in troubleshooting slow startup or crashes during startup. Allowed values for *x* are:

0: Regular startup.

1: Write diagnostic information to "Igor Debug Log.txt" in the Diagnostics folder of the Igor Pro preferences folder.

2: Write diagnostic information to an Igorlogging window.

/START was added in Igor Pro 9.00.

/UNATTENDED Suppresses certain interactions that are inconvenient for unattended operations. An example is the About Autosave dialog that appears when Igor is launched until the user clicks “Do not show this message again”.

/ /UNATTENDED was added in Igor Pro 9.00.

X Executes the commands in the parameter. Only one parameter is allowed with /X. Use semicolons to separate Igor commands within the parameter.

Details

If an existing instance of Igor is running, the command is sent to the existing instance if you omit the /I flag and you include /X, /SN, or a path to a file. Otherwise, a new instance of Igor is launched.

Registering a License

You can register an Igor license using the /SN, /KEY, and /NAME flags. All of these flags must be present to successfully register a license. The optional /ORG parameter defaults to "".

These batch file commands register Igor Pro with the given serial number and license activation key:

```
<IGOR> /SN=1234567 /KEY="ABCD-EFGH-IJKL-MNOP-QRST-UVWX-Y" /NAME="Jack" /ORG="Acme Scientific" /QUIT
```

Network Communication

The following sections contain material related to the network communication and Internet-related capabilities of Igor Pro:

URLs on page IV-267

Safe Handling of Passwords on page IV-270

Network Timeouts and Aborts on page IV-270

Network Connections From Multiple Threads on page IV-271

File Transfer Protocol (FTP) on page IV-272

Hypertext Transfer Protocol (HTTP) on page IV-276

URLs

URLs, or Uniform Resource Locators, are compact strings that represent a resource available via the Internet. The description of the URL standard is described in RFC1738 (http://www.rfc-editor.org/rfc/rfc1738.txt) and updated in RFC3986 (http://www.rfc-editor.org/rfc/rfc3986.txt).

Each URL is composed of several different parts, most of which are optional:

```
<scheme>://<username>:<password>@<host>:<port>/<path>?<query>
```

Some examples of valid URLs are:

```
http://www.example.com
http://www.example.com/afolder?key1=45&key2=66
http://myusername:Passw0rD@www.example.com:8010/index.html
ftp://ftp.wavemetrics.com
```

```
file:///C:\\Data\\Trial1\\control.ibw (on Windows only)
file:///Users/bob/Data/Trial1/control.ibw (on Macintosh only)
```

For most operations and functions that take a *urlStr* parameter, only the scheme and host parts of the URL are required. See the **Supported Network Schemes** section for information on which schemes are supported by which operations and functions, and which port is used by default if it is not provided as part of the URL.

Usernames and Passwords

You can provide a username and password as part of the URL. However authentication credentials may not be supported by all schemes (such as file://). Some operations allow you to provide a username and password by using a flag, such as the /U and /W flags with **FTPDownload** or the /AUTH flag with **URLRequest**.

If a URL contains a username and password in the URL and the authentication flags are also used, the values specified in the flags override values provided in the URL.

If you do not provide a username and password as part of the URL, and you do not use the authentication flags, then no authentication is attempted. An exception to this rule is that the FTP operations will login to the FTP server using "anonymous" as the username and a generic email address as the password.

If either the username or password contains special or reserved characters, those characters must be percent-encoded.

Supported Network Schemes

Different operations and functions support different schemes:

Operation	Supported Schemes	Default Port
FetchURL and **URLRequest**	http	80
	https	443
	ftp	21
	file	Not applicable
FTP operations*	ftp	21

* Includes **FTPUpload**, **FTPDownload**, **FTPDelete**, and **FTPCreateDirectory**.

Percent Encoding

Percent encoding is a way to encode characters in URLs that would otherwise have a special meaning or could be misinterpreted by servers. For example, a space character in a URL is encoded as "%20" using a percent character followed by the hex code for a space in the ASCII character set.

Most URLs contain only the letters A-Z and a-z, the digits 0-9, and a few other characters such as the underscore (_), hyphen (-), period (.), and tilde (~).

A URL may also contain "reserved characters" that may have special meaning depending on the way that they are used. Every URL contains the reserved characters ":" and "/" and may also contain one or more of the following reserved characters: !*'();@&=+$,?#[].

All operations and functions provided by Igor Pro that accept a URL string parameter expect that the URL has already been percent-encoded as necessary.

In most cases you don't need to worry about percent encoding because most URLs don't use reserved characters except for their special meaning. If you need to use a reserved character in a way that differs from the character's special meaning, you must percent-encode the character. You can use the **URLEncode** function for this purpose.

It is important that you not pass your entire URL to URLEncode to be encoded because that URL will not be understood by a server. URLEncode percent-encodes all reserved characters in the string you pass to it, because it cannot distinguish between reserved characters used for their special meaning and reserved characters used outside of their special meaning. Instead, you must pass each piece of the URL through URLEncode so that the final URL uses the correct syntax.

As an example, we'll use URLEncode to properly encode a URL that contains the following parts:

Part Name	Example
Scheme	http
Username	A. MacGyver
Password	yj@!2M
Host	www.example.com
Path	/tape/duct
Query	discount=10%&color=red

Without any percent-encoding, the URL is:

```
http://A. MacGyver:yj@!2M@www.example.com/tape/duct?discount=10%&color=red
```

If this URL were passed to **FetchURL**, the result would be an error because the URL contains several reserved characters that are not intended to be used in their standard way. For example, the "@" character indicates the separation between the username:password information and the start of the host name, but in this case the password itself also contains the "@" character. In addition, the "%" character is typically used to indicate that the next two characters represent a percent-encoded character, but in this example it is also part of the query. Finally, the username contains a space character. The space character is not technically a reserved character, but should be percent-encoded to ensure that it is handled correctly.

The following table shows the values of the parts of the URL that need to be percent-encoded by passing them through the URLEncode function:

Part Name	Encoded Value
Username	A%2E%20MacGyver
Password	yj%40%212M
Host	www.example.com
Path	/tape/duct
Query	discount=10%25&color=red

The properly percent-encoded URL is:

```
http://A%2E%20MacGyver:yj%40%212M@www.example.com/tape/duct?discount=10%25&color=red
```

For keyword-value pairs that make up the query part, each keyword and value must be percent-encoded separately because the "=" character that separates the key from the value and the "&" character that separates the pairs in the list must not be percent-encoded.

For more information on percent-encoding and reserved characters, see http://en.wikipedia.org/wiki/Percent-encoding.

Safe Handling of Passwords

Some operations and functions support the use of a username and password when making a network connection. If you use sensitive passwords you must take certain precautions to prevent them from being accidentally revealed.

1. Always use the /V=0 flag when using a username or password with the /U (username) and /W (password) flags or the /AUTH flag. Otherwise, the debugging information that is printed to the history area will contain those values and anyone who sees the experiment could see them.
2. Do not hard code username or password values into procedures, since anyone with access to the procedure file could read them.
3. Do not store username or password values in global variables. Since global variables are saved with an experiment, if someone else had access to your experiment they could see this information.

Here is an example of how a username and sensitive password can be used in a secure manner:

```
Function SafeLogin()
   String username = ""
   String password = ""
   Prompt username, "Username"
   Prompt password, "Password"
   DoPrompt "Enter username and password", username, password
   if (V_flag == 1)
      // User hit cancel button, so do nothing.
      return 0
   endif

   // Percent-encode in case username and password contain reserved characters.
   String encodedUser = URLEncode(username)
   String encodedPass = URLEncode(password)

   String theURL
   sprintf theURL, "http://%s:%s@www.example.com", encodedUser, encodedPass
   String response = FetchURL(theURL)
   // NOTE: For FTP operations and URLRequest, make sure to use /V=0 so that
   // the username and password are not printed to the history.

   return 0
End
```

Note that the user is prompted to provide the username and password when the function is called and that only local string variables are used to store the username and password. The values in those string variables are not stored once the function is done executing.

Note also that the password is not hidden during entry in the dialog. Igor currently does not provide a way to do this.

Network Timeouts and Aborts

Some network calls may return an error code to Igor if they timeout. Depending on the specific operation or function, there can be a number of causes for a timeout.

If a network connection cannot be made after a period of time it will timeout. The amount of time allowed for a connection to be established is dependent on several factors.

You can always abort a network operation or function by pressing the **User Abort Key Combinations**.

Network Connections From Multiple Threads

All network-related operations and functions are thread-safe, which means that they can be called from multiple preemptive threads at the same time. This capability can be useful when:

- You want to retrieve information from several different URLs as quickly as possible.
- You want to do a long download or other operation in the background to avoid tieing Igor up.

The following example illustrates the first of these cases. It uses **FetchURL** to retrieve a list of the most frequently downloaded books from the Project Gutenberg web site. It then uses FetchURL to download the entire text of the top four books and prints the number of bytes in each.

```
ThreadSafe Function GetThePage(url)
   String url

   String response = FetchURL(url)
   return strlen(response)
End

Function ListGutenbergTopBooks()
   String topBooksURL = "http://www.gutenberg.org/browse/scores/top"
   String baseURL = "http://www.gutenberg.org/files/"

   // Get the contents of the page.
   String response = FetchURL(topBooksURL)
   Variable error = GetRTError(1)
   if (error || numtype(strlen(response)) != 0)
      Print "Error getting the list of most popular books."
      return 0
   endif

   String topBooksHTML = response

   // Remove all line endings.
   topBooksHTML = ReplaceString("\n", topBooksHTML, "")
   topBooksHTML = ReplaceString("\r", topBooksHTML, "")

   // Parse the page to get the section of the page
   // with the list of the most popular books from yesterday.
   // This could break if the format of the web page changes.
   String regExp = "(?i)<h2 id=\"books-last1\">.*?<ol>(.*?)</ol>"
   String topYesterdayHTML = ""
   SplitString/E=regExp topBooksHTML, topYesterdayHTML
   if (V_flag != 1)
      Print "Error parsing the top 100 books section."
      return 0
   endif

   // Replace the line endings.
   topYesterdayHTML = ReplaceString("</li><li>", topYesterdayHTML, "\r")

   // Create a wave to store text info about the top four books.
   Variable numBooksToUse = 4
   Make/O/T/N=(numBooksToUse, 2) topBooksInfo

   Make/O/N=(numBooksToUse) byteCounts

   Variable n
   String bookNumStr
   Variable bookNum
   String titleAuthor
   String thisLine
   Variable pos
   String bookURL = ""
   regExp = "(?i)a href=\".*?(\d+)\">(.+?)</a>"
   for (n=0; n<numBooksToUse; n+=1)
      // For each book we're going to look at, get the
      // partial URL and the title/author text.
      thisLine = StringFromList(n, topYesterdayHTML, "\r")
      SplitString/E=regExp thisLine, bookNumStr, titleAuthor
      if (V_flag != 2)
         Print "Error parsing the URL and title/author information."
         return 0
      endif

      // Remove the (###) stuff at the end of titleAuthor if it's there.
      pos = strsearch(titleAuthor, "(", 0)
      if (pos > 0)
         titleAuthor = titleAuthor[0, pos - 1]
      endif

      bookNum = str2num(bookNumStr)

      // Store the information about the book in the text wave.
      sprintf bookURL, "%s%d/%d.txt", baseURL, bookNum, bookNum
      topBooksInfo[n][0] = bookURL
      topBooksInfo[n][1] = titleAuthor
   endfor

   // Download each book (using multiple threads if possible)
   // and count the number of bytes in each.
   MultiThread byteCounts = GetThePage(topBooksInfo[p][0])

   // Print the results.
   Print "The top four books by download from yesterday are:"
   for (n=0; n<numBooksToUse; n+=1)
      Printf "%s (%d bytes)\r", topBooksInfo[n][1], byteCounts[n]
   endfor
End
```

Here is an example of what the output was when this help file was written:

The top four books by download from yesterday are:

```
Ulysses by James Joyce (1573044 bytes)
Alice's Adventures in Wonderland by Lewis Carroll (167529 bytes)
Piper in the Woods by Philip K. Dick (62214 bytes)
Pride and Prejudice by Jane Austen (704160 bytes)
```

File Transfer Protocol (FTP)

The **FTPDownload**, **FTPUpload**, **FTPDelete**, and **FTPCreateDirectory** operations support simple transfers of files and directories over the Internet.

Since Igor’s SaveNotebook operation can generate HTML files from notebooks, it is possible to write an Igor procedure that downloads data, analyzes it, graphs it, and uploads an HTML file to a directory used by a Web server. You can then use the BrowseURL operation to verify that everything worked as expected. For a demo of some of these features, choose File→Example Experiments→Feature Demos→Web Page Demo.

FTP Limitations

All FTP operations run “synchronously”. This means that, if the operation executes in the main thread, Igor can not do anything else. However, it is possible to perform these operations using an Igor preemptive thread so that they execute in the background and you can continue to use Igor for other purposes. For more information, see **Network Connections From Multiple Threads** on page IV-271.

Igor does not currently provide any way for the user to browse the remote server from within Igor itself.

Igor does not provide any secure way to store passwords. Consequently, you should not use Igor for FTP in situations where tight security is required. See **Safe Handling of Passwords** on page IV-270 for an example of how to securely prompt the user for a password.

Igor does not provide any support for using proxy servers. Proxy servers are security devices that stand between the user and the Internet and permit some traffic while prohibiting other traffic. If your site uses a proxy server, FTP operations may fail. Your network administrator may be able to provide a solution.

Igor does not include operations for listing a server directory or changing its current directory.

Downloading a File

The following function transfers a file from an FTP server to the local hard disk:

```
Function DemoFTPDownload()
   String url = "ftp://ftp.wavemetrics.net/welcome.msg"
   String localFolder = SpecialDirPath("Desktop",0,0,0)
   String localPath = localFolder + "DemoFTPDownloadFile.txt"
   FTPDownload/U="anonymous"/W="password" url, localPath
End
```

The output directory must already exist on the local hard disk. The target file may or may not exist on the local hard disk. If it does not exist, the FTPDownload command creates it. If it does exist, FTPDownload asks if you want to overwrite it. To overwrite it without being asked, use the /O flag.

Warning: If you elect to overwrite it, all previous contents of the local target file are obliterated.

FTPDownload presents a dialog asking you to specify the local file name and location in the following cases:

1. You use the /I (interactive) flag.
2. The parent directory specified by the local path does not exist.
3. The specified local file exists and you do not use the /O (overwrite) flag.

Downloading a Directory

The following function transfers a directory from an FTP server to the local hard disk:

```
Function DemoFTPDownloadDirectory()
   String url = "ftp://ftp.wavemetrics.net/Utilities"
   String localFolder = SpecialDirPath("Desktop",0,0,0)
   String localPath = localFolder + "DemoFTPDownloadDirectory"
   FTPDownload/D/U="anonymous"/W="password" url, localPath
End
```

The /D flag specifies that you are transferring a directory.

The output directory may or may not already exist on the local hard disk. If it does not exist, the FTPDownload command creates it. If it does exist, FTPDownload asks if you want to overwrite it. To overwrite it without being asked, use the /O flag.

Warning: If you elect to overwrite it, all previous contents of the local directory are obliterated.

If the local path that you specify ends with a colon or backslash, FTPDownload presents a dialog asking you to specify the local directory because it is looking for the name of the directory to be created on the local hard disk.

FTPDownload presents a dialog asking you to specify the local directory in the following cases:

1. You use the /I (interactive) flag.
2. The parent directory specified by the local path does not exist.
3. The specified directory (DemoFTPDownloadFolder in the example above) exists and you have not used the /O (overwrite) flag.
4. FTPDownload gets an error when it tries to create the specified directory. This could happen, for example, if you don't have write privileges for the parent directory.

Uploading a File

The following function uploads a file to an FTP server:

```
Function DemoFTPUploadFile()
   String url = "ftp://ftp.wavemetrics.com/pub"
   String localFolder = SpecialDirPath("Desktop",0,0,0)
   String localPath = localFolder + "DemoFTPUploadFile.txt"
   FTPUpload/U="username"/W="password" url, localPath
End
```

To successfully execute this, you need a real user name and a real password.

Note: The /O flag has no effect on the FTPUpload operation when uploading a file. FTPUpload *always* overwrites an existing server file, whether /O is used or not.

Warning: If you overwrite a server file, all previous contents of the file are obliterated.

To overwrite an existing file on the server, you must have permission to delete files on that server. The server administrator determines what permission a particular user has.

FTPUpload presents a dialog asking you to specify the local file in the following cases:

1. You use the /I (interactive) flag.
2. The local parent directory or the local file does not exist.

Uploading a Directory

The following function uploads a directory to an FTP server:

```
Function DemoFTPUploadDirectory()
   String url = "ftp://ftp.wavemetrics.com/pub"
   String localFolder = SpecialDirPath("Desktop",0,0,0)
   String localPath = localFolder + "DemoFTPUploadDirectory"
   FTPUpload/D/U="username"/W="password" url, localPath
End
```

To successfully execute this, you need a real user name and a real password. Also, the server would have to allow uploading directories.

Note: FTPUpload *always* overwrites an existing server directory, whether /O is used or not.

Warning: If you omit /O or specify /O or /O=1, all previous contents of the directory are obliterated.

If you specify /O=2, FTPUpload performs a merge of the directory contents. This means that files and directories in the source overwrite files and directories on the server that have the same name, but files and directories on the server whose names do not conflict with those in the source directory are not modified.

To overwrite an existing directory on the server, you must have permission to delete directories on that server. The server administrator determines what permission a particular user has.

If the local path that you specify ends with a colon or backslash, FTPUpload presents a dialog asking you to specify the local directory because it is looking for the name of the directory to be uploaded.

FTPUpload presents a dialog asking you to specify the local directory in the following cases:

1. You use the /I (interactive) flag.
2. The specified directory or any of its parents do not exist.

If you don’t have permission to remove and to create directories on the server, FTPUpload will fail and return an error.

Creating a Directory

The **FTPCreateDirectory** operation creates a new directory on an FTP server.

If the directory already exists on the server, the operation does nothing. This is not treated as an error, though the V_Flag output variable is set to -1 to indicate that the directory already existed.

If you don't have permission to create directories on the server, FTPCreateDirectory fails and returns an error.

Deleting a Directory

The **FTPDelete** operation with the /D flag deletes a directory on an FTP server.

If you don't have permission to delete directories on the server, or if the specified directory does not exist on the server, FTPDelete fails and returns an error.

FTP Transfer Types

The FTP protocol supports two types of transfers: image and ASCII. Image transfer is appropriate for binary files. ASCII transfer is appropriate for text files.

In an image transfer, also called a binary transfer, the data on the receiving end will be a replica of the data on the sending end. In an ASCII transfer, the receiving FTP agent changes line terminators to match the local convention. On Macintosh and Unix, the conventional line terminator is linefeed (LF, ASCII code 0x0A). On Windows, it is carriage-return plus linefeed (CR+LF, ASCII code 0x0D + ASCII code 0x0A).

If you transfer a text file using an image transfer, the file may not use the local conventional line terminator, but the data remains intact. Igor Pro can display text files that use any of the three conventional line terminators, but some other programs, especially older programs, may display the text incorrectly.

On the other hand, if you transfer a binary file, such as an Igor experiment file, using an ASCII transfer, the file will almost certainly be corrupted. The receiving FTP agent will convert any byte that happens to have the value 0x0D to 0x0A or vice versa. If the local convention calls for CRLF, then a single byte 0x0D will be changed to two bytes, 0x0D0A. In either case, the file will become unusable.

FTP Troubleshooting

FTP involves a lot of hardware and software on both ends and a network in between. This provides ample opportunity for errors.

Here are some tips if you experience errors using the FTP operations.

1. Use an FTP client or web browser to connect to the FTP site. This confirms that your network is operating, the FTP server is operating, and that you are using the correct URL.
2. Use an FTP client or web browser to verify that the user name and password that you are using is correct or that the server allows anonymous FTP access.

   Many web browser accept URLs of the form:

   ```
   ftp://username:password@ftp.example.com
   ```

   However the password is not transferred securely.

3. Use an FTP client or web browser to verify that the directory structure of the FTP server is what you think it is.
4. Using an FTP client or web browser, do the operation that you are attempting to do with Igor. This verifies that you have sufficient permissions on the server.
5. Use /V=7 to tell the Igor operation to display status information in the history area.
6. Try the simplest transfer you can. For example, try to download a single file that you know exists on the server.
7. If you have access to the FTP server, examine the FTP server log for clues.

Hypertext Transfer Protocol (HTTP)

The **FetchURL** function supports simple URL requests over the Internet from web or FTP servers and to local files. For example, you can use FetchURL to get the source code of a web page in text form, and then process the text to extract specific information from the response.

The **URLRequest** operation supports both simple URL requests and more complicated requests such as using the http POST, PUT, and DELETE methods. It also provides experimental support for using a proxy server.

HTTP Limitations

At this time, **FetchURL** and **BrowseURL** routines work with the HTTP protocol.

Currently not supported are features such as using network proxy servers, using the HTTP POST method to submit forms and upload files to a web server, and making secure network connections using the Secure Socket Layer (SSL) protocol.

Downloading a Web Page Via HTTP

This example uses FetchURL to download the contents of the WaveMetrics home page into a string, and then counts the number of times that the string "Igor" occurs in the text of the page.

```
Function DownloadWebPageExample()
   String webPageText = FetchURL("http://www.wavemetrics.com")
   if (numtype(strlen(webPageText)) == 2)
      Print "There was an error while downloading the web page."
   endif
   Variable count, pos
   do
      pos = strsearch(webPageText, "Igor", pos, 2)
      if (pos == -1)
         break    // No more occurrences of "Igor"
      else
         pos += 1
         count += 1
      endif
   while (1)
   Printf "The text \"Igor\" was found %d times on the web page.\r", count
End
```

Downloading a File Via HTTP

This example uses FetchURL to download a file from a web server. Because FetchURL does not support storing the downloaded data into a file directly, we store the data in memory and then use Igor to write that data to a file on disk.

Though the example uses a URL that begins with http://, FetchURL also supports https://, ftp:// and file://. You could use the code below with a different URL to download a file from an FTP server or even to access a local on-disk file.

```
Function DownloadWebFileExample()
   String url = "http://www.wavemetrics.net/IgorManual.zip"

   // Based on the URL, determine what the destination
   // file name should be. This will be the default in the
   // Save As... dialog.
   String urlStrParam = RemoveEnding(url, "/")
   Variable parts = ItemsInList(urlStrParam, "/")
   String destFileNameStr = StringFromList(parts - 1, urlStrParam, "/")
   if (strlen(destFileNameStr) < 1)
      Print "Error: Could not determine the name of the destination file."
      return 0
   endif

   Variable refNum
   Open/D/M="Save File As..."/T="????" refNum as destFileNameStr
   String fullFilePath = S_fileName

   if (strlen(fullFilePath) > 0) // No error and user didn't cancel in dialog.
      // Open the selected file so that it can later be written to.
      Open/Z/T="????" refNum as fullFilePath
      if (V_flag != 0)
         Print "There was an error opening the local destination file."
      else
         String response = FetchURL(url)
         Variable error = GetRTError(1)
         if (error == 0 && numtype(strlen(response)) == 0)
            FBinWrite refNum, response
            Close refNum
            Print "The file was successfully downloaded as " + fullFilePath
         else
            Close refNum
            DeleteFile/Z fullFilePath  // Clean up the empty file.
            Print "There was an error downloading the file."
         endif
      endif
   endif
End
```

Making a Query Via HTTP

Another use for HTTP requests is to get the server's response to a query. Many simple web forms use the HTTP GET method, which both FetchURL and URLRequest support. For example, you can simulate the submission of the basic Google search form using the following code.

```
Function WebQueryExample()
   String keywords
   String baseURL = "http://www.gutenberg.org/ebooks/search/"

   // Prompt the user to enter search keywords.
   Prompt keywords, "Enter search term"
   DoPrompt "Search Gutenberg.org", keywords
   if (V_flag == 1)
      return -1   // User clicked cancel button
   endif

   // Pass the search terms through URLEncode to
   // properly percent-encode them.
   keywords = URLEncode(keywords)

   // Build the full URL.
   String url = ""
   sprintf url, "%s?query=%s", baseURL, keywords

   // Fetch the results.
   String response
   response = FetchURL(url)
   Variable error = GetRTError(1)
   if (error != 0 || numtype(strlen(response)) != 0)
      Print "Error fetching search results."
      return -1
   endif

   // Try to extract the thumbnail image of the first result.
   String regExp = "(?s)<img class=\"cover-thumb\" src=\"(.+?)\".*"
   String firstURL
   SplitString/E=regExp response, firstURL
   firstURL = TrimString(firstURL)
   firstURL = "http:" + firstURL
   if (V_flag == 1)
      BrowseURL firstURL
   else
      Print "Could not extract the first result from the"
      Print "results page. Your search terms might not"
      Print "have given any results, or the format of"
      Print "the results may have changed so that the"
      Print "first result cannot be extracted."
      return -1
   endif

   return 0
End
```

Examples using the POST method can be found in the section ***The HTTP POST Method*** of the documentation for the **URLRequest** operation.

HTTP Troubleshooting

Here are some tips if you experience errors using **FetchURL** or **URLRequest**:

1. Use a web browser to connect to the site. This confirms that your network is operating, the server is operating, and that you are using the correct URL.
2. FetchURL and URLRequest generate an error if it cannot connect to the destination server, which could happen if your computer is not connected to the network or if the target URL contains an invalid host name or port number.

   However if the URL contains an invalid path or if the destination URL requires you to provide a username and password, these operations will likely not generate an error. The reason is these errors typically result in a web page being returned, though not the one you expected. If you need to check that a call to FetchURL returned a valid web page and not an error web page, you must do that in your own code. One possibility would be to try searching the page for key phrases, such as "File Not Found" or "Page Not Found".

   With URLRequest, you can inspect the value of the V_responseCode output variable. For successful HTTP requests, this value will usually be 200. A different value may indicate that there was an error making the request.

Operation Queue

Igor implements an operation queue that supports deferred execution of commands and some special commands for dealing with files, procedures, and experiments.

Igor services the operation queue when no procedures are running and the command line is empty. If the operation queue is not empty, Igor then executes the oldest command in the queue.

You can append a command to the operation queue using

```
Execute/P <command string>
```

The /P flag tells Igor to post the command to operation queue instead of executing it immediately.

You can also specify the /Q (quiet) or /Z (ignore error) flags. See **Execute/P** operation (page V-204) for details about /Q and /Z.

The command string can contain either a special command that is unique to the operation queue or ordinary Igor commands. The special commands are:

INSERTINCLUDE *procedureSpec*

DELETEINCLUDE *procedureSpec*

AUTOCOMPILE ON

AUTOCOMPILE OFF

COMPILEPROCEDURES

NEWEXPERIMENT

LOADFILE *filePath*

LOADFILEINNEWINSTANCE *filePath*

LOADHELPEXAMPLE *filePath*

MERGEEXPERIMENT *filePath*

RELOAD CHANGED PROCS (added in Igor Pro 9.00)

Note: The special operation queue keywords must be all caps and must have exactly one space after the keyword.

INSERTINCLUDE and DELETEINCLUDE insert or delete #include lines in the main procedure window. *procedureSpec* is whatever you would use in a #include statement except for “#include” itself.

AUTOCOMPILE ON and AUTOCOMPILE OFF require Igor Pro 8.03 or later. See **Auto-Compiling** on page III-404 for details.

COMPILEPROCEDURES does just what it says, compiles procedures. You must call it after operations such as INSERTINCLUDE that modify, add, or remove procedure files.

NEWEXPERIMENT closes the current experiment without saving.

LOADFILE opens the file specified by *filePath*. *filePath* is either a full path or a path relative to the Igor Pro Folder. The file can be any file that Igor can open. If the file is an experiment file, execute NEWEXPERIMENT first to avoid displaying a “Do you want to save” dialog. If you want to save the changes in an experiment before loading another, you can use the standard SaveExperiment operation.

LOADFILEINNEWINSTANCE opens the file specified by *filePath* in a new instance of Igor. *filePath* is either a full path or a path relative to the Igor Pro Folder. The file can be any file that Igor can open. LOADFILEINNEWINSTANCE was added in Igor Pro 7.01.

LOADHELPEXAMPLE opens the file specified by *filePath* in a new instance of Igor. It works the same as LOADFILEINNEWINSTANCE and is recommended for **Notebook Action Special Characters** that load example experiments, such as those used throughout Igor’s help files. LOADHELPEXAMPLE was added in Igor Pro 8.00.

MERGEEXPERIMENT merges the experiment file specified by *filePath* into the current experiment. Before using this, make sure you understand the caveats regarding merging experiments. See **Merging Experiments** on page II-19 for details.

RELOAD CHANGED PROCS, added in Igor Pro 9.00, causes Igor to reload procedure windows from their disk files if they have been modified outside of Igor. This is useful if you run a system that modifies procedure files using external software which is part of a system running from Igor. The sequence of events is:

1. Most likely, you use ExecuteScriptText to trigger the external modifications.

2. You use Execute/P "RELOAD CHANGED PROCS " to reload the modified procedures.

3. You use Execute/P "COMPILEPROCEDURES " to compile the modified procedures.

Operation Queue Example

```
Function DemoQueue()
   Execute/P "INSERTINCLUDE <Multipeak Fitting>"
   Execute/P "COMPILEPROCEDURES "
   Execute/P "fStartMultipeakFit2()"
   Execute/P "Sleep 00:00:04"
   Execute/P "NEWEXPERIMENT "
   Execute/P "LOADFILE :Examples:Feature Demos:Live mode.pxp"
   Execute/P "DoWindow/F Graph0"
   Execute/P "StartButton(\"StartButton\")"
End
```

Operation Queue For Loading Packages

One important use of the operation queue is providing easy access to useful procedure packages. The "Igor Pro Folder/Igor Procedures" folder contains a procedure file named “DemoLoader.ipf” that creates the Packages submenus found in various menus. To try it out, choose one of the items from the Analysis→Packages menu.

DemoLoader.ipf is an independent module. To examine it, execute:

```
SetIgorOption IndependentModuleDev=1
```

and then use the Windows→Procedure Windows submenu.

User-Defined Hook Functions

Igor calls specific user-defined functions, called “hook” functions, if they exist, when it performs certain actions. This allows savvy programmers to customize Igor’s behavior.

In some cases the hook function may inform Igor that the action has been completely handled, and that Igor shouldn’t perform the action. For example, you could write a hook function to load data from a certain kind of text file that Igor can not handle directly.

This section discusses general hook functions that do not apply to a particular window. For information on window-specific events, see Window Hook Functions.

There are two ways to get Igor to call your general hook function. The first is by using a predefined function name. For example, if you create a function named AfterFileOpenHook, Igor will automatically call it after opening a file. The second way is to explicitly tell Igor that you want it to call your hook using the **SetIgorHook** operation.

If you use a predefined hook function name, you should make the function static (private to the file containing it) so that other procedure files can use the same predefined name.

Here are the predefined hook functions.

Action	Hook Function Called
Procedures were successfully compiled	**AfterCompiledHook**
A file or experiment was opened	**AfterFileOpenHook**

Action	Hook Function Called
The Windows-only "MDI frame" (main application window) was resized	**AfterMDIFrameSizedHook**
A target window was created	**AfterWindowCreatedHook**
The debugger window is about to open	**BeforeDebuggerOpensHook**
An experiment is about to be saved	**BeforeExperimentSaveHook**
A file or XOP is about to be opened	**BeforeFileOpenHook**
A modification to a procedure window is about to cause procedures to be uncompiled	**BeforeUncompiledHook** (Igor Pro 8.03 or later)
HDF5 dataset for wave is about to be written	**HDF5SaveDataHook** (Igor Pro 9.00 or later)
Igor is about to open a new experiment	**IgorBeforeNewHook**
Igor is about to quit	**IgorBeforeQuitHook**
Igor is building and enabling menus or about to handle a menu selection	**IgorMenuHook**
Igor is about to quit	**IgorQuitHook**
Igor launching or creating a new experiment	**IgorStartOrNewHook**

To create hook functions, you must write functions with the specified names and store them in any procedure file. If you store the procedure file in "Igor Pro User Files/Igor Procedures" (see **Igor Pro User Files** on page II-31 for details), Igor will automatically open the file and compile the functions when it starts up and will execute the IgorStartOrNewHook function if it exists.

To allow for multiple procedure files to define the same predefined hook function, you should declare your hook function static. For example:

```
static Function IgorStartOrNewHook(igorApplicationNameStr)
   String igorApplicationNameStr
```

The use of the static keyword makes the function private to the procedure file containing it and allows other procedure files to have their own static function with the same name.

Igor calls static hook functions after the **SetIgorHook** functions are called. The static hook functions themselves are called in the order in which their procedure file was opened. You should not rely on any execution order among the static hook functions. However, any hook function which returns a nonzero result prevents remaining hook functions from being called and prevents Igor from performing its usual processing of the hook event. In most cases hook functions should exercise caution in returning any value other than 0. For hook functions only, returning a NaN or failing to return a value (which returns a NaN) is considered the same as returning 0.

The following sections describe the individual hook functions in detail.

AfterCompiledHook

`AfterCompiledHook()`

AfterCompiledHook is a user-defined function that Igor calls after the procedure windows have all been compiled successfully.

You can use AfterCompiledHook to initialize global variables or data folders, among other things.

The function result from AfterCompiledHook must be 0. All other values are reserved for future use.

See Also

SetIgorHook, **BeforeUncompiledHook**, **User-Defined Hook Functions** on page IV-280.

AfterFileOpenHook

AfterFileOpenHook(*refNum*, *fileNameStr*, *pathNameStr*, *fileTypeStr*, *fileCreatorStr*, *fileKind*)

AfterFileOpenHook is a user-defined function that Igor calls after it has opened a file because the user dragged it onto the Igor icon or into Igor or double-clicked it.

AfterFileOpenHook is not called when a file is opened via a menu.

Windows system files with .bin, .com, .dll, .exe, and .sys extensions aren't passed to the hook functions.

The parameters contain information about the file, which has already been opened for read-only access.

AfterFileOpenHook's return value is ignored unless *fileKind* is 9. If the returned value is zero, the default action is performed.

Parameters

refNum is the file reference number. You use this number with file I/O operations to read from the file. Igor closes the file when the user-defined function returns, and *refNum* becomes invalid. The file is opened for read-only; if you want to write to it, you must close and reopen it with write access. *refNum* will be -1 for experiment files and XOPs. In this case, Igor has not opened the file for you.

fileNameStr contains the name of the file.

pathNameStr contains the name of the symbolic path. *pathNameStr* is not the value of the path. Use the **PathInfo** operation to determine the path's value.

fileTypeStr contains the Macintosh file type, if applicable. File type codes are obsolete. Use the file name extension to determine if you want to handle the file. You can use **ParseFilePath** to obtain the extension from *fileNameStr*.

fileCreatorStr contains the Macintosh creator code, if applicable. Creator codes are obsolete so ignore this parameter.

fileKind is a number that identifies what kind of file Igor thinks it is. Values for *fileKind* are listed in the next section.

AfterFileOpenHook *fileKind* Parameter

This table describes the AfterFileOpenHook function *fileKind* parameter.

If the user's AfterFileOpenHook function returns 0, Igor performs the default action listed in the table:

Kind of File	*fileKind*	Default Action, if Any
Unknown	0	
Igor Experiment, packed (stationery, too)	1	
Igor Experiment, unpacked (stationery, too)	2	
Igor XOP	3	
Igor Binary Wave File	4	
Igor Text (data and commands)	5	
Text, no numbers detected in first two lines	6	
General Numeric text (no tabs)	7	
Numeric text Tab-Separated-Values	8	
Numeric text Tab-Separated-Values, MIME	9	Display loaded data in a new table and a new graph.
Text, with tabs	10	
Igor Notebook (unformatted or formatted)	11	
Igor Procedure	12	
Igor Help	13	

Details

AfterFileOpenHook's return value is ignored, except when *fileKind* is 9 (Numeric text, Tab-Separated-Values, MIME). If you return a value of 0, Igor executes the default action, which displays the loaded data in a table and a graph. If you return a value of 1, Igor does nothing.

Another way to disable the MIME-TSV default action is define a global variable named V_no_MIME_TSV_Load (in the root data folder) and set its value to 1. In this case any file of *fileKind* = 9 is reassigned a *fileKind* of 8.

The default action for *fileKind* = 9 makes Igor a MIME-TSV document Helper Application for Web browsers such as Netscape or Internet Explorer.

The exact criteria for Igor to consider a file to be of kind 9 are:

- *fileTypeStr* must be "TEXT" or "WMT0" (that's a zero, not an oh).
- Either the first line of the file must begin with a # character, or the name of the file must end with ".tsv" in either lower or upper case.
- The first line must contain one or more column titles. If the line starts with a # character, the first column title must not start with "include", "pragma" or the ! character. Spaces are allowed in the titles, but if two or more title columns are present, they must be separated by one tab character.
- The second line must contain one or more numbers. If two or more numbers, they must be separated by one tab character, and the first line's words must also be separated by tabs.

When the MIME-TSV file contains one column of data, it is graphed as a series of Y values.

Short columns (less than 50 values) are graphed with lines and markers, longer columns with lines only. Preferences are turned on when the graph is made.

Two columns are assumed to be X followed by Y, and are graphed as Y versus X. More columns do not affect the graph, though they are shown in the table.

Example

```
// This hook function prints the first line of opened TEXT files
// into the history area
Function AfterFileOpenHook(refNum,file,pathName,type,creator,kind)
   Variable refNum,kind
   String file,pathName,type,creator
   // Check that the file is open (read only), and of correct type
   if( (refNum >= 0) && (CmpStr(type,"TEXT")==0) )  // also "text", etc.
      String line1
      FReadLine refNum, line1 // Read the line (and carriage return)
      Print line1       // Print line in the history area.
   endif
   return 0             // don't prevent MIME-TSV from displaying
End
```

See Also

BeforeFileOpenHook and **SetIgorHook**.

BeforeDebuggerOpensHook

`BeforeDebuggerOpensHook(`*`errorInRoutineStr`*`, `*`stoppedByBreakpoint`*`)`

BeforeDebuggerOpensHook is a user-defined function that Igor calls when the debugger window is about to be opened, whether by hitting a breakpoint or when Debug on Error is enabled.

BeforeDebuggerOpensHook can be used to prevent the debugger window opening for certain error codes or in selected user-defined functions when Debug on Error is enabled. This is a feature for advanced programmers only. Most programmers will not need it.

This hook does not work well for macros or procs, because their runtime errors don't automatically open the debugger, but instead present an error dialog from which the user manually enters the debugger by clicking the Debug button.

Parameters

errorInRoutineStr contains the name of the routine (function or macro) the debugger will be stopping in as a fully-qualified name, comprised of at least "`ModuleName#RoutineName`", suitable for use with FunctionInfo.

If the routine is in a regular module procedure window (see **Regular Modules** on page IV-236), *errorInRoutineStr* will be a triple name such as "`MyIM#MyModule#MyFunction`".

stoppedByBreakpoint is 0 if the debugger is about to be shown because of Debug on Error, or non-zero if the debugger encountered a user-set breakpoint (see **Setting Breakpoints** on page IV-213).

If a breakpoint exists at the line where an error caused the debugger to appear, *stoppedByBreakpoint* will be non-zero, even though the cause was Debug on Error.

Details

If BeforeDebuggerOpensHook returns 0 or NaN (or doesn't return a value), the debugger window is opened normally.

If it returns 1, the debugger window is not shown and program execution continues.

All other return values are reserved for future use.

Example

The following hypothetical example:

1. Prevents breakpoints from bringing up the debugger, unless DEBUGGING is defined.
2. Prints the name of the routine with the error, and the error message.
3. Beeps before the debugger appears.

```
Function ProvokeDebuggerInFunction()
   DebuggerOptions enable=1, debugOnError=1     // Enable debug on error

   ProvokeDebugger()
```

```
End

Function ProvokeDebugger()
   Variable var=0 // Put a breakpoint here.
                  // Without a #define DEBUGGING, the breakpoint is skipped.
   Make/O $""     // Cause an error
   Print "Back from bad Make command in function"
End

static Function BeforeDebuggerOpensHook(pathToErrorFunction,isUserBreakpoint)
   String pathToErrorFunction
   Variable isUserBreakpoint

   #ifndef DEBUGGING
      if( isUserBreakpoint )
         return 1 // Ignore user breakpoints we forgot to clear.
                  // Don't use this during development!
      endif
   #endif

   Print "stackCrawl = ", GetRTStackInfo(0)
   Print "FunctionInfo = ", FunctionInfo(pathToErrorFunction)

   // Don't clear errors unless you're preventing the debugger from appearing
   Variable clearErrors= 0
   Variable rtErr= GetRTError(clearErrors)   // Get the error #

   Variable substitutionOption= exists(pathToErrorFunction)== 3 ? 3 : 2
   String errorMessage= GetErrMessage(rtErr,substitutionOption)

   Beep     // Audible cue that the debugger is showing up!

   Print "Error \""+errorMessage+"\" in "+pathToErrorFunction+""

   return 0 // Return 0 to show the debugger; an unexpected error occurred.
End

•ProvokeDebuggerInFunction()  // Execute this in the command line
  stackCrawl =
      ProvokeDebuggerInFunction;ProvokeDebugger;BeforeDebuggerOpensHook;
  FunctionInfo =
      NAME:ProvokeDebugger;PROCWIN:Procedure;MODULE:;INDEPENDENTMODULE:;...
  Error "Expected name" in ProcGlobal#ProvokeDebugger
  Back from bad Make command in function
```

See Also

SetWindow, **SetIgorHook**, and **User-Defined Hook Functions** on page IV-280

Static Functions on page IV-105, **Regular Modules** on page IV-236, **Independent Modules** on page IV-238

FunctionInfo, **GetRTStackInfo**, **GetRTError**, **GetRTErrMessage**

Conditional Compilation on page IV-108

AfterMDIFrameSizedHook

AfterMDIFrameSizedHook(*param*)

AfterMDIFrameSizedHook is a user-defined function that Igor calls when the Windows-only "MDI frame" (main application window) has been resized.

AfterMDIFrameSizedHook can be used to resize windows to fit the new frame size. See **GetWindow** kwFrame and **MoveWindow**.

Parameters

param is one of the following values:

Size Event	*param*
Normal resize	0
Minimized	1
Maximized	2
Moved	3

Details

This function is not called on Macintosh.

Resizing the MDI frame by the top left corner calls AfterMDIFrameSizedHook twice: first for the move (param = 3) and then for the normal resize (param = 0).

Igor currently ignores the value returned by AfterMDIFrameSizedHook. Return 0 in case Igor uses this value in the future.

See Also

SetWindow, **GetWindow**, **SetIgorHook**, and **User-Defined Hook Functions** on page IV-280.

AfterWindowCreatedHook

`AfterWindowCreatedHook(`*`windowNameStr`*`, `*`winType`*`)`

AfterWindowCreatedHook is a user-defined function that Igor calls when a target window is first created.

AfterWindowCreatedHook can be used to set a window hook on target windows created by the user or by other procedures.

Parameters

windowNameStr contains the name of the created window.

winType is the type of the window, the same value as returned by **WinType**.

Details

“Target windows” are graphs, tables, layout, panels, and notebook windows.

AfterWindowCreatedHook is not called when an Igor experiment is being opened.

Igor ignores the value returned by AfterWindowCreatedHook.

See Also

SetWindow, **SetIgorHook**, and **User-Defined Hook Functions** on page IV-280.

BeforeExperimentSaveHook

`BeforeExperimentSaveHook(`*`refNum`*`, `*`fileNameStr`*`, `*`pathNameStr`*`, `*`fileTypeStr`*`, `*`fileCreatorStr`*`, `*`fileKind`*`)`

BeforeExperimentSaveHook is a user-defined function that Igor calls when an experiment is about to be saved by Igor.

Igor ignores the value returned by BeforeExperimentSaveHook.

Parameters

refNum identifies what is causing the experiment to be saved:

Cause	*refNum*
Save Experiment (File menu or SaveExperiment)	1
Save Experiment As (File menu or SaveExperiment)	2
Save Experiment Copy (File menu or SaveExperiment/C)	3
Autosave Experiment (direct mode)	17 (0x10 + 1)
Autosave Experiment Copy (indirect mode)	19 (0x10 + 3)

Before Igor Pro 9.00, *refNum* was always -1.

fileNameStr contains the name of the file.

pathNameStr contains the name of the symbolic path. *pathNameStr* is not the value of the path. Use the **PathInfo** operation to determine the path's value.

fileTypeStr contains the Macintosh file type, if applicable. File type codes are obsolete. Use the file name extension to determine if you want to handle the file. You can use **ParseFilePath** to obtain the extension from *fileNameStr*

fileCreatorStr contains the Macintosh creator code, if applicable. Creator codes are obsolete so ignore this parameter.

Variable *fileKind* is a number that identifies what kind of file Igor will be saving:

Kind of File	***fileKind***
Igor Experiment, packed*	1
Igor Experiment, unpacked*	2

* Including stationery experiment files.

Details

You can determine the full directory and file path of the experiment by calling the **PathInfo** operation with $*pathNameStr*.

Example

This example prints the full file path of the about-to-be-saved experiment to the history area, and deletes all unused symbolic paths.

```
#pragma rtGlobals=1        // treat S_path as local string variable

Function BeforeExperimentSaveHook(rN,fileName,path,type,creator,kind)
   Variable rN,kind
   String fileName,path,type,creator

   PathInfo $path             // puts path value into (local) S_path
   Printf "Saved \"%s\" experiment\r",S_path+fileName

   KillPath/A/Z               // Delete all unneeded symbolic paths
End
```

See Also

The **SetIgorHook** operation.

BeforeFileOpenHook

BeforeFileOpenHook(*refNum*, *fileNameStr*, *pathNameStr*, *fileTypeStr*, *fileCreatorStr*, *fileKind*)

BeforeFileOpenHook is a user-defined function that Igor calls when a file *is* about to be opened because the user dragged it onto the Igor icon or into Igor or double-clicked it.

BeforeFileOpenHook is not called when a file is opened via a menu.

Windows system files with .bin, .com, .dll, .exe, and .sys extensions aren't passed to the hook functions.

The value returned by BeforeFileOpenHook informs Igor whether the hook function handled the open event and therefore Igor should not perform its default action. In some cases, this return value is ignored, and Igor performs the default action anyway.

Parameters

refNum is the file reference number. You use this number with file I/O operations to read from the file. Igor closes the file when the user-defined function returns, and *refNum* becomes invalid. The file is opened for read-only; if you want to write to it, you must close and reopen it with write access. *refNum* will be -1 for experiment files and XOPs. In this case, Igor has not opened the file for you.

fileNameStr contains the name of the file.

pathNameStr contains the name of the symbolic path. *pathNameStr* is not the value of the path. Use the **PathInfo** operation to determine the path's value.

fileTypeStr contains the Macintosh file type, if applicable. File type codes are obsolete. Use the file name extension to determine if you want to handle the file. You can use **ParseFilePath** to obtain the extension from *fileNameStr*

fileCreatorStr contains the Macintosh creator code, if applicable. Creator codes are obsolete so ignore this parameter.

fileKind is a number that identifies what kind of file Igor thinks it is. Values for *fileKind* are listed in the next section.

BeforeFileOpenHook *fileKind* Parameter

This table describes the BeforeFileOpenHook function *fileKind* parameter.

Kind of File	*fileKind*	Default Action, if Any
Unknown	0	
Igor Experiment, packed *	1	(Hook not called)
Igor Experiment, unpacked*	2	(Hook not called)
Igor XOP	3	
Igor Binary Wave File	4	Data loaded
Igor Text (data and commands)	5	Data loaded, commands executed
Text, no numbers detected in first two lines	6	Opened as unformatted notebook
General Numeric text (no tabs)	7	Data loaded as general text
Numeric text Tab-Separated-Values	8	Data loaded as delimited text
Numeric text Tab-Separated-Values, MIME	9	Display loaded data in a new table and a new graph.
Text, with tabs	10	Opened as unformatted notebook
Igor Notebook (unformatted or formatted)	11	Opened as notebook
Igor Procedure	12	*Always* opened as procedure file
Igor Help	13	*Always* opened as help file

* Including stationery experiment files.

Details

BeforeFileOpenHook must return 1 if Igor is not to take action on the file (it won't be opened), or 0 if Igor is permitted to take action on the file. Igor ignores the return value for *fileKind* values of 3, 12, and 13. The hook function is not called for Igor experiments (*fileKind* values of 1 and 2).

Igor always closes the file when the user-defined function returns, and *refNum* becomes invalid (don't store the value of *refNum* in a global for use by other routines, since the file it refers to has been closed).

Example

This example checks the first line of the file about to be opened to determine whether it has a special, presumably user-specific, format. If it does, then LoadMyFile (another user-defined function) is called to load it. LoadMyFile presumably loads this custom data file, and returns 1 if it succeeded. If it returns 0 then Igor will open it using the Default Action from the above table.

```
Function BeforeFileOpenHook(refNum,fileName,path,type,creator,kind)
   Variable refNum,kind
   String fileName,path,type,creator

   Variable handledOpen=0
   if( CmpStr(type,"TEXT")==0 )              // text files only
      String line1
      FReadLine refNum, line1 // First line (and carriage return)
      if( CmpStr(line1[0,4],"XYZZY") == 0 )  // My special file
```

```
         FSetPos refNum, 0             // rewind to start of file
         handledOpen= LoadMyFile(refNum)  // returns 1 if loaded OK
      endif
   endif
   return handledOpen   // 1 tells Igor not to open the file
End
```

See Also

AfterFileOpenHook, **SetIgorHook**.

BeforeUncompiledHook

`BeforeUncompiledHook(`*`changeCode`*`, `*`procedureWindowTitleStr`*`, `*`textChangeStr`*`)`

BeforeUncompiledHook is a user-defined function that Igor calls before procedures enter the uncompiled state after a change to the procedures.

You can use BeforeUncompiledHook to shut down background tasks or threads before the user functions they depend on go away. You can use **AfterCompiledHook** to restart them.

BeforeUncompiledHook was added in Igor Pro 8.03.

Parameters

changeCode is one of the following values:

Pending Change	*changeCode*	Scenarios
Text deletion	1	Delete/backspace key, cut, saved recreation macro, merge experiment
Text insertion	2	User typing, paste insert, Execute/P "INSERTINCLUDE "
Text replacement	3	User typing, paste over selected text
Open procedure file	4	File→Open Procedure, OpenProc
Close procedure file	5	Procedure close icon click, CloseProc
SetIgorOption poundDefine	6	SetIgorOption poundDefine causes a recompile
SetIgorOption poundUndefine	7	SetIgorOption poundUndefine causes a recompile

procedureWindowTitleStr contains the title of the procedure window whose text is about to change. If the procedure window is in an independent module, the title is followed by

```
[<nameOfIndependentModule>]
```

as described in documentation for the **WinList** function.

The content of *textChangeStr* depends on *changeCode*:

changeCode	*textChangedStr*
1	""
2	Inserted text
3	Replacement text
4	""
5	""
6	*name* defined by SetIgorOption poundDefine=*name*
7	*name* undefined by SetIgorOption poundUndefine=*name*

Details

In most cases your BeforeUncompiledHook function should return 0.

Returning non-zero prevents the pending change to the text from taking effect for changeCode = 1, 2, or 3, though only when procedures are already compiled.

The returned value is ignored for other values of *changeCode*.

BeforeUncompiledHook is not called before a new experiment is opened, before the experiment is reverted, or before Igor quits. Use **IgorBeforeNewHook** and **IgorBeforeQuitHook** for those scenarios.

See Also

User-Defined Hook Functions on page IV-280, **SetIgorHook**, **AfterCompiledHook**, **Operation Queue** on page IV-278.

HDF5SaveDataHook

`HDF5SaveDataHook(`*`s`*`)`

HDF5SaveDataHook is a user-defined function that Igor calls when saving a wave to an HDF5 file. It allows advanced users to control compression of the dataset written for the wave.

HDF5SaveDataHook was added in Igor Pro 9.00.

NOTE: HDF5SaveDataHook is an experimental feature for advanced users only. The feature may be changed or removed. If you find it useful, please let us know, and send your function and an explanation of what purpose it serves.

Parameters

s is an HDF5SaveDataHookStruct.

Details

See **Using HDF5SaveDataHook** on page II-215 for details.

IgorBeforeNewHook

`IgorBeforeNewHook(`*`igorApplicationNameStr`*`)`

IgorBeforeNewHook is a user-defined function that Igor calls before a new experiment is opened in response to the New Experiment, Revert Experiment, or Open Experiment menu items in the File menu.

You can use IgorBeforeNewHook to clean up the current experiment, or to avoid losing unsaved data even if the user chooses to not save the current experiment.

Igor ignores the value returned by IgorBeforeNewHook.

Parameters

igorApplicationNameStr contains the name of the currently running Igor Pro application.

See Also

IgorStartOrNewHook and **SetIgorHook**.

IgorBeforeQuitHook

`IgorBeforeQuitHook(`*`unsavedExp,unsavedNotebooks,unsavedProcedures`*`)`

IgorBeforeQuitHook is a user-defined function that Igor calls just before Igor is about to quit, before any save-related dialogs have been presented.

Parameters

unsavedExp is 0 if the experiment is saved, non-zero if unsaved.

unsavedNotebooks is the count of unsaved notebooks.

unsavedProcedures is the count of unsaved procedures.

The save state of packed procedure and notebook files is part of *unsavedExp*, not *unsavedNotebooks* or *unsavedProcedures*. This applies to adopted procedure and notebook files and new procedure and notebook windows that have never been saved.

Details

IgorBeforeQuitHook should normally return 0. In this case, Igor presents the "Do you want to save" dialog, and if the user approves, proceeds with the quit, which includes calling IgorQuitHook.

If IgorBeforeQuitHook returns 1, then the normal save-and-quit process is aborted and Igor quits immediately. The current experiment, notebooks, and procedures are not saved, no dialogs are presented to the user, and IgorQuitHook is not called.

If IgorBeforeQuitHook returns 2, then the normal save-and-quit process is aborted and Igor does not quit. The current experiment, notebooks, and procedures are not saved, no dialogs are presented to the user, and IgorQuitHook is not called. This return value was first supported in Igor Pro 8.05.

See Also

IgorQuitHook and **SetIgorHook**.

IgorMenuHook

IgorMenuHook(*isSelection*, *menuStr*, *itemStr*, *itemNo*, *activeWindowStr*, *wType*)

IgorMenuHook is a user-defined function that Igor calls just before and just after menu selection, whether by mouse or keyboard.

Parameters

isSelection is 0 before a menu item has been selected and 1 after a menu item has been selected.

When *isSelection* is 1, *menuStr* is the name of the selected menu. It is always in English, regardless of the localization of Igor. When *isSelection* is 0, *menuStr* is "".

When *isSelection* is 1, *itemStr* is the name of the selected menu item. When *isSelection* is 0, *itemStr* is "".

When *isSelection* is 1, *itemNo* is the one-based item number of the selected menu item. When *isSelection* is 0, *itemNo* is 0.

activeWindowStr identifies the active window. See details below.

wType identifies the kind of window that *activeWindowStr* identifies. It returns the same values as the **WinType** function.

activeWindowStr Parameter

activeWindowStr identifies the window to which the menu selection will apply. It can be a window name, window title, or special keyword, as follows:

Window	***activeWindowStr***
Target window	Window name. The target window is that top graph, table, page layout, notebook, control panel, Gizmo plot, or XOP target window.
Command window	`kwCmdHist` (as used with **GetWindow**).
Procedure window	Window title as shown in the window’s title bar. The built-in procedure window is “Procedure”.
XOP non-target window	The window title as shown in the window’s title bar.

See **Window Names and Titles** on page II-45 for a discussion of the distinction.

Details

IgorMenuHook is called with *isSelection* set to 0 after all the menus have been enabled and before a mouse click or keyboard equivalent is handled.

The return value should normally be 0. If the return value is nonzero (1 is usual) then the active window’s hook function (see **SetWindow** operation on page V-865) is not called for the enablemenu event.

IgorMenuHook is called with *isSelection* set to 1 after the menu has been selected and before Igor has acted on the selection.

If the IgorMenuHook function returns 0, Igor proceeds to call the active window’s hook function for the menu event. (If the window hook function exists and returns nonzero, Igor ignores the menu selection. Otherwise Igor handles the menu selection normally.)

If the IgorMenuHook function returns nonzero (1 is recommended), Igor does not call the remaining hook functions and Igor ignores the menu selection.

Example

This example invokes the Export Graphics menu item when Command-C (*Macintosh*) or Ctrl+C (*Windows*) is selected for all graphs, preventing Igor from performing the usual Copy.

```
Function IgorMenuHook(isSel, menuStr, itemStr, itemNo, activeWindowStr, wt)
   Variable isSel
   String menuStr, itemStr
   Variable itemNo
   String activeWindowStr
   Variable wt

   Variable handled= 0
   if( Cmpstr(menuStr,"Edit") == 0 && CmpStr(itemStr,"Copy") == 0 )
      if( wt == 1 )                                  // graph
         // DoIgorMenu would cause recursion, so we defer execution
         Execute/P/Q/Z "DoIgorMenu \"Edit\", \"Export Graphics\""
         handled= 1
      endif
   endif

   return handled
End
```

See Also

SetWindow, **Execute**, and **SetIgorHook**.

IgorQuitHook

`IgorQuitHook(`*`igorApplicationNameStr`*`)`

IgorQuitHook is a user-defined function that Igor calls when Igor is about to quit.

The value returned by IgorQuitHook is ignored.

Parameters

igorApplicationNameStr contains the name of the currently running Igor Pro application (including the .exe extension under Windows).

Details

You can determine the full directory and file path of the Igor application by calling the **PathInfo** operation with the Igor path name. See the example in **IgorStartOrNewHook** on page IV-292.

See Also

IgorBeforeQuitHook and **SetIgorHook**.

IgorStartOrNewHook

`IgorStartOrNewHook(`*`igorApplicationNameStr`*`)`

IgorStartOrNewHook is a user-defined function that Igor calls when starting up and when creating a new experiment. It is also called if Igor is launched as a result of double-clicking a saved Igor experiment.

Igor ignores the value returned by IgorStartOrNewHook.

Parameters

igorApplicationNameStr contains the name of the currently running Igor Pro application (including the .exe extension under Windows).

Details

You can determine the full directory and file path of the Igor application by calling the **PathInfo** operation with the Igor path name.

Example

This example prints the full path of Igor application whenever Igor starts up or creates a new experiment:

```
Function IgorStartOrNewHook(igorApplicationNameStr)
   String igorApplicationNameStr

   PathInfo Igor              // puts path value into (local) S_path
   printf "\"%s\" (re)starting\r", S_path + igorApplicationNameStr
End
```

See Also

IgorBeforeNewHook and **SetIgorHook**.

Window User Data

The window user data feature provides a way for packages that create or manage windows to store per-window settings. You can store arbitrary data with a window using the userdata keyword with the **SetWindow**.

Each window has a primary, unnamed user data that is used by default.

You can also store an unlimited number of different user data strings by specifying a name for each different user data string. The name can be any legal Igor name. It should be distinct to prevent clashes between packages.

Packages should use named user data.

You can retrieve information from the default user data using the **GetWindow**. To retrieve named user data, you must use the **GetUserData**.

Here is a simple example of user data using the top window:

```
SetWindow kwTopWin,userdata= "window data"
Print GetUserData("","","")
```

Although there is no size limit to how much user data you can store, it does have to be stored as part of the recreation macro for the window when experiments are saved. Consequently, huge user data can slow down experiment saving and loading.

You can also attach user data to traces in graphs using the userData keyword of the ModifyGraph operation and to controls using the userData keyword of the various control operations.

Window Hook Functions

A window hook function is a user-defined function that receives notifications of events that occur in a specific window. Your window hook function can detect and respond to events of interest. You can then allow Igor to also process the event or inform Igor that you have handled it.

This section discusses window hook functions that apply to a specific window. For information on general events hooks, see **User-Defined Hook Functions** on page IV-280.

To handle window events, you first write a window hook function and then use the **SetWindow** operation to install the hook on a particular window. This example shows how you would detect arrow key events in a particular window. To try it, paste the code below into the procedure window and then execute `DemoWindowHook()`:

```
Function MyWindowHook(s)
   STRUCT WMWinHookStruct &s

   Variable hookResult = 0 // 0 if we do not handle event, 1 if we handle it.

   switch(s.eventCode)
      case 11:            // Keyboard event
         switch (s.keycode)
            case 28:
               Print "Left arrow key pressed."
               hookResult = 1
               break
            case 29:
               Print "Right arrow key pressed."
```

```
                hookResult = 1
                break
            case 30:
                Print "Up arrow key pressed."
                hookResult = 1
                break
            case 31:
                Print "Down arrow key pressed."
                hookResult = 1
                break
            default:
                // The keyText field requires Igor Pro 7 or later
                // See Keyboard Events on page IV-300
                Printf "Key pressed: %s\r", s.keyText
                break
         endswitch
         break
   endswitch

   return hookResult // If non-zero, we handled event and Igor will ignore it.
End

Function DemoWindowHook()
   DoWindow/F DemoGraph    // Does graph exist?
   if (V_flag == 0)
      Display /N=DemoGraph // Create graph
      SetWindow DemoGraph, hook(MyHook)=MyWindowHook  // Install window hook
   endif
End
```

The window hook function receives a WMWinHookStruct structure as a parameter. WMWinHookStruct is a built-in structure that contains all of the information you might need to respond to an event. One of its fields, the eventCode field, specifies what kind of event occurred.

If your hook function returns 1, this tells Igor that you handled the event and Igor does not handle it. If your hook function returns 0, this tells Igor that you did not handle the event, so Igor does handle it.

This example uses a named window hook. In this case the name is MyHook. More than one procedure file can install a hook on a given window. The purpose of the name is to allow a package to install and remove its own hook function without disturbing the hook functions of other packages. Choose a distinct hook function name that is unlikely to conflict with other hook names.

Earlier versions of Igor supported only one unnamed hook function. This meant that only one package could hook any particular window. Unnamed hook functions are still supported for backward compatibility but new code should always use named hook functions.

Window Hooks and Subwindows

Except for the modified event (see **Modified Events** on page IV-299), Igor calls window hook functions for top-level windows only, not for subwindows. If you want to hook a subwindow, you must set the hook on the top-level window. In the hook function, test to see if the subwindow is active. For example, this code, at the start of a window hook function, insures that the hook runs only if a subwindow named G0 is active.

```
GetWindow $s.winName activeSW
String activeSubwindow = S_value
if (CmpStr(activeSubwindow,"G0") != 0)
   return 0
endif
```

Exterior panels (see **Exterior Control Panels** on page III-443) are top-level windows even though they are subwindows. To hook an exterior subwindow, you must install the hook on the exterior panel using subwindow syntax.

Named Window Hook Functions

A named window hook function takes one parameter - a `WMWinHookStruct` structure. This built-in structure provides your function with information about the status of various window events.

The named window hook function has this format:

```
Function MyWindowHook(s)
   STRUCT WMWinHookStruct &s

   Variable hookResult = 0

   switch(s.eventCode)
      case 0:              // Activate
         // Handle activate
         break

      case 1:              // Deactivate
         // Handle deactivate
         break

      // And so on . . .
   endswitch

   return hookResult       // 0 if nothing done, else 1
End
```

If you handle a particular event and you want Igor to ignore it, return 1 from the hook function. However, you cannot make Igor ignore a window kill event - once the kill event is received the window will be killed.

Named Window Hook Events

Here are the events passed to a named window hook function:

eventCode	eventName	Notes
0	"Activate"	
1	"Deactivate"	
2	"Kill"	Returning 1 when you receive this event does not cause Igor to ignore the event. At this point, you cannot prevent the window from being killed. See the killVote event to prevent the window being killed.
3	"Mousedown"	
4	"Mousemoved"	
5	"Mouseup"	
6	"Resize"	
7	"Cursormoved"	See **Cursors — Moving Cursor Calls Function** on page IV-339.
8	"Modified"	A modification to the window has been made. See **Modified Events** on page IV-299.
9	"Enablemenu"	
10	"Menu"	
11	"Keyboard"	See **Keyboard Events** on page IV-300.
12	"moved"	
13	"renamed"	

eventCode	eventName	Notes
14	“subwindowKill”	One of the window’s subwindows is about to be killed.
15	“hide”	The window or one of its subwindows is about to be hidden. See **Window Hook Show and Hide Events** on page IV-304.
16	“show”	The window or one of its subwindows is about to be unhidden. See **Window Hook Show and Hide Events** on page IV-304.
17	“killVote”	Window is about to be killed. Return 2 to prevent the window from being killed, otherwise return 0. **Note**: Don’t delete data structures during this event, use killVote only to decide whether the window kill should actually happen. Delete data structures in the kill event. See **Window Hook Deactivate and Kill Events** on page IV-303.
18	“showTools”	
19	“hideTools”	
20	“showInfo”	
21	“hideInfo”	
22	“mouseWheel”	
23	“spinUpdate”	This event is sent only to windows marked via DoUpdate/E=1 as progress windows. It is sent when Igor spins the beachball cursor. See **Progress Windows** on page IV-156 for details.
24	"tableEntryAccepted"	This event is sent to tables only. It is sent when the user manually accepts text entered in the table entry line, for example by clicking the checkmark button or pressing the Enter or Return keys. It is also sent by a **ModifyTable** `entryMode=1` command. This hook event was added in Igor Pro 8.03.
25	"tableEntryCancelled"	This event is sent to tables only. It is sent when the user cancels text entry, for example by clicking the X button or pressing the Escape key. It is also sent by a **ModifyTable** `entryMode=0` command. This hook event was added in Igor Pro 8.03.
26	"earlyKeyboard"	See **EarlyKeyboard Events** on page IV-302.

WMWinHookStruct

The `WMWinHookStruct` structure has members as described in the following tables:

Base `WMWinHookStruct` Structure Members

Member	Description
`char winName[MAX_PATH_LENGTH+1]`	hcSpec of the affected (sub)window.
`STRUCT Rect winRect`	Local coordinates of the affected (sub)window.
`STRUCT Point mouseLoc`	Mouse location.
`double ticks`	Tick count when event happened.
`Int32 eventCode`	See see **eventCode** table on page IV-295.
`char eventName[255+1]`	Name-equivalent of `eventCode`, see **eventCode** table on page IV-295. Added in Igor 5.03.
`Int32 eventMod`	Bitfield of modifiers. See description for `MODIFIERS:`*`flags`*.

Members of `WMWinHookStruct` Structure Used with `menu` Code

Member	Description
`char menuName[255+1]`	Name of menu (in English) as used by **SetIgorMenuMode**.
`char menuItem[255+1]`	Text of the menu item as used by SetIgorMenuMode

Members of `WMWinHookStruct` Structure Used with `keyboard and earlyKeyboard` Code

Member	Description
`Int32 keycode`	ASCII value of key struck. Function keys are not available but navigation keys are translated to specific values and will be the same on Macintosh and Windows. This field can not represent non-ASCII text such as accented characters. Use keyText instead.
`Int32 specialKeyCode`	See **Keyboard Events** on page IV-300. This field was added in Igor Pro 7.
`char keyText[16]`	UTF-8 representation of key struck. This field was added in Igor Pro 7.
`char focusCtrl[MAX_WIN_PATH+1]`	Used only with **EarlyKeyboard Events**. This field was added in Igor Pro 9.

Members of `WMWinHookStruct` Structure Used with `cursormoved` Code

Member	Description
`char traceName[MAX_OBJ_NAME+1]`	The name of the trace or image to which the moved cursor is attached or which supplies the X (and Y) values. Can be `""` if the cursor is free.
`char cursorName[2]`	Cursor name A through J.
`double pointNumber`	Point number of the trace or the X (row) point number of the image where the cursor is attached. If the cursor is “free”, `pointNumber` is actually the fractional relative *xValue* as used in the **Cursor**/F/P command.
`double yPointNumber`	Valid only when the cursor is attached to a two-dimensional item such as an image, contour, or waterfall plot, or when the cursor is free. If attached to an image, contour, or waterfall plot, `yPointNumber` is the Y (column) point number of the image where the cursor is attached. If the cursor is “free”, `yPointNumber` is actually the fractional relative *yValue* as used in the **Cursor**/F/P command.
`Int32 isFree`	Has value of 1 if the cursor is not attached to anything, or value of 0 if it is attached to a trace, image, contour, or waterfall.

Members of `WMWinHookStruct` Structure Used with `mouseWheel` Code

Member	Description
`double wheelDy`	Vertical lines to scroll. Typically +1 or -1.
`double wheelDx`	Horizontal lines to scroll. Typically +1 or -1. On Windows, horizontal mouse wheel requires Vista.

Members of `WMWinHookStruct` Used with `renamed` Code

Member	Description
`char oldWinName[MAX_OBJ_NAME+1]`	Old name of the window or subwindow. Not the absolute path *hcSpec*, just the name.

User-Modifiable Members of `WMWinHookStruct` Structure

Member	Description
`Int32 doSetCursor`	Set to 1 to change cursor to that specified by `cursorCode`.
`Int32 cursorCode`	See **Setting the Mouse Cursor**.

Mouse Events

Igor sends mouse down and mouse up events with the `eventCode` field of the `WMWinHookStruct` structure set to 3 or 5. With rare exceptions, your hook function should act on the mouse up event. Consider, for instance, a click on a button- the button's action should occur when you release the mouse button.

One exception is for a contextual click, that is, right-click on Windows, or Ctrl-click on Macintosh. On Macintosh, a contextual menu should be displayed on the mouse-down event. On Windows, the contextual menu should be shown on mouse-up, but in Igor confoundingly, the mouse-down is not delivered until the mouse-up has occurred. Thus, displaying a contextual menu on mouse-down on Windows will do the Right Thing.

A mouse click on a table cell causes Igor to send a mouse down event to your hook function before Igor acts on the click, allowing you to block the action by returning a non-zero result. Igor then sends a mouse down event after Igor has acted on the click.

In Igor6, the mouse down event was sent only when the selection was finished, that is, after the mouse up event occurred. If you have existing code that uses the mouse down event to get a table selection, you need to change your code to use the mouse up event.

Modified Events

A modified event is sent when a modification to the window has been made. This is sent to graph and notebook windows only.

As of Igor Pro 9.00, modified events are sent to both top-level graphs and notebooks and also to graph and notebook subwindows. That means that a window hook on a control panel window could receive a modified event for a notebook or graph subwindow in the panel. In that case, the winName member of the WMWinHookStruct structure is set to the subwindow path.

It is an error to try to kill a graph or notebook window or subwindow from the window hook during the modified event.

The modified event is issued in Igor's outer loop when it is idling (i.e., after any processing is finished). This means that, if you modify a graph or notebook programmatically, the modified event is not sent to hook functions until Igor returns to idling.

Most changes to the graph are reported by the modified event, but not all:

Action	**Sends Modified Event**
Dragging axis	Yes
Dragging plot area	Yes
Dragging trace	Sent only after drag ends
Dragging annotation	Yes - see note 1
Adding, removing or changing a control	No
Showing or hiding drawing tools	No - see note 1
Showing or hiding info panel	No - see note 1

1. At present, only in top-level windows, not in graph subwindows.

2. But see events 18, 19, 20, and 21 in **Named Window Hook Events** on page IV-295.

In Igor Pro 9.00 and later, if you modify a notebook from a modified event, this does not cause recursion into your hook function.

If your hook function modifies a graph window, this may trigger another modify event after your function returns from the first. Without something to stop it, this will create a cascade of modified events. Checking for recursion in your hook function does not solve the problem because subsequent hook event calls occur after your hook function has returned.

To break the chain, your hook function needs to modify the graph window only if necessary. For example, assume that your hook function needs to change something when the range of an axis in a graph changes. You need to store the axis range from the last time your function was called using user data (see **GetUserData**). If, in the current call, the axis range has not changed, your function must return without making any changes, thereby breaking the chain.

Keyboard Events

The `WMWinHookStruct` structure has three members used with keyboard and earlyKeyboard events. A fourth member, `focusCtrl`, is used only with earlyKeyboard events, described below.

The `keycode` field works with ASCII characters and some special keys such as keyboard navigation keys.

The `specialKeyCode` fields works with navigation keys, function keys and other special keys. `specialKeyCode` is zero for normal text such as letters, numbers and punctuation.

The `keyText` field works with ASCII characters and non-ASCII characters such as accented characters.

The `specialKeyCode` and `keyText` fields were added in Igor Pro 7. New code that does not need to run with earlier Igor versions should use these new fields instead of the keycode field. See **Keyboard Events Example** on page IV-301 for an example.

Here are the codes for the `specialKeyCode` and `keyCode` fields:

Key	specialKeyCode	keyCode	Note
F1 through F39	1 through 39	Not supported	Function keys
LeftArrow	100	28	
RightArrow	101	29	
UpArrow	102	30	
DownArrow	103	31	
PageUp	104	11	
PageDown	105	12	
Home	106	1	
End	107	4	
Return	200	13	
Enter	201	3	
Tab	202	9	
BackTab	203	Not supported	Tab with Shift pressed
Escape	204	27	
Delete	300	8	Backspace key
ForwardDelete	301	127	
Clear	302	Not supported	
Insert	303	Not supported	
Help	400	Not supported	
Break	401	Not supported	Pause/Break key

Key	specialKeyCode	keyCode	Note
Print	402	Not supported	
SysReq	403	Not supported	

Keyboard Events Example

This example illustrates the use of the various keyboard event fields in the WMWinHookStruct structure. It requires Igor Pro 7 or later.

```
Function KeyboardWindowHook(s)
   STRUCT WMWinHookStruct &s

   Variable hookResult = 0    // 0 if we do not handle event, 1 if we handle it.

   String message = ""

   switch(s.eventCode)
      case 11:                // Keyboard event
         String keyCodeInfo
         sprintf keyCodeInfo, "s.keycode = 0x%04X", s.keycode
         if (strlen(message) > 0)
            message += "\r"
         endif
         message +=keyCodeInfo

         message += "\r"
         String specialKeyCodeInfo
         sprintf specialKeyCodeInfo, "s.specialKeyCode = %d", s.specialKeyCode
         message +=specialKeyCodeInfo
         message += "\r"

         String keyTextInfo
         sprintf keyTextInfo, "s.keyText = \"%s\"", s.keyText
         message +=keyTextInfo

         String text = "\\Z24" + message
         Textbox /C/N=Message/W=KeyboardEventsGraph/A=MT/X=0/Y=15 text

         hookResult = 1    // We handled keystroke
         break
   endswitch

   return hookResult // If non-zero, we handled event and Igor will ignore it.
End

Function DemoKeyboardWindowHook()
   DoWindow/F KeyboardEventsGraph      // Does graph exist?
   if (V_flag == 0)
      // Create graph
      Display /N=KeyboardEventsGraph as "Keyboard Events"

      // Install hook
      SetWindow KeyboardEventsGraph, hook(MyHook)=KeyboardWindowHook

      String text = "\\Z24" + "Press a key"
      Textbox /C/N=Message/W=KeyboardEventsGraph/A=MT/X=0/Y=15 text
   endif
End
```

EarlyKeyboard Events

The earlyKeyboard event was added in Igor Pro 9.00.

The earlyKeyboard event is like the keyboard event but is sent, to graph and control panel windows only, before other components of those windows get the keyboard event. Its purpose is to let you filter key presses in graphs and panels before they reach a control.

The keycode, specialKeyCode, and keyText fields work the same as with the keyboard event.

The earlyKeyboard event sets the focusCtrl field which was added in Igor Pro 9.00.

If the event is earlyKeyboard and the window or its subwindows have a control with keyboard focus, the focusCtrl field is set to the name of the control and the winName field is set to the path of the window or subwindow that contains the control. If you return a non-zero result from the hook function when it receives the earlyKeyboard event, you prevent the control from receiving the keyboard event.

Only graphs and control panels receive earlyKeyboard events. Other window types receive normal keyboard events before any use is made of the keyboard events, making the earlyKeyboard event redundant.

Setting the Mouse Cursor

An advanced programmer can use a named window hook function to change the mouse cursor.

You might want to do this, for example, if your window hook function intercepts mouse events on certain items (e.g., waves) and performs custom actions. By setting a custom mouse cursor you indicate to the user that clicking the items results in different-from-normal actions.

See the Mouse Cursor Control example experiment - in Igor choose File→Example Experiments→Programming→Mouse Cursor Control.

Panel Done Button Example

This example uses a window hook and button action procedure to implement a panel dialog with a Done button such that the panel can't be closed by clicking the panel's close widget, but can be closed by the Done button's action procedure:

```
Proc ShowDialog()
   PauseUpdate; Silent 1         // building window...
   NewPanel/N=Dialog/W=(225,105,525,305) as "Dialog"
   Button done,pos={119,150},size={50,20},title="Done"
   Button done,proc=DialogDoneButtonProc
   TitleBox warning,pos={131,83},size={20,20},title=""
   TitleBox warning,anchor=MC,fColor=(65535,16385,16385)
   SetWindow Dialog hook(dlog)=DialogHook, hookevents=2
EndMacro

Function DialogHook(s)
   STRUCT WMWinHookStruct &s
   Variable statusCode= 0
      strswitch( s.eventName )
      case "killVote":
         TitleBox warning win=$s.winName, title="Press the Done button!"
         Beep
         statusCode=2                  // prevent panel from being killed.
         break
      case "mousemoved":               // to reset the warning
         TitleBox warning win=$s.winName, title=""
         break
   endswitch
   return statusCode
End
```

```
Function DialogDoneButtonProc(ba) : ButtonControl
   STRUCT WMButtonAction &ba
   switch( ba.eventCode )
      case 2:             // mouse up
         // turn off the named window hook
         SetWindow $ba.win hook(dlog)=$""
         // kill the window AFTER this routine returns
         Execute/P/Q/Z "KillWindow " + ba.win
         break
   endswitch
   return 0
End
```

Window Hook Deactivate and Kill Events

The actions caused by these events (eventCode 2, 14, 15, 16 and 17) potentially affect multiple subwindows.

If you kill a subwindow, the root window's hook functions receives a subwindowKill event for that subwindow and any child subwindows.

If you kill a root window, the root window's hook function(s) receives a subwindowKill event for each child subwindow, and then the root window's hook function(s) receive a kill event.

Likewise, hiding and showing windows can result in subwindows being hidden or shown. In each case, the window hook function receives a hide or show event for each affected window or subwindow.

The winName member of WMWinHookStruct will be set to the full subwindow path of the subwindow that is affected.

Events for an exterior subwindow are a special case. See **Hook Functions for Exterior Subwindows** on page IV-305.

The hook functions attached to an exterior subwindow will receive a subwindowKill event if the exterior subwindow is killed as a result of killing the parent window. But it will receive a regular kill event if it is killed directly. Normal subwindows always receive only subwindowKill events.

The kill-related events are sent in this order when a window or subwindow is killed:

1. A killVote event is sent to the root window's hook function(s). If any hook function returns 2, no further events are generated and the window is not killed.
2. If the window is not a subwindow and wasn't created with /K=1, /K=2 or /K=3, the standard window close dialog appears. If the close is cancelled, the window is not killed, the window will receive an activate event when the dialog is dismissed, and no further events are generated. Otherwise, proceed to step 3.
3. If the window being killed has subwindows, starting from the bottom-most subwindow and working back toward the window being killed:

3a. If the subwindow is a panel, action procedures for controls contained in the subwindow are called with event -1, “control being killed”.

3b. The root window's hook function(s) receive a subwindowKill event for the subwindow. If any hook function returns 1, no further subwindow hook events or control being killed events are sent, but the window killing process continues.

   Steps 3a and 3b are repeated for each subwindow until the window or subwindow being killed is reached.

4. If the killed window is a root window, a kill event is sent to the root window's hook function(s). If any hook function returns 2, no further events are generated and the window is not killed. This method of preventing a window from closing is to be avoided: use the killVote event or the window-equivalent of `NewPanel/K=2`.

   Prior to Igor 7, you could return 2 when the window hook received a kill event to prevent the killing of the window. This is no longer supported. Use the killVote event instead.

There are several ways to prevent a window being killed. You might want to do this in order to enforce use of a Done or Do It button, or to prevent killing a control panel while some hardware action is taking place.

The best method is to use /K=2 when creating the window (see **Display** or **NewPanel**). Then the only way to kill the window is via the DoWindow/K command, or KillWindow command. In general, you would provide a button that kills the window after checking for any conditions that would prevent it.

The KillVote event is more flexible but harder to use. It gives your code a chance to decide whether or not killing is allowed. This means the user can close and kill the window with the window close box when it is allowed.

Returning 2 for the window kill event is not recommended. If you have old code that uses this method, we strongly recommend changing it to return 2 for the killVote event. New code should never return 2 for the kill event.

As of Igor Pro 7, returning 2 for the window kill event does not prevent the window from being killed. If you have old code that uses this technique, change it to return 2 for the killVote event instead.

Window Hook Show and Hide Events

Igor sends the show event to your hook function when the affected window is about to be shown but is still hidden. Likewise, Igor sends the hide event when the window is about to be hidden but is still visible. Other events, notably resize or move events, may be triggered by showing or hiding a window and may be sent before the change in visibility actually occurs. Here is an example that illustrates this issue:

```
Function MyHookFunction(s)
   STRUCT WMWinHookStruct &s

   strswitch(s.eventName)
      case "resize":
         GetWindow $(s.winName) hide
         if (V_value)
            Print "Resized while hidden"
         else
            Print "Resized while visible"
         endif
         break

      case "moved":
         GetWindow $(s.winName) hide
         if (V_value)
            Print "Moved while hidden"
         else
            Print "Moved while visible"
         endif
         break

      case "hide":
         print "Hide event"
         break

      case "show":
         print "Show event"
         break
   endswitch

   return 0    // Don't interfere with Igor's handling of events
End

Function MakePanelWithHook()
   NewPanel/N=MyPanel/HIDE=1
   SetWindow MyPanel, hook(myHook)=MyHookFunction
End
```

If you run the MakePanelWithHook function on the command line, you see nothing because the panel is hidden. Now select Windows→Other Windows→MyPanel. The following is printed in the history:

```
Show event
Moved while hidden
Resized while hidden
```

The hook function received the Move and Resize events *after* the Show event, but before the window actually became visible.

Hook Functions for Exterior Subwindows

A regular subwindow lives inside a host window and receives events through a hook function attached to its host window.

An exterior subwindow is different because, although it is a subwindow (it is controlled by a host window), unlike a regular subwindow, it has its own actual window and therefore you can attach a hook function directly to it. A hook function attached to the root window does not receive events for an exterior subwindow. To handle events for an exterior subwindow, you must attach a hook function to the exterior subwindow itself. For example:

```
// Make a panel
NewPanel/N=RootPanel

// Make an exterior subwindow attached to RootPanel
NewPanel/HOST=RootPanel/EXT=0

// Attach a hook function to the exterior subwindow
SetWindow RootPanel#P0 hook(myhook)=MyHookFunction
```

Unnamed Window Hook Functions

Unnamed window hook functions are supported for backward compatibility only. New code should use named window hook functions. See **Named Window Hook Functions** on page IV-295.

Each window can have one unnamed hook function. You designate a function as the unnamed window hook function using the **SetWindow** operation with the hook keyword.

The unnamed hook function is called when various window events take place. The reason for the hook function call is stored as an event code in the hook function’s infoStr parameter.

The hook function is not called during experiment creation or load time so as to prevent the hook function from failing because the experiment is not fully recreated.

The hook function has the following syntax:

```
Function procName(infoStr)
   String infoStr
   String event= StringByKey("EVENT",infoStr)
   …
   return statusCode    // 0 if nothing done, else 1
End
```

`infoStr` is a string containing a semicolon-separated list of ***key:value*** pairs:

Key	Value
EVENT	*eventKey* See list of *eventKey* values below.
HCSPEC	Absolute path of the window or subwindow. See **Subwindow Command Concepts** on page III-92.
`MODIFIERS`	Bit flags as follows: Bit 0: Set if mouse button is down. Bit 1: Set if Shift is down. Bit 2: Set if Option (*Macintosh*) or Alt (*Windows*) is down. Bit 3: Set if Command (*Macintosh*) or Ctrl (*Windows*) is down. Bit 4: Contextual menu click: right-click or Control-click (*Macintosh*), or right-click (*Windows*). See **Setting Bit Parameters** on page IV-12 for details about bit settings.
`OLDWINDOW`	Previous name of the window or subwindow (for renamed event). Not the old absolute path *hcSpec*, just the name. WINDOW and HCSPEC contain the new name and new *hcSpec*.
`WINDOW`	Name of the window.

The value accompanying the EVENT keyword is one of the following:

eventKey	Meaning
`activate`	Window has just been activated.
`copy`	Copy menu item has been selected.
`cursormoved`	A graph cursor was moved. This event is sent only if bit 2 of the **SetWindow** operation hookevents flag is set.
`deactivate`	Window has just been deactivated.
`enablemenu`	Menus are being built and enabled.
`hide`	Window or subwindows about to be hidden.
`hideInfo`	The window info panel or window has just been hidden by **HideInfo**.
`hideTools`	The window tool palette or window has just been hidden by **HideTools**.
`kill`	Window is being killed. As of Igor Pro version 7, returning 2 as the hook function result no longer prevents Igor from killing the window. Use the killVote event instead.
`killVote`	Window is about to be killed. Return 2 to prevent that, otherwise return 0. See **Window Hook Deactivate and Kill Events** on page IV-303.
`menu`	A built-in menu item has been selected.
`modified`	A modification to the window has been made. See **Modified Events** on page IV-299.
mousedown	Mouse button was clicked. This event is sent only if bit 0 of the **SetWindow** operation hookevents flag is set.

eventKey	**Meaning**
mousemoved	The mouse moved. This event is sent only if bit 1 of the **SetWindow** operation hookevents flag is set.
mouseup	Mouse button was released. This event is sent only if bit 0 of the **SetWindow** operation hookevents flag is set.
moved	Window has just been moved.
renamed	Window has just been renamed. The previous name is available under the OLDWINDOW key.
resize	Window has just been resized.
show	Window or subwindow is about to be unhidden.
showInfo	The window info panel or window has just been shown by **ShowInfo**.
showTools	The window tool palette or window has just been shown by **ShowTools**.
subwindowKill	One of the window's subwindows is about to be killed.

If mouse events are enabled then the following key:value pairs will also be present in infoStr:

Key	**Value**
MOUSEX	X coordinate in pixels of the mouse.
MOUSEY	Y coordinate in pixels of the mouse.
TICKS	Time event happened.

Note that a mouseup event may or may not correspond to a previous mousedown. If the user clicks in the window, drags out and releases the button then the mouseup event will be missing. If the user clicks in another window, drags into this one and then releases then a mouseup will be sent that had no previous mousedown.

In the case of mousedown or mousemoved messages, a nonzero return value will skip normal processing of the message. This is most useful with mousedown.

The cursormoved event is not reported if Option (*Macintosh*) or Alt (*Windows*) is held down.

If the cursormoved event is enabled then the following key:value pairs will also be present in infoStr:

Key	**Value**
CURSOR	Name of the cursor that moved (A through J).
TNAME	Name of the trace the cursor is attached to (invalid if ISFREE=1).
ISFREE	1 if the cursor is "free" (not attached to a trace), 0 if it is attached to a trace or image.
POINT	Point number of the trace if not a free cursor. If the cursor is attached to an image, value is the row number of the image. If a free cursor, value is the fraction of the plot width, 0 being the left edge of the plot area, and 1 being the right edge.
YPOINT	Column number if the cursor is attached to an image, NaN if attached to a trace. If a free cursor, value is the fraction of the plot height, 0 being the top edge, and 1 being the bottom edge.

When the a menu event is reported then the following key:value pairs will also be present in infoStr:

Key	Value
MENUNAME	Name of menu (in English) as used by **SetIgorMenuMode**.
MENUITEM	Text of menu item as used by **SetIgorMenuMode**.

The enablemenu event does not pass MENUNAME or MENUITEM.

The menu and enablemenu messages are not sent when drawing tools are in use in a graph or layout or when waves are being edited in a graph.

Returning a value of 0 for the enablemenu message is recommended, though the return value is (currently) ignored.

You can use the **SetIgorMenuMode** operation to alter the enable state of Igor's built-in menus in a way you find appropriate for the window. If you do this, usually you will also handle the menu message and perform your idea of an appropriate action.

Note: Dynamic user-defined menus (see **Dynamic Menu Items** on page IV-129) are built and enabled by using string functions in the menu definitions.

Returning a value of 0 for any menu message allows Igor to perform the normal action. Returning any other value (1 is commonly used) tells Igor to skip performing the normal action.

See the user function description with **IgorMenuHook** on page IV-291 for details on the sequence of menu building, enabling, and handling.

Custom Marker Hook Functions

You can define custom marker shapes for use with graph traces. To do this, you must define a custom marker hook function, activate it by calling SetWindow with the markerHook keyword, and set a trace to use it via the ModifyGraph operation marker keyword.

A custom marker hook function takes one parameter - a WMMarkerHookStruct structure. This structure provides your function with information you need to draw a marker.

The function prototype used with a custom marker hook has the format:

```
Function MyMarkerHook(s)
   STRUCT WMMarkerHookStruct &s
   <code to draw marker>
   ...
   return statusCode       // 0 if nothing done, else 1
End
```

Your function can use the DrawXXX operations to draw the marker. The function is called each time the marker is drawn and should not do anything other than drawing the marker. The function should return 1 if it handled the marker or 0 if not.

Because the user-defined function runs during a drawing operation that cannot be interrupted without crashing Igor, the debugger cannot be invoked while it is running. Consequently breakpoints set in the function are ignored. Use **Debugging With Print Statements** on page IV-212 instead.

The marker number range, which you specify via the SetWindow markerHook call, can be any positive integers less than 1000 and can overlap built-in marker numbers.

WMMarkerHookStruct

The WMMarkerHookStruct structure has the following members:

WMMarkerHookStruct Structure Members

Member	Description
`Int32 usage`	0= normal draw, 1= legend draw (others reserved).
`Int32 marker`	Marker number minus start (i.e., starts from zero).
`float x,y`	Location of desired center of marker
`float size`	Half width/height of marker
`Int32 opaque`	1 if marker should be opaque
`float penThick`	Stroke width
`STRUCT RGBColor mrkRGB`	Fill color
`STRUCT RGBColor eraseRGB`	Background color
`STRUCT RGBColor penRG`	Stroke color
`WAVE ywave`	Trace's y wave
`double ywIndex`	Point number; ywave[wyIndex] is the y value where the marker is being drawn.
`char winName[MAX_HostChildSpec+1]`	Full path to window or subwindow
`char traceName[MAX_OBJ_INST+1]`	Full name of trace or "" if no trace

When your marker function is called, the pen thickness and colors of the drawing environment of the target window are already set consistent with the penThick, mrkRGB, eraseRGB and penRGB members.

The winName and traceName members were added in Igor Pro 9.00 to provide access to trace userData. See **Trace User Data** on page IV-89.

Marker Hook Example

Here is an example that draws audiology symbols:

```
Function AudiologyMarkerProc(s)
   STRUCT WMMarkerHookStruct &s

   if( s.marker > 3 )
      return 0
   endif

   Variable size= s.size - s.penThick/2

   if( s.opaque )
      SetDrawEnv linethick=0,fillpat=-1
      DrawRect s.x-size,s.y-size,s.x+size,s.y+size
      SetDrawEnv linethick=s.penThick
   endif
   SetDrawEnv fillpat= 0      // polys are not filled

   if( s.marker == 0 )        // 90 deg U open to the right
      DrawPoly s.x+size,s.y-size,1,1,{size,-size,-size,-size,-size,size,size,size}
   elseif( s.marker == 1 )    // 90 deg U open to the left
      DrawPoly s.x-size,s.y-size,1,1,{-size,-size,size,-size,size,size,-size,size}
   elseif( s.marker == 2 )    // Cap Gamma
      DrawPoly s.x+size,s.y-size,1,1,{size,-size,-size,-size,-size,size}
   elseif( s.marker == 3 )    // Cap Gamma reversed
      DrawPoly s.x-size,s.y-size,1,1,{-size,-size,size,-size,size,size}
   endif
   return 1
End

Window Graph1() : Graph
   PauseUpdate; Silent 1      // building window...
   Make/O/N=10 testw=sin(x)
   Display /W=(35,44,430,252) testw,testw,testw,testw
   ModifyGraph offset(testw#1)={0,-0.2},offset(testw#2)={0,-0.4},
               offset(testw#3)={0,-0.6}
   ModifyGraph mode=3,marker(testw)=100,marker(testw#1)=101,marker(testw#2)=102,
               marker(testw#3)=103
   SetWindow kwTopWin,markerHook={AudiologyMarkerProc,100,103}
EndMacro
```

See also the Custom Markers Demo experiment - in Igor choose File→Example Experiments→Feature Demos 2→Custom Markers Demo.

Tooltip Hook Functions

Igor displays tooltips for traces and images in graphs if you have selected Graph→Show Trace Info Tags and for table data cells if you have selected Table→Show Column Info Tags. You can also set help text for user-defined controls. In Igor Pro 9.00 or later, you can customize those tooltips by creating a tooltip hook function and activating it for a graph or control panel using **SetWindow**.

A tooltip hook function is similar to a window hook function (see **Named Window Hook Functions** on page IV-295). Igor calls the tooltip hook function for a graph when the mouse hovers over a trace or image, and for a table when the mouse hovers over a cell associated with a wave. The tooltip hook function is also called for a graph or control panel when the mouse hovers over a control.

Unlike a window hook function, you can associate a tooltip hook function with either a top-level graph or control panel window or with a control panel subwindow.

A tooltip hook function takes one parameter - a **WMTooltipHookStruct** structure. The structure contains fields describing the trace, image, wave or control for which tooltip help is being requested and fields that allow you to return information to Igor.

When Igor calls a tooltip hook function, the tooltip field or the structure is preset to the text that Igor has composed for the tooltip. The function can alter it, add to it, or replace it completely. If you want to specify the displayed tooltip, set the tooltip field to your desired text and return 1 from the hook function. Otherwise return 0. Other return values are reserved for future use.

Here is a tooltip hook function that generates tool tips for traces in graphs:

```
Function MyGraphTraceTooltipHook(s)
   STRUCT WMTooltipHookStruct &s

   Variable hookResult = 0    // 0 tells Igor to use the standard tooltip

   // traceName is set only for graphs and only if the mouse hovered near a trace
   if (strlen(s.traceName) > 0)
      hookResult = 1          // 1 tells Igor to use our custom tooltip
      WAVE w = s.yWave        // The trace's Y wave
      s.tooltip = GetWavesDataFolder(w, 2)    // Replace Igor's tooltip
      if (WaveDims(w) > 0)
         s.tooltip += "[" + num2str(s.row) + "]"
         if (WaveDims(w) > 1)
            s.tooltip += "[" + num2str(s.column) + "]"
         endif
      endif
   endif

   return hookResult
End
```

Use the SetWindow operation to establish the tooltip hook function for a given window, like this:

```
SetWindow Graph0, tooltipHook(MyTooltipHook) = MyGraphTraceTooltipHook
```

You can clear the tooltip function like this:

```
SetWindow Graph0, tooltipHook(MyTooltipHook) = $""
```

Because the hook function runs during an operation that cannot be interrupted without crashing Igor, the debugger cannot be invoked while it is running. Consequently breakpoints set in the function are ignored. Use **Debugging With Print Statements** on page IV-212 instead.

Tooltip Tracking Rectangle

When your function is called, the WMTooltipHookStruct trackRect field is set to a tracking rectangle in local window coordinates. When the tooltip is displayed, the tooltip system tracks the mouse. If the mouse leaves the tracking rectangle, the tooltip is hidden. If the the mouse enters another area appropriate for display of a tooltip, the hook function is called again.

In a graph, the tracking rectangle is a 20x20 point rectangle around the mouse location. In a table, the tracking rectangle corresponds to the bounds of the table cell under the mouse. If the mouse hovers over a control, the tracking rectangle is the bounding box of the control.

The trackRect field is both an input and an output. You can modify it to change when the tooltip disappears, but this is rarely necessary. You might want to do it if, for instance, you want to define help text for different parts of a control.

HTML Tags in Tooltips

You can control formatting of the tooltip by including certain HTML tags in the text. The subset of tags supported is described here: http://doc.qt.io/qt-5/richtext-html-subset.html

To use HTML tags, start your text with "<html>", end it with "</html>", and set the structure field isHTML to 1.

For example, this code:

```
s.tooltip = "<html><font size=\"+2\">Large</font> and <b>bold</b> text</html>"
s.isHTML=1
```

results in this tooltip: This tooltip has large and bold text

The Tooltip Row, Column, Layer and Chunk Fields

These fields specify the wave element of the wave associated with the graph trace, graph image, or table cell under the mouse. A value of -1 indicates that the dimension is not used.

Tooltip Display Duration

By default, Igor displays a tooltip for 10 seconds or longer for long messages. After that time, the tooltip disappears.

You can adjust the time by setting the WMTooltipHookStruct duration_ms field. When your function is called, that field is set to -1, which selects the default duration. To get an effectively permanent tooltip, set duration_ms to a large number in units of milliseconds. For example, setting duration_ms to 600000 causes the tooltip to be displayed for ten minutes.

Regardless of the duration, Igor hides the tooltip if you click the mouse or move it out of the tracking rectangle.

Data Acquisition

Igor Pro provides a number of facilities to allow working with live data:

- Live mode traces in graphs
- FIFOs and Charts
- Background task
- External operations and external functions
- Controls and control panels
- User-defined functions

Live mode traces in graphs are useful when you acquiring complete waveforms in a single short operation and you want to update a graph many times per second to create an oscilloscope type display. See **Live Graphs and Oscilloscope Displays** on page II-347 for details.

FIFOs and Charts are used when you have a continuous stream of data that you want to capture and, perhaps, monitor. See **FIFOs and Charts** on page IV-313 details.

You can set up a background task that periodically performs data acquisition while allowing you to continue to work with Igor in the foreground. The background operations are *not* done using interrupts and therefore are easily disrupted by foreground operations. Background tasks are useful only for relatively infrequent tasks that can be quickly accomplished and do not cause a cascade of graph updates or other things that take a long time. See **Background Tasks** on page IV-319 for details.

You can create an instrument-like front panel for your data acquisition setup using user-defined controls in a panel window. Refer to Chapter III-14, **Controls and Control Panels**, for details. There are many example experiments that can be found in the Examples folder.

Igor Pro comes with an XOP named VDT2 for communicating with instruments via serial port (RS232), another XOP named NIGPIB2 for communicating via General Purpose Interface Bus (GPIB), and another XOP named VISA for communicating with VISA-compatible instruments. See the Igor Pro Folder:More Extensions:Data Acquisition folder.

Sound I/O can be done using the built-in **SoundInRecord** and **PlaySound** operations.

The **NewCamera**, **GetCamera** and **ModifyCamera** operations support frame grabbing.

WaveMetrics produces the NIDAQ Tools software package for doing data acquisition using National Instruments cards. NIDAQ Tools is built on top of Igor using all of the techniques mentioned in this section. Information about NIDAQ Tools is available via the WaveMetrics Web site <http://www.wavemetrics.com/Products/NIDAQTools/nidaqtools.htm>.

Third parties have created data acquisition packages that use other hardware. Information about these is also available at <http://www.wavemetrics.com/Products/thirdparty/thirdparty.htm>.

If an XOP package is not available for your hardware you can write your own. For this, you will need to purchase the XOP Toolkit product from WaveMetrics. See **Creating Igor Extensions** on page IV-208 for details.

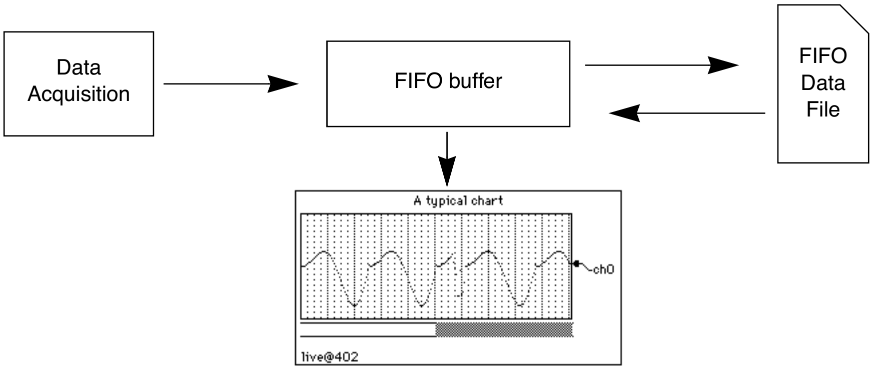
FIFOs and Charts

This section will be of interest principally to programmers writing data acquisition packages.

Most people who use FIFOs and chart recorder controls will do so via packages provided by expert Igor programmers. For information on using, as opposed to programming, chart controls, see **Using Chart Recorder Controls** on page IV-317.

FIFO Overview

A FIFO is an invisible data objects that can act as a First-In-First-Out buffer between a data source and a disk file. Data is placed in a FIFO either via the AddFIFOData operation or via an XOP package designed to interface to a particular piece of hardware. Chart recorder controls provide a graphical view of a portion of the data in a FIFO. When data acquisition is complete a FIFO can operate as a bidirectional buffer to a disk file. This allows the user to review the contents of a file by scrolling the chart "paper" back and forth. FIFOs can be used without a chart but charts have no use without a FIFO to monitor.

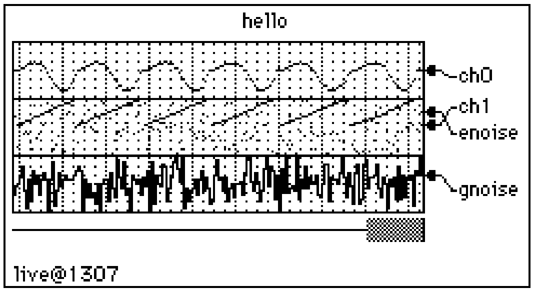


A FIFO can have an arbitrary number of channels each with its own number type, scaling, and units. All channels of a given FIFO share a common "timebase".

Chart Recorder Overview

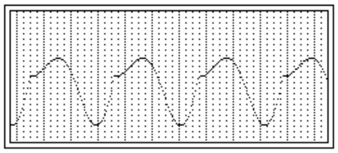
Chart recorder controls can be used to emulate a mechanical chart recorder that writes on paper with moving pens as the paper scrolls by under the pens. Charts can be used to monitor data acquisition processes or to examine a long data record. Although programming a chart is quite involved, using a chart is very easy.

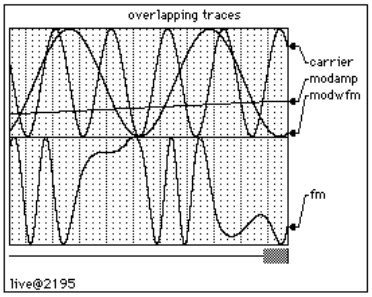
Here is a typical chart recorder control:



The First-In-First-Out (FIFO) buffer is an invisible Igor component that buffers the data coming from data acquisition hardware and software and also writes the data to a file. The data that is streaming through the FIFO can be observed using a chart recorder control. When data acquisition is finished the process can be reversed with data coming back out of the file and into the FIFO where it can be reviewed using the chart. The FIFO file is optional but if missing then all data pushed out the end of the FIFO is lost.

Chart recorder controls can take on quite a number of forms from the simple to the sophisticated:





A given chart recorder control can monitor an arbitrary selection of channels from a single FIFO. Each trace can have its own display gain, color and line style and can either have its own area on the "paper" or can share an area with one or more other traces. There can be multiple chart recorder controls active an one time in one or more panel or graph windows.

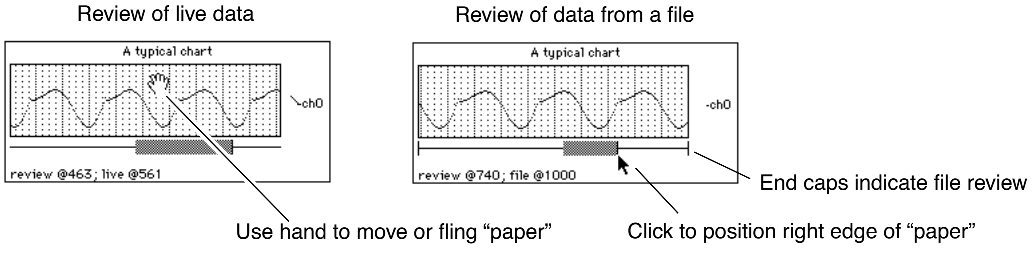
Chart recorders can display an image strip when hooked up to a FIFO channel defined using the optional vectPnts parameter to NewFIFOChan. An example experiment, Image Strip FIFO Demo, is provided to illustrate how to use this feature.

Chart recorders can operate in two modes — live and review. When a chart is in live mode and data acquisition is in progress, the chart "paper" scrolls by from right to left under the influence of the acquisition process. When in review mode, you are in control of the chart. When you position the mouse over the chart area you will see that the cursor turns into a hand. You can move the chart paper right or left by dragging with the hand. If you give the paper a push it will continue scrolling until it hits the end.

You can place the chart in review mode even as data acquisition is in progress by clicking in the paper with the hand cursor. To go back to live mode, give the paper a hard push to the left. When the paper hits the end then the chart will go to into live mode. You can also go back to live mode by clicking anywhere in the margins of the chart.

Depending on the exact details of the data acquisition hardware and software you may run the risk of corrupting the data if you use review mode while acquisition is in progress. The person that created the hardware and software system you are using should have provided guidelines for the use of review mode during acquisition. In general, if the acquisition process is paced by hardware then it should be OK to use review mode.

In the chart recorder graphics above, you may have noticed the line directly under the scrolling paper area. This line represents the current extent of data while the gray bar represents the data that is being shown in the chart. The right edge of the gray bar represents the right edge of the section of data being shown in the chart window. The above example is shown in live mode. Here are two examples shown in review mode:



End caps indicate file review

Use hand to move or fling "paper"

Click to position right edge of "paper"

While data acquisition is in progress, the horizontal line represents the extent of the data in the FIFO's memory. After acquisition is over then the line includes all of the data in the FIFO's output file, if any.

If you are in review mode while data acquisition is taking place, you will notice that the gray bar indicates the view area is moving even though the paper appears to be motionless. This is because the FIFO is moving out from under the chart. Eventually it will reach a position where the chart display can not be valid since the data it wants to display has been flushed off the end of the FIFO. When this happens the view area will go blank. Because it is very time-consuming for Igor to try to keep the chart updated in this situation your data acquisition rate may suffer.

Chart recorder controls sometimes try to auto-configure themselves to match their FIFO. Generally this action is exactly what you want and is unobtrusive. Here are the rules that charts use:

When the FIFO becomes invalid or if it ceases to exist then the chart marks itself as being in auto-configure mode. If the FIFO then becomes valid the chart will read the FIFO information and configure itself to monitor all channels. It tries to set the ppStrip parameter to a value appropriate for the deltaT value of the FIFO. It does so by assuming a desirable update rate of around 10 strips per second. Thus, for example, if deltaT was 1 millisecond then ppStrip would be set to 100. The moral is: deltaT had better be valid or weird values of ppStrip may be created.

Any chart recorder channel configuration commands executed after the FIFO becomes invalid but before the FIFO becomes valid again will prevent auto-configuration from taking place.

Programming with FIFOs

You can create a FIFO by using the NewFIFO operation. When you are done using a FIFO you use the KillFIFO operation. A freshly created FIFO is not useful until either channels are created with the NewFIFOChan operation or until the FIFO is attached to a disk file for review using a variant of the CtrlFIFO operation.

You can obtain information about a FIFO using the FIFOStatus operation and you can extract data from a FIFO using the FIFO2Wave operation. Once a FIFO is set up and ready to accept data, you can insert data using the AddFIFOData operation. Alternately, you can insert data using an XOP package.

Once data is stored in a file you can review the data using a FIFO or extract data using user-defined functions. See the example experiment, “FIFO File Parse”, for sample utility routines.

Here are the operations and functions used in FIFO programming:

NewFIFO **KillFIFO**

NewFIFOChan **CtrlFIFO**

FIFO2Wave **AddFIFOData**

AddFIFOVectData **FIFOStatus**

As with background tasks, FIFOs are considered transient objects — they are not saved and restored as part of an experiment.

A FIFO does not need to be attached to a file to be useful. Note, however, that the oldest data is lost when a FIFO overflows.

A FIFO set up to acquire data does not become valid until the start command is issued. Chart controls will report invalid FIFOs on their status line. FIFO2Wave will give an error if it is invoked on an invalid FIFO. A stopped FIFO remains valid until the first command is issued that could potentially change the FIFO’s setup.

Data in a running FIFO is written to disk when Igor notices that the FIFO is half full or when the AddFIFOData command is issued and the FIFO is full. The amount of time it takes to write data to disk can be quite considerable and at the same time unpredictable. If the computer disk cache size is large then writes to disk will be less frequent but when they do occur they will take a long time. This will matter to you most if you are attempting to take data rapidly using software, perhaps using an Igor background task.

If you are taking data via interrupt transfer to an intermediate buffer of adequate size or if your hardware has an adequate internal buffer then the disk write latency may not be a concern. If dead time due to disk writes is a concern then you may want to decrease the size of the disk cache and you may want to run with a relatively small FIFO. Note that if you change the size of the disk cache you may have to reboot for the change to take effect.

When the stop command is given to a running FIFO then it goes into review mode and remains valid. If the FIFO is attached to a file then the entire contents of the file can be reviewed or be transferred to a wave using the FIFO2Wave command.

The act of attaching a FIFO to an existing file for review using the rfile keyword of the CtrlFIFO command reads in the file contents and sets itself up for review. You should not use the NewFIFOChan command or any of the other CtrlFIFO keywords except size. Here is all that is required to review a preexisting file:

```
Variable refnum
Open/R/P=mypath refnum as "my file"
NewFIFO dave
CtrlFIFO dave,rfile=refnum
```

If any chart controls have been set up to monitor FIFO dave then they will automatically configure themselves to display all the channels of dave using default parameters.

The connection between FIFOs and chart controls relies on Igor’s dependency manager. The dependency manager does not automatically run during function execution — you have to explicitly call it by executing the DoUpdate command.

The dependency manager sends messages to a chart control when:

- A FIFO is created
- A FIFO is killed
- A FIFO becomes valid (start command)
- Data is added to a FIFO

In particular, if inside a user function, you kill a FIFO and then create it again you should call DoUpdate after the kill so that the chart control notices the kill and can get ready for the creation.

FIFO File Format

This section is provided for programmers who want to create FIFO files as input for Igor with their own programs or to analyze data stored in Igor-generated FIFO files. Such applications require familiarity with C programming.

The FIFO file uses C structures named FIFOFileHeader, ChartChunkInfo, ChartChanInfo, and FIFOSplitFileHeader. These are defined in the file NamedFIFO.h which is shipped in "Igor Pro Folder\Miscellaneous\More Documentation". NamedFIFO.h is the primary documentation for the FIFO file format.

The FIFOFileHeader, ChartChunkInfo, and FIFOSplitFileHeader structures contain version fields. These structures may evolve in the future, so your code should check them and fail gracefully if you encounter an unexpected value.

Igor supports integrated FIFO files, in which the header and data are in the same file, and split FIFO files, in which the header and data are in separate files.

An integrated FIFO file consists of a FIFOFileHeader structure followed by a ChartChunkInfo structure and finally by chunks of data until the end of the file. A user-defined function that can parse an integrated FIFO file can be found in the "FIFO File Parse" example experiment.

The split format uses the FIFOSplitFileHeader structure to allow the raw data to reside in its own file rather than having to be in the same file as the header. This facilitates the use of Igor to review large binary files generated by third-party programs.

See the "Wave Review Chart Demo" example experiment for sample code for both the integrated and split FIFO file formats.

Another way to use a FIFO and chart control to review a raw binary file is to use the rdfile keyword with the **CtrlFIFO** command.

FIFO and Chart Demos

```
Igor Pro Folder:Examples:Feature Demos:FIFO Chart Demo FM.pxp
Igor Pro Folder:Examples:Feature Demos:FIFO Chart Overhead.pxp
Igor Pro Folder:Examples:Feature Demos:Wave Review Chart Demo.pxp
Igor Pro Folder:Examples:Imaging:Image Strip FIFO Demo.pxp
```

Using Chart Recorder Controls

The information provided here pertains to using rather than programming a chart recorder control. For information on programming chart controls, see **FIFOs and Charts** on page IV-313.

An Igor chart recorder control works in conjunction with a FIFO to display data as it is acquired or to review data that has previously been acquired.

Chart Reorder Control Basics

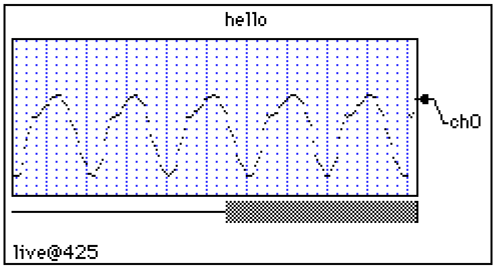
An Igor chart recorder control is neither an analytical tool nor a presentation quality graphic. It is meant only for real time monitoring of incoming data or to review data from a FIFO file. When you want an analytical or presentation quality graph you must transfer the data to a wave and then use a conventional Igor graph.

An Igor chart recorder control emulates a mechanical chart recorder that writes on paper with moving pens as the paper scrolls by under the pens. It differs from a real chart recorder in that the paper of the latter moves at a constant velocity whereas the “paper” of an Igor chart moves only when data becomes available in the FIFO it is monitoring. If data is placed in the FIFO at a constant rate then the “paper” will scroll by at a constant rate. However, since there can be no guarantee that the data is coming in at a constant rate, we refer to the horizontal axis not in terms of time but rather in terms of data sample number.

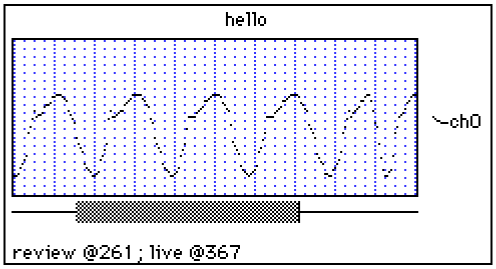
A given chart recorder control can monitor an arbitrary selection of channels from a single FIFO. Each chart trace can have its own display gain, color and line style and can either have its own area on the “paper” or can share an area with one or more other traces. There can be multiple charts active an one time in one or more control panel or graph windows.

Operating a Chart Recorder

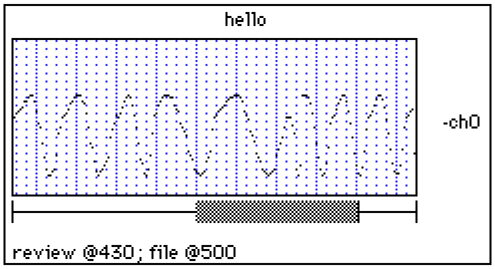
Here is a typical chart recorder while taking data:



And here is the same chart recorder while reviewing data even though data acquisition is still taking place:



And here we are while reviewing data from the file after data acquisition is complete:



Notice the positioning strip just under the chart and above the status line. It consists of a horizontal line and a horizontal, gray bar. The line, called the positioning line, represents the extent of available data. The bar, called the positioning bar, represents the currently displayed region of this data.

While data acquisition is in progress, the available data is the data in the FIFO's memory. After the acquisition is over then the available data includes all of the data in the FIFO's output file, if any. The vertical bars at the ends of the positioning line indicate we are reviewing from a file.

You can instantly jump to any portion of the data by clicking on the positioning line. The spot that you click on indicates the part of the available data that you want to view. After clicking you can drag the "paper" region around.

The chart recorder will be in one of two modes: live mode or review mode. While data acquisition is under way, the chart recorder will display incoming data if it is in live mode. If it is in review mode, you can review previously acquired data.

Clicking on the positioning line or in the positioning bar puts the chart recorder into review mode even if data acquisition is taking place. To exit review mode and go into live mode, simply click anywhere in the chart recorder outside of the "paper" and the positioning strip. Of course, if you are not acquiring data you can not go into live mode.

Another way to go into review mode and navigate is to grab the "paper" with the mouse and fling it to the left or right. It will keep going until it hits the end of available data. The speed at which the "paper" moves depends on how hard you fling it.

If you are in review mode while data acquisition is taking place, you will notice that the positioning bar indicates the view area is moving even though the "paper" appears to be motionless. This is because the FIFO is moving out from under the chart. Eventually it will reach a position where the chart display can not be valid since the data it wants to display has been flushed off the end of the FIFO. When this happens the paper will go blank. Because it is very time consuming for Igor to try to keep the chart updated in this situation, your data acquisition rate may suffer. To get an idea of what kind of data rates can be sustained using an Igor background task, spend some time experimenting with the "FIFO/Chart Overhead" example experiment.

Chart Recorder Control Demos

```
Igor Pro Folder:Examples:Feature Demos:FIFO Chart Demo FM.pxp
Igor Pro Folder:Examples:Feature Demos:FIFO Chart Overhead.pxp
Igor Pro Folder:Examples:Feature Demos:Wave Review Chart Demo.pxp
Igor Pro Folder:Examples:Imaging:Image Strip FIFO Demo.pxp
```

Background Tasks

Background tasks allow procedures to run periodically "in the background" while you continue to interact normally with Igor. This is useful for data acquisition, simulations and other processes that run indefinitely, over long periods of time, or need to run at regular intervals. Using a background task allows you to continue to interact with Igor while your data acquisition or simulation runs.

Originally Igor supported just one unnamed background task controlled using the **CtrlBackground** operation (page V-118). New code should use the **CtrlNamedBackground** operation (page V-119) to create named background tasks instead, as shown in the following sections. You can run any number of named backgrounds tasks.

In addition to the documentation provided here, the Background Task Demo experiment provides sample code that is designed to be redeployed for other projects. We recommend reading this documentation first and then opening the demo by choosing File→Example Experiments→Programming→Background Task Demo.

Background Task Example #1

You create and control background tasks using the **CtrlNamedBackground** operation. The main parameters of CtrlNamedBackground are the background task name, the name of a procedure to be called periodically, and the period. Here is a simple example:

```
Function TestTask(s)     // This is the function that will be called periodically
   STRUCT WMBackgroundStruct &s

   Printf "Task %s called, ticks=%d\r", s.name, s.curRunTicks
   return 0             // Continue background task
End

Function StartTestTask()
   Variable numTicks = 2 * 60    // Run every two seconds (120 ticks)
   CtrlNamedBackground Test, period=numTicks, proc=TestTask
   CtrlNamedBackground Test, start
End

Function StopTestTask()
   CtrlNamedBackground Test, stop
End
```

You start this background task by calling `StartTestTask()` from the command line or from another procedure. StartTestTask creates a background task named Test, sets the period which is specified in units of ticks (1 tick = 1/60th of a second), and specifies the user-defined function to be called periodically (TestTask in this example).

You stop the Test background task by calling `StopTestTask().`

As shown above, the background procedure takes a **WMBackgroundStruct** parameter. In most cases you won't need to access it.

Background Task Exit Code

The background procedure (TestTask in the example above) returns an exit code to Igor. The code is one of the following values:

0: The background procedure executed normally.

1: The background procedure wants to stop the background task.

2: The background procedure encountered an error and wants to stop the background task.

Normally the background procedure should return 0 and the background task will continue to run. If you return a non-zero value, Igor stops the background task. You can tell Igor to terminate the background task by returning the value 1 from the background function.

If you forget to add a return statement to your background procedure, this acts like a non-zero return value and stops the background task.

Background Task Period

The CtrlNamedBackground operation's period keyword takes an integer parameter expressed in ticks. A tick is approximately 1/60th of a second. Thus the timing of Igor background tasks has a nominal resolution of 1/60th of a second.

You can override the specified period in the background task procedure by writing to the nextRunTicks field of the **WMBackgroundStruct** structure. This is needed only if you want your procedure to run at irregular intervals.

The actual time between calls to the background procedure is not guaranteed. Igor runs the background task from its outer loop, when Igor is doing nothing else. If you do something in Igor that takes a long time, for example performing a lengthy curve fit, running a user-defined function that takes a long time, or saving a large experiment, Igor's outer loop does not run so the background task will not run. If you do something that causes a compilation of Igor procedures to fail, the background task is not called. On Macintosh, the background task is not called while a menu is displayed or while the mouse button is pressed.

If you need your background task to continue running even if you edit other procedures in Igor, you need to make your project an independent module. See **Independent Modules** on page IV-238 for details.

If you need precise timing that can not be interrupted, things get much more complicated. You need to do your data acquisition in an Igor thread running in an independent module or in a thread created by an XOP that you write. See **ThreadSafe Functions and Multitasking** on page IV-329 for details.

The shortest supported period is one tick. The minimum actual period for the background task depends on your hardware and what your background task is doing. If you set the period too low for your background task, interacting with Igor becomes sluggish.

It is very easy to bog your computer down using background tasks. If the background task takes a long time to execute or if it triggers something that takes a long time (like a wave dependency formula or updating a complex graph) then it may appear that the system is hung. It is not, but it may take longer to respond to user actions than you are willing to wait.

Background Task Limitations

The principal limitation of Igor background tasks is that they are stopped while other operations are taking place. Thus, although you can type commands into the command line without disrupting the background task, when you press Return the task is stopped until execution of the command line is finished.

Background tasks do not run if procedures are in an uncompiled state. If you need your background task to continue running even if you edit other procedures in Igor, you need to make your project an independent module. See **Independent Modules** on page IV-238 for details.

On Macintosh, the background task does not run when the mouse button is pressed or when a menu is displayed.

Background Tasks and Errors

If a background task procedure contains a bug, it will typically generate an error each time the procedure runs. Normally an error generates an error dialog. If this happened over and over again, it would prevent you from fixing the bug.

Igor handles such repeated errors as follows: The first time an error occurs during the execution of the background task procedure, Igor displays an error dialog. On subsequent errors, Igor prints an error message in the history. After printing 10 such error messages, Igor stops printing messages. When you click a control, execute a command from the command line or execute a command through a menu item, the process starts over.

If the Igor debugger is enabled and Debug on Error is turned on, Igor will break into the debugger each time an error occurs in the background task procedure. You may have to turn Debug on Error off to give you time to stop the background task. You can do this from within the debugger by right-clicking.

Background Tasks and Dialogs

By default, a background task created by CtrlNamedBackground does not run while a dialog is displayed. You can change this behavior using the CtrlNamedBackground dialogsOK keyword.

If you allow background tasks to run while an Igor dialog is present, you should ensure that your background task does not kill anything that a dialog might depend on. It should not kill waves or variables. It should never directly modify a window (except for a status panel) and especially should never remove something from a window (such as a trace from a graph). Otherwise your background task may kill something that the dialog depends on which will cause a crash.

Background Task Tips

Background tasks should be designed to execute quickly. They do not run in separate threads threads and they hang Igor’s event processing as long as they run. For maximum responsiveness, your task procedure should take no more than a fraction of a second to run even when the period is long. If you have to perform a lengthy computation, let the user know what is going on, perhaps via a message in a status control panel.

Background tasks should never attempt to put up dialogs or directly wait for user input. If you need to get the attention of the user, you should design your system to include a status control panel with an area for messages or some other change in appearance. If you need to wait for the user, you should do so by monitoring global variables set by nonbackground code such as a button procedure in a panel.

Your task procedure should always leave the current data folder unchanged on exit.

Background Task Example #2

Here is an example that uses many of the concepts discussed above. The task prints a message in the history area at one second intervals five times, performs a “lengthy calculation”, and then waits for the user to give the go-ahead for another run.

The task does its own timing and consequently is set to run at the maximum rate (60 times per second). The task procedure, MyBGTask, tests to see if one second has elapsed since the last time it printed a message. In a real application, you might test to see if some external event has occurred.

To try the example, copy the code below to the Procedure window and execute:

```
BGDemo()
```

```
Function BGDemo()
   DoWindow/F BGDemoPanel        // bring panel to front if it exists
   if( V_Flag != 0 )
      return 0                   // panel already exists
   endif

   String dfSav= GetDataFolderDFR()         // so we can leave current DF as we found it
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S root:Packages:MyDemo  // our variables go here
   // still here if no panel, create globals if needed
   if( NumVarOrDefault("inited",0) == 0 )
      Variable/G inited= 1

      Variable/G lastRunTicks= 0 // value of ticks function last time we ran
      Variable/G runNumber= 0              // incremented each time we run
      // message displayed in panel using SetVariable...
      String/G message="Task paused. Click Start to resume."

      Variable/G running=0                 // when set, we do our thing
   endif

   SetDataFolder dfSav
   NewPanel /W=(150,50,449,163)
   DoWindow/C BGDemoPanel                  // set panel name
   Button StartButton,pos={21,12},size={50,20},proc=BGStartStopProc,title="Start"
   SetVariable msg,pos={21,43},size={300,17},title=" ",frame=0
   SetVariable msg,limits={-Inf,Inf,1},value= root:Packages:MyDemo:message
End

Function MyBGTask(s)
   STRUCT WMBackgroundStruct &s

   NVAR running= root:Packages:MyDemo:running

   if( running == 0 )
      return 0                      // not running -- wait for user
   endif

   NVAR lastRunTicks= root:Packages:MyDemo:lastRunTicks

   if( (lastRunTicks+60) >= ticks )
      return 0                      // not time yet, wait
   endif

   NVAR runNumber= root:Packages:MyDemo:runNumber

   runNumber += 1

   printf "Hello from the background, #%d\r",runNumber

   if( runNumber >= 5 )
      runNumber= 0
      running= 0                    // turn ourself off after five runs

      // run again when user says to
      Button StopButton,win=BGDemoPanel,rename=StartButton,title="Start"

      // Simulate a long calculation after a run
      String/G root:Packages:MyDemo:message="Performing long calculation. Please wait."
      ControlUpdate /W=BGDemoPanel msg
      DoUpdate /W=BGDemoPanel       // Required on Macintosh for control to be redrawn

      Variable t0= ticks
      do
         if (GetKeyState(0) & 32)
            Print "Lengthy process aborted by Escape key"
            break
         endif
      while( ticks < (t0+60*3) )    // delay for 3 seconds

      String/G root:Packages:MyDemo:message="Task paused. Click Start to resume."
   endif

   lastRunTicks= ticks

   return 0
End

Function BGStartStopProc(ctrlName) : ButtonControl
   String ctrlName

   NVAR running= root:Packages:MyDemo:running
   if( CmpStr(ctrlName,"StartButton") == 0 )
      running= 1
      Button $ctrlName,rename=StopButton,title="Stop"
      String/G root:Packages:MyDemo:message=""
      CtrlNamedBackground MyBGTask, proc=MyBGTask, period=1, start
   endif
```

```
    if( CmpStr(ctrlName,"StopButton") == 0 )
        running= 0
        Button $ctrlName,rename=StartButton,title="Start"
        CtrlNamedBackground MyBGTask, stop
        String/G root:Packages:MyDemo:message="Task paused. Press Start to resume."
    endif
End
```

Background Task Example #3

For another example including code that you can easily redeploy for your own project, open the Background Task Demo experiment by choosing File→Example Experiments→Programming→Background Task Demo.

Old Background Task Techniques

Originally Igor supported just one unnamed background task. This is still supported for backward compatibility but new code should use CtrlNamedBackground to create and control named background tasks instead.

The unnamed background task is designated using **SetBackground**, controlled using **CtrlBackground** and killed using **KillBackground**. The **BackgroundInfo** operation returns information about the unnamed background task.

The SetBackground, CtrlBackground, KillBackground and BackgroundInfo operations work only with the unnamed background task. For named background tasks, the CtrlNamedBackground operation provides all necessary functionality.

By default, a background task created by CtrlBackground does not run while a dialog is displayed. You can change this behavior using the CtrlBackground dialogsOK keyword.

Automatic Parallel Processing with TBB

TBB stands for “Threading Building Blocks”. It is an Intel technology that facilitates the use of multiple processors on a given task. The home page for TBB is:

https://www.threadingbuildingblocks.org

Starting with Igor Pro 7, some Igor operations, such as CurveFit, DSPPeriodogram, and ImageProfile, automatically use TBB. To see which operations use TBB, choose Help→Command Help, click Show All, and then check the Automatically Multithreaded Only checkbox.

You don’t need to do anything to take advantage of automatic multithreading with TBB. It happens automatically.

Running on multiple threads reduces the time required for number crunching tasks when the benefit of using multiple processors exceeds the overhead. Operations that use TBB automatically use multiple threads only when the size of the data or the complexity of the problem crosses a certain threshold. Igor is programmed to use a reasonable threshold for each supported operation. You can control the threshold using the **MultiThreadingControl** operation.

Automatic Parallel Processing with MultiThread

Intermediate-level Igor programmers can make use of multiple processors to speed up wave assignment statements in user-defined functions. To do this, simply insert the keyword MultiThread in front of a normal wave assignment. For example, in a function:

```
Make wave1
Variable a=4
MultiThread wave1= sin(x/a)
```

The expression, on the right side of the assignment statement, is compiled as threadsafe even if the host function is not.

Because of the overhead of spawning threads, you should use MultiThread only when the destination has a large number of points or the expression takes a significant amount of time to evaluate. Otherwise, you may see a performance penalty rather than an improvement.

The assignment is, by default, automatically parceled into as many threads as there are processors, each evaluating the right-hand expression for a different output point. You can specify the number of threads that you want to use using MultiThread /NT=(<number of threads>). In most cases, you should omit /NT to get the default behavior.

The MultiThread keyword causes Igor to evaluate the expression for multiple output points simultaneously. Do not make any assumptions as to the order of processing and certainly do not try to use a point from the destination wave other than the current point in the expression. For example, do not do something like this:

```
wave1 = wave1[p+1] - wave1[p-1]  // Result is indeterminate
```

Expressions like give unexpected results even in the absence of threading.

Here is a simple example to try on your own machine:

```
Function TestMultiThread(n)
   Variable n                    // Number of wave points

   Make/O/N=(n) testWave

   // To prime processor data cache so comparison will be valid
   testWave= 0

   Variable t1,t2
   Variable timerRefNum

   // First, non-threaded
   timerRefNum = StartMSTimer
   testWave= sin(x/8)
   t1= StopMSTimer(timerRefNum)

   // Now, automatically threaded
   timerRefNum = StartMSTimer
   MultiThread testWave= sin(x/8)
   t2= StopMSTimer(timerRefNum)

   Variable processors = ThreadProcessorCount
   Print "On a machine with",processors,"cores,MultiThread is", t1/t2,"faster"
End
```

Here is the output on a Mac Pro:

```
•TestMultiThread(100)
  On a machine with  8  cores, MultiThread is  0.059746  faster

•TestMultiThread(10000)
  On a machine with  8  cores, MultiThread is  3.4779  faster

•TestMultiThread(1000000)
  On a machine with  8  cores, MultiThread is  6.72999  faster

•TestMultiThread(10000000)
  On a machine with  8  cores, MultiThread is  8.11069  faster
```

The first result shows that the MultiThread keyword slowed the assignment down. This is because the assignment involved a small number of points and MultiThread has some overhead.

The remaining results illustrate that MultiThread can provide increased speed for assignments involving large waves.

In the last result, the speed improvement factor was greater than the number of processors. This is explained by the fact that, once running, a threadsafe expression has slightly less overhead than a normal expression.

If the right-hand expression involves calling user-defined functions, those functions must be threadsafe (see **ThreadSafe Functions** on page IV-106) and must also follow these rules:

1. Do not do anything to waves that are passed as parameters that might disturb memory. For example, do not change the number of points in the wave or change its data type or kill it or write to a text wave.
2. Do not write to a variable that is passed by reference.
3. Note that any waves or global variables created by the function will disappear when then wave assignment is finished.
4. Each thread has its own private data folder tree. You can not use WAVE, NVAR or SVAR to access objects in the main thread.

Failure to heed rule #1 will likely result in a crash.

Although it is legal to use the MultiThread mechanism in a threadsafe function that is already running in a preemptive thread via **ThreadStart**, it is not recommended and will likely result in a substantial loss of speed.

For an example using MultiThread, open the Mandelbrot demo experiment file by choosing “File→Example Experiments→Programming→MultiThreadMandelbrot”.

Data Folder Reference MultiThread Example

Advanced programmers can use waves containing data folder references and wave references along with MultiThread to perform multithreaded calculations more involved than evaluating an arithmetic expression. Here we use **Free Data Folders** (see page IV-96) to facilitate multithreading.

In this example, we extract each of the planes of a 3D wave, perform a filtering operation on the planes, and then finally assemble the planes into an output 3D wave. The main function, Test, executes a multithreaded assignment statement where the expression includes a call to a subroutine named Worker.

Because MultiThread is used, multiple instances of Worker execute simultaneously on different cores. Each instance runs in its own thread, working on a different plane. Each instance returns one filtered plane in a wave named M_ImagePlane in a thread-specific free data folder. The use of free data folders allows each instance of Worker to work on its own M_ImagePlane wave without creating a name conflict.

When the multithreaded assignment is finished, the main function assembles an output 3D wave.

```
// Extracts a plane from the 3D input wave, filters it, and returns the
// filtered output as M_ImagePlane in a new free data folder
ThreadSafe Function/DF Worker(w3DIn, plane)
   WAVE w3DIn
   Variable plane

   DFREF dfSav= GetDataFolderDFR()

   // Create a free data folder to hold the extracted and filtered plane
   DFREF dfFree= NewFreeDataFolder()
   SetDataFolder dfFree

   // Extract the plane from the input wave into M_ImagePlane.
   // M_ImagePlane is created in the current data folder
   // which is a free data folder.
   ImageTransform/P=(plane) getPlane, w3DIn
   Wave M_ImagePlane             // Created by ImageTransform getPlane

   // Filter the plane
   WAVE wOut= M_ImagePlane
   MatrixFilter/N=21 gauss,wOut

   SetDataFolder dfSav

   // Return a reference to the free data folder containing M_ImagePlane
   return dfFree
End

Function Demo(numPlanes)
   Variable numPlanes

   // Create a 3D wave and fill it with data
   Make/O/N=(200,200,numPlanes) src3D= (p==(2*r))*(q==(2*r))

   // Create a wave to hold data folder references returned by Worker.
   // /DF specifies the data type of the wave as "data folder reference".
   Make/DF/N=(numPlanes) dfw

   Variable timerRefNum = StartMSTimer

   MultiThread dfw= Worker(src3D,p)

   Variable elapsedTime = StopMSTimer(timerRefNum) / 1E6

   Printf "Statement took %g seconds for %d planes\r", elapsedTime, numPlanes

   // At this point, dfw holds data folder references to 50 free
   // data folders created by Worker. Each free data folder holds the
   // extracted and filtered data for one plane of the source 3D wave.

   // Create an output wave named out3D by cloning the first filtered plane
   DFREF df= dfw[0]
   Duplicate/O df:M_ImagePlane, out3D

   // Concatenate the remaining filtered planes onto out3D
   Variable i
   for(i=1; i<numPlanes; i+=1)
      df= dfw[i]     // Get a reference to the next free data folder
      Concatenate {df:M_ImagePlane}, out3D
   endfor

   // Redimension out3D to contain the correct number of planes
   Redimension/N=(-1, -1, numPlanes) out3D

   // Assign values to the remaining planes of out3D
   Variable i
   for(i=1; i<numPlanes; i+=1)
      df= dfw[i]                           // Reference to next free data folder
      WAVE thisPlaneWave = df:M_ImagePlane// Reference to wave in data folder
      out3D[][][i] = thisPlaneWave[p][q]
   endfor

   // dfw holds references to the free data folders. By killing dfw,
   // we kill the last reference to the free data folders which causes
   // them to be automatically deleted. Because there are no remaining
   // references to the various M_ImagePlane waves, they too are
   // automatically deleted.
   KillWaves dfw
End
```

To run the demo, execute:

```
Demo(50)
```

On an eight-core Mac Pro, this took 4.1 seconds without the MultiThread keyword and 0.6 seconds with the MultiThread keyword for a speedup of about 6.8 times.

Wave Reference MultiThread Example

In the preceding example, free data folders were used to hold data processed by threads. Since each free data folder held just a single wave, the example can be simplified by using free waves instead of free data folders. So here we perform the same threaded filtering of planes using free waves.

Because MultiThread is used, multiple instances of Worker execute simultaneously on different cores. Each instance runs in its own thread, working on a different plane. Each instance returns one filtered plane in a free wave named M_ImagePlane. The use of free waves allows each instance of Worker to work on its own M_ImagePlane wave without creating a name conflict.

This version of the example relies on the fact that a wave in a free data folder becomes a free wave when the free data folder is automatically deleted. See **Free Wave Lifetime** on page IV-92 for details.

```
ThreadSafe Function/WAVE Worker(w3DIn, plane)
   WAVE w3DIn
   Variable plane

   DFREF dfSav= GetDataFolderDFR()

   // Create a free data folder and set it as the current data folder
   SetDataFolder NewFreeDataFolder()

   // Extract the plane from the input wave into M_ImagePlane.
   // M_ImagePlane is created in the current data folder
   // which is a free data folder.
   ImageTransform/P=(plane) getPlane, w3DIn
   Wave M_ImagePlane             // Created by ImageTransform getPlane

   // Filter the plane
   WAVE wOut= M_ImagePlane
   MatrixFilter/N=21 gauss,wOut

   // Restore the current data folder
   SetDataFolder dfSav

   // Since the only reference to the free data folder created above
   // was the current data folder, there are now no references it.
   // Therefore, Igor has automatically deleted it.
   // Since there IS a reference to the M_ImagePlane wave in the free
   // data folder, M_ImagePlane is not deleted but becomes a free wave.

   return wOut       // Return a reference to the free M_ImagePlane wave
End

Function Demo(numPlanes)
   Variable numPlanes

   // Create a 3D wave and fill it with data
   Make/O/N=(200,200,numPlanes) srcData= (p==(2*r))*(q==(2*r))
   // Create a wave to hold data folder references returned by Worker.
   // /WAVE specifies the data type of the wave as "wave reference".
   Make/WAVE/N=(numPlanes) ww

   Variable timerRefNum = StartMSTimer

   MultiThread ww= Worker(srcData,p)

   Variable elapsedTime = StopMSTimer(timerRefNum) / 1E6

   Printf "Statement took %g seconds for %d planes\r", elapsedTime, numPlanes

   // At this point, ww holds wave references to 50 M_ImagePlane free waves
   // created by Worker. Each M_ImagePlane holds the extracted and filtered
   // data for one plane of the source 3D wave.

   // Create an output wave named out3D by cloning the first filtered plane
   WAVE w= ww[0]
   Duplicate/O w, out3D

   // Concatenate the remaining filtered planes onto out3D
   Variable i
   for(i=1;i<numPlanes;i+=1)
      WAVE w= ww[i]
      Concatenate {w}, out3D
   endfor

   // Create a 3D output wave by concatenating the filtered planes
   Concatenate/O {ww}, out3D

   // ww holds references to the free waves. By killing ww, we kill
   // the last reference to the free waves which causes them to be
   // automatically deleted.
   KillWaves ww
End
```

To run the demo, execute:

```
Demo(50)
```

Structure Array MultiThread Example

In a preceding example, free data folders were used to hold data processed by threads. A somewhat simpler approach is to use one or more structures to pass input data and to receive output data. The following example uses a single structure for both input and output. An array of these structures stored in a wave ensures that each thread works on its own data. After the calculation, the results are extracted. The net result for this simple example is nothing more than: `dataOutput = sin(p)`.

```
Structure ThreadIOData
   // Input to thread
   double x

   // Output from thread
   double out
EndStructure

Function Demo()
   if (IgorVersion() < 6.36)
      // This example crashes in Igor Pro 6.35 or before
      // because of a bug in StructGet/StructPut
      Abort "Function requires Igor Pro 6.36 or later."
   endif

   STRUCT ThreadIOData ioData

   // Prepare input
   Make/O ioDataArray      // This wave will be redimensioned by StructPut
   Variable i, imax=100
   for(i=0; i<imax; i+=1)
      ioData.x = i                        // Set input data
      StructPut ioData, ioDataArray[i]    // Pack structure into wave column
   endfor

   // Generate output
   Make/O/N=(imax) threadOutput
   MultiThread threadOutput = Worker(ioDataArray, p)

   // Extract output
   Make/O/N=(imax) outputData
   for(i=0; i<imax; i+=1)
      StructGet ioData, ioDataArray[i]
      outputData[i] = ioData.out
   endfor

   KillWaves ioDataArray, threadOutput
End

ThreadSafe Function Worker(w, point)
   WAVE w
   Variable point

   STRUCT ThreadIOData ioData
   StructGet ioData, w[point]       // Extract structure from wave column

   ioData.out = sin(ioData.x)       // Calculate of output data

   StructPut ioData, w[point]       // Pack structure into wave column

   // The return value from the thread worker function is accessible
   // via ThreadReturnValue. It is not used in this example.
   return point
End
```

To run the demo, execute:

```
Demo()
```

ThreadSafe Functions and Multitasking

Igor supports two multitasking techniques that are easy to use:

- **Automatic Parallel Processing with TBB**
- **Automatic Parallel Processing with MultiThread**

This section discusses the third technique, **ThreadSafe Functions**, which expert programmers can use to create complex, preemptive multitasking background tasks.

Preemptive multitasking uses the following functions and operations:

ThreadProcessorCount

ThreadStart

ThreadGroupCreate

ThreadGroupPutDF

ThreadGroupGetDF (deprecated)

ThreadGroupWait

ThreadGroupRelease

ThreadGroupGetDFR

ThreadReturnValue

To run a threadsafe function preemptively, you first create a thread group using **ThreadGroupCreate** and then call **ThreadStart** to start your worker function. Usually you will use the same function for each thread of a group although they can be different.

The worker function must be defined as threadsafe and must return a real or complex numeric result. The return value can be obtained after the function finishes by calling **ThreadReturnValue**.

The worker function can take variable and wave parameters. It can not take pass-by-reference parameters or data folder reference parameters.

Any waves you pass to the worker are accessible to both the main thread and to your preemptive thread. Such waves are marked as being in use by a thread and Igor will refuse to perform any operations that could change the size of the wave.

You can determine if any threads of a group are still running by calling **ThreadGroupWait**. Use zero for the "milliseconds to wait" parameter to just test if all threads are finished. Use a larger value to cause the main thread to sleep until all threads are finished. If you know the maximum time the threads should take, you can use that value and print an error message or take other action if the threads don't finish in time.

When ThreadGroupWait is called, Igor updates certain internal variables including variables that track whether a thread has finished and what result it returned. Therefore you must call ThreadGroupWait before calling ThreadReturnValue.

Once you are finished with a given thread group, call **ThreadGroupRelease**.

If your threads can run for a long time, you should detect aborts and make your threads quit. See **Aborting Threads** on page IV-337 for details.

The Igor debugger can not be used with threadsafe functions. See **Debugging ThreadSafe Code** on page IV-225 for details.

The hard part of using multithreading is devising a scheme for partitioning your data processing algorithms into threads.

Thread Data Environment

When a thread is started, Igor creates a root data folder for that thread. This root data folder and any data objects that the thread creates in it are private to the thread. This constitutes a separate data hierarchy for each thread.

Data is transferred, when you request it, from the main thread to a preemptive thread and vice-versa using input and output queues. The "currency" of these queues is the data folder, which provides considerable flexibility for passing data to threads and for retrieving results. Each thread group has an input queue to which the main thread may post data and an output queue from which the main thread may retrieve results.

The terms "input" and "output" are relative to the preemptive thread. The main thread posts a data folder to the input queue to send input to the preemptive thread. The preemptive thread retrieves the data folder from the input queue. After processing, the preemptive thread may post a data folder to the output queue. The main thread reads output from the preemptive thread by retrieving the data folder from the output queue.

Use **ThreadGroupPutDF** to post data folders and **ThreadGroupGetDFR** to retrieve them. These are called from both the main thread and from preemptive threads.

ThreadGroupPutDF clips the specified data folder, and everything it contains, out of the source thread's data hierarchy and puts it in the queue. From the standpoint of the source thread, it is as if KillDataFolder had been called. While a data folder resides in a queue, it is not accessible by any thread. See the documentation for **ThreadGroupPutDF** for some warnings about its use.

ThreadGroupGetDFR removes the data folder from the queue and returns it, as a free data folder, to the calling thread. Because it is a free data folder, Igor will automatically delete it when there are no more references to it, for example, when the thread returns.

Except for waves passed to the thread worker function as parameters and the thread worker's return value, the input and output queues are the only way for a thread to share data with the main thread. Examples below illustrate the use of these queues.

Parallel Processing - Group-at-a-Time Method

In this example, we attempt to improve the speed of filling columns of a 2D wave with a sin function. The traditional method is compared with parallel processing. Notice how much more complicated the multi-threaded version, MTFillWave, is compared to the single threaded STFillWave.

```
ThreadSafe Function MyWorkerFunc(w,col)
    WAVE w
    Variable col

    w[][col]= sin(x/(col+1))

    return stopMSTimer(-2)             // Time when we finished
End

Function MTFillWave(dest)
    WAVE dest

    Variable ncol= DimSize(dest,1)
    Variable i,col,nthreads= ThreadProcessorCount
    Variable threadGroupID= ThreadGroupCreate(nthreads)

    for(col=0; col<ncol;)
        for(i=0; i<nthreads; i+=1)
            ThreadStart threadGroupID,i,MyWorkerFunc(dest,col)
            col+=1
            if( col>=ncol )
                break
            endif
        endfor

        do
            Variable threadGroupStatus= ThreadGroupWait(threadGroupID,100)
        while( threadGroupStatus != 0 )
    endfor
    Variable dummy= ThreadGroupRelease(threadGroupID)
End

Function STFillWave(dest)
    WAVE dest

    Variable ncol= DimSize(dest,1)
    Variable col

    for(col= 0;col<ncol;col+=1)
        MyWorkerFunc(dest,col)
    endfor
End

Function ThreadTest(rows)
    Variable rows

    Variable cols=10

    make/o/n=(rows,cols) jack

    Variable i

    for(i=0;i<10;i+=1)        // get any pending pause events out of the way
    endfor
    Variable ttime= stopMSTimer(-2)

    Variable t0= stopMSTimer(-2)
    MTFillWave(jack)
    Variable t1= stopMSTimer(-2)
    STFillWave(jack)
    Variable t2= stopMSTimer(-2)

    ttime= (stopMSTimer(-2) - ttime)*1e-6

    // Times are in microseconds
    printf "ST: %d, MT: %d; ",t2-t1,t1-t0
    printf "speed up factor: %.3g; total time= %.3gs\r",(t2-t1)/(t1-t0),ttime
End
```

The empty loop above is necessary because of periodic pauses in execution when Igor checks for user aborts. If a pause was pending, we want to get it out of the way beforehand to avoid it affecting the first timing test.

After starting Igor Pro, there is initially some extra overhead associated with creating new threads. Consequently, in the test results to follow, the first test is run twice.

Results for Mac Mini 1.66 GHz Core Duo, OS X 10.4.6:

```
•ThreadTest(100)
  ST: 223, MT: 1192; speed up factor: 0.187; total time= 0.00146s
•ThreadTest(100)
  ST: 211, MT: 884; speed up factor: 0.239; total time= 0.0011s
•ThreadTest(1000)
  ST: 1991, MT: 1821; speed up factor: 1.09; total time= 0.00381s
•ThreadTest(10000)
  ST: 19857, MT: 11921; speed up factor: 1.67; total time= 0.0318s
•ThreadTest(100000)
  ST: 199174, MT: 113701; speed up factor: 1.75; total time= 0.313s
•ThreadTest(1000000)
  ST: 2009948, MT: 1146113; speed up factor: 1.75; total time= 3.16s
```

As you can see, when there is sufficient work to be done, the speed up factor approaches the theoretical maximum of 2 for dual processors.

Now on the same computer but booting into Windows XP Pro:

```
•ThreadTest(100)
  ST: 245, MT: 523; speed up factor: 0.468; total time= 0.000776s
•ThreadTest(100)
  ST: 399, MT: 247; speed up factor: 1.61; total time= 0.000655s
•ThreadTest(1000)
  ST: 3526, MT: 1148; speed up factor: 3.07; total time= 0.00468s
•ThreadTest(10000)
  ST: 34830, MT: 10467; speed up factor: 3.33; total time= 0.0453s
•ThreadTest(100000)
  ST: 350253, MT: 99298; speed up factor: 3.53; total time= 0.45s
•ThreadTest(1000000)
  ST: 2837645, MT: 1057275; speed up factor: 2.68; total time= 3.89s
```

So, what is happening here? The speed-up factors for Windows XP are greater than for Mac OS X, but mostly because the ST version is much slower. We do not known why the ST version runs more slowly — the Benchmark 2.01 example experiment shows similar values for OS X vs. XP on this same computer.

Parallel Processing - Thread-at-a-Time Method

In the previous section, we dispatched a group of threads, waited for them to all finish, and then dispatched another group of threads. Using that technique, a slow thread in the group would cause all of the group's threads to wait.

In this section, we dispatch a thread anytime there is a free thread in the group. This technique requires Igor Pro 6.23 or later.

The only thing that changes from the preceding example is that the MTFillWave function is replaced with this MTFillWaveThreadAtATime function:

```
Function MTFillWaveThreadAtATime(dest)
   WAVE dest

   Variable ncol= DimSize(dest,1)
   Variable col,nthreads= ThreadProcessorCount
   Variable threadGroupID= ThreadGroupCreate(nthreads)
   Variable dummy

   for(col=0; col<ncol; col+=1)
      // Get index of a free thread - Requires Igor Pro 6.23 or later
      Variable threadIndex = ThreadGroupWait(threadGroupID,-2) - 1
      if (threadIndex < 0)
         dummy = ThreadGroupWait(threadGroupID, 50)// Let threads run a while
         col -= 1                   // Try again for the same column
         continue                   // No free threads yet
      endif
      ThreadStart threadGroupID, threadIndex, MyWorkerFunc(dest,col)
   endfor

   // Wait for all threads to finish
   do
      Variable threadGroupStatus = ThreadGroupWait(threadGroupID,100)
   while(threadGroupStatus != 0)

   dummy = ThreadGroupRelease(threadGroupID)
End
```

The ThreadGroupWait statement suspends the main thread for a while so that the preemptive threads get more processor time. The parameter 50 is the number of milliseconds to wait. You should tune this for your application.

Input/Output Queues

In this example, data folders containing a data wave and a string variable that specifies the task to be performed are created and posted to the thread group's input queue. The thread worker function waits for an input data folder to become available. It then processes the input and posts an output data folder to the thread group's output queue from which it is retrieved by the main thread.

```
ThreadSafe Function MyWorkerFunc()
   do
      do
         DFREF dfr = ThreadGroupGetDFR(0,1000)// Get free data folder from input queue
         if (DataFolderRefStatus(dfr) == 0)
            if( GetRTError(2) ) // New in 6.20 to allow this distinction:
               Print "worker closing down due to group release"
            else
               Print "worker thread still waiting for input queue"
            endif
         else
            break
         endif
      while(1)

      SVAR todo = dfr:todo
      WAVE jack = dfr:jack

      NewDataFolder/S outDF

      Duplicate jack,outw // WARNING: outw must be cleared. See WAVEClear below
      String/G did= todo
      if( CmpStr(todo,"sin") )
         outw= sin(outw)
      else
         outw= cos(outw)
      endif
      // Clear outw so Duplicate above does not try to use it and to allow
      // ThreadGroupPutDF to succeed.
      WAVEClear outw

      ThreadGroupPutDF 0,:   // Put current data folder in output queue

      KillDataFolder dfr     // We are done with the input data folder
   while(1)

   return 0
End

Function DemoThreadQueue()
   Variable i,ntries= 5,nthreads= 2

   Variable threadGroupID = ThreadGroupCreate(nthreads)

   for(i=0;i<nthreads;i+=1)
      ThreadStart threadGroupID,i,MyWorkerFunc()
   endfor

   for(i=0;i<ntries;i+=1)
      NewDataFolder/S forThread
      String/G todo
      if( mod(i,3) == 0 )
         todo= "sin"
      else
         todo= "cos"
      endif
      Make/N= 5 jack= x + gnoise(0.1)

      WAVEClear jack

      ThreadGroupPutDF threadGroupID,: // Send current data folder to input queue
   endfor

   for(i=0;i<ntries;i+=1)
      do
         // Get results in free data folder
         DFREF dfr= ThreadGroupGetDFR(threadGroupID,1000)
         if ( DatafolderRefStatus(dfr) == 0 )
            Print "Main still waiting for worker thread results."
         else
            break
         endif
      while(1)

      SVAR did = dfr:did
      WAVE outw = dfr:outw

      Print "task= ",did,"results= ",outw

      // The next two statements are not really needed as the same action
      // will happen the next time through the loop or, for the last iteration,
      // when this function returns.
      WAVEClear outw      // Redundant because of the WAVE statement above
      KillDataFolder dfr  // Redundant because dfr refers to a free data folder
   endfor

   // This terminates the MyWorkerFunc by setting an abort flag
   Variable tstatus= ThreadGroupRelease(threadGroupID)
   if( tstatus == -2 )
      Print "Thread would not quit normally, had to force kill it. Restart Igor."
   endif
End
```

Typical output:

```
•DemoThreadQueue()
   task=   sin  results=
outw[0]= {0.994567,0.660904,-0.516692,-0.996884,-0.63106}
   task=   cos  results=
outw[0]= {0.0786631,0.709576,0.873524,0.0586175,-0.718122}
   task=   cos  results=
outw[0]= {-0.23686,0.848603,0.871922,0.0992451,-0.856209}
   task=   sin  results=
outw[0]= {0.999734,0.531563,-0.172071,-0.931296,-0.750942}
   task=   cos  results=
outw[0]= {-0.166893,0.767707,0.925874,0.114511,-0.662994}
   worker closing down due to group release
   worker closing down due to group release
```

Parallel Processing With Large Datasets

In the preceding section we synthesized the input data. In the real-world, your input data would most-likely be in an existing wave and you would have to copy it to a data folder to put into the input queue.

If your input data is very large, for example, a 3D stack of images, copying would require too much memory. In that case, a good choice is to pass the input directly to the thread using parameters to the thread worker function and use the output queue to return output to the main thread.

To do this you can use the **Parallel Processing - Thread-at-a-Time Method** and the output queue to return results.

Preemptive Background Task

In this example, we create a single worker thread that runs while the user does other things. A normal cooperative background task retrieves results from the preemptive thread. Although the background task will sometimes be blocked (as described in **Background Tasks** on page IV-319) the preemptive worker thread will always be running or waiting for data.

Another example of this kind of multitasking can be found in the “Slow Data Acq” demo experiment referenced under **More Multitasking Examples** on page IV-339.

In some cases, it may be possible to run two instances of Igor instead of using a preemptive background task. Running two instances is far simpler, so use that approach if it is feasible.

We put the code for the background tasks in an independent module (see **The IndependentModule Pragma** on page IV-55) so that the user can recompile procedures, which is done automatically when a recreation macro is created, without stopping the background task.

You might use a preemptive background task is when you have lengthy computations but want to continue to do other things, such as creating graphics for publication. Although you can do anything you want while the task runs in the experiment, if you load a different experiment, the thread is killed.

For this example, our “lengthy computation” is simply creating a wave of sine values which is not lengthy at all and consequently there is no reason for using a preemptive thread in this case. To simulate a lengthy computation, the code delays for a few seconds before posting its results.

The named background task checks the output queue every 10 ticks, when it is not blocked, and updates a graph with data retrieved from the queue.

Independent modules can not be defined in the built-in procedure window so paste the following code in a new procedure window:

```
#pragma IndependentModule= PreemptiveExample

ThreadSafe Function MyWorkerFunc()
   do
      DFREF dfr = ThreadGroupGetDFR(0,inf)
      if( DataFolderRefStatus(dfr) == 0 )
         return -1                // Thread is being killed
      endif

      WAVE frequencies = dfr:frequencies    // Array of frequencies to calculate
      Variable i, n= numpnts(frequencies)

      for(i=0;i<n;i+=1)
         NewDataFolder/S resultsDF
         Make jack= sin(frequencies[i]*x)
         Variable t0= ticks
         do
            // waste cpu for a few seconds
         while(ticks < (t0+120))

         // ThreadGroupPutDF requires that no waves in the data folder be referenced
         WAVEClear jack

         ThreadGroupPutDF 0,:              // Send current data folder to input queue
      endfor

      KillDataFolder dfr      // We are done with the input data folder
   while(1)

   return 0
End

Function DisplayResults(s)     // Called from cooperative background task
   STRUCT WMBackgroundStruct &s

   DFREF dfSav= GetDataFolderDFR()

   SetDataFolder root:testdf
   NVAR threadGroupID
   DFREF dfr = ThreadGroupGetDFR(threadGroupID,0) // Get free data folder from queue
   if( DataFolderRefStatus(dfr) != 0 )
      // Make free data folder a regular data folder in root:testdf
      MoveDataFolder dfr, :

   // Give data folder a unique name
      String dfName = UniqueName("Results", 11, 0)
      RenameDataFolder dfr, $dfName

      WAVE jack = dfr:jack       // This is the output from the thread
      AppendToGraph/W=ThreadResultsGraph jack
   endif

   SetDataFolder dfSav
   return 0
End
```

And put this in the main procedure window:

```
Function DemoPreemptiveBackgroundTask()
   DFREF dfSav= GetDataFolderDFR()

   NewDataFolder/O/S root:testdf // thread group ID and result datafolders go here

   Variable/G threadGroupID= ThreadGroupCreate(1)

   ThreadStart threadGroupID,0,PreemptiveExample#MyWorkerFunc()

   // MyWorkerFunc is now running and waiting for input data
   // now, let's give it something to do
   NewDataFolder/S tasks
   Make/N=10 frequencies= 1/(10+p/2+enoise(0.2))// array of frequencies to calculate
   WAVEClear frequencies

   ThreadGroupPutDF threadGroupID,:        // thread is now crunching away

   // Results will be appended to this graph
   Display /N=ThreadResultsGraph as "Thread Results"

   // ...by this named task
   CtrlNamedBackground ThreadResultsTask,period=10,proc=PreemptiveExample#DisplayResults,start

   SetDataFolder dfSav     // restore current df
End

Function PostMoreFreqs()
   NVAR threadGroupID = root:testdf:threadGroupID

   NewDataFolder/S moretasks
   Make/N=50 frequencies= 1/(15+p/2+enoise(0.2))  // array of frequencies to calculate
   WAVEClear frequencies

   ThreadGroupPutDF threadGroupID,:               // thread continues crunching
End
```

Open the Data Browser and then, on the command line, execute:

```
DemoPreemptiveBackgroundTask()
```

After the action stops, send more tasks to the background thread by executing

```
PostMoreFreqs()
```

While this is running, experiment with creating graphs, using dialogs, creating functions, etc. Note that both tasks run indefinitely.

To start over you need to stop the preemptive background task, stop the named background task, kill the graph, and delete the data. This function, which you can paste into the main procedure window, will do it.

```
Function StopDemo()
   NVAR threadGroupID = root:testdf:threadGroupID

   // Stop preemptive thread
   Variable status = ThreadGroupRelease(threadGroupID)

   // Stop named background task
   CtrlNamedBackground ThreadResultsTask, stop

   // Kill graph
   DoWindow /K ThreadResultsGraph

   // Kill data
   KillDataFolder root:testdf
End
```

Aborting Threads

If a procedure abort occurs, via a user abort or a programmed abort (see **Aborting Functions** on page IV-112), we would like all pre-emptive threads to stop running. In most cases, this is not something you need to worry about because the threads, by the nature of what they are doing, are guaranteed to stop in a relatively short period of time. It becomes an issue for threads that can run for arbitrarily long periods of time. In such cases, it is a good idea to handle the abort as explained in this section.

To signal running pre-emptive threads to stop, the thread-coordinating function, running in the main thread, must detect the abort and call ThreadGroupRelease. ThreadGroupRelease signals the pre-emptive threads to quit. For example:

```
// A thread worker function that may take a long time
ThreadSafe Function ThreadWorkerFunc(threadNumber, numWaves, numPoints)
   Variable threadNumber, numWaves, numPoints

   Printf "Thread worker %d starting\r", threadNumber

   try
      int i
      for(i=0; i<numWaves; i+=1)
         Make/FREE/N=(numPoints) junk=gnoise(1) // Busy work to take up time
      endfor
   catch
      // ThreadGroupRelease, called from the main thread,
      // causes this catch block to execute
      Printf "Thread worker %d aborted after %d iterations\r", threadNumber, i
      return -1
   endtry

   Printf "Thread worker %d quitting normally\r", threadNumber
   return 0
End

Function ThreadCoordinatingFunction(numThreads, numWaves, numPoints)
   int numThreads, numWaves, numPoints    // Runs in main thread
   Variable timerRefNum = StartMSTimer
   Variable elapsedTime

   Variable threadGroupID = ThreadGroupCreate(numThreads)

   // Start all threads
   int i
   for(i=0; i<numThreads; i+=1)
      ThreadStart threadGroupID, i, ThreadWorkerFunc(i, numWaves, numPoints)
   endfor

   // Wait for threads to finish or for abort to occur
   Variable threadGroupResult
   try
      do
         Variable threadGroupStatus = ThreadGroupWait(threadGroupID,100)
         DoUpdate // Needed for Printf output from worker to appear in history
      while(threadGroupStatus != 0)
   catch
      // If a user or programmed abort occurs, this executes and stops threads
      elapsedTime = StopMSTimer(timerRefNum) / 1E6       // In seconds
      Printf "Aborted after %g seconds\r", elapsedTime
      DoUpdate // Needed for Printf output from worker to appear in history

      // Signal running threads to stop
      threadGroupResult = ThreadGroupRelease(threadGroupID)
      return -1
   endtry

   // Here if threads ran to normal completion
   threadGroupResult = ThreadGroupRelease(threadGroupID)
   elapsedTime = StopMSTimer(timerRefNum) / 1E6    // In seconds
   Printf "ThreadCoordinatingFunction completed in %g seconds\r", elapsedTime
   return threadGroupResult
End
```

To try this example, execute this from the command line:

```
ThreadCoordinatingFunction(5, 1E3, 1E5)   // Completes in about 25 seconds
```

After about 25 seconds, it completes normally. If you click the Abort button in Igor's status bar, the code in the catch block runs. The ThreadGroupRelease call in the catch block signals running threads to quit. After a short period of time, all threads have quit and the coordinating function returns.

If Igor signals a thread to quit but it fails to do so, Igor forces the thread to quit.

Aborting Threads Gracefully

When an abort occurs, whether initiated by the user (see **User Abort Key Combinations** on page IV-51) or programmatically using the **AbortOnValue**, **AbortOnRTE** or **Abort** operations, Igor sets an internal per-thread abort flag. The abort flag causes Igor to short-circuit loops and subroutine calls so that the function to be aborted finishes quickly.

If a thread worker function needs to run cleanup code in the event of an abort then the it must include a try-catch-endtry block (see **try-catch-endtry Flow Control**). This section explains what you must do to enable the catch code to run without interference.

By default, ThreadGroupRelease attempts to stop running threads by marking them with an unclearable abort flag. "Unclearable" means that, if a thread worker function has a catch block, when the abort flag is set, the catch block is called but the abort is still in effect which interferes with the code in the catch block.

As of Igor Pro 9, you can provide the optional beGraceful flag when calling ThreadGroupRelease. If you specify beGraceful=1, ThreadGroupRelease sets a clearable abort flag. "Clearable" means that, if a thread worker function has a catch block, the abort flag is cleared when the catch block is called. This allows code in the catch block to execute without interference. The catch block must include a return statement so that the thread worker function returns.

The interaction between Igor and threads when an abort occurs is complex. You don't need to understand more that is explained in the preceding paragraphs of this section. Advanced programmers who want to understand the details can find them in the Threads and Aborts example experiment.

More Multitasking Examples

More multitasking examples can be found in the following example experiments:

The Multithreaded LoadWave demo experiment in “Igor Pro Folder/Examples/Programming”.

The Multithreaded Mandelbrot demo experiment in “Igor Pro Folder/Examples/Programming”.

The Multiple Fits in Threads demo experiment in “Igor Pro Folder/Examples/Curve Fitting”.

The Slow Data Acq demo experiment in “Igor Pro Folder/Examples/Programming”.

The Thread-at-a-Time demo experiment in “Igor Pro Folder/Examples/Programming”.

Cursors — Moving Cursor Calls Function

You can write a hook function which Igor calls whenever a cursor is moved.

Graph-Specific Cursor Moved Hook

The preferred way to do this is to use SetWindow to designate a window hook function for a specific graph window (see **Window Hook Functions** on page IV-293). In your window hook function, look for the cursormoved event. Your hook function receives a **WMWinHookStruct** structure containing fields that describe the cursor and its properties.

For a demo of this technique, choose File→Example Experiments→Techniques→Cursor Moved Hook Demo.

Global Cursor Moved Hook

This section describes an old technique in which you create a hook function that is called any time a cursor is moved in any graph. This technique is more difficult to implement and kludgy, so it is no longer recommended.

You can write a hook function named CursorMovedHook. Igor automatically calls it whenever any cursor is moved in any graph, unless Option (*Macintosh*) or Alt (*Windows*) is pressed.

The CursorMovedHook function takes one string argument containing information about the graph, trace or image, and cursor in the following format:

```
GRAPH:graphName;CURSOR:<A - J>;TNAME:traceName; MODIFIERS:modifierNum;
ISFREE:freeNum;POINT:xPointNumber; [YPOINT:yPointNumber;]
```

The *traceName* value is the name of the graph trace or image to which the cursor is attached.

The *modifierNum* value represents the state of some of the keyboard keys summed together:

1 If Command (*Macintosh*) or Ctrl (*Windows*) is pressed.

2 If Control (*Macintosh only*) is pressed.

4 If Shift is pressed.

8 If Caps Lock is pressed.

The Option key (*Macintosh*) or Alt key (*Windows*) is not represented because it prevents the hook from being called.

The YPOINT keyword and value are present only when the cursor is attached to a two-dimensional item such as an image, contour, or waterfall plot or when the cursor is free.

If cursor is free, POINT and YPOINT values are fractional relative positions (see description in **Cursor** operation on page V-121). If TNAME is empty, fields POINT, ISFREE and YPOINT are not present.

This example hook function simply prints the information in the history area:

```
Function CursorMovedHook(info)
   String info
   Print info
End
```

Whenever any cursor on any graph is moved, this CursorMovedHook function will print something like the following in the history area:

```
GRAPH:Graph0;CURSOR:A;TNAME:jack;MODIFIERS:0;ISFREE:0;POINT:6;
```

Cursor Globals

On older technique involving globals named S_CursorAInfo and S_CursorBInfo is no longer recommended. For details, see the “Cursor Globals” subtopic in the Igor6 help files.

Profiling Igor Procedures

You can find bottlenecks in your procedures using profiling.

Profiling is supported by the FunctionProfiling.ipf file. To use it, add this to your procedures:

```
#include <FunctionProfiling
```

Then choose Windows→Procedures→FunctionProfiling.ipf and read the instructions in the file.

Crashes

A crash results from a software bug and prevents a program from continuing to run. Crashes are highly annoying at best and, at worst, can cause you to lose valuable work.

WaveMetrics uses careful programming practices and extensive testing to make Igor as reliable and bug-free as we can. However in Igor as in any complex piece of software it is impossible to exterminate all bugs. Also, crashes can sometimes occur in Igor because of bugs in other software, such as printer drivers, video drivers or system extensions.

We are committed to keeping Igor a solid and reliable program. If you experience a crash, we would like to know about it.

When reporting a crash to WaveMetrics, please start by choosing Help-Contact Support. This provides us with important information such as your Igor version and your OS version.

Please include the following in your report:

- A description of what actions preceded the crash and whether it is reproducible.
- A recipe for reproducing the crash, if possible.
- A crash log (described below), if possible.

In most cases, to fix a crash, we need to be able to reproduce it.

Crash Logs on Mac OS X

When a crash occurs on Mac OS X, most of the time the system is able to generate a crash log. You can usually find it at:

```
/Users/<user>/Library/Logs/DiagnosticReports/Igor Pro_<date>_<machinename>.crash
```

where <user> is your user name.

The /Users/<user>/Library folder is hidden. To reveal the DiagnosticReports folder:

1. Choose Finder->Go to Folder
2. Enter ~/Library/Logs/DiagnosticReports
3. Click Go

Send this log as an attachment when reporting a crash.

Crashes On Windows

When a crash occurs on Windows, Igor attempts to write a crash report that may help WaveMetrics determine and fix the cause of the crash. If Igor is able to write a crash report, it displays a dialog showing the location of the crash report on disk. The location is in your Igor preferences folder and will be something like:

```
C:\Users\<user>\AppData\Roaming\WaveMetrics\Igor Pro 8\Diagnostics\Igor Crash Reports.txt
```

If the "Igor Crash Reports.txt" file already exists when a crash occurs, Igor appends a new report to the existing file.

Igor may also write a minidump file to the same folder. A minidump file, which has a ".dmp" extension, contains additional information about the crash. There will typically be one minidump file for each crash.

When a crash occurs, Igor attempts to open the Diagnostics folder on your desktop to make it easy for you to find the report and minidump files.

Please send the "Igor Crash Reports.txt" file, along with the minidump files related to the current crash, as email attachments to WaveMetrics support. If possible, include instructions for reproducing the crash and any other details that may help us understand what you were doing leading to the crash.

Once the problem has been resolved, if you want to reduce clutter and reclaim disk space, you can delete the Diagnostics folder and its contents. Igor will recreate the folder if necessary.

There may be cases where Igor is not able to write a crash report. This would happen if a library, security software or the operating system has overridden Igor's crash handler for some reason.

Volume V Reference

Table of Contents

Built-In Operations by Category 1
Graphs.. 1
Contour and Image Plots 1
Tables 1
Layouts 2
Gizmo 2
Subwindows 2
Other Windows 2
All Windows 2
Wave Operations... 2
Analysis 3
Matrix Operations 3
Analysis of Functions 3
Signal Processing.. 4
Image Analysis 4
Statistics 4
Geometry 4
Drawing 5
Programming & Utilities.. 5
Files & Paths.. 5
HDF5 6
Data Folders 6
Movies & Sound 6
Controls & Cursors 6
FIFOs 6
Printing 6
Built-In Functions by Category 7
Numbers 7
Trig 7
Exponential 7
Complex.. 7
Rounding 7
Conversion 7
Time and Date... 7
Matrix Analysis 7
Wave Analysis 8
About Waves... 8
Special 8
Statistics 10
Windows.. 10
Strings 11
Names 11
Lists 11
Programming... 11
Data Folders 12
I/O (files, paths, and PICTs).. 12
HDF5 12
Built-In Keywords 13

- Procedure Declarations 13
- Procedure Subtypes 13
- Object References 13
- Function Local Variable Keywords 13
- Flow Control 13
- Other Programming Keywords 13

Built-in Structures 14

Hook Functions 14

Alphabetic Listing of Functions, Operations and Keywords 15

Igor Reference

This volume contains detailed information about Igor Pro’s built-in operations, functions, and keywords. They are listed alphabetically after the category sections that follow.

External operations (XOPs) and external functions (XFUNCs) are not covered here. For information about them, use the Command Help tab of the Igor Help Browser.

Built-In Operations by Category

Note: Some operations may appear in more than one category.

Graphs

AddWavesToBoxPlot
AddWavesToViolinPlot
AppendBoxPlot
AppendText
AppendToGraph
AppendToLayout
AppendViolinPlot
CheckDisplayed
ColorScale
ColorTab2Wave
ControlBar
Cursor
DefaultFont
DefineGuide
DelayUpdate
DeleteAnnotations
Display
DoUpdate
DoWindow
ErrorBars
GetAxis
GetMarquee
GetSelection
GetWindow
GraphNormal
GraphWaveDraw
GraphWaveEdit
HideInfo
HideTools
KillFreeAxis
KillWindow
Label
Legend
ModifyBoxPlot
ModifyFreeAxis
ModifyGraph
ModifyViolinPlot
ModifyWaterfall
MoveSubwindow
MoveWindow
NewFreeAxis
NewWaterfall
PauseUpdate
PrintGraphs
RemoveFromGraph
RenameWindow
ReorderImages
ReorderTraces
ReplaceText
ReplaceWave
ResumeUpdate
SaveGraphCopy
SetActiveSubwindow
SetAxis
SetMarquee
SetWindow
ShowInfo
ShowTools
StackWindows
Tag
TextBox
TickWavesFromAxis
TileWindows

Contour and Image Plots

AppendImage
AppendMatrixContour
AppendToLayout
AppendXYZContour
CheckDisplayed
ColorScale
ColorTab2Wave
DefineGuide
DeleteAnnotations
DoUpdate
DoWindow
FindContour
HideTools
ImageLoad
ImageSave
KillWindow
ModifyContour
ModifyImage
MoveSubwindow
NewImage
PauseUpdate
RemoveContour
RemoveImage
ReplaceWave
SetActiveSubwindow
SetWindow
ShowTools
Tag
TileWindows

Tables

AppendToLayout
AppendToTable
CheckDisplayed
DelayUpdate
DoUpdate
DoWindow
Edit
GetSelection
GetWindow
KillWindow
ModifyTable
MoveWindow
PauseUpdate
PrintTable
RemoveFromTable
RenameWindow
ResumeUpdate
SaveTableCopy
SetWindow
StackWindows
TileWindows

Layouts

AppendLayoutObject
AppendText
AppendToLayout
ColorScale
DefaultFont
DelayUpdate
DeleteAnnotations
DoUpdate
DoWindow
GetMarquee
GetSelection
GetWindow
HideTools
KillWindow
Layout
LayoutPageAction
LayoutSlideShow
Legend
ModifyLayout
MoveWindow
NewLayout
PauseUpdate
PrintLayout
RemoveFromLayout
RemoveLayoutObjects
RenameWindow
ReplaceText
ResumeUpdate
SetMarquee
SetWindow
ShowTools
Stack
StackWindows
TextBox
Tile
TileWindows

Gizmo

AppendToGizmo
ExportGizmo
GetGizmo
GizmoInfo
GizmoScale
ModifyGizmo
NewGizmo
RemoveFromGizmo

Subwindows

DefineGuide
GetMarquee
KillWindow
MoveSubwindow
RenameWindow
SetActiveSubwindow
SetMarquee

Other Windows

CloseHelp
CloseProc
CreateBrowser
DisplayProcedure
DoWindow
GetCamera
GetSelection
GetWindow
GetWindowBrowserSelection
HideProcedures
HideTools
KillWindow
ModifyBrowser
ModifyCamera
ModifyPanel
ModifyProcedure
MoveWindow
NewCamera
NewNotebook
NewPanel
Notebook
NotebookAction
OpenHelp
OpenNotebook
PrintNotebook
RenameWindow
SaveNotebook
SetWindow
ShowTools
StackWindows

All Windows

Append
AutoPositionWindow
DoWindow
GetSelection
GetUserData
GetWindow
KillWindow
Modify
MoveWindow
Remove
RenameWindow
SetWindow
StackWindows
TileWindows

Wave Operations

AddMovieAudio
Append
AppendToGraph
AppendToTable
BezierToPolygon
CheckDisplayed
ColorTab2Wave
Concatenate
CopyDimLabels
CopyScales
DeletePoints
Display
Duplicate
Edit
Extract
FIFO2Wave
FindPointsInPoly
FindSequence
FindValue
GBLoadWave
GraphWaveDraw
GraphWaveEdit
InsertPoints
JCAMPLoadWave
KillWaves
LoadData
LoadWave
Make
MLLoadWave
MoveWave
Note
PlaySound
PolygonOp
Redimension
Remove
RemoveFromGraph
RemoveFromTable
Rename
ReplaceWave
Reverse

- Rotate
- Save
- SetDimLabel
- SetScale
- SetWaveLock
- SetWaveTextEncoding
- SoundLoadWave
- SoundSaveWave
- SplitWave
- WAVEClear
- WaveStats
- wfprintf
- XLLoadWave

Analysis

- APMath
- BoundingBall
- BoxSmooth
- Convolve
- Correlate
- ConvexHull
- Cross
- CurveFit
- CWT
- Differentiate
- DPSS
- DSPDetrend
- DSPPeriodogram
- DWT
- EdgeStats
- EstimatePeakSizes
- FastGaussTransform
- FastOp
- FFT
- FilterFIR
- FilterIIR
- FindContour
- FindDuplicates
- FindLevel
- FindLevels
- FindPeak
- FindAPeak
- FindPointsInPoly
- FindRoots
- FindValue
- FMaxFlat
- FuncFit
- FuncFitMD
- Hanning
- HCluster
- HilbertTransform
- Histogram
- ICA
- IFFT
- IndexSort
- InstantFrequency
- Integrate
- Integrate1D
- Integrate2D
- IntegrateODE
- Interpolate2
- Interp3DPath
- Interpolate3D
- JointHistogram
- Loess
- LombPeriodogram
- MakeIndex
- MultiTaperPSD
- NeuralNetworkRun
- NeuralNetworkTrain
- Optimize
- PCA
- PrimeFactors
- PolygonOp
- Project
- PulseStats
- RatioFromNumber
- Remez
- Resample
- Smooth
- SmoothCustom
- Sort
- SortColumns
- SphericalInterpolate
- SphericalTriangulate
- STFT
- SumDimension
- SumSeries
- TextHistogram
- Triangulate3D
- Unwrap
- WaveMeanStdv
- WaveStats
- WaveTransform
- WignerTransform
- WindowFunction

Matrix Operations

- Concatenate
- Extract
- FFT
- IFFT
- ImageFilter
- ImageFromXYZ
- Loess
- MatrixBalance
- MatrixConvolve
- MatrixConvolve
- MatrixCorr
- MatrixEigenV
- MatrixFactor
- MatrixFilter
- MatrixGaussJ
- MatrixGLM
- MatrixInverse
- MatrixLinearSolve
- MatrixLinearSolveTD
- MatrixLLS
- MatrixLUBkSub
- MatrixLUD
- MatrixLUDTD
- MatrixMultiply
- MatrixMultiplyAdd
- MatrixOp
- MatrixRank
- MatrixReverseBalance
- MatrixSchur
- MatrixSolve
- MatrixSparse
- MatrixSVBkSub
- MatrixSVD
- MatrixTranspose
- Reverse
- SplitWave
- SumDimension
- WaveTransform

Analysis of Functions

- FindRoots
- Integrate1D
- IntegrateODE
- Optimize
- SumSeries

Signal Processing

BoxSmooth
Convolve
Correlate
CWT
DPSS
DSPDetrend
DSPPeriodogram
DWT
EdgeStats
EstimatePeakSizes
FFT
FilterFIR
FilterIIR
FindLevel
FindLevels
FindAPeak
FindPeak
FMaxFlat
Hanning
HilbertTransform
IFFT
ImageWindow
InstantFrequency
LinearFeedbackShiftRegister
LombPeriodogram
MultiTaperPSD
PulseStats
Remez
Resample
Rotate
Smooth
SmoothCustom
STFT
Unwrap
WignerTransform
WindowFunction

Image Analysis

ColorScale
ColorTab2Wave
DWT
ImageAnalyzeParticles
ImageBlend
ImageBoundaryToMask
ImageComposite
ImageEdgeDetection
ImageFileInfo
ImageFilter
ImageFocus
ImageFromXYZ
ImageGenerateROIMask
ImageGLCM
ImageHistModification
ImageHistogram
ImageInfo
ImageInterpolate
ImageLineProfile
ImageLoad
ImageMorphology
ImageNameList
ImageNameToWaveRef
ImageRegistration
ImageRemoveBackground
ImageRestore
ImageRotate
ImageSave
ImageSeedFill
ImageSnake
ImageSkeleton3D
ImageStats
ImageThreshold
ImageTransform
ImageUnwrapPhase
v
Loess
MatrixFilter

Statistics

EdgeStats
FPClustering
Histogram
ICA
ImageHistModification
ImageHistogram
ImageStats
JointHistogram
KMeans
PCA
PulseStats
SetRandomSeed
StatsAngularDistanceTest
StatsANOVA1Test
StatsANOVA2NRTest
StatsANOVA2RMTest
StatsANOVA2Test
StatsChiTest
StatsCircularCorrelationTest
StatsCircularMeans
StatsCircularMoments
StatsCircularTwoSampleTest
StatsCochranTest
StatsContingencyTable
StatsDIPTest
StatsDunnettTest
StatsFriedmanTest
StatsFTest
StatsHodgesAjneTest
StatsJBTest
StatsKDE
StatsKendallTauTest
StatsKSTest
StatsKWTest
StatsLinearCorrelationTest
StatsLinearRegression
StatsMultiCorrelationTest
StatsNPMCTest
StatsNPNominalSRTest
StatsQuantiles
StatsRankCorrelationTest
StatsResample
StatsSample
StatsScheffeTest
StatsShapiroWilkTest
StatsSignTest
StatsSRTest
StatsTTest
StatsTukeyTest
StatsVariancesTest
StatsWatsonUSquaredTest
StatsWatsonWilliamsTest
StatsWheelerWatsonTest
StatsWilcoxonRankTest
StatsWRCorrelationTest
TextHistogram
WaveMeanStdv
WaveStats

Geometry

BezierToPolygon
BoundingBall
ConvexHull
FindPointsInPoly
Interp3DPath
Interpolate3D
PolygonOp
Project
SphericalInterpolate
SphericalTriangulate
Triangulate3D

Drawing

BezierToPolygon
DrawAction
DrawArc
DrawBezier
DrawLine
DrawOval
DrawPICT
DrawPoly
DrawRect
DrawRRect
DrawText
DrawUserShape
GraphNormal
GraphWaveDraw
GraphWaveEdit
HideTools
SetDashPattern
SetDrawEnv
SetDrawLayer
ShowTools
Text2Bezier
ToolsGrid

Programming & Utilities

Abort
BackgroundInfo
Beep
BuildMenu
ChooseColor
CloseProc
CtrlBackground
CtrlNamedBackground
DefaultGUIFont
DefaultGUIControls
Debugger
DebuggerOptions
DefaultTextEncoding
DelayUpdate
DisplayHelpTopic
DisplayProcedure
DoAlert
DoIgorMenu
DoUpdate
DoXOPIdle
Execute
Execute/P
ExecuteScriptText
ExperimentInfo
ExperimentModified
GetLastUserMenuInfo
GetMouse
Grep
HideIgorMenus
HideProcedures
IgorVersion
KillBackground
KillStrings
KillVariables
LoadPackagePreferences
MarkPerfTestTime
MeasureStyledText
ModifyProcedure
MoveString
MoveVariable
MoveWave
MultiThreadingControl
ParseOperationTemplate
PauseForUser
PauseUpdate
Preferences
PrintSettings
PutScrapText
Quit
Rename
ResumeUpdate
SavePackagePreferences
SetBackground
SetFormula
SetIdlePeriod
SetIgorHook
SetIgorMenuMode
SetIgorOption
SetProcessSleep
SetRandomSeed
SetWaveLock
ShowIgorMenus
Silent
Sleep
Slow
SplitString
sprintf
sscanf
String
StructFill
StructGet
StructPut
ThreadGroupPutDF
ThreadStart
ToCommandLine
Variable
WAVEClear
WaveTracking

Files & Paths

AdoptFiles
BrowseURL
Close
CopyFile
CopyFolder
CreateAliasShortcut
DeleteFile
DeleteFolder
FBinRead
FBinWrite
fprintf
FReadLine
FGetPos
FSetPos
FStatus
FTPCreateDirectory
FTPDelete
FTPDownload
FTPUpload
GBLoadWave
GetFileFolderInfo
Grep
ImageFileInfo
ImageLoad
ImageSave
JCAMPLoadWave
KillPath
KillPICTs
KillWaves
LoadData
LoadPICT
LoadWave
MLLoadWave
MoveFile
MoveFolder
NewNotebook
NewPath
Open
OpenNotebook
OpenProc
PathInfo
ReadVariables
RemovePath
RenamePath
RenamePICT
Save
SaveData
SaveExperiment

SaveGizmoCopy
SaveGraphCopy
SaveNotebook
SavePICT
SaveTableCopy
SetFileFolderInfo
UnzipFile
URLRequest
wfprintf
XLLoadWave

HDF5

HDF5CloseFile
HDF5CloseGroup
HDF5Control
HDF5CreateFile
HDF5CreateGroup
HDF5CreateLink
HDF5DimensionScale
HDF5Dump
HDF5DumpErrors
HDF5FlushFile
HDF5ListAttributes
HDF5ListGroup
HDF5LoadData
HDF5LoadGroup
HDF5LoadImage
HDF5OpenFile
HDF5OpenGroup
HDF5SaveData
HDF5SaveGroup
HDF5SaveImage
HDF5UnlinkObject

Data Folders

cd
Dir
DuplicateDataFolder
KillDataFolder
MoveDataFolder
MoveVariable
MoveWave
NewDataFolder
pwd
RenameDataFolder
ReplaceWave
root
SetDataFolder

Movies & Sound

AddMovieAudio
AddMovieFrame
Beep
CloseMovie
ImageFileInfo
NewMovie
PlayMovie
PlayMovieAction
PlaySnd
PlaySound
SoundInRecord
SoundInSet
SoundInStartChart
SoundInStatus
SoundInStopChart
SoundLoadWave
SoundSaveWave

Controls & Cursors

Button
Chart
CheckBox
ControlBar
ControlInfo
ControlUpdate
Cursor
CustomControl
DefaultGUIFont
DefaultGUIControls
GetUserData
GroupBox
HideInfo
HideTools
KillControl
ListBox
ListBoxControl
ModifyControl
ModifyControlList
NewPanel
popup
PopupContextualMenu
PopupMenu
PopupMenuControl
SetVariable
ShowInfo
ShowTools
Slider
TabControl
TitleBox
ValDisplay

FIFOs

AddFIFOData
AddFIFOVectData
Chart
ControlInfo
CtrlFIFO
FIFO2Wave
FIFOStatus
KillFIFO
NewFIFO
NewFIFOChan
SoundInStartChart

Printing

Print
printf
PrintGraphs
PrintLayout
PrintNotebook
PrintSettings
PrintTable
sprintf
wfprintf

Built-In Functions by Category

Note: some functions may appear in more than one category.

Numbers

e
Inf
NaN
numtype
Pi
VariableList

Trig

acos
asin
atan
atan2
cos
cot
csc
sawtooth
sec
sin
sinc
sqrt
tan

Exponential

acosh
alog
asinh
atanh
cosh
coth
cpowi
csch
exp
ln
log
sech
sinh
tanh

Complex

cabs
cequal
cmplx
conj
cpowi
imag
magsqr
p2rect
r2polar
real

Rounding

abs
cabs
ceil
floor
limit
max
min
mod
round
sign
trunc

Conversion

char2num
cmplx
ConvertGlobalStringTextEncoding
ConvertTextEncoding
date2secs
imag
LowerStr
magsqr
NormalizeUnicode
num2char
num2istr
num2str
p2rect
pnt2x
r2polar
real
Secs2Date
Secs2Time
str2num
UpperStr
x2pnt

Time and Date

CreationDate
date
dateToJulian
date2secs
DateTime
JulianToDate
ModDate
Secs2Date
Secs2Time
StartMSTimer
StopMSTimer
ticks
time

Matrix Analysis

MatrixCondition
MatrixDet
MatrixDot
MatrixRank
MatrixTrace

Wave Analysis

area
areaXY
BinarySearch
BinarySearchInterp
centerOfMass
centerOfMassXY
centerOfMass
centerOfMassXY
ContourZ
FakeData
faverage
faverageXY
GeometricMean
interp
Interp2D
Interp3D
mean
median
p
poly
poly2D
PolygonArea
q
r
s
sum
t
Variance
WaveMax
WaveMin
WaveMinAndMax
x
y
z

About Waves

BinarySearch
BinarySearchInterp
ContourInfo
ContourNameToWaveRef
ContourZ
CreationDate
CsrInfo
CsrWave
CsrWaveRef
CsrXWave
CsrXWaveRef
deltax
DimDelta
DimOffset
DimSize
EqualWaves
exists
FindDimLabel
GetDimLabel
GetWavesDataFolder
GetWavesDataFolderDFR
hcsr
ImageInfo
ImageNameToWaveRef
IndexToScale
leftx
ModDate
NameOfWave
NewFreeWave
note
numpnts
p
pcsr
pnt2x
q
qcsr
r
rightx
s
ScaleToIndex
t
TagVal
TagWaveRef
TraceInfo
TraceNameToWaveRef
vcsr
WaveCRC
WaveDims
WaveExists
WaveHash
WaveInfo
WaveList
WaveModCount
WaveName
WaveRefIndexed
WaveRefsEqual
WaveTextEncoding
WaveType
WaveUnits
x
x2pnt
xcsr
XWaveName
XWaveRefFromTrace
y
zcsr

Special

airyA
airyAD
airyB
airyBD
Besseli
Besselj
Besselk
Bessely
bessI
bessJ
bessK
bessY
beta
betai
binomial
binomialln
binomialNoise
chebyshev
chebyshevU
CosIntegral
dawson
digamma
Dilogarithm
ei
EllipticE
EllipticK
enoise
erf
erfc
erfcw
erfcx
expInt
ExpIntegralE1
expNoise
factorial
Faddeeva
fresnelCos
fresnelCS
fresnelSin
gamma
gammaInc
gammaNoise
gammln
gammp
gammq
Gauss
Gauss1D
Gauss2D
gcd
gnoise
hermite
hermiteGauss
hyperG0F1
hyperG1F1
hyperG2F1
hyperGNoise

hyperGPFQ
inverseErf
inverseErfc
JacobiCn
JacobiSn
laguerre
laguerreA
laguerreGauss
LambertW
legendreA
logNormalNoise
lorentzianNoise
MandelbrotPoint
MarcumQ
MPFXEMGPeak
MPFXExpConvExpPeak
MPFXGaussPeak
MPFXLorentzianPeak
MPFXVoigtPeak
poissonNoise
poly
poly2D
SinIntegral
sphericalBessJ
sphericalBessJD
sphericalBessY
sphericalBessYD
sphericalHarmonics
sqrt
VoigtFunc
VoigtPeak
zeta
ZernikeR

Statistics

binomialln
binomialNoise
enoise
erf
erfc
expNoise
faverage
faverageXY
gamma
gammaInc
gammaNoise
gammln
gammp
gammq
gnoise
inverseErf
inverseErfc
lorentzianNoise
logNormalNoise
mean
max
min
norm
poissonNoise
StatsCorrelation
StatsBetaCDF
StatsBetaPDF
StatsBinomialCDF
StatsBinomialPDF
StatsCauchyCDF
StatsCauchyPDF
StatsChiCDF
StatsChiPDF
StatsCMSSDCDF
StatsCorrelation
StatsDExpCDF
StatsDExpPDF
StatsErlangCDF
StatsErlangPDF
StatsErrorPDF
StatsEValueCDF
StatsEValuePDF
StatsExpCDF
StatsExpPDF
StatsFCDF
StatsFPDF
StatsFriedmanCDF
StatsGammaCDF
StatsGammaPDF
StatsGeometricCDF
StatsGeometricPDF
StatsGEVCDF
StatsGEVPDF
StatsHyperGCDF
StatsHyperGPDF
StatsInvBetaCDF
StatsInvBinomialCDF
StatsInvCauchyCDF
StatsInvChiCDF
StatsInvCMSSDCDF
StatsInvDExpCDF
StatsInvEValueCDF
StatsInvExpCDF
StatsInvFCDF
StatsInvFriedmanCDF
StatsInvGammaCDF
StatsInvGeometricCDF
StatsInvKuiperCDF
StatsInvLogisticCDF
StatsInvLogNormalCDF
StatsInvMaxwellCDF
StatsInvMooreCDF
StatsInvNBinomialCDF
StatsInvNCChiCDF
StatsInvNCFCDF
StatsInvNormalCDF
StatsInvParetoCDF
StatsInvPoissonCDF
StatsInvPowerCDF
StatsInvQCDF
StatsInvQpCDF
StatsInvRayleighCDF
StatsInvRectangularCDF
StatsInvSpearmanCDF
StatsInvStudentCDF
StatsInvTopDownCDF
StatsInvTriangularCDF
StatsInvUSquaredCDF
StatsInvVonMisesCDF
StatsInvWeibullCDF
StatsKuiperCDF
StatsLogisticCDF
StatsLogisticPDF
StatsLogNormalCDF
StatsLogNormalPDF
StatsMaxwellCDF
StatsMaxwellPDF
StatsMedian
StatsMooreCDF
StatsNBinomialCDF
StatsNBinomialPDF
StatsNCChiCDF
StatsNCChiPDF
StatsNCFCDF
StatsNCFPDF
StatsNCTCDF
StatsNCTPDF
StatsNormalCDF
StatsNormalPDF
StatsParetoCDF
StatsParetoPDF
StatsPermute
StatsPoissonCDF
StatsPoissonPDF
StatsPowerCDF
StatsPowerNoise
StatsPowerPDF
StatsQCDF
StatsQpCDF
StatsRayleighCDF
StatsRayleighPDF
StatsRectangularCDF
StatsRectangularPDF
StatsRunsCDF
StatsSpearmanRhoCDF
StatsStudentCDF
StatsStudentPDF
StatsTopDownCDF
StatsTriangularCDF
StatsTriangularPDF
StatsTrimmedMean
StatsUSquaredCDF
StatsVonMisesCDF
StatsVonMisesPDF
StatsWaldCDF
StatsWaldPDF
StatsWeibullCDF
StatsWeibullPDF
StudentA
StudentT
sum
Variance
WaveMax
WaveMin
WaveMinAndMax
wnoise

Windows

AnnotationInfo
AnnotationList
AxisInfo
AxisLabel
AxisList
AxisValFromPixel
ChildWindowList
ContourInfo
CsrInfo
CsrWave
CsrXWave
GetBrowserLine
GetBrowserSelection
GuideInfo
GuideNameList
GizmoScale
hcsr
ImageInfo
LayoutInfo
PanelResolution

pcsr	PixelFromAxisVal	qcsr	SpecialCharacterInfo
SpecialCharacterList	TagVal	TraceInfo	vcsr
WinList	WinName	WinRecreation	WinType
xcsr	XWaveName	zcsr	

Strings

AddListItem	Base64Decode	Base64Encode	char2num
CmpStr	FontSizeHeight	FontSizeStringWidth	GrepList
GrepString	IndexedDir	IndexedFile	ListToTextWave
LowerStr	num2char	num2istr	num2str
PadString	PossiblyQuoteName	RemoveEnding	RemoveFromList
RemoveListItem	ReplaceStringByKey	ReplicateString	SelectString
str2num	StringByKey	StringCRC	StringFromList
StringList	StringMatch	StringToUnsignedByteWave	strlen
strsearch	TextFile	TrimString	UnPadString
UpperStr	URLDecode	URLEncode	WaveDims
WaveDataToString	WhichListItem		

Names

CheckName	CleanupName	ContourNameList	ContourNameToWaveRef
ControlNameList	CreateDataObjectName	CTabList	FontList
FunctionList	GetDefaultFont	GetIndependentModuleName	GetIndexedObjName
GetWavesDataFolder	GetWavesDataFolderDFR	ImageNameList	ImageNameToWaveRef
IndependentModuleList	IndexedDir	IndexedFile	MacroList
NameOfWave	StringList	TextEncodingCode	TextEncodingName
TraceFromPixel	TraceNameList	TraceNameToWaveRef	UniqueName
VariableList	WaveList	WaveName	WinList
WinName	XWaveName		

Lists

AnnotationList	AxisList	ChildWindowList	ContourNameList
ControlNameList	CountObjects	CountObjectsDFR	DataFolderDir
DataFolderList	FindListItem	FontList	FunctionInfo
FunctionList	GetIndexedObjName	GetWindow	GuideNameList
ImageNameList	IndependentModuleList	ItemsInList	ListMatch
ListToTextWave	ListToWaveRefWave	MacroList	NumberByKey
OperationList	PathList	PICTList	RemoveByKey
RemoveFromList	RemoveListItem	ReplaceNumberByKey	ReplaceStringByKey
SortList	StringByKey	StringFromList	StringList
TableInfo	TraceNameList	VariableList	WaveList
WaveRefIndexed	WaveRefWaveToList	WhichListItem	WinList

Programming

CaptureHistory	CaptureHistoryStart	ControlNameList	exists
FakeData	FuncRefInfo	FunctionInfo	FunctionPath
GetDefaultFont	GetDefaultFontSize	GetDefaultFontStyle	GetEnvironmentVariable

GetErrMessage GetFormula GetKeyState GetRTError
GetRTErrMessage GetRTLocation GetRTLocInfo GetRTStackInfo
GetScrapText GuideInfo GuideNameList Hash
i IgorInfo ilim jlim
MacroInfo MacroPath NameOfWave numtype
NumVarOrDefault NVAR_Exists PanelResolution
ParamIsDefault PICTInfo PixelFromAxisVal ProcedureText
ProcedureVersion ScreenResolution SelectNumber SelectString
SetEnvironmentVariable SpecialDirPath StartMSTimer StopMSTimer
StringCRC StrVarOrDefault SVAR_Exists TableInfo
TagVal ThreadGroupCreate ThreadGroupGetDF ThreadGroupGetDFR
ThreadGroupRelease ThreadGroupWait ThreadProcessorCount ThreadReturnValue
UnsetEnvironmentVariable WaveCRC WinType

Data Folders

CountObjects DataFolderDir DataFolderExists DataFolderList
DataFolderRefChanges DataFolderRefsEqual DataFolderRefStatus GetDataFolder
GetDataFolderDFR GetIndexedObjName GetWavesDataFolder GetWavesDataFolderDFR
NewFreeDataFolder

I/O (files, paths, and PICTs)

FetchURL IndexedDir IndexedFile ParseFilePath
PathList PICTInfo PICTList SpecialDirPath
TextFile URLDecode URLEncode

HDF5

HDF5AttributeInfo HDF5DatasetInfo HDF5LibraryInfo HDF5LinkInfo
HDF5TypeInfo

Built-In Keywords

Procedure Declarations

- End
- EndMacro
- EndStructure
- Function
- Macro
- Picture
- Proc
- Structure
- Window

Procedure Subtypes

- ButtonControl
- CameraWindow
- CDFFunc
- CheckBoxControl
- CursorStyle
- FitFunc
- GizmoPlot
- Graph
- GraphMarquee
- GraphStyle
- GridStyle
- Layout
- LayoutMarquee
- LayoutStyle
- ListBoxControl
- Panel
- PopupMenuControl
- SetVariableControl
- SliderControl
- TabControl
- Table
- TableStyle

Object References

- DFREF
- FUNCREF
- NVAR
- STRUCT
- SVAR
- WAVE

Function Local Variable Keywords

- Complex
- Double
- Int
- Int64
- String
- STRUCT
- UInt64
- Variable

Flow Control

- AbortOnRTE
- AbortOnValue
- break
- catch
- continue
- default
- do-while
- endtry
- for-endfor
- for-var-in-wave
- if-elseif-endif
- if-endif
- return
- strswitch-case-endswitch
- switch-case-endswitch
- try
- try-catch-endtry

Other Programming Keywords

- #define
- #if-#elif-#endif
- #if-#endif
- #ifdef-#endif
- #ifndef-#endif
- #include
- #pragma
- #undef
- Constant
- DefaultTab
- DoPrompt
- GalleryGlobal
- hide
- IgorVersion
- IndependentModule
- Menu
- ModuleName
- MultiThread
- Override
- popup
- ProcGlobal
- Prompt
- root
- rtGlobals
- Static
- StrConstant
- String
- Submenu
- TextEncoding
- ThreadSafe
- Variable
- version

Built-in Structures

HDF5DataInfo
HDF5DatatypeInfo
HDF5LinkInfoStruct
HDF5SaveDataHookStruct
Point
PointF
Rect
RectF
RGBColor
RGBAColor
WMAxisHookStruct
WMBackgroundStruct
WMButtonAction
WMCheckboxAction
WMCustomControlAction
WMDrawUserShapeStruct
WMFitInfoStruct
WMGizmoHookStruct
WMListboxAction
WMMarkerHookStruct
WMPopupAction
WMSetVariableAction
WMSliderAction
WMTabControlAction
WMWinHookStruct

These symbols are used to specify array sizes in the structure definitions:

```
Constant MAX_OBJ_NAME 255

Constant MAX_WIN_PATH 400

Constant MAX_UNITS 49

Constant MAXCMDLEN 2500
```

Prior to Igor Pro 8.00, MAX_OBJ_NAME was 31, MAX_WIN_PATH was 200, and MAXCMDLEN was 1000.

Hook Functions

See Chapter IV-10, **Advanced Topics**, **User-Defined Hook Functions** on page IV-280.

AfterCompiledHook
AfterFileOpenHook
AfterMDIFrameSizedHook
AfterWindowCreatedHook
BeforeDebuggerOpensHook
BeforeExperimentSaveHook
BeforeFileOpenHook
HDF5SaveDataHook
IgorBeforeNewHook
IgorBeforeQuitHook
IgorMenuHook
IgorQuitHook
IgorStartOrNewHook

Alphabetic Listing of Functions, Operations and Keywords

This section alphabetically lists all built-in functions, operations and keywords. Much of this information is also accessible online in the Command Help tab of the Igor Help Browser.

External operations (XOPs) and external functions (XFUNCs) are not covered here. For information about them, use the Command Help tab of the Igor Help Browser and the XOP help file in the same folder as the XOP file.

Reference Syntax Guide

In the descriptions of functions and operations that follow, italics indicate parameters for which you can supply numeric or string expressions. Non-italic keywords must be entered literally as they appear. Commas, slashes, braces and parentheses in these descriptions are always literals. Brackets surround optional flags or parameters. Ellipses (…) indicate that the preceding element may be repeated a number of times.

Italicized parameters represent values you supply. Italic words ending with "*Name*" are names (wave names, for example), and those ending with "*Str*" are strings. Italic words ending with "*Spec*" (meaning "specification") are usually further defined in the description. If none of these endings are employed, the italic word is a numeric expression, such as a literal number, the name of a variable or function, or some valid combination.

Strings and names are different, but you can use a string where a name is expected using "string substitution": precede a string expression with the $ operator. See **String Substitution Using $** on page IV-18.

A syntax description may span several lines, but the actual command you create must occupy a single line.

Many operations have optional "flags". Flags that accept a value (such as the Make operation's /N=*n* flag) sometimes require additional parentheses. For example:

```
Make/N=1 aNewWave
```

is acceptable because here *n* is the literal "1". To use a numeric expression (anything other than a literal number) for *n*, parentheses are needed:

```
Make/N=(numberOfPoints) aNewWave              // error if no parentheses!
```

For more about using functions, operations and keywords, see Chapter IV-1, **Working with Commands**, Chapter IV-2, **Programming Overview**, and Chapter IV-10, **Advanced Topics**.

#define

```
#define symbol
```

The #define statement is a conditional compilation directive that defines a *symbol* for use only with #ifdef or #ifndef expressions. #undef removes the definition.

Details

The defined *symbol* exists only in the file where it is defined; the only exception is in the main procedure window where the scope covers all other procedures except independent modules. See **Conditional Compilation** on page IV-108 for information on defining a global *symbol*.

#define cannot be combined inline with other conditional compilation directives.

See Also

The #**undef**, #**ifdef**-#**endif**, and #**ifndef**-#**endif** statements.

Conditional Compilation on page IV-108.

#if-#elif-#endif

```
#if expression1
   <TRUE part 1>
#elif expression2
   <TRUE part 2>
[…]
[#else
   <FALSE part>]
#endif
```

In a #if-#elif-#endif conditional compilation statement, when an expression evaluates as TRUE (absolute value > 0.5), then only code corresponding to the TRUE part of that expression is compiled, and then the conditional statement is exited. If all expressions evaluate as FALSE (zero) then *FALSE part* is compiled when present.

Details

Conditional compiler directives must be either entirely outside or inside function definitions; they cannot straddle a function fragment. Conditionals cannot be used within Macros.

See Also

Conditional Compilation on page IV-108 for more usage details.

#if-#endif

```
#if expression
   <TRUE part>
[#else
   <FALSE part>]
#endif
```

A #if-#endif conditional compilation statement evaluates *expression*. If *expression* is TRUE (absolute value > 0.5) then the code in *TRUE part* is compiled, or if FALSE (zero) then the optional *FALSE part* is compiled.

Details

Conditional compiler directives must be either entirely outside or inside function definitions; they cannot straddle a function fragment. Conditionals cannot be used within Macros.

See Also

Conditional Compilation on page IV-108 for more usage details.

#ifdef-#endif

```
#ifdef symbol
   <TRUE part>
[#else
   <FALSE part>]
#endif
```

A #ifdef-#endif conditional compilation statement evaluates *symbol*. When *symbol* is defined the code in *TRUE part* is compiled, or if undefined then the optional *FALSE part* is compiled.

Details

Conditional compiler directives must be either entirely outside or inside function definitions; they cannot straddle a function fragment. Conditionals cannot be used within Macros.

symbol must be defined before the conditional with #define.

See Also

The #**define** statement and **Conditional Compilation** on page IV-108 for more usage details.

#ifndef-#endif

```
#ifndef symbol
   <TRUE part>
[#else
   <FALSE part>]
#endif
```

An #ifndef-#endif conditional compilation statement evaluates *symbol*. When *symbol* is undefined the code in *TRUE part* is compiled, or if defined then the optional *FALSE part* is compiled.

Details

Conditional compiler directives must be either entirely outside or inside function definitions; they cannot straddle a function fragment. Conditionals cannot be used within Macros.

symbol must be defined before the conditional with #define.

See Also

The #**define** statement and **Conditional Compilation** on page IV-108 for more usage details.

#include

```
#include "file spec" or <file spec>
```

A #include statement in a procedure file automatically opens another procedure file. You should use #include in any procedure file that you write if it requires that another procedure file be open. A #include statement must always appear flush against the left margin in a procedure window.

Parameters

The ".ipf" extension must be omitted from both "*file spec*" and <*file spec*>.

The <*file spec*> syntax is used to include a WaveMetrics procedure file in "Igor Pro Folder/WaveMetrics Procedures". *file spec* is the name of the procedure file without the extension.

The "*file spec*" syntax is used to include a user procedure file typically in "Igor Pro User Files/User Procedures". If *file spec* is the name of or a partial path to the file, Igor interprets it as relative to the "User Procedures" folder. If *file spec* is a full path then it specifies the exact path to the file without the extension.

See Also

The Include Statement on page IV-166 for usage details.

Igor Pro User Files on page II-31.

#pragma

```
#pragma pragmaName = value
```

#pragma introduces a compiler directive, which is a message to the Igor procedure compiler. A #pragma statement must always appear flush against the left margin in a procedure window.

Igor ignores unknown pragmas such as pragmas introduced in later versions of the program.

Currently Igor supports the following pragmas:

```
#pragma rtGlobals = value
#pragma version = versionNumber
#pragma IgorVersion = versionNumber
#pragma hide = value
#pragma ModuleName = name
#pragma IndependentModule = name
#pragma rtFunctionErrors = value
#pragma TextEncoding = "textEncodingName"  // Igor Pro 7.00
#pragma DefaultTab = {<mode>,<width in points>,<width in spaces>} // Igor Pro 9.00
```

See Also

Pragmas on page IV-52

The rtGlobals Pragma on page IV-52

The version Pragma on page IV-54

The IgorVersion Pragma on page IV-54

The hide Pragma on page IV-54

The ModuleName Pragma on page IV-54

The IndependentModule Pragma on page IV-55

The rtFunctionErrors Pragma on page IV-55

The TextEncoding Pragma on page IV-55

The Default Tab Pragma For Procedure Files on page III-406

#undef

`#undef` ***`symbol`***

A #undef statement removes a nonglobal *symbol* created previously by #define. See **Conditional Compilation** on page IV-108 for information on undefining a global *symbol.*

See Also

The #**define** statement and **Conditional Compilation** on page IV-108 for more usage details.

Abort

`Abort` `[`***`errorMessageStr`***`]`

The Abort operation aborts procedure execution.

Parameters

The optional *errorMessageStr* is a string expression, which, if present, specifies the message to be displayed in the error alert.

Details

Abort provides a way for a procedure to abort execution when it runs into an error condition.

See Also

Aborting Functions on page IV-112 , **Aborting Macros** on page IV-124, and **Flow Control for Aborts** on page IV-48. The **DoAlert** operation.

AbortOnRTE

`AbortOnRTE`

The AbortOnRTE flow control keyword raises an abort when a runtime error has occurred in a user-defined function.

AbortOnRTE should be used after a command that might give rise to a runtime error.

You can place AbortOnRTE immediately after a command that might give rise to a runtime error that you want to handle instead of allowing Igor to handle it by halting procedure execution. Use a **try-catch-endtry** block to catch the abort, if it occurs.

AbortOnRTE has very low overhead and should not significantly slow program execution.

Details

In terms of programming style, you should consider using AbortOnRTE (preceded by a semicolon) on the same line as the command that may give rise to an abort condition.

When using AbortOnRTE after a related sequence of commands, then it should be placed on its own line.

When used with try-catch-endtry, you should place a call to **GetRTError**(1) in your catch section to clear the runtime error.

Example

Abort if the wave does not exist:

```
WAVE someWave; AbortOnRTE
```

See Also

Flow Control for Aborts on page IV-48 and **AbortOnRTE Keyword** on page IV-49 for further details.

The **try-catch-endtry** flow control statement.

AbortOnValue

`AbortOnValue` ***`abortCondition`*****`,`** ***`abortCode`***

The AbortOnValue flow control keyword will abort function execution when the *abortCondition* is nonzero and it will then return the numeric *abortCode*. No dialog will be displayed when such an abort occurs.

Parameters

abortCondition can be any valid numeric expression using comparison or logical operators.

abortCode is a nonzero numeric value returned to any abort or error handling code by AbortOnValue whenever it causes an abort.

Details

When used with try-catch-endtry, you should place a call to **GetRTError**(1) in your catch section to clear the runtime error.

See Also

Flow Control for Aborts on page IV-48 and **AbortOnValue Keyword** on page IV-49 for further details.

The **AbortOnRTE** keyword and the **try-catch-endtry** flow control statement.

abs

`abs(`***`num`*****`)`**

The abs function returns the absolute value of the real number *num*. To calculate the absolute value of a complex number, use the cabs function.

See Also

The **cabs** function.

acos

`acos(`***`num`*****`)`**

The acos function returns the inverse cosine of *num* in radians in the range [0,π].

In complex expressions, *num* is complex and acos returns a complex value.

See Also

cos

acosh

`acosh(`***`num`*****`)`**

The acosh function returns the inverse hyperbolic cosine of *num*. In complex expressions, *num* is complex and acosh returns a complex value.

AddFIFOData

`AddFIFOData` *`FIFOName`*`,` *`FIFO_channelExpr`* `[,` *`FIFO_channelExpr`*`]`...

The AddFIFOData operation evaluates *FIFO_channelExpr* expressions as double precision floating point and places the resulting values into the named FIFO.

Details

There must be one *FIFO_channelExpr* for each channel in the FIFO.

See Also

FIFOs are used for data acquisition. See **FIFOs and Charts** on page IV-313.

Other operations used with FIFOs: **NewFIFO**, **NewFIFOChan**, **CtrlFIFO**, and **FIFOStatus**.

AddFIFOVectData

`AddFIFOVectData` *`FIFOName`*`,` *`FIFO_channelKeyExpr`* `[,` *`FIFO_channelKeyExpr`*`]`...

The AddFIFOVectData operation is similar to AddFIFOData except the expressions use a keyword to allow either a single numeric value for a normal channel or a wave containing the data for a special image vector channel.

Details

There must be one *FIFO_channelKeyExpr* for each channel in the FIFO.

A *FIFO_channelKeyExpr* may be one of:

```
num = numericExpression
vect = wave
```

For best results, the wave should have the same number of points as used to define the FIFO channel and the same number type. See the **NewFIFOChan** operation.

See Also

FIFOs and Charts on page IV-313.

AddListItem

`AddListItem(`*`itemStr`*`,` *`listStr`* `[,` *`listSepStr`* `[,` *`itemNum`*`]])`

The AddListItem function returns *listStr* after adding *itemStr* to it. *listStr* should contain items separated by *listSepStr*, such as "abc;def;".

Use AddListItem to add an item to a string containing a list of items separated by a string (usually a single ASCII character), such as those returned by functions like **TraceNameList** or **AnnotationList**, or to a line from a delimited text file.

listSepStr and *itemNum* are optional; their defaults are ";" and 0, respectively.

Details

By default *itemStr* is added to the start of the list. Use the optional list index *itemNum* to add *itemStr* at a different location. The returned list will have *itemStr* at the index *itemNum* or at ItemsInList(*returnedListStr*)-1 when *itemNum* equals or exceeds ItemsInList(*listStr*).

itemNum can be any value from -infinity (`-Inf`) to infinity (`Inf`). Values from -infinity to 0 prepend *itemStr* to the list, and values from ItemsInList(*listStr*) to infinity append *itemStr* to the list.

itemStr may be `""`, in which case an empty item (consisting of only a separator) is added.

If *listSepStr* is `""`, then *listStr* is returned unchanged (unless *listStr* contains only list separators, in which case an empty string (`""`) is returned).

listStr is treated as if it ends with a *listSepStr* even if it doesn't.

In Igor6, only the first byte of *listSepStr* was used. In Igor7 and later, all bytes are used.

Examples

```
Print AddListItem("hello","kitty;cat;")        // prints "hello;kitty;cat;"
Print AddListItem("z", "b,c,", ",", 1)         // prints "b,z,c,"
Print AddListItem("z", "b,c,", ",", 999)       // prints "b,c,z,"
Print AddListItem("z", "b,c,", ",", Inf)       // prints "b,c,z,"
Print AddListItem("", "b-c-", "-")             // prints "-b-c-"
```

See Also

The **FindListItem**, **FunctionList**, **ItemsInList**, **RemoveByKey**, **RemoveFromList**, **RemoveListItem**, **StringFromList**, **StringList**, **TraceNameList**, **VariableList**, and **WaveList** functions.

AddMovieAudio

`AddMovieAudio` ***`soundWave`***

The AddMovieAudio operation adds audio samples to the audio track of the currently open movie.

In Igor Pro 7.00 and later, this operation is not supported on Macintosh.

Parameters

soundWave contains audio samples with an amplitude from -128 to +127 and with the same time scale as the prototype *soundWave* used to open the movie.

Flags

/Z	Suppresses error reporting. If you use /Z, check the V_Flag output variable to see if the operation succeeded.

Details

You can create movies with 16-bit and stereo sound by providing a sound wave in the appropriate format. To specify 16-bit sound, the wave type must be signed 16-bit integer (/W flag in **Make** or **Redimension**). To specify stereo, use a wave with two columns (or any other number of channels as desired).

Output Variables

`V_Flag`	Set to 0 if the operation succeeded or to a non-zero error code. V_Flag is set only if you use the /Z flag.

See Also

Movies on page IV-245, **NewMovie**, **AddMovieFrame**

AddMovieFrame

`AddMovieFrame [/PICT=`*`pictName`*`]`

The AddMovieFrame operation adds the top graph, page layout, Gizmo window, or the specified picture to the currently open movie.

Support for page layout and Gizmo windows was added in Igor Pro 7.00.

When you write a procedure to generate a movie, you need to call the **DoUpdate** operation after all modifications to the target window and before calling AddMovieFrame. This allows Igor to process any changes you have made to the window.

In Igor7 or later, the target window at the time you call NewMovie is remembered and is used by AddMovieFrame even if it is not the target window when you call AddMovieFrame.

If the /PICT flag is provided, then the specified picture from the picture gallery (see **Pictures** on page III-509) is used in place of the target window.

Flags

/Z	Suppresses error reporting. If you use /Z, check the V_Flag output variable to see if the operation succeeded.

Output Variables

`V_Flag`	Set to 0 if the operation succeeded or to a non-zero error code. V_Flag is set only if you use the /Z flag.

See Also

Movies on page IV-245, **NewMovie**, **AddMovieAudio**

AddWavesToBoxPlot

```
AddWavesToBoxPlot [/W=winName /T=traceName /INST=traceInstance] wave [, wave ]
    . . .
```

Adds additional 1D waves to a pre-existing box plot trace created by AppendBoxPlot.

AddWavesToBoxPlot was added in Igor Pro 8.00.

Because a box plot trace may require a number of waves to define each data set in the trace, and because wave names may be quite long, the AddWavesToBoxPlot operation is provided to add waves to a list begun by AppendBoxPlot.

Flags

/T=*traceName*

/INST=*traceInstance* These flags specify the name and instance number of an existing box plot trace to which waves will be added. You can use /T without /INST, in which case a trace with instance number zero will be used. Do not use /INST without /T.

See **Creating Graphs** on page II-277 for information about trace names and trace instance numbers.

In the absence of both /T and /INST, the default is to use the top box plot trace found on the graph. That would be the most recently added box plot trace.

/W=*winName* Appends to the named graph window or subwindow. When omitted, AddWavesToBoxPlot operates on the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

If your original AppendBoxPlot command included an X wave, the total number of waves in the list of box plot data set waves cannot be greater than the number of points in your X wave.

If the box plot trace is defined by a multicolumn wave, you cannot add additional waves using this operation.

See Also

Box Plots on page II-331, **AppendBoxPlot**, **ModifyBoxPlot**

AddWavesToViolinPlot

```
AddWavesToViolinPlot [/W=winName /T=traceName /INST=traceInstance] wave [, wave
    ] . . .
```

Adds additional 1D waves to a pre-existing violin plot trace created by AppendViolinPlot.

AddWavesToViolinPlot was added in Igor Pro 8.00.

Because a violin plot trace may require a number of waves to define each data set in the trace, and because wave names may be quite long, the AddWavesToViolinPlot operation is provided to add waves to a list begun by AppendViolinPlot.

Flags

/T=*traceName*

/INST=*traceInstance*	These flags specify the name and instance number of an existing violin plot trace to which waves will be added. You can use /T without /INST, in which case a trace with instance number zero will be used. Do not use /INST without /T. See **Creating Graphs** on page II-277 for information about trace names and trace instance numbers. In the absence of both /T and /INST, the default is to use the top violin plot trace found on the graph. That would be the most recently added violin plot trace.
/W=*winName*	Appends to the named graph window or subwindow. When omitted, AppendViolinPlot operates on the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

If your original AppendViolinPlot command included an X wave, the total number of waves in the violin plot trace cannot be greater than the number of points in your X wave.

If the violin plot trace is defined by a multicolumn wave, you cannot add additional waves using this operation.

See Also

Violin Plots on page II-337, **AppendViolinPlot**, **ModifyViolinPlot**

AdoptFiles

AdoptFiles [***flags***]

The AdoptFiles operation adopts external files and waves into the current experiment.

When the experiment is next saved, the files and waves are saved in the experiment file for a packed experiment or in the experiment folder for an unpacked experiment. References to the external files are eliminated.

AdoptFiles cannot be called from a function except via Execute/P.

Flags

/A	Adopts all external notebooks and user procedure files and all waves in the experiment. WaveMetrics Procedure files are not adopted. /A is equivalent to /NB/UP/DF.
/DF	Adopts all waves saved external to the experiment.
/DF=*dataFolderPathStr*	Adopts all waves saved external to the experiment that are in the specified data folder.
/I	Shows the Adopt All dialog and adopts what the user selects there.
/NB	Adopts all external notebook files.
/UP	Adopts all external user procedure files.

/W=*winTitleOrName*	Adopts the specified notebook or procedure file. /W was added in Igor Pro 7.02.
	winTitleOrName is a name, not a string, so you construct /W like this:
	`/W=$"New Polar Graph.ipf"`
	or:
	`/W=Notebook0`
	When working with independent modules, *winTitleOrName* is a procedure window title followed by a space and, in brackets, an independent module name. See **Independent Modules** on page IV-238 for details.

/WP	Adopts all WaveMetrics Procedure procedure files.

/WV=*wave*	Adopts only the specified wave.

Details

Only files and waves saved external to the current experiment are adopted. See **References to Files and Folders** on page II-24 for a discussion of such standalone files.

The number of objects actually adopted is returned in V_Flag.

To adopt just one wave, use:

```
AdoptFiles/WV=wave
```

To adopt just one notebook or procedure window use AdoptFiles/W=*winTitleOrName*.

Command Line and Macro Examples

```
// Using AdoptFiles from the command line or from a macro
AdoptFiles/I                  // Show the Adopt All dialog.
AdoptFiles/A/WP               // Adopt everything that can be adopted.
AdoptFiles/DF/NB/UP/WP        // Adopt everything that can be adopted.
AdoptFiles/DF=root:subfolder  // Adopt any externally saved waves in root:subfolder.
AdoptFiles/W=$"Proc0.ipf"     // Adopt Proc0.ipf if it is saved externally.
AdoptFiles/WV=GetWavesDataFolder(wave0,2)   // Adopt wave0 if it is saved externally.
```

Function Examples

```
// Using AdoptFiles from a user-defined function - you must use Execute/P
Execute/P "AdoptFiles/A"         // Schedule adoption of all user files and waves
Execute/P "AdoptFiles/WV="+GetWavesDataFolder(w,2)    // Schedule adoption of wave w
```

See Also

Adopt All on page II-25, **Adopting Notebook and Procedure Files** on page II-25, **Avoiding Shared Igor Binary Wave Files** on page II-24, **Operation Queue** on page IV-278.

airyA

`airyA(`*`x`*`[, `*`accuracy`*`])`

The airyA function returns the value of the Airy $Ai(x)$ function:

$$Ai(x) = \frac{1}{\pi}\sqrt{\frac{x}{3}}K_{1/3}\left(\frac{2}{3}x^{3/2}\right),$$

where K is the modified Bessel function.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **airyAD** and **airyB** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

airyAD

`airyAD(`*`x`*`[, `*`accuracy`*`])`

The airyAD function returns the value of the derivative of the Airy function.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **airyA** function.

airyB

airyB(*x* [, *accuracy*])

The airyB function returns the value of the Airy $Bi(x)$ function:

$$Bi(x)=\sqrt{\frac{x}{3}}\left[I_{-1/3}\left(\frac{2}{3}x^{3/2}\right)+I_{1/3}\left(\frac{2}{3}x^{3/2}\right)\right],$$

where I is the modified Bessel function.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **airyBD** and **airyA** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

airyBD

airyBD(*x* [, *accuracy*])

The airyBD function returns the value of the derivative $Bi'(x)$ of the Airy function.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **airyB** function.

alog

alog(*num*)

The alog function returns 10^{num}.

AnnotationInfo

AnnotationInfo(*winNameStr*, *annotationNameStr* [, *options*])

The AnnotationInfo function returns a string containing a semicolon-separated list of information about the named annotation in the named graph or page layout window or subwindow.

The main purpose of AnnotationInfo is to use a tag or textbox as an input mechanism to a procedure. This is illustrated in the “Tags as Markers Demo” sample experiment, which includes handy utility functions (supplied by AnnotationInfo Procs.ipf).

Parameters

winNameStr can be `""` to refer to the top graph or layout window or subwindow.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

options is an optional parameter that controls the text formatting in the annotation output. The default value is 0.

Omit *options* or use 0 for *options* to escape the returned annotation text, which is appropriate for printing the output to the history or for using the text in an Execute operation.

Use 1 for *options* to not escape the returned annotation text because you intend to extract the text for use in a subsequent command such as Textbox or Tag.

Details

The string contains thirteen pieces of information. The first twelve pieces are prefaced by a keyword and colon and terminated with a semicolon. The last piece is the annotation text, which is prefaced with a keyword and a colon but is not terminated with a semicolon.

Keyword	Information Following Keyword
ABSX	X location, in points, of the anchor point of the annotation. For graphs, this is relative to the top-left corner of the graph window. For layouts, it is relative to the top-left corner of the page.
ABSY	Y location, in points, of the anchor point of the annotation. For graphs, this is relative to the top-left corner of the graph window. For layouts, it is relative to the top-left corner of the page.
ATTACHX	For tags, it is the X value of the wave at the point where the tag is attached, as specified with the Tag operation. For textboxes, color scales, and legends, this will be zero and has no meaning.
AXISX	X location of the anchor point of the annotation. For tags or color scales in graphs, it is in terms of the X axis against which the tagged wave is plotted. For textboxes and legends in graphs, it is in terms of the first X axis. For layouts, this has no meaning and is always zero.
AXISY	Y location of the anchor point of the annotation. For layouts, this has no meaning and is always zero. For tags or color scales in graphs, it is in terms of the Y axis against which the tagged wave is plotted. For textboxes and legends in graphs, it is in terms of the first Y axis.
AXISZ	Z value of the image or contour level trace to which the tag is attached or NaN if the trace is not a contour level trace or the annotation is not a tag.
COLORSCALE	Parameters used in a ColorScale operation to create the annotation.
FLAGS	Flags used in a Tag, Textbox, ColorScale, or Legend operation to create the annotation.
RECT	The outermost corners of the annotation (values are in points): RECT:*left*, *top*, *right*, *bottom*
TEXT	Text that defines the contents of the annotation or the main axis label of a color scale.
TYPE	Annotation type: “Tag”, “TextBox”, “ColorScale”, or “Legend”.
XWAVE	For tags, it is the name of the X wave in the XY pair to which the tag is attached. If the tag is attached to a single wave rather than an XY pair, this will be empty. For textboxes, color scales, and legends, this will be empty and has no meaning.
XWAVEDF	For tags, the full path to the data folder containing the X wave associated with the trace to which the tag is attached. For textboxes, color scales, and legends, this will be empty and has no meaning.
YWAVE	For tags, it is the name of the trace or image to which the tag is attached. See **ModifyGraph (traces)** and **Instance Notation** on page IV-20 for discussions of trace names and instance notation. For color scales, it is the name of the wave displayed in associated the contour plot, image plot, f(z) trace, or the name of the color scale’s cindex wave. For textboxes and legends, this will be empty and has no meaning.
YWAVEDF	Full path to the data folder containing the Y wave or blank if the annotation is not a tag or color scale.

AnnotationList

`AnnotationList(`*`winNameStr`*`)`

The AnnotationList function returns a semicolon-separated list of annotation names from the named graph or page layout window or subwindow.

Parameters

winNameStr can be `""` to refer to the top graph or layout window.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

APMath

APMath [*flags*] *destStr* = *Expression*

The APMath operation provides arbitrary precision calculation of basic mathematical expressions. It converts the final result into the assigned string *destStr*, which can then be printed or used to represent a value (at the given precision) in another APMath operation.

Parameters

destStr	Specifies a destination string for the assignment expression. If *destStr* is not an existing variable, it is created by the operation. When executing in a function, *destStr* will be a local variable if it does not already exist.
Expression	Algebraic expression containing constants, local, global, and reference variables or strings, as well as wave elements together with the operators shown below.

APMath Operators

+	Scalar addition	Lowest precedence
-	Scalar subtraction	Lowest precedence
*	Scalar multiplication	Medium precedence
/	Scalar division	Medium precedence
^	Exponentiation	Highest precedence

APMath Functions on Scalar Parameters

The following functions are supported for arbitrary precision math on scalar parameters:

sqrt(*x*)	Square root of *x*.
cbrt(*x*)	Cube root of *x*.
pi	Value of π (without parentheses).
sin(*x*)	Sine of *x*.
cos(*x*)	Cosine of *x*.
tan(*x*)	Tangent of *x*.
asin(*x*)	Inverse sine of *x*.
acos(*x*)	Inverse cosine of *x*.
atan(x)	Inverse tangent of *x*.
atan2(*y*,*x*)	Inverse tangent of *y*/*x*.
log(*x*)	Logarithm of *x*.
log10(*x*)	Logarithm based 10 of *x*.
exp(*x*)	Exponential function e^*x*.
pow(*x*,*n*)	*x* to the power *n* (*n* not necessarily integer).
sinh(*x*)	Hyperbolic sine of *x*.
cosh(*x*)	Hyperbolic cosine of *x*.
tanh(*x*)	Hyperbolic tangent of *x*.
asinh(*x*)	Inverse hyperbolic sine of *x*.
acosh(*x*)	Inverse hyperbolic cosine of *x*.
atanh(*x*)	Inverse hyperbolic tangent of *x*.

`ceil(x)` Smallest integer larger than *x*.

`comp(x,y)` Returns 0 for *x* == *y*, 1 if *x* > *y* and -1 if *y* > *x*.

`factorial(n)` Factorial of integer *n*.

`floor(x)` Greatest integer smaller than *x*.

`gcd(x,y)` Greatest common divisor of *x* and *y*.

`lcd(x,y)` Lowest common denominator of *x* and *y* (given by *x***y*/gcd(*x*,*y*).

`sgn(x)` Sign of *x* or zero if *x* == 0.

`binomial(n,k)` Returns the the binomial function for integers n and k.

`Bernoulli(n)` Returns the Bernoulli number Bn (with Bn(1)=-1/2).

`Stirling2(n,k)` Returns the Stirling number of the second kind.

APMath Functions on Wave Parameters

The following functions are supported for arbitrary precision math on waves.

These restrictions apply to all of these APMath functions on waves:

1. The parameter *w* must be a simple wave reference. It can not be a data folder path to a wave or a $ expression pointing to a wave.
2. The wave can be a real numeric wave or a text wave containing arbitrary precision numbers in string form.
3. Complex waves are not allowed.
4. 64-bit integer waves are not allowed.
5. The functions return an error if the wave contains NaNs or INFs.
6. Multidimensional waves are treated as 1D.

`kurtosis(w)` Returns the kurtosis of the entire wave *w*. See **WaveStats** for a discussion of kurtosis. The kurtosis function was added in Igor Pro 8.00.

`mean(w)` Returns the mean of the entire wave *w*. The mean function was added in Igor Pro 8.00.

`skew(w)` Returns the skewness of the entire wave *w*. See **WaveStats** for a discussion of skewness. The skew function was added in Igor Pro 8.00.

`sum(w)` Returns the sum of the entire wave *w*. The sum function was added in Igor Pro 8.00.

`variance(w)` Returns the variance of the entire wave *w*. See **WaveStats** for a discussion of variance. The variance function was added in Igor Pro 8.00.

Flags

/EX=*exDigits* Specifies the number of extra digits added to the precision digits (/N) for intermediate steps in the calculation.

/N=*numDigits* Specifies the precision of the final result. To add digits to the intermediate computation steps, use /EX.

/V Verbose mode; prints the result in the history in addition to performing the assignment.

/Z No error reporting.

Details

By default, all arbitrary precision math calculations are performed with *numDigits*=50 and *exDigits*=6, which yields a final result using at least 56 decimal places. Because none of the built-in variable types can express numbers with such high accuracy, the arbitrary precision numbers must be stored as strings. The operation automatically converts between strings and constants. It evaluates all of the numerical functions listed above using the specified accuracy. If you need functions that are not supported by this operation, you may have to precompute them and store the results in a local variable.

The operation stores the result in *destStr*, which may or may not exist prior to execution. When you execute the operation from the command line, *destStr* becomes a global string in the current data folder if it does not already exist. If it exists, then the result of the operation overwrites its value (as with any normal string assignment). In a user function, *destStr* can be a local string, an SVAR, or a string passed by reference. If *destStr* is not one of these then the operation creates a local string by that name.

Arbitrary precision math calculations are much slower (by a factor of about 300) than equivalent floating point calculations. Execution time is a function of the number of digits, so you should use the /N flag to limit the evaluation to the minimum number of required digits.

Output Variables

In Igor Pro 8.00 or later, the APMath operation returns information in the following output variables:

`V_Flag`	Set to 0 if the operation succeeds or to a non-zero error code.
`V_Value`	Set to a double-precision representation of the output string.

Examples

Evaluate pi to 50 digits:

```
APMath/V aa = pi
```

Evaluate ratios of large factorials:

```
APMath/V aa = factorial(500)/factorial(499)
```

Evaluate ratios of large exponentials:

```
APMath/V aa = exp(-1000)/exp(-1001)
```

Division of mixed size values:

```
APMath/V aa = 1-sgn(1-(1-0.00000000000000000001234)/(1-0.000000000000000000012345)))
```

you'll get a different result trying to evaluate this using double precision.

Difference between built-in pi and the arbitrary precision pi:

```
Variable/G biPi = pi
APMath/V aa = biPi-pi
```

Precision control:

```
Function test()
   APMath aa = pi                // Assign 50 digit pi to the string aa.
   APMath/V bb = aa              // Create local string bb equal to aa.
   APMath/V bb = aa-pi           // Subtract arb. prec. pi from aa.
                                 // note the default exDigits=6.
   APMath/V/N=50/ex=0 bb = aa-pi // setting exDigits=0.
End
```

Numerical recreation:

```
APMath/V/N=16 aa = 111111111^2
```

The solution for the sum of three cubes problem for the number 42:

```
APMath/V aa = pow((-80538738812075974),3) + pow(80435758145817515,3) +
   pow(12602123297335631,3)
```

Append

`Append`

The Append operation is interpreted as **AppendToGraph**, **AppendToTable**, or **AppendToLayout**, depending on the target window. This does not work when executing a user-defined function. Therefore we now recommend that you use **AppendToGraph**, **AppendToTable**, or **AppendLayoutObject** rather than Append.

AppendBoxPlot

```
AppendBoxPlot [ flags ] wave[, wave, ...] [vs xWave]
```

The AppendBoxPlot operation appends a box plot trace to the target or named graph. A box plot is a way to display a summary of the distribution of data values. Another way to display a summary of data distribution is via a violin plot.

AppendBoxPlot was added in Igor Pro 8.00.

A box plot trace is treated within Igor as a single graph trace, and many of the operations such as removing from a graph or reordering traces work the same with a box plot trace as with any other graph trace.

There is no DisplayBoxPlot operation. Use Display followed by AppendBoxPlot.

Parameters

The data for a single box in a box plot trace comes from either one entire 1D wave or from a single column of a multi-column wave. The number of box plots in the trace is determined by either the number of 1D waves in the list of waves, or by the number of columns in the single multi-column wave. It is not permitted to mix 1D and multi-column waves.

You can list up to 100 individual 1D waves. If you want more boxes than 100 in a box plot trace you must use a multi-column wave or add to the list using the AddWavesToBoxPlot operation.

If you do not provide *xWave*, each box plot is positioned on a numeric axis at X=0, 1, etc. If the data is in a multi-column wave, positioning comes by default from the X scaling of the matrix wave. Providing a numeric *xWave* allows you to position each box plot at an arbitrary position on the X axis. A text *xWave* displays the box plots using a category axis. If you use the /CATL flag with a multi-column wave, the result is a category X axis using the dimension labels as the category labels. See **Category Plots** on page II-355 and **Dimension Labels** on page II-93.

xWave must have at least as many points as you have 1D Y waves or columns in a multi-column wave. Because a list of waves may become too long for the command line, you can use an *xWave* that is longer than the list. Use **AddWavesToBoxPlot** to complete the list.

Flags

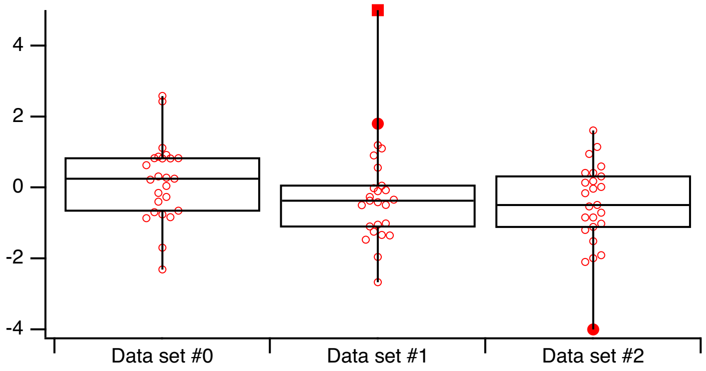
/L/R/B/T	These axis flags are the same as used by **AppendToGraph**.
/CATL[=*doCatLabels*]	Use the column dimension labels to produce box plots on a category axis using the column dimension labels as the category labels. *doCatLabels* is 0 or 1 and /CATL is equivalent to /CATL=1.
/TN=*traceName*	Allows you to provide a custom name for a trace. This is useful when displaying waves with the same name but from different data folders. See **User-defined Trace Names** on page IV-89 for details, except that the /TN flag for AppendBoxPlot comes in the normal position in the command, after the command name.
/VERT[=*doVert*]	Arranges the individual box plots vertically along the Y axis. *doVert* is 0 or 1 and /VERT is equivalent to /VERT=1. /VERT is similar to ModifyGraph SwapXY but on a trace-by-trace basis. To make a box plot with horizontal boxes, use either ModifyGraph SwapXY or AppendBoxPlot/VERT.
/W=*winName*	Appends to the named graph window or subwindow. When omitted, AppendBoxPlot is directed to the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with winName, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

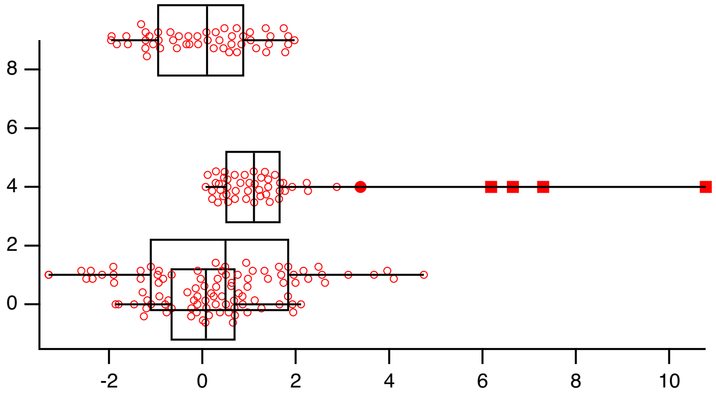
Some aspects of the overall appearance of a box plot trace can be set using ModifyGraph. By default, the line color is black and markers for non-outlier data points are hollow circles 0.7 times the size of the normal trace marker. Using the Modify Box Plot dialog or the ModifyBoxPlot operation, you can choose to use trace color, line size, line dash style, marker and marker size as set by ModifyGraph just as for a regular trace. Detailed control of these characteristics for various parts of a box plot trace is provided by **ModifyBoxPlot**. Markers to represent outliers, far outliers, and non-outliers override the marker set by ModifyGraph.

Examples

```
// Demo vertical box plot
Make/O/N=(25,3) multicol                // A three-column wave with 25 rows
SetRandomSeed(.4)
multicol = gnoise(1)                    // Three normally-distributed datasets
multicol[20][1] = 5                     // A "far" outlier
multicol[13][2] = -4                    // An outlier
Make/O/N=3/T labels                     // A text wave to make a category plot
labels = "Dataset #"+num2str(p)         // Labels for the X axis of a category plot
Display; AppendBoxPlot multicol vs labels
```



```
// Demo horizontal box plot
Make/O/N=50 ds1, ds2, ds3, ds4
Make/O/N=4 dsX
ds1 = gnoise(1)
ds2 = gnoise(2)
ds3 = logNormalNoise(0, 1)
ds4 = enoise(2)
dsX = p^2
Display; AppendBoxPlot ds1,ds2,ds3,ds4 vs dsX
ModifyGraph swapXY=1                    // horizontal boxes
ModifyGraph margin(top)=20              // top margin may be too small
```



See Also

Display, **AppendToGraph**, **ModifyGraph (traces)**, **ModifyBoxPlot**

Box Plots on page II-331, **Violin Plots** on page II-337

AppendImage

AppendImage [**/G=*g*/W=*winName***][***axisFlags***] ***matrix*** [**vs** {***xWaveName***, ***yWaveName***}]

The AppendImage operation appends the matrix as an image to the target or named graph. By default the image is plotted versus the left and bottom axes.

Parameters

matrix is either an NxM 2D wave for false color or indexed color images, or it can be a 3D NxMx3 wave containing a layer of data for red, a layer for green and a layer for blue. It can also be a 3D NxMx4 wave with the fourth plane containing alpha values.

If *matrix* contains multiple planes other than three or four or if it contains three or four and multiple chunks, the **ModifyImage** plane keyword can be used to specify the desired subset to display.

If you provide *xWaveName* and *yWaveName*, *xWaveName* provides X coordinate values, and *yWaveName* provides Y coordinate values. This makes an image with uneven pixel sizes. In both cases, you can use * to specify calculated values based on the dimension scaling of *matrix*. See **Details** if you use *xWaveName* or *yWaveName*.

Flags

axisFlags Flags /L, /R, /B, and /T are the same as used by **AppendToGraph**.

/G=*g* Controls the interpretation of three-plane images as direct RGB.

g=1 Suppresses the auto-detection of three or four plane images as direct (RGB) color.

g=0 Same as no /G flag (default).

/W=*winName* Appends to the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

When appending an image to a graph each image data point is displayed as a rectangle. You can supply optional X and Y waves to define the coordinates of the rectangle edges. *These waves need to contain one more data point than the X (row) or Y (column) dimension of the matrix*. The waves must also be either strictly increasing or strictly decreasing. See **Image X and Y Coordinates** on page II-388 for details.

For false color, the values in the matrix are linearly mapped into a color table. See the ModifyImage ctab keyword. For indexed color, the values in the matrix are interpreted as Z values to be looked up in a user-supplied 3 column matrix of colors. See the ModifyImage cindex keyword. Direct color NxMx3 waves contain the actual red, green, and blue values for each pixel. NxMx4 waves add an alpha channel. If the number type is unsigned bytes, then the range of intensity ranges from 0 to 255. For all other number types, the intensity ranges from 0 to 65535.

By default, nondirect color matrices are initially displayed as false color using the Grays color table and autoscale mode.

If the matrix is complex, the image is displayed in terms of the magnitude of the Z value, that is, sqrt(real^2 + imag^2).

See Also

Image X and Y Coordinates on page II-388, **Color Blending** on page III-498.

The **NewImage**, **ModifyImage**, and **RemoveImage** operations. For general information on image plots see Chapter II-16, **Image Plots**.

AppendLayoutObject

```
AppendLayoutObject [flags] objectType objectName
```

The AppendLayoutObject operation appends a single object to the top layout or to the layout specified via the /W flag. It targets the active page or the page specified by the /PAGE flag.

Unlike the AppendToLayout operation, AppendLayoutObject can be used in user-defined functions. Therefore, AppendLayoutObject should be used in new programming instead of AppendToLayout.

Parameters

objectType identifies the type of object to be appended. It is one of the following keywords: `graph`, `table`, `picture`, `gizmo`.

objectName is the name of the graph, table, picture or Gizmo window to be appended.

Use a space between *objectType* and *objectName*. A comma is not allowed.

Flags

/D=*fidelity*
Draws layout objects in low fidelity (*fidelity*=0) or high fidelity (*fidelity*=1; default). This affects drawing on the screen only, not exporting or printing. Low fidelity is somewhat faster but less accurate and should be used only for graphs that take a very long time to draw.

/F=*frame*
Specifies the type of frame enclosing the object.

frame =1	Single frame (default).
frame =2	Double frame.
frame =3	Triple frame.
frame =4	Shadow frame.

/T=*trans*
Sets the transparency of the object background to opaque (*trans* =0; default) or transparent (*trans* =1).

For transparency to be effective, the object itself must also be transparent. Annotations have their own transparent/opaque settings. Graphs are transparent only if their backgrounds are white. PICTs may have been created transparent or opaque. Opaque PICTs cannot be made transparent.

/R=(*l*, *t*, *r*, *b*)
Sets the size and position of the object. If omitted, the object is placed with a default size and position. *l*, *t*, *r*, and *b* are the left, top, right, and bottom coordinates of the object, respectively. Coordinates are expressed in units of points, relative to the top/left corner of the paper.

/PAGE=*page*
Appends the object to the specified page.

Page numbers start from 1. To target the active page, omit /PAGE or use *page*=0.

The /PAGE flag was added in Igor Pro 7.00.

/W=*winName*
Appends the object to the named page layout window. If /W is omitted or if *winName* is `$""`, the top page layout is used.

See Also

NewLayout, **ModifyLayout**, **LayoutPageAction**, **RemoveLayoutObjects**, **TextBox**, **Legend**

AppendMatrixContour

```
AppendMatrixContour [axisFlags][/F=formatStr /W=winName] zWave
   [vs {xWave, yWave}]
```

The AppendMatrixContour operation appends to the target or named graph a contour plot of a matrix of z values with autoscaled contour levels, using the Rainbow color table.

Note: There is no DisplayContour operation. Use `Display; AppendMatrixContour`.

Parameters

zWave must be a matrix (2D wave).

To contour a set of XYZ triplets, use **AppendXYZContour**.

If you provide the *xWave* and *yWave* specification, *xWave* provides X values for the rows, and *yWave* provides Y values for the columns. This results in an “uneven grid” of Z values.

If you omit the *xWave* and *yWave* specification, Igor uses the *zWave*’s X and Y scaled indices as the X and Y values. Igor also uses the *zWave*’s scaled indices if you use * (asterisk symbol) in place of *xWave* or *yWave*.

In a macro, to modify the appearance of contour levels before the contour is calculated and displayed with the default values, append "`;DelayUpdate`" and immediately follow the AppendMatrixContour command with the appropriate **ModifyContour** commands. All but the last ModifyContour command should also have `;DelayUpdate` appended. DelayUpdate is not needed in a function, but DoUpdate is useful in a function to force the contour traces to be built immediately rather than the default behavior of waiting until all functions have completed.

On the command line, the Display command and subsequent AppendMatrixContour commands and any ModifyContour commands can be typed all on one line with semicolons between:

```
Display; AppendMatrixContour MyMatrix; ModifyContour ...
```

Flags

axisFlags Flags /L, /R, /B, /T are the same as used by **AppendToGraph**.

/F=*formatStr* Determines the names assigned to the contour level traces. See **Details**.

/W=*winName* Appends to the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

AppendMatrixContour creates and displays contour level traces. You can modify these all together using the Modify Contour Appearance dialog or individually using the Modify Trace Appearance dialog. In most cases, you will not need to modify the individual traces.

By default, Contour level traces are automatically named with names that show the *zWave* and the contour level, for example, “zWave=1.5”. You will see these trace names in the Modify Trace Appearance dialog and in Legends. In most cases, the default trace names will be just fine.

If you want to control the names of the contour level traces (which you might want to do for names in a Legend), use the /F=*formatStr* flag. This flag uses a format string as described for the **printf** operation. The default format string is `"%.17s=%g"`, resulting in trace names such as “zWave=1.5”. *formatStr* must contain at least `%f` or `%g` (used to insert the contour level) or `%d` (used to insert the zero-based index of the contour level). Include `%s`, to insert the *zWave* name.

Here are some examples of format strings.

formatStr	**Examples of Resulting Name**	**Format**
`"%g"`	“100”, “1e6”, “-2.05e-2”	(<level>)
`"z=%g"`	“z=100”, “z=1e6”, “z=-2.05e-2”	(z=<level>)
`"%s %f"`	“zWave 100.000000”	(<wave>, space, <level>)
`"[%d]=%g"`	“[0]=100”, “[1]=1e6”	([<index>]=<level>)

Examples

```
Make/O/N=(25,25) w2D                    // Make a matrix
SetScale x -1, 1, w2D                   // Set row scaling
SetScale y -1, 1, w2D                   // Set column scaling
w2D = sin(x) * cos(y)                   // Store values in the matrix
Display; AppendMatrixContour w2D
ModifyContour w2D autoLevels={*,*,9}    // Roughly 9 automatic levels
```

See also

Display, **AppendToGraph**, **AppendXYZContour**, **ModifyContour**, **RemoveContour**, **FindContour**.

For general information on contour plots, see Chapter II-15, **Contour Plots**.

AppendText

AppendText [**/W=*winName*/N/NOCR** [**=*n***]] ***textStr***

The AppendText operation appends a carriage return and *textStr* to the most recently created annotation, or to the named annotation in the target or graph or layout window. Annotations include tags, textboxes, color scales, and legends.

Parameters

textStr can contain escape codes to control font, font size and other stylistic variations. See **Annotation Escape Codes** on page III-53 for details.

Flags

/N=*name*	Appends *textStr* to the named tag or textbox.
/NOCR[=*n*]	Omits the initial appending of a carriage return (allows a long line to be created with multiple AppendText commands). /NOCR=0 is the same as no /NOCR, and /NOCR=1 is the same as just /NOCR.
/W=*winName*	Appends to an annotation in the named graph, layout window, or subwindow. Without /W, AppendText appends to an annotation in the topmost graph or layout window or subwindow. This must be the first flag specified when AppendText is used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

A textbox, tag, or legend can contain at most 100 lines. A color scale can have at most one line, and this line is the color scale's main axis label.

See Also

The **Tag**, **TextBox**, **ColorScale**, **ReplaceText**, and **Legend** operations.

Annotation Escape Codes on page III-53.

AppendToGizmo

AppendToGizmo [***flags***] ***keyword*** [**=*value***]

The AppendToGizmo operation appends a Gizmo object or attribute operation to the top Gizmo window or to the Gizmo window specified by the /N flag.

Documentation for the AppendToGizmo operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "AppendToGizmo"
```

AppendToGraph

AppendToGraph [***flags***] ***waveName*** [, ***waveName***]...[**vs** ***xwaveName***]

The AppendToGraph operation appends the named waves to the target or named graph. By default the waves are plotted versus the left and bottom axes.

Parameters

The *waveNames* parameters are the names of existing waves.

vs *xwaveName* plots the data values of *waveNames* against the data values of *xwaveName*.

If you are appending a new trace to an existing category plot, *xwaveName* must be the same as the one already controlling the plot's X axis. If the existing X axis uses dimension labels from a Y wave, using the '_labels_' keyword, then *xwaveName* must be set to '_labels_'.

If you are appending a new category plot using a different X axis, *xwaveName* can refer any suitable text wave, or it may be '_labels_' to use dimension labels from the Y wave.

Subsets of data, including individual rows or columns from a matrix, may be specified using **Subrange Display Syntax** on page II-321.

You can provide a custom name for a trace by appending /TN=traceName to the waveName specification. This is useful when displaying waves with the same name but from different data folders. See **User-defined Trace Names** on page IV-89 for more information.

Flags

/B [=*axisName*] Plots X coordinates versus the standard or named bottom axis.

/C=(*r*,*g*,*b*[,*a*]) Sets the color of appended traces. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

/L [=*axisName*] Plots Y coordinates versus the standard or named left axis.

/NCAT Causes trace to be plotted normally on what otherwise is a category plot. X values are just category numbers but can be fractional. Category numbers start from zero. This can be used to overlay the original data points for a box plot.

See **Combining Numeric and Category Traces** on page II-362 for details.

/Q Uses a special, quick update mode when appending to a pair of existing axes. A side effect of this mode is that waves that are appended are marked as not modified. This will prevent other graphs containing these waves, if any, from being updated properly.

/R [=*axisName*] Plots Y coordinates versus the standard or named right axis.

/T [=*axisName*] Plots X coordinates versus the standard or named top axis.

/TN=*traceName* Allows you to provide a custom trace name for a trace. This is useful when displaying waves with the same name but from different data folders. See **User-defined Trace Names** on page IV-89 for details.

/VERT Plots data vertically. Similar to SwapXY (**ModifyGraph (axes)**) but on a trace-by-trace basis.

/W=*winName* Appends to the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

See Also

The **Display** operation.

AppendToLayout

AppendToLayout [***flags***] ***objectSpec*** [, ***objectSpec***]…

The AppendToLayout operation appends the specified objects to the top layout.

The AppendToLayout operation can not be used in user-defined functions. Use the AppendLayoutObject operation instead.

Parameters

The optional *objectSpec* parameters identify a graph, table, textbox or PICT to be added to the layout. An object specification can also specify the location and size of the object, whether the object should have a frame or not, whether it should be transparent or opaque, and whether it should be displayed in high fidelity or not. See the **Layout** operation for details.

Flags

/G=*g* Specifies grout, the spacing between tiled objects. Units are points unless /I, /M, or /R are specified.

/I *objectSpec* coordinates are in inches.

/M	*objectSpec* coordinates are in centimeters.
/R	*objectSpec* coordinates are in percent of printing part of the page.
/S	Stacks objects.
/T	Tiles objects.

See Also

The **Layout** and **AppendLayoutObject** operations for use with user-defined functions.

AppendToTable

```
AppendToTable [/W=winName] columnSpec [, columnSpec]…
```

The AppendToTable operation appends the specified columns to the top table. *columnSpec*s are the same as for the **Edit** operation; usually they are just the names of waves.

Flags

/W=*winName*	Appends columns to the named table window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

See Also

Edit for details about *columnSpec*s, and **RemoveFromTable**.

AppendViolinPlot

```
AppendViolinPlot [ flags ] wave[, wave, ...] [vs xWave]
```

The AppendViolinPlot operation appends a violin plot trace to the target or named graph. A violin plot (also called a “bean” plot) is a way to display a summary of the distribution of data values using a kernel density estimator curve (see **StatsKDE**). Another way to display a summary of data distribution is via a box plot.

AppendViolinPlot was added in Igor Pro 8.00.

A violin plot trace is treated within Igor as a single graph trace, and many of the operations such as removing from a graph or reordering traces work the same with a violin plot trace as with any other graph trace.

There is no DisplayViolinPlot operation. Use Display followed by AppendViolinPlot.

Parameters

The data for a single violin plot in a violin plot trace comes from either one entire 1D wave or from a single column of a multi-column wave. The number of violin plots in the trace is determined by either the number of 1D waves in the list of waves, or by the number of columns in the single multi-column wave. It is not permitted to mix 1D and multi-column waves.

You can list up to 100 individual 1D waves. If you want more violin plots than 100 in a violin plot trace you must use a multi-column wave or add to the list using the **AddWavesToViolinPlot** operation.

If you do not provide *xWave*, each violin plot is positioned on a numeric axis at X=0, 1, etc. If the data is in a multi-column wave, positioning comes by default from the X scaling of the matrix wave. Providing a numeric *xWave* allows you to position each violin plot at an arbitrary position on the X axis. A text *xWave* displays the violin plots using a category axis. If you use the /CATL flag with a multi-column wave, the result is a category X axis using the dimension labels as the category labels. See **Category Plots** on page II-355 and **Dimension Labels** on page II-93.

xWave must have at least as many points as you have 1D Y waves or columns in a multi-column wave. Because a list of waves may become too long for the command line, you can use an *xWave* that is longer than the list. Use **AddWavesToViolinPlot** to complete the list.

Flags

/L/R/B/T These axis flags are the same as used by **AppendToGraph**.

/CATL[=*doCatLabels*] Use the column dimension labels to produce violin plots on a category axis using the column dimension labels as the category labels. *doCatLabels* is 0 or 1 and /CATL is equivalent to /CATL=1.

/TN=*traceName* Allows you to provide a custom name for a trace. This is useful when displaying waves with the same name but from different data folders. See **User-defined Trace Names** on page IV-89 for details, except that the /TN flag for AppendViolinPlot comes in the normal position in the command, after the command name.

/VERT[=*doVert*] Arranges the individual violin plots vertically along the Y axis. *doVert* is 0 or 1 and /VERT is equivalent to /VERT=1.

/VERT is similar to ModifyGraph SwapXY but on a trace-by-trace basis. To make a violin plot with horizontal violins, use either ModifyGraph SwapXY or AppendViolinPlot/VERT.

/W=*winName* Appends to the named graph window or subwindow. When omitted, AppendViolinPlot is directed to the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

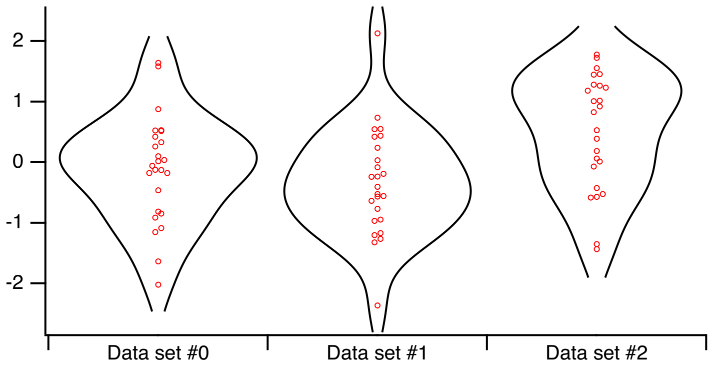
When identifying a subwindow with winName, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

Some aspects of the overall appearance of a violin plot trace can be set using ModifyGraph. By default, the line color is black and markers for non-outlier data points are hollow circles half the size of the normal trace marker. Using the Modify Violin Plot dialog or the ModifyViolinPlot operation, you can choose to use trace color, line size, line dash style, marker and marker size as set by ModifyGraph just as for a regular trace. Detailed control of these characteristics for various parts of a violin plot trace is provided by **ModifyViolinPlot**.

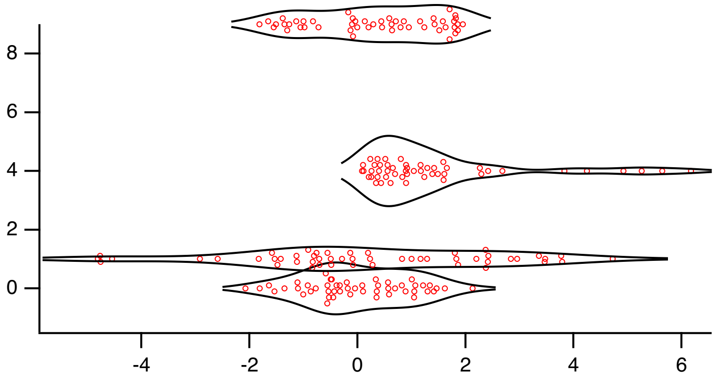
Examples

```
// Demo vertical violin plot
Make/O/N=(25,3) multicol               // A three-column wave with 25 rows
SetRandomSeed(.5)
multicol = gnoise(1)                   // Three normally-distributed datasets
Make/O/N=3/T labels                    // A text wave to make a category plot
labels = "Dataset #"+num2str(p)        // Labels for the X axis of a category plot
Display; AppendViolinPlot multicol vs labels
```



```
// Demo horizontal violin plot
Make/O/N=50 ds1, ds2, ds3, ds4
Make/O/N=4 dsX
ds1 = gnoise(1)
ds2 = gnoise(2)
ds3 = logNormalNoise(0, 1)
```

```
ds4 = enoise(2)
dsX = p^2
Display; AppendViolinPlot ds1,ds2,ds3,ds4 vs dsX
ModifyGraph swapXY=1                    // Horizontal boxes
ModifyGraph margin(top)=20              // Top margin may be too small
```



See Also

Display, **AppendToGraph**, **ModifyGraph (traces)**, **ModifyViolinPlot**

Box Plots on page II-331, **Violin Plots** on page II-337

AppendXYZContour

AppendXYZContour [**/W=*winName* /F=*formatStr***][***axisFlags***] ***zWave*** [**vs** {***xWave***, ***yWave***}]

The AppendXYZContour operation appends to the target or named graph a contour of a 2D wave consisting of XYZ triples with autoscaled contour levels and using the Rainbow color table.

To contour a matrix of Z values, use **AppendMatrixContour**.

Note: There is no DisplayContour operation. Use `Display; AppendXYZContour`.

Parameters

If you provide the *xWave* and *yWave* specification, *xWave* provides X values for the rows, and *yWave* provides Y values for the columns, *zWave* provides Z values and all three waves must be 1D. All must have at least four rows and must have the same number of rows.

If you omit the *xWave* and *yWave* specification, *zWave* must be a 2D wave with 4 or more rows and 3 or more columns. The first column is X, the second is Y, and the third is Z. Any additional columns are ignored.

If any of X, Y, or Z in a row is blank, (NaN), that row is ignored.

In a macro, to modify the appearance of contour levels before the contour is calculated and displayed with the default values, append "`;DelayUpdate`" and immediately follow the AppendXYZContour command with the appropriate **ModifyContour** commands. All but the last ModifyContour command should also have `;DelayUpdate` appended. DelayUpdate is not needed in a function, but DoUpdate is useful in a function to force the contour traces to be built immediately rather than the default behavior of waiting until all functions have completed.

On the command line, the **Display** command and subsequent AppendXYZContour commands and any **ModifyContour** commands can be typed all on one line with semicolons between:

```
Display; AppendXYZContour zWave; ModifyContour ...
```

Flags

axisFlags	Flags /L, /R, /B, and /T are the same as used by **AppendToGraph**.
/F=*formatStr*	Determines names assigned to the contour level traces. This is the same as for **AppendMatrixContour**.

/W=*winName* Appends to the named graph window or subwindow. When omitted, action affects the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

AppendXYZContour creates and displays contour level traces. You can modify these as a group using the Modify Contour Appearance dialog or individually using the Modify Trace Appearance dialog. In most cases, you will have no need to modify the traces individually.

See **AppendMatrixContour** for a discussion of how the contour level traces are named.

Examples

```
Make/O/N=(100) xW, yW, zW               // Make X, Y, and Z waves
xW = sawtooth(2*PI*p/10)                // Generate X values
yW = trunc(p/10)/10                     // Generate Y values
zW = sin(2*PI*xW)*cos(2*PI*yW)          // Generate Z values
Display; AppendXYZContour zW vs {xW, yW}; DelayUpdate
ModifyContour zW autoLevels={*,*,9}     // roughly 9 automatic levels
```

See Also

The **Display** operation. **AppendToGraph** for details about other axis flags. The **AppendMatrixContour**, **ModifyContour**, and **RemoveContour** operations. For general information on contour plots, see Chapter II-15, **Contour Plots**.

area

`area(`*`waveName`*` [, `*`x1`*`, `*`x2`*`])`

The area function returns the signed area between the named wave and the line y=0 from x=*x1* to x=*x2* using trapezoidal integration, accounting for the wave's X scaling. If your data are in the form of an XY pair of waves, see **areaXY**.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If *x1* or *x2* are not within the X range of *waveName*, area limits them to the nearest X range limit of *waveName*.

If any values in the X range are NaN, area returns NaN.

The function returns NaN if the input wave has zero points.

Reversing the order of *x1* and *x2* changes the sign of the returned area.

The area function is intended to work on 1D real or complex waves only.

The area function returns a complex result for a complex input wave. The real part of the result is the area of the real components in the input wave, and the imaginary part of the result is the area of the imaginary components.

Examples

```
Make/O/N=100 data; SetScale/I x 0,Pi,data
data=sin(x)
Print area(data,0,Pi)          // the entire X range, and no more
Print area(data)               // same as -infinity to +infinity
Print area(data,Inf,-Inf)      // +infinity to -infinity
```

The following is printed to the history area:

```
Print area(data,0,Pi)          // the entire X range, and no more
  1.99983
Print Print area(data)         // same as -infinity to +infinity
  1.99983
Print area(data,Inf,-Inf)      // +infinity to -infinity
  -1.99983
```

The `-Inf` value was limited to 0 and `Inf` was limited to `Pi` to keep them within the X range of data.

See Also

The figure “Comparison of area, faverage and mean functions over interval (12.75,13.32)”, in the **Details** section of the **faverage** function.

Integrate, **areaXY**, **faverage**, **faverageXY**, **Poly2D Example 3**

areaXY

`areaXY(`*`XWaveName, YWaveName`*` [, `*`x1, x2`*`])`

The areaXY function returns the signed area between the named *YWaveName* and the line y=0 from x=*x1* to x=*x2* using trapezoidal integration with X values supplied by *XWaveName*.

This function is identical to the **area** function except that it works on an XY wave pair and does not work with complex waves.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If *x1* or *x2* are outside the X range of *XWaveName*, areaXY limits them to the nearest X range limit of *XWaveName*.

If any values in the Y range are NaN, areaXY returns NaN.

If any values in the entire X wave are NaN, areaXY returns NaN.

The function returns NaN if the input wave has zero points.

Reversing the order of *x1* and *x2* changes the sign of the returned area.

If *x1* or *x2* are not found in *XWaveName*, a Y value is found by linear interpolation based on the two bracketing X values and the corresponding values from *YWaveName*.

The values in *XWaveName* may be increasing or decreasing. AreaXY assumes that the values in *XWaveName* are monotonic. If they are not monotonic, Igor does not complain, but the result is not meaningful. If any X values are NaN, the result is NaN.

See the figure “Comparison of area, faverage and mean functions over interval (12.75,13.32)”, in the **Details** section of the **faverage** function.

The areaXY operation is intended to work on 1D waves only.

Examples

```
Make/O/N=101 Xdata, Ydata
Xdata = x*pi/100
Ydata = sin(Xdata[p])
Print areaXY(Xdata, Ydata,0,Pi)         // the entire X range, and no more
Print areaXY(Xdata, Ydata)              // same as -infinity to +infinity
Print areaXY(Xdata, Ydata,Inf,-Inf)     // +infinity to -infinity
```

The following is printed to the history area:

```
Print areaXY(Xdata, Ydata,0,Pi)         // the entire X range, and no more
  1.99984
Print areaXY(Xdata, Ydata)              // same as -infinity to +infinity
  1.99984
Print areaXY(Xdata, Ydata,Inf,-Inf)     // +infinity to -infinity
  -1.99984
```

The `-Inf` value was limited to `0`, and `Inf` was limited to `Pi` to stay within the X range of data.

See Also

Integrate, **area**, **faverage**, **faverageXY**, **Poly2D Example 3**

asin

`asin(`*`num`*`)`

The asin function returns the inverse sine of num in radians in the range [-π/2,π/2].

In complex expressions, *num* is complex, and asin returns a complex value.

See Also

sin

asinh

asinh(*num*)

The asinh function returns the inverse hyperbolic sine of *num*. In complex expressions, *num* is complex, and asinh returns a complex value.

atan

atan(*num*)

The atan function returns the inverse tangent of *num* in radians. In complex expressions, *num* is complex, and atan returns a complex value. Results are in the range -π/2 to π/2.

See Also

tan, **atan2**

atan2

atan2(*y1, x1*)

The atan2 function returns the angle in radians whose tangent is *y1/x1*. Results are in the range -π to π.

See Also

tan, **atan**

atanh

atanh(*num*)

The atanh function returns the inverse hyperbolic tangent of *num*. In complex expressions, *num* is complex, and atanh returns a complex value.

AutoPositionWindow

AutoPositionWindow [**/E/M=*m*/R=*relWindow***][***windowName***]

The AutoPositionWindow operation positions the window specified by *windowName* relative to the next lower window of the same kind or relative to the window given by the /R flag. If *windowName* is not specified, AutoPositionWindow acts on the target window.

Flags

/E — Uses entire area of the monitor. Otherwise, it takes into account the command window.

/M=*m* — Specifies the window positioning method.

m=0: Positions *windowName* to the right of the other window, if possible. If there is no room, then it positions *windowName* just below the other window but at the left edge of the display area. If that is not possible, then the position is not affected.

m=1: Positions *windowName* just under the other window lined up on the left edge, if possible. If there is no room, then it positions *windowName* just to the right of the other window lined up on the bottom edges. If neither are possible then it positions *windowName* as far to the bottom and right as it will go.

/R=*relWindow* — Positions *windowName* relative to *relWindow*.

AxisInfo

`AxisInfo(`*`graphNameStr`*`, `*`axisNameStr`*`)`

The AxisInfo function returns a string containing a semicolon-separated list of information about the named axis in the named graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

axisNameStr is the name of the graph axis.

Details

The string contains groups of information. Each group is prefaced by a keyword and colon, and terminated with a semicolon. The keywords are:

Keyword	Information Following Keyword
AXFLAG	Flag used to select the axis in any of the operations that display waves (Display, AppendMatrixContour, AppendImage, etc.).
AXTYPE	Axis type, such as “left”, “right”, “top”, or “bottom”.
CATWAVE	Wave supplying the categories for the axis if this is a category plot.
CATWAVEDF	Full path to data folder containing category wave.
CTRACE	Name of the trace controlling the names axis. See **Trace Names** on page II-282 for background information. This field was added in Igor Pro 9.00.
CWAVE	Name of wave controlling named axis.
CWAVEDF	Full path to data folder containing controlling wave.
FONT	Actual name of font used to draw axis tick labels.
FONTSIZE	Actual size of font used to draw axis tick labels, in points.
FONTSTYLE	Actual font style used to draw axis tick labels, in points. See **ModifyGraph (axes)** on page V-626 fstyle for the meaning of this integer value.
HOOK	Name set by ModifyFreeAxis with hook keyword.
ISCAT	Truth that this is a category axis (used in a category plot).
ISTFREE	Truth that this is truly free axis (created via NewFreeAxis).
MASTERAXIS	Name set by ModifyFreeAxis with master keyword.
RECREATION	List of keyword commands as used by ModifyGraph command. The format of these keyword commands is: *keyword*(x)=*modifyParameters*;
SETAXISCMD	Full SetAxis command.
SETAXISFLAGS	Flags that would be used with the SetAxis function to set the particular auto-scaling behavior that the axis uses. If the axis uses a manual axis range, SETAXISFLAGS is blank.
UNITS	Axis units, if any.

The format of the RECREATION information is designed so that you can extract a keyword command from the keyword up to the “`;`”, prepend “`ModifyGraph`”, replace the “`x`” with the name of an actual axis and then **Execute** the resultant string as a command.

Examples

```
Make/O data=x;Display data
Print StringByKey("CWAVE", AxisInfo("","left"))       // Prints data
```

```
// Get a numeric value from the RECREATION keyword nested keyword values
String info = AxisInfo("","left")
String key = ";RECREATION:"
Variable index = strsearch(info,key,0)
String recreation = info[index+strlen(key),inf]
Print recreation                          // Prints catGap(x)=0.1;barGap(x)=0.1;...
Print NumberByKey("barGap(x)",recreation,"=")     // Prints 0.1
```

See Also

GetAxis, **SetAxis**, **AxisLabel**, **StringByKey**, **NumberByKey**

The "Readback ModifyStr" procedure file is useful for parsing strings returned by AxisInfo.

AxisLabel

AxisLabel(*graphNameStr*, *axisNameStr* [, *escapeBackslashes*])

The AxisLabel function returns a string containing the axis label for the named axis in the named graph window or subwindow. The string returned is suitable for use with the Label operation. The AxisLabel function is primarily intended for copying the label from one axis to another.

The AxisLabel function was added in Igor Pro 9.00.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow via *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

axisNameStr is the name of the axis.

escapeBackslashes is an optional parameter which defaults to 0 (false).

Details

Specify *escapeBackslashes*=1 when using the result to create a command passed to the **Execute** operation. Otherwise omit *escapeBackslashes* or specify *escapeBackslashes*=0.

The "Axis Utilities.ipf" WaveMetrics procedure contains an AxisLabelText function that is very similar to AxisLabel. If you use AxisLabelText, you can replace it with AxisLabel for use with Igor Pro 9 or later.

Examples

```
// Label Graph0 left axis with something that has a backslash in it
Make/O data=x
Display/N=Graph0 data
Label/W=Graph0 left, "Area \\E (UV*Sec)"

// Reuse the left axis label from Graph0 for Graph1
Display/N=Graph1 data
String lblStr = AxisLabel("Graph0","left")
Label/W=Graph1 left, lblStr
```

See Also

AxisInfo, **Label**, **Backslashes in Annotation Escape Sequences** on page III-58

AxisList

AxisList(*graphNameStr*)

The AxisList function returns a semicolon-separated list of axis names from the named graph window or subwindow.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Examples

```
Make/O data=x;Display/L/T data
Print AxisList("")              // prints left;top;
```

AxisValFromPixel

`AxisValFromPixel(`*`graphNameStr`*`, `*`axNameStr`*`, `*`pixel`*`)`

The AxisValFromPixel function returns an axis value corresponding to the local graph pixel coordinate in the graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

If the specified axis is not found and if the name is “left” or “bottom” then the first vertical or horizontal axis will be used. Sources for *pixel* value may be the GetWindow operation or a user window hook with the mousemoved and mousedown event messages (see the **SetWindow** operation).

If *graphNameStr* references a subwindow, *pixel* is relative to top left corner of base window, not the subwindow.

Axis ranges and other graph properties are computed when the graph is redrawn. Since automatic screen updates are suppressed while a user-defined function is running, if the graph was recently created or modified, you must call DoUpdate to redraw the graph so you get accurate axis information.

See Also

The **PixelFromAxisVal** and **TraceFromPixel** functions; the **GetWindow** and **SetWindow** operations.

BackgroundInfo

`BackgroundInfo`

The BackgroundInfo operation returns information about the current unnamed background task.

BackgroundInfo works only with the unnamed background task. New code should used named background tasks instead. See **Background Tasks** on page IV-319 for details.

Details

Information is returned via the following variables:

`V_flag`	0: No background task is defined. 1: Background task is defined, but not running (is idle). 2: Background task is defined and is running.
`V_period`	DeltaTicks value set by CtrlBackground. This is how often the background task runs.
`V_nextRun`	Ticks value when the task will run again. 0 if the task is not scheduled to run again.
`S_value`	Text of the numeric expression that the background task executes, as set by SetBackground.

See Also

The **SetBackground**, **CtrlBackground**, **CtrlNamedBackground**, **KillBackground**, and **SetProcessSleep** operations, and the **ticks** function. See **Background Tasks** on page IV-319 for usage details.

Base64Decode

`Base64Decode(`*`inputStr`*`)`

The Base64Decode function returns a decoded copy of the Base64-encoded string *inputStr*. The contents of *inputStr* are not checked for validity. Any invalid characters in *inputStr* are skipped, and decoding continues with subsequent characters.

The algorithm used to encode Base64-encoded data is defined in RFC 4648 (http://www.ietf.org/rfc/rfc4648.txt).

For an explanation of Base64 encoding, see https://en.wikipedia.org/wiki/Base64.

The Base64Decode function was added in Igor Pro 8.00.

Example

```
String encodedString = "SWdvciBpcyBncmVhdCE="
Print Base64Decode(encodedString)
  Igor is great!
```

See Also

Base64Encode, **URLRequest**

Base64Encode

Base64Encode(*inputStr*)

The Base64Encode function returns a copy of *inputStr* encoded as Base64.

The algorithm used to encode Base64-encoded data is defined in RFC 4648 (http://www.ietf.org/rfc/rfc4648.txt).

For an explanation of Base64 encoding, see https://en.wikipedia.org/wiki/Base64.

The Base64Encode function was added in Igor Pro 8.00.

Example

```
String theString = "Igor is great!"
Print Base64Encode(theString)
  SWdvciBpcyBncmVhdCE=
```

See Also

Base64Decode, **URLRequest**

Beep

Beep

The Beep operation plays the current alert sound (*Macintosh*) or the system beep sound (*Windows*).

Besseli

Besseli(*n*,*z*)

The Besseli function returns the modified Bessel function of the first kind, I*n*(*z*), of order *n* and argument *z*. Replaces the bessI function, which is supported for backwards compatibility only.

If *z* is real, Besseli returns a real value, which means that if *z* is also negative, it returns NaN unless *n* is an integer.

For complex *z* a complex value is returned, and there are no restrictions on *z* except for possible overflow.

Details

The calculation is performed using the SLATEC library. The function supports fractional and negative orders *n*, as well as real or complex arguments *z*.

See Also

The **Besselj**, **Besselk**, and **Bessely** functions.

Besselj

Besselj(*n*,*z*)

The Besselj function returns the Bessel function of the first kind, J*n* (*z*), of order *n* and argument *z*. Replaces the bessJ function, which is supported for backwards compatibility only.

If *z* is real, Besselj returns a real value, which means that if *z* is also negative, it returns NaN unless *n* is an integer.

For complex *z* a complex value is returned, and there are no restrictions on *z* except for possible overflow.

Details

The calculation is performed using the SLATEC library. The function supports fractional and negative orders *n*, as well as real or complex arguments *z*.

See Also

The **Besseli**, **Besselk**, and **Bessely** functions.

Besselk

Besselk(*n*,*z*)

The Besselk function returns the modified Bessel function of the second kind, K*n*(*z*), of order *n* and argument *z*. Replaces the bessK function, which is supported for backwards compatibility only.

If *z* is real, Besselk returns a real value, which means that if *z* is also negative, it returns NaN unless *n* is an integer.

For complex *z* a complex value is returned, and there are no restrictions on *z* except for possible overflow.

Details

The calculation is performed using the SLATEC library. The function supports fractional orders *n*, as well as real or complex arguments *z*.

See Also

The **Besseli**, **Besselj**, and **Bessely** functions.

Bessely

Bessely(*n*,*z*)

The Bessely function returns the Bessel function of the second kind, Y*n*(*z*), of order *n* and argument *z*. Replaces the bessY function, which is supported for backwards compatibility only.

If *z* is real, Bessely returns a real value, which means that if *z* is also negative, it returns NaN unless *n* is an integer.

For complex *z* a complex value is returned, and there are no restrictions on *z* except for possible overflow.

Details

The calculation is performed using the SLATEC library. The function supports fractional and negative orders *n*, as well as real or complex arguments *z*.

See Also

The **Besseli**, **Besselj**, and **Besselk** functions.

bessI

bessI(*n*, *x* [, *algorithm* [, *accuracy*]])

Obsolete — use **Besseli**.

The bessI function returns the modified Bessel function of the first kind, I*n*(*x*) of order *n* and argument *x*.

For real *x*, the optional parameter *algorithm* selects between a faster, less accurate calculation method and slower, more accurate methods. In addition, when *algorithm* is zero or absent, the order *n* is truncated to an integer.

When *algorithm* is included and is 1, *accuracy* can be used to specify the desired fractional accuracy. See Details about algorithms.

If *x* is complex, a complex result is returned. In this case, *algorithm* and *accuracy* are ignored. The order *n* can be fractional, and must be real.

Details

The *algorithm* parameter has three options, each selecting a different calculation method:

Algorithm	What You Get
0 (default)	Uses a calculation method that has fractional accuracy better than 10^{-6} everywhere and is generally better than 10^{-8}. This method does not handle fractional order *n*; the order is truncated to an integer before the calculation is performed. Algorithm 0 is fastest by a large margin.
1	Allows fractional order. The calculation is performed using methods described in *Numerical Recipes in C*, 2nd edition, pp. 240-245. Using algorithm 1, *accuracy* specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 1e-7 (that is, 10^{-7}), that means that you wish that the absolute value of (f_{actual} - $f_{returned}$)/f_{actual} be better than 10^{-7}. Asking for less accuracy gives some increase in speed.

Algorithm	What You Get
	You pay a heavy price for higher accuracy or fractional order. When *algorithm* is nonzero, calculation time is increased by an order of magnitude for small *x*; at larger *x* the penalty is even greater. If accuracy is greater than 10^{-8} and *n* is an integer, algorithm 0 is used. The algorithm calculates bessI and bessK simultaneously. Both values are stored, and if a call to bessI is followed by a call to bessK (or bessK is followed by bessI) with the same *n*, *x*, and *accuracy* the previously-stored value is returned, making the second call very fast.
2	Fractional order is allowed. The calculation is performed using code from the SLATEC library. The accuracy achievable is often better than algorithm 1, but not always. Algorithm 2 is 1.5 to 3 times faster than algorithm 1, but still slower than algorithm 0. The accuracy parameter is ignored.

The achievable accuracy of algorithms 1 and 2 is a complicated function *n* and *x*. To see a summary of achievable accuracies choose File→Example Experiments→Testing and Misc→Bessel Accuracy menu item.

bessJ

`bessJ(`*`n`*`, `*`x`*` [, `*`algorithm`*` [, `*`accuracy`*`]])`

Obsolete — use **Besselj**.

The bessJ function returns the Bessel function of the first kind, J*n*(*x*) of order *n* and argument *x*.

For real *x*, the optional parameter *algorithm* selects between a faster, less accurate calculation method and slower, more accurate methods. In addition, when *algorithm* is zero or absent, the order *n* is truncated to an integer.

When *algorithm* is included and is 1, *accuracy* can be used to specify the desired fractional accuracy. See Details about algorithms.

If *x* is complex, a complex result is returned. In this case, *algorithm* and *accuracy* are ignored. The order *n* can be fractional, and must be real.

Details

See the **bessI** function for details on algorithms, accuracy and speed of execution.

When *algorithm* is 1, pairs of values for bessJ and bessY are calculated simultaneously. The values are stored, and a subsequent call to bessY after a call to bessJ (or vice versa) with the same *n*, *x*, and *accuracy* will be very fast.

bessK

`bessK(`*`n`*`, `*`x`*` [, `*`algorithm`*` [, `*`accuracy`*`]])`

Obsolete — use **Besselk**.

The bessK function returns the modified Bessel function of the second kind, K*n*(*x*) of order *n* and argument *x*.

For real *x*, the optional parameter *algorithm* selects between a faster, less accurate calculation method and slower, more accurate methods. In addition, when *algorithm* is zero or absent, the order *n* is truncated to an integer.

When *algorithm* is included and is 1, *accuracy* can be used to specify the desired fractional accuracy. See Details about algorithms.

If *x* is complex, a complex result is returned. In this case, *algorithm* and *accuracy* are ignored. The order *n* can be fractional, and must be real.

Details

See the **bessI** function for details on algorithms, accuracy and speed of execution.

When *algorithm* is 1, pairs of values for bessJ and bessY are calculated simultaneously. The values are stored, and a subsequent call to bessY after a call to bessJ (or vice versa) with the same *n*, *x*, and *accuracy* will be very fast.

bessY

bessY(*n*, *x* [, *algorithm* [, *accuracy*]])

Obsolete — use **Bessely**.

The bessY function returns the Bessel function of the second kind, Y*n*(*x*) of order *n* and argument *x*.

For real *x*, the optional parameter *algorithm* selects between a faster, less accurate calculation method and slower, more accurate methods. In addition, when *algorithm* is zero or absent, the order *n* is truncated to an integer.

When *algorithm* is included and is 1, *accuracy* can be used to specify the desired fractional accuracy. See Details about algorithms.

If *x* is complex, a complex result is returned. In this case, *algorithm* and *accuracy* are ignored. The order *n* can be fractional, and must be real.

Details

See the **bessI** function for details on algorithms, accuracy and speed of execution.

When *algorithm* is 1, pairs of values for bessJ and bessY are calculated simultaneously. The values are stored, and a subsequent call to bessY after a call to bessJ (or vice versa) with the same *n*, *x*, and *accuracy* will be very fast.

beta

beta(*a*, *b*)

The beta function returns for real or complex arguments as

$$B(a,b)=\frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)},$$

with *Re*(*a*), *Re*(*b*)>0. Γ is the gamma function.

See Also

The **gamma** function.

betai

betai(*a*, *b*, *x* [, *accuracy*])

The betai function returns the regularized incomplete beta function I*x*(*a*,*b*),

$$I_x(a,b)=\frac{B(x;a,b)}{B(a,b)}.$$

Here

$$B(x;a,b)=\int_0^x t^{a-1}(1-t)^{b-1}\,dt.$$

where *a*,*b* > 0, and 0 ≤ *x* ≤ 1.

Optionally, *accuracy* can be used to specify the desired fractional accuracy.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of (f_{actual} - $f_{returned}$)/f_{actual} be less than 10^{-7}.

Larger values of *accuracy* (poorer accuracy) result in evaluation of fewer terms of a series, which means the function executes somewhat faster.

A single-precision level of accuracy is about $3x10^{-7}$, double-precision is about $2x10^{-16}$. The betai function will return full double-precision accuracy for small values of *a* and *b*. Achievable accuracy declines as *a* and *b* increase:

a	*b*	*x*	betai	Accuracy Achievable
1	1.5	0.5	0.646447	2x10^{-16} (full double precision)
8	10	0.5	0.685470	6x10^{-16}
20	21	0.5	0.562685	2x10^{-15}
20	21	0.1	1.87186x10^{-10}	5x10^{-15}

BezierToPolygon

BezierToPolygon [***flags*** **]** ***bezXWave*, *bezYWave***

The BezierToPolygon operation creates an XY pair of waves approximating the Bezier curves described by bezXWave and bezYWave.

The BezierToPolygon operation was added in Igor Pro 9.00.

Flags

/DSTX=*destX*	Specifies the X destination wave to be created or overwritten. If you omit /DSTX, *destX* defaults to W_PolyX.
/DSTY=*destY*	Specifies the Y destination wave to be created or overwritten. If you omit /DSTY, *destY* defaults to W_PolyY.
/FREE	Creates output waves as free waves (see **Free Waves** on page IV-91). /FREE is allowed only in functions. If you use /DSTX or /DSTY then the specified parameter must be either a simple name or a valid wave reference.
/NSEG=nseg	The number of segments used to render each Bezier segment from 1 and 500. The default of 20 is usually sufficient.

Details

The Bezier waves *bezXWave* and *bezYWave* must be 1-dimensional real-valued floating point waves of the same length and type.

Each Bezier curve is a minimum of 1 segment comprised of 4 XY pairs. A Bezier curve of n segments consists of 1+n*3 XY pairs.

bezXWave and *bezYWave* may have NaN values between Bezier segments but not within a segment. BezierToPolygon issues an error at runtime if the data in the input waves does not conform to these requirements. NaNs in the input waves are copied to the polygon output waves.

If you omit /DSTX the output polygon X data is written to W_PolyX in the current data folder. If you omit /DSTY the output polygon Y data is written to W_PolyY in the current data folder. The output waves are created or redimensioned as single-precision or double-precision floating point waves to match the type of *bezXWave* and *bezYWave*.

Example

```
Function DemoBezierToPolygon()
   Make/O wx={0.5, 0.6, 0.9, 1}
   Make/O wy={0.0, 0.2, 0.5, 0.1}
   BezierToPolygon wx,wy
   Execute "BezierToPolygonExample()"
End

Window BezierToPolygonExample() : Graph
   PauseUpdate; Silent 1
   Display /W=(237,45,1419,669)/K=1  wy vs wx
   AppendToGraph W_PolyY vs W_PolyX
   ModifyGraph expand=-3
   ModifyGraph mode(wy)=4
   ModifyGraph marker(wy)=19
   ModifyGraph rgb(wy)=(1,16019,65535)
   Legend/C/N=text0/J/X=15.66/Y=68.34 "\\s(wy)\\[1 Bezier Control Points"
   AppendText "\\K(48059,48059,48059)\\y+15\\L1700\\X1\\M\\K(0,0,0) DrawBezier
                                        \n\\s(W_PolyY) Polygon Approximation to Bezier"
```

```
    SetDrawLayer UserBack
    SetDrawEnv xcoord= bottom,ycoord= left,save
    SetDrawEnv linethick= 3,linefgc= (48059,48059,48059)
    DrawBezier wx[0],wy[0],1,1,wx,wy
    SetDrawLayer UserFront
EndMacro
```

See Also

DrawBezier, **DrawPoly**, **Drawing Polygons and Bezier Curves** on page III-69

BinarySearch

BinarySearch(*waveName*, *val*)

The BinarySearch function performs a binary search of the one-dimensional *waveName* for the value *val*. BinarySearch returns an integer point number p such that *waveName*[p] and *waveName*[p+1] bracket *val*. If *val* is in *waveName*, then *waveName*[p]==*val*.

Details

BinarySearch is useful for finding the point in an XY pair that corresponds to a particular X coordinate.

WaveName must contain monotonically increasing or decreasing values.

BinarySearch returns -1 if *val* is not within the range of values in the wave, but would numerically be placed before the first value in the wave.

BinarySearch returns -2 if *val* is not within the range of values in the wave, but would fall after the last value in the wave.

BinarySearch returns -3 if the wave has zero points.

Examples

```
Make/O data = {1, 2, 3.3, 4.9}       // Monotonic increasing
Print BinarySearch(data,3)           // Prints 1
// BinarySearch returns 1 because data[1] <= 3 < data[2].
```

```
Make/O data = {9, 4, 3, -6}          // Monotonic decreasing
Print BinarySearch(data,2.5)         // Prints 2
// BinarySearch returns 2 because data[2] >= 2.5 > data[3].
Print BinarySearch(data,10)          // Prints -1, precedes first value
Print BinarySearch(data,-99)         // Prints -2, beyond last value
```

See Also

The **BinarySearchInterp** and **FindLevel** operations. See **Indexing and Subranges** on page II-76.

BinarySearchInterp

BinarySearchInterp(*waveName*, *val*)

The BinarySearchInterp function performs a binary interpolated search of the named wave for the value *val*. The returned value, pt, is a floating-point point index into the named wave such that *waveName*[pt] == *val*.

Details

BinarySearchInterp is useful for finding the point in an XY pair that corresponds to a particular X coordinate.

WaveName must contain monotonically increasing or decreasing values.

When the named wave does not actually contain the value *val*, BinarySearchInterp locates a value below *val* and a value above *val* and uses reverse linear interpolation to figure out where *val* would fall if a straight line were drawn between them. It includes that fractional amount in the resulting point index.

BinarySearchInterp returns NaN if *val* is not within the range of values in the wave.

Examples

```
Make/O data = {1, 2, 3.3, 4.9}          // Monotonic increasing
Print BinarySearchInterp(data,3)        // Prints 1.76923
Print data[1.76923]                     // Prints 3
```

```
Make/O data = {9, 4, 3, 1}              // Monotonic decreasing
Print BinarySearchInterp(data,2.5)      // Prints 2.25
Print data[2.25]                        // Prints 2.5
```

See Also

The **BinarySearch** and **FindLevel** operations. See **Indexing and Subranges** on page II-76.

binomial

`binomial(n, k)`

The binomial function returns the ratio:

$$\frac{n!}{k!(n-k)!}$$

It is assumed that *n* and *k* are integers and $0 \le k \le n$ and ! denotes the factorial function.

Note that although the binomial function is an integer-valued function, a double-precision number has 53 bits for the mantissa. This means that numbers over 2^{52} (about $4.5\text{x}10^{15}$) will be accurate to about one part in $2\text{x}10^{16}$.

If you encounter overflow when the arguments are large you can use **APMath** or **binomialln**. For example:

```
• Print binomial(2800,333)
  inf
• APMath/V result = binomial(2800,333)
  9.0019826685021446499850285037304473382191760312233OE+441

• Print/D binomialln(2800,333)
  1017.63747085995
• APMath/V result = log(binomial(2800,333))
  1.0176374708599488934351415800704506410635781326878 1E+3
```

binomialln

`binomialln(a, b)`

The binomialln function returns the natural log of the binomial coefficient for *a* and *b*.

$$\text{binomialln}(a,b) = \ln(a!) - \ln(b!) - \ln((a-b)!)$$

See Also

Chapter III-12, **Statistics** for an overview of the various functions and operations; **binomial**, **StatsBinomialPDF**, **StatsBinomialCDF**, and **StatsInvBinomialCDF**.

binomialNoise

`binomialNoise(n, p)`

The binomialNoise function returns a pseudo-random value from the binomial distribution

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}, \qquad \begin{array}{c} 0 \le p \le 1 \\ x = 0,1,2...n \end{array}$$

whose mean is *np* and variance is *np*(1-*p*).

When *n* is large such that p^n is zero to machine accuracy the function returns NaN. When *n* is large such that *np*(1-*p*)>5 and 0.1<*p*<0.9 you can replace the binomial variate with a normal variate with mean *np* and standard deviation sqrt(*n***p**(1-*p*)).

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat the same sequence. For repeatable “random” numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-390.

Chapter III-12, **Statistics** for an overview of the various functions and operations.

BoundingBall

BoundingBall [/F/Z] ***scatterWave***

The BoundingBall operation calculates a bounding circle or the bounding sphere for a set of scatter points. The operation accepts 2D waves that have two, three or more columns; data in the additional columns are ignored.

When *scatterWave* consists of two columns the operation computes the bounding circle. Otherwise it computes the bounding 3D sphere.

Parameters

scatterWave is a two-dimensional wave with X coordinates in column 0, Y in column 1, and optional Z coordinates in column 2.

Flags

/F	This flag applies to 3D scatter only. It uses an algorithm from "An Efficient Bounding Sphere" by Jack Ritter originally from *Graphics Gems*. Unfortunately it does not give an accurate bounding ball but something that is sufficiently large. This algorithm is less accurate but it produces a ball which is sufficiently large to contain all the points.
/Z	No error reporting.

Details

The center and radius of the bounding sphere are stored in the variables: V_CenterX, V_CenterY, V_CenterZ, and V_Radius.

If you are not using the /F flag, the operation also accepts a 2 column wave consisting of X, Y pairs for calculating the center and radius of a bounding circle in the plane.

Example

```
Make/N=(33,2) ddd=enoise(4)                // Create random data
BoundingBall ddd
Display ddd[][1] vs ddd[][0]
ModifyGraph mode=3
Make/n=360 xxx,yyy
yyy=v_centerY+V_radius*cos(p*2*pi/360)
xxx=v_centerX+V_radius*sin(p*2*pi/360)
AppendToGraph yyy vs xxx
```

References

Glassner, Andrew S., (Ed.), *Graphics Gems*, 833 pp., Academic Press, San Diego, 1990.

BoxSmooth

BoxSmooth ***box*****,** ***srcWave*****,** ***smoothedWave***

The BoxSmooth operation replaces *smoothedWave* with a smoothed copy of *srcWave*. The waves must both exist.

The BoxSmooth operation is used primarily by Igor Technical Note #20 and its variants. For most purposes, use the more flexible **Smooth** operation instead of BoxSmooth.

Parameters

box is the number of *srcWave* points averaged to form each *smoothedWave* point. If you specify an even number, the next-higher odd number is used.

Details

BoxSmooth is equivalent to the **Smooth** operation with the /B flag, except that BoxSmooth does not compute the result in-place like Smooth does. This command:

```
BoxSmooth box, srcWave, smoothedWave
```

is equivalent to:

```
Duplicate/O srcWave, smoothedWave
Smooth/B/DIM=-1/E=3/F=0 box, smoothedWave
```

break

`break`

The break flow control keyword immediately terminates execution of a loop, switch, or strswitch. Execution then continues with code following the loop, switch, or strswitch.

See Also

Break Statement on page IV-48, **Switch Statements** on page IV-43, and **Loops** on page IV-45 for usage details.

BrowseURL

`BrowseURL [/Z]` ***`urlStr`***

The BrowseURL operation opens the Web browser or FTP browser on your computer and asks it to display a particular Web page or to connect to an FTP server.

BrowseURL sets a variable named V_flag to zero if the operation succeeds and to nonzero if it fails. This, in conjunction with the /Z flag, can be used to allow procedures to continue to execute if an error occurs.

Parameters

urlStr specifies a Web page or FTP server directory to be browsed. It is constructed of a naming scheme (e.g., “http://” or “ftp://”), a computer name (e.g., “www.wavemetrics.com” or “ftp.wavemetrics.com” or “38.170.234.2”), and a path (e.g., “/Test/TestFile1.txt”). See **Examples** for sample usage.

Flags

/Z	Errors are not fatal. Will not abort procedure execution if the URL is bad or if the server is down. Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded or nonzero if it failed. Syntactic errors, such as omitting the URL altogether or omitting quotes, are still fatal.

Examples

```
// Browse a Web page.
   String url = "http://www.wavemetrics.com/News/index.html"
   BrowseURL url
```

```
// Browse an FTP server.
   String url = "ftp://ftp.wavemetrics.com/pub/test"
   BrowseURL url
```

See Also

URLRequest

BuildMenu

`BuildMenu` ***`menuNameStr`***

The BuildMenu operation rebuilds the user-defined menu items in the specified menu the next time the user clicks in the menu bar.

Parameters

menuNameStr is a string expression containing a menu name or "`All`".

Details

Call BuildMenu when you’ve defined a custom menu using string variables for the menu items. After you change the string variables, call BuildMenu to update the menu.

BuildMenu "`All`" rebuilds all the menu items and titles and updates the menu bar.

Under the current implementation, if menuNameStr is not "`All`", Igor will rebuild *all* user-defined menu items if BuildMenu is called for *any* user-defined menu.

See Also

Dynamic Menu Items on page IV-129.

Button

Button [**/Z**] ***ctrlName*** [***keyword* = *value*** [, ***keyword* = *value*** …]]

The Button operation creates or modifies the named button control.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the Button control to be created or changed.

align=*alignment* Sets the alignment mode of the control. The alignment mode controls the interpretation of the *leftOrRight* parameter to the pos keyword. The align keyword was added in Igor Pro 8.00.

If *alignment*=0 (default), *leftOrRight* specifies the position of the left end of the control and the left end position remains fixed if the control size is changed.

If *alignment*=1, *leftOrRight* specifies the position of the right end of the control and the right end position remains fixed if the control size is changed.

appearance={*kind* [, *platform*]} Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind=default:	Appearance determined by **DefaultGUIControls**.
kind=native:	Creates standard-looking controls for the current computer platform.
kind=os9:	Igor Pro 5 appearance (quasi-Macintosh OS 9 controls that look the same on Macintosh and Windows).
platform=Mac:	Changes the appearance of controls only on Macintosh; affects the experiment whenever it is used on Macintosh.
platform=Win:	Changes the appearance of controls only on Windows; affects the experiment whenever it is used on Windows.
platform=All:	Changes the appearance on both Macintosh and Windows computers.

disable=*d* Sets the state of the control. *d* is a bit field: bit 0 (the least significant bit) is set when the control is hidden. Bit 1 is set when the control is disabled:

d=0:	Normal (visible), enabled.
d=1:	Hidden.
d=2:	Visible and disabled. Drawn in grayed state, also disables action procedure.
d=3:	Hidden and disabled.

See the **ModifyControl** example for setting the bits individually.

fColor=(*r*,*g*,*b*[,*a*]) Sets color of the button. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. While accepted as an input, *a* has no effect.

Specify fColor=(0,0,0) to get the default button color. If you want a black button use fColor=(1,1,1). To get the default blue button appearance, use fColor(0,0,65535).To set the color of the title text, see valueColor.

focusRing=*fr* Enables or disables the drawing of a rectangle indicating keyboard focus:

fr=0:	Focus rectangle will not be drawn.
fr=1:	Focus rectangle will be drawn (default).

On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.

font="*fontName*" Sets button font, e.g., `font="Helvetica"`.

fsize=*s* Sets font size.

fstyle=*fs* Specifies the font style. *fs* is a bitwise parameter with each bit controlling one aspect of the font style:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*} Specifies the help for the control.

helpStr is limited to 1970 bytes (255 in Igor Pro 8 and before).

You can insert a line break by putting "`\r`" in a quoted string.

labelBack=(*r*,*g*,*b*[,*a*]) or 0

Sets the background color for the control when using a picture to define the appearance of the button (see the picture keyword).

The labelBack keyword was added in Igor Pro 9.00.

r, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

For transparency to work, the picture must be inherently transparent. For example, each pixel in a PNG picture has its own internal alpha value so it can be inherently transparent or inherently opaque.

If you want the button background to be actually transparent, use the labelBack color to set the background color of the picture. In most cases, use transparent white: labelBack=(65535, 65535, 65535, 0).

If you omit labelBack or specify labelBack=0 then the button background color is the background color of the window in which the button is drawn.

noproc No procedure is executed when clicking the button.

picture= *pict* Draws the button using the named picture. The picture is taken to be three side-by-side frames that show the control appearance in the normal state, when the mouse is down, and in the disabled state. The picture may be either a global (imported) picture or a Proc Picture (see **Proc Pictures** on page IV-56).

By default, the button's rectangle is filled with the background color of the window it is drawn in before the picture is drawn. You can use the labelBack keyword to control the button's background color and transparency.

In Igor6, the size keyword is ignored when a picture is used with a button control. To make it easier to size graphics for high-resolution screens, as of Igor7, the size keyword is respected in this case.

pos={*leftOrRight*,*top*} Sets the position in **Control Panel Units** of the top/left corner of the control if its alignment mode is 0 or the top/right corner of the control if its alignment mode is 1. See the align keyword above for details.

pos+={*dx*,*dy*} Offsets the position of the button in **Control Panel Units**.

proc=*procName* Names the procedure to execute when clicking the button.

rename=*newName* Gives the button a new name.

size={*width*,*height*} Sets *width* and *height* of button in **Control Panel Units**.

title=*titleStr* Sets title of button (text that appears in the button) to the specified string expression. If not given then title will be "New". If you use `""` the button will contain no text.

Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.

userdata(*UDName*)=*UDStr*
Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to create named user data.

Names starting with “WM_” are reserved for WaveMetrics.

userdata(*UDName*)+=*UDStr*
Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

Names starting with “WM_” are reserved for WaveMetrics.

valueColor=(*r*,*g*,*b*[,*a*])

Sets initial color of the button's text (title). *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

To further change the color of the title text, use escape sequences as described for title=*titleStr*.

win=*winName*
Specifies which window or subwindow contains the named button control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The target window must be a graph or panel.

Button Action Procedure

The action procedure for a Button control takes a predefined structure `WMButtonAction` as a parameter to the function:

```
Function ActionProcName(B_Struct) : ButtonControl
   STRUCT WMButtonAction &B_Struct
   …
   return 0
End
```

The “`: ButtonControl`” designation tells Igor to include this procedure in the Procedure pop-up menu in the Button Control dialog.

See **WMButtonAction** for details on the WMButtonAction structure.

Although the return value is not currently used, action procedures should always return zero.

You may see an old format button action procedure in old code:

```
Function procName(ctrlName) : ButtonControl
   String ctrlName
   …
   return 0
End
```

This old format should not be used in new code.

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

The **ControlInfo** operation for information about the control.

The **GetUserData** function for retrieving named user data.

ButtonControl

`ButtonControl`

ButtonControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined button control. See **Procedure Subtypes** on page IV-204 for details. See **Button** for details on creating a button control.

cabs

`cabs(`*`z`*`)`

The cabs function returns the real-valued absolute value of complex number *z*.

See Also

The **magsqr** function.

CameraWindow

`CameraWindow`

CameraWindow is a procedure subtype keyword that identifies a macro as being a camera window recreation macro. It is automatically used when Igor creates a window recreation macro for a camera window. See **Procedure Subtypes** on page IV-204 and **Saving and Recreating Graphs** on page II-350 for details.

CaptureHistory

`CaptureHistory(`*`refnum, stopCapturing`*`)`

The CaptureHistory function returns a string containing text from the history area of the command window since a matching call to the **CaptureHistoryStart** function.

Parameters

refnum is a number returned from a call to CaptureHistoryStart. It identifies the starting point in the history for the returned string.

Set *stopCapturing* to nonzero to indicate that no more history should be captured for the given refnum. Subsequent calls to CaptureHistory with the same *refnum* will result in an error.

Set *stopCapturing* to zero to retrieve history text captured so far. Further calls to CaptureHistory with the same reference number will return this text, plus any additional history text added subsequently.

Details

You can have multiple captures active at one time. Each call to CaptureHistoryStart will return a unique reference number identifying a start point in the history. The capture corresponding to each reference number can be terminated at any time, regardless of the order of the CaptureHistoryStart calls.

CaptureHistoryStart

`CaptureHistoryStart()`

The CaptureHistoryStart function returns a reference number to identify a starting point in the history area text. Subsequently, the CaptureHistory function can be used to retrieve captured history text. See **CaptureHistory** for details.

catch

`catch`

The catch flow control keyword marks the beginning of code in a try-catch-endtry flow control construct for handling any abort conditions.

See Also

The **try-catch-endtry** flow control statement for details.

cd

`cd `*`dataFolderSpec`*

The cd operation sets the current data folder to the specified data folder. It is identical to the longer-named SetDataFolder operation.

cd is named after the UNIX "change directory" command.

See Also

SetDataFolder, **pwd**, **Dir**, **Data Folders** on page II-107

CDFFunc

`CDFFunc`

CDFFunc is a procedure subtype keyword that identifies a function as being suitable for calling from the **StatsKSTest** operation.

ceil

`ceil(`*`num`*`)`

The ceil function returns the closest integer greater than or equal to *num*.

The result for INF and NAN is undefined.

See Also

The **round**, **floor**, and **trunc** functions.

centerOfMass

`centerOfMass(`*`srcWave`*` [,`*`x1`*`,`*`x2`*`])`

The centerOfMass function returns the 1D center of mass for *srcWave* X values from x=*x1* to x=x2.

The centerOfMass function was added in Igor Pro 9.00.

Center of mass and center of gravity in a uniform gravity field are different terms for the same calculation. When the masses are of uniform density, the center of mass is also identical to the geometric centroid.

Details

The center of mass is defined as

$$centerMass = \frac{\sum x_i y_i}{\sum y_i},$$

where the summation is over all the points in *srcWave* or over the X range specified by the optional parameters x1 and x2.

Each term in the numerator above can be written as

$$x_i y_i = \left[DimOffset(srcWave,0) + i \cdot DimDelta(srcWave,0) \right] srcWave[i].$$

In this notation, y_i represents an individual mass at x = x_i, and the returned value x_c is the X location of the center of the aggregate mass.

See Also

centerOfMassXY, **mean**, **area**, **SumDimension**, **ImageAnalyzeParticles**

centerOfMassXY

`centerOfMassXY(waveX, waveY)`

The centerOfMassXY function returns the 1D center of mass xc for the pair of waves.

The centerOfMassXY function was added in Igor Pro 9.00.

You can obtain the center of mass in the orthogonal direction (yc) by reversing the order of arguments to the function.

Details

The center of mass is defined as

$$centrMassXY = \frac{\sum waveA[i]\ \ waveB[i]}{\sum waveB[i]}.$$

See Also

centerOfMass, **mean**, **areaXY**, **SumDimension**, **ImageAnalyzeParticles**

cequal

`cequal(`*`z1, z2`*`)`

The cequal function determines the equality of two complex numbers *z1* and *z2*. It returns 1 if they are equal, or 0 if not.

This is in contrast to the == operator, which compares only the real components of *z1* and *z2*, ignoring the imaginary components.

Examples

```
Function TestComplexEqualities()
   Variable/C z1= cmplx(1,2), z2= cmplx(1,-2)
   // This test compares only the real parts of z1 and z2:
   if( z1 == z2 )
      Print "== match"
   else
      Print "no == match"
   endif
   // This test compares both real and imaginary parts of z1 and z2:
   if( cequal(z1,z2) )
      Print "cequal match"
   else
      Print "no cequal match"
   endif
End
```

```
•TestComplexEqualities()
  == match
  no cequal match
```

See Also

The **imag**, **real**, and **cmplx** functions.

char2num

`char2num(`*`str`*`)`

The char2num function returns a numeric code representing the first byte of *str* or the first character of *str*.

If *str* contains zero bytes, char2num returns NaN.

If *str* contains exactly one byte, char2num returns the value of that byte, treated as a signed byte. For backward compatibility with Igor6, if the input is a single byte in the range 0x80..0xFF, char2num returns a negative number.

If *str* contains more than one byte, char2num returns a number which is the Unicode code point for the first character in *str* treated as UTF-8 text. If *str* does not start with a valid UTF-8 character, char2num returns NaN.

Prior to Igor Pro 7.00, char2num always returned the value of the first byte, treated as a signed byte.

Examples

```
Function DemoChar2Num()
   String str

   str = "A"
   Printf "Single ASCII character: %02X\r", char2num(str)          // Prints 0x41

   str = "ABC"
   Printf "Multiple ASCII characters: %02X\r", char2num(str)       // Prints 0x41

   str = U+2022        // Bullet character
   Printf "Valid UTF-8 text: U+%04X\r", char2num(str)               // Prints U+2022
   Printf "First byte value: %02X\r", char2num(str[0]) & 0xFF       // Prints E2
   Printf "Second byte value: %02X\r", char2num(str[1]) & 0xFF      // Prints 80

   str = num2char(0xE2, 0) + num2char(0x41, 0)                      // Invalid UTF-8 text
   Printf "Invalid UTF-8 text: U+%04X\r", char2num(str)             // Prints NaN

   str = ""
   Printf "Empty string: %g\r", char2num(str)                       // Prints NaN
End
```

See Also

The **num2char**, **str2num** and **num2str** functions.

Text Encodings on page III-459.

Chart

`Chart` `[`**`/Z`**`]` ***`ctrlName`*** `[`***`keyword = value`*** `[,` ***`keyword = value`*** `…]]`

The Chart operation creates or modifies a chart control. Charts are generally used in conjunction with data acquisition. Charts do not have to be connected to a FIFO, but they are not useful until they are.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the Chart control to be created or changed.

The following keyword=value parameters are supported:

align=*alignment* Sets the alignment mode of the control. The alignment mode controls the interpretation of the *leftOrRight* parameter to the pos keyword. The align keyword was added in Igor Pro 8.00.

If *alignment*=0 (default), *leftOrRight* specifies the position of the left end of the control and the left end position remains fixed if the control size is changed.

If *alignment*=1, *leftOrRight* specifies the position of the right end of the control and the right end position remains fixed if the control size is changed.

chans={*ch#*, *ch#*,…} List of FIFO channel numbers that Chart is to monitor.

color(*ch#*)=(*r*,*g*,*b*[,*a*]) Sets the color of the specified trace. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

ctab=*colortableName* When a channel is connected to an image strip FIFO channel, the data is displayed as an image using this built-in color table. Valid names are the same as used in images. Invalid name will result in the default Grays color table being used.

disable=*d* Sets user editability of the control.

d=0: Normal.

d=1: Hide.

d=2: Disable user input.

Charts do not change appearance because they are read-only. When disabled, the hand cursor is not shown.

fbkRGB=(*r*,*g*,*b*[,*a*]) Sets frame background color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

fgRGB=(*r*,*g*,*b*[,*a*]) Sets foreground color (text, etc.). *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

fifo=*FIFOName* Sets which named FIFO the chart will monitor. See the **NewFIFO** operation.

font="*fontName*" Sets the font used in the chart, e.g., `font="Helvetica"`.

fsize=*s* Sets font size for chart.

fstyle=*fs*
: Specifies the font style. *fs* is a bitwise parameter with each bit controlling one aspect of the font style:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

gain(*ch#*)=*g*
: Sets the display gain *g* of the specified channel relative to nominal. Values greater than unity expand the display.

gridRGB=(*r*,*g*,*b*[,*a*])
: Sets grid color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

help={*helpStr*}
: Specifies help for the control.

helpStr is limited to 1970 bytes (255 in Igor Pro 8 and before).

You can insert a line break by putting "`\r`" in a quoted string.

jumpTo=*p*
: Jumps to point number *p*. This works in review mode only.

lineMode(*ch#*)=*lm*
: Sets the display line mode for the given channel.

lm=0: Dots mode. Draws values as dots. However, if the number of dots in a strip exceeds maxDots then Igor draws a vertical line from the min to the max of the values packed into the strip.

lm=1: Lines mode. Draws a vertical line encompassing the min and the max of the points in a given strip along with the last point of the preceding strip. Since which strip is the preceding strip depends on the direction of motion then the appearance may slightly shift depending on which direction the chart is moving.

lm=2: Dots mode. Draws values as dots. However, if the number of dots in a strip exceeds maxDots then Igor draws a vertical line from the min to the max of the values packed into the strip.

mass=*m*
: Sets the "feel" of the chart paper when you move it with the mouse. The larger the mass *m*, the slower the chart responds. Odd values cause the movement of the paper to stop the instant the mouse is clicked while even values continue with the illusion of mass.

maxDots=*md*
: Controls whether points in a given vertical strip of the chart are displayed as dots or as a solid line. See lineMode above. Default is 20.

offset(*ch#*)=*o*
: Sets the display offset of the specified channel. The offset value *o* is subtracted from the data before the gain is applied.

oMode=*om*
: Chart operation mode.

om=0: Live mode.

om=1: Review mode.

pbkRGB=(*r*,*g*,*b*[,*a*])
: Sets plot area background color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

pos={*leftOrRight*,*top*}
: Sets the position in **Control Panel Units** of the top/left corner of the control if its alignment mode is 0 or the top/right corner of the control if its alignment mode is 1. See the align keyword above for details.

ppStrip=*pps*
: Number of data points packed into each vertical strip of the chart.

rSize(*ch#*)=*rs*
: Sets the relative vertical size allocated to the given channel. Nominal is unity. If the value of *rs* is zero then this channel shares space with the previous channel.

sMode=*sm* Status line mode.

- *sm*=0: Turns off fancy status line and positioning bar.
- *sm*=1: Normal mode.
- *sm*=2: Uses alternate style for bar.

sRate=*sr* Sets the scroll rate (vertical strips/second). If the chart control is in review mode negative numbers scroll in reverse.

title=*titleStr* Specifies the chart title. Use "" for no title.

uMode=*um* Status line mode.

- *um*=1: Fast update with no bells and whistles.
- *um*=2: Status line and positioning bar.
- *um*=3: Status line, positioning bar, and animated pens.

win=*winName* Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

The target window must be a graph or panel.

The action of some of the Chart keywords depends on whether or not data acquisition is taking place. If the chart is in review mode then all keywords cause the chart to be redrawn. If data acquisition is taking place and the chart is in live mode then some keywords affect new data but do not attempt to update the part of the "paper" that has already been drawn. The following keywords affect only new data during live mode:

```
ppStrip, maxDots, gain, offset, color, lineMode
```

See Also

Charts on page III-415 and **FIFOs and Charts** on page IV-313.

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

The **ControlInfo** operation for information about the control.

The **GetUserData** function for retrieving named user data.

chebyshev

chebyshev(*n*, *x*)

The chebyshev function returns the Chebyshev polynomial of the first kind and of degree *n*.

The Chebyshev polynomials satisfy the recurrence relation:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

with:

$$T_0(x) = 1$$
$$T_1(x) = x$$
$$T_2(x) = 2x^2 - 1.$$

The orthogonality of the polynomial is expressed by the integral:

$$\int_{-1}^{1}\frac{T_n(x)T_m(x)}{\sqrt{1-x^2}}dx=\begin{cases}0 & m\neq n\\ \pi/2 & m=n\neq 0\\ \pi & m=m=0\end{cases}\quad .$$

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

See Also

chebyshevU.

chebyshevU

chebyshevU(*n*, *x*)

The chebyshevU function returns the Chebyshev polynomial of the second kind, degree *n* and argument *x*.

The Chebyshev polynomial of the second kind satisfies the recurrence relation

```
U(n+1,x)=2xU(n,x)-U(n-1,x)
```

which is also the recurrence relation of the Chebyshev polynomials of the first kind.

The first 10 polynomials of the second kind are:

$U(0,x)=1$

$U(1,x)=2x$

$U(2,x)=4x^2-1$

$U(3,x)=8x^3-4x$

$U(4,x)=16x^4-12x^2+1$

$U(5,x)=32x^5-32x^3+6x$

$U(6,x)=64x^6-80x^4+24x-1$

$U(7,x)=128x^7-192x^5+80x^3-8x$

$U(8,x)=256x^8-448x^6+240x^4-40x^2+1$

$U(9,x)\,512x^9-1024x^{\wedge 7}+672x^5-160x^3+10x$

See Also

The **chebyshev** function.

CheckBox

CheckBox [**/Z**] ***ctrlName*** [***keyword = value*** [, ***keyword = value*** …]]

The CheckBox operation creates or modifies a checkbox, radio button or disclosure triangle in the target or named window, which must be a graph or control panel.

ctrlName is the name of the checkbox.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the CheckBox control to be created or changed.

The following keyword=value parameters are supported:

align=*alignment*

Sets the alignment mode of the control. The alignment mode controls the interpretation of the *leftOrRight* parameter to the pos keyword. The align keyword was added in Igor Pro 8.00.

If *alignment*=0 (default), *leftOrRight* specifies the position of the left end of the control and the left end position remains fixed if the control size is changed.

If *alignment*=1, *leftOrRight* specifies the position of the right end of the control and the right end position remains fixed if the control size is changed.

appearance={*kind* [, *platform*]}
Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or *All*.

See **Button** and **DefaultGUIControls** for more appearance details.

disable=*d*
Sets user editability of the control.

d=0: Normal.
d=1: Hide.
d=2: Disable user input.

fsize=*s*
Sets font size for checkbox.

fColor=(*r*,*g*,*b*[,*a*])
Sets the initial color of the title. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

To further change the color of the title text, use escape sequences as described for title=*titleStr*.

focusRing=*fr*
Enables or disables the drawing of a rectangle indicating keyboard focus:

fr=0: Focus rectangle will not be drawn.
fr=1: Focus rectangle will be drawn (default).

On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.

help={*helpStr*}
Sets the help for the control.

helpStr is limited to 1970 bytes (255 in Igor Pro 8 and before).

You can insert a line break by putting "`\r`" in a quoted string.

mode=*m*
Specifies checkbox appearance.

m=0: Default checkbox appearance.
m=1: Display as a radio button control.
m=2: Display as a disclosure triangle (*Macintosh*) or treeview expansion node (*Windows*).

noproc
Specifies that no procedure is to execute when clicking the checkbox.

picture= *pict*
Draws the checkbox using the named picture. The picture is taken to be six side-by-side frames which show the control appearance in the normal state, when the mouse is down, and in the disabled state. The first three frames are used when the checked state is false and the next three show the true state. The picture may be either a global (imported) picture or a Proc Picture (see **Proc Pictures** on page IV-56).

pos={*leftOrRight*,*top*}
Sets the position in **Control Panel Units** of the top/left corner of the control if its alignment mode is 0 or the top/right corner of the control if its alignment mode is 1. See the align keyword above for details.

pos+={*dx*,*dy*}
Offsets the position of the checkbox in **Control Panel Units**.

proc=*procName*
Specifies the procedure to execute when the checkbox is clicked.

rename=*newName*
Renames the checkbox to *newName*.

side=s
Sets the location of the title relative to the box:

s =0: Checkbox is on the left, title is on the right (default).
s =1: Checkbox is on the right, title is on the left.

size={*width*,*height*}
Sets checkbox size in **Control Panel Units**.

title=*titleStr*
Sets title of checkbox to the specified string expression. The title is the text that appears in the checkbox. If not given or if `""` then the title will be “New”.

Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.

userdata(*UDName*)= *UDStr*
Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+= *UDStr*
Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

value=*v*
Specifies whether the checkbox is selected (*v*=1) or not (*v*=0).

variable=*varName*
Specifies a global numeric variable to be set to the current state of a checkbox whenever it is clicked or when it is set by the value parameter. The variable is two-way: setting the variable also changes the state of the checkbox.

wave=*waveName*
Specifies a point from a wave to be set to the current state of a checkbox when it is clicked or when it is set by the value keyword. The point is specified using standard bracket notation with either a numeric point number or a row label, e.g., `value=awave[4]` or `value=awave[%alabel]`. You may also use a 2D, 3D, or 4D wave and specify a column, layer, and chunk index or dimension label in addition to the row index. This feature was added in Igor Pro 9.00.

win=*winName*
Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Flags

/Z
No error reporting.

Details

The target window must be a graph or panel.

Checkbox Action Procedure

The action procedure for a CheckBox control can takes a predefined structure `WMCheckboxAction` as a parameter to the function:

```
Function ActionProcName(CB_Struct) : CheckBoxControl
   STRUCT WMCheckboxAction &CB_Struct
   …
   return 0
End
```

The “`: CheckboxControl`” designation tells Igor to include this procedure in the Procedure pop-up menu in the Checkbox Control dialog.

See **WMCheckboxAction** for details on the WMCheckboxAction structure.

Although the return value is not currently used, action procedures should always return zero.

You may see an old format checkbox action procedure in old code:

```
Function procName(ctrlName,checked) : CheckBoxControl
   String ctrlName
   Variable checked        // 1 if selected, 0 if not
   …
   return 0
End
```

This old format should not be used in new code.

When using radio button controls, it is the responsibility of the Igor programmer to turn off other radio buttons when one of a group of radio buttons is pressed.

Examples

```
// Create a radio button group

Window Panel0() : Panel
   PauseUpdate; Silent 1         // building window …
   NewPanel /W=(150,50,353,212)
   Variable/G gSelectedRadioButton = 1
   CheckBox radioButton1,pos={52,25},size={78,15},title="Radio 1"
   CheckBox radioButton1,value=1,mode=1,proc=MyRadioButtonProc
   CheckBox radioButton2,pos={52,45},size={78,15},title="Radio 2"
   CheckBox radioButton2,value=0,mode=1,proc=MyRadioButtonProc
   CheckBox radioButton3,pos={52,65},size={78,15},title="Radio 3"
   CheckBox radioButton3,value= 0,mode=1,proc=MyRadioButtonProc
EndMacro

static Function HandleRadioButtonClick(controlName)
   String controlName

   NVAR gSelectedRadioButton = root:gSelectedRadioButton

   strswitch(controlName)
      case "radioButton1":
         gSelectedRadioButton = 1
         break
      case "radioButton2":
         gSelectedRadioButton = 2
         break
      case "radioButton3":
         gSelectedRadioButton = 3
         break
   endswitch
   CheckBox radioButton1, value = gSelectedRadioButton==1
   CheckBox radioButton2, value = gSelectedRadioButton==2
   CheckBox radioButton3, value = gSelectedRadioButton==3
End

Function MyRadioButtonProc(cb) : CheckBoxControl
   STRUCT WMCheckboxAction& cb

   switch(cb.eventCode)
      case 2:          // Mouse up
         HandleRadioButtonClick(cb.ctrlName)
      break
   endswitch

   return 0
End
```

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

The **ControlInfo** operation for information about the control.

The **GetUserData** function for retrieving named user data.

CheckBoxControl

`CheckBoxControl`

CheckBoxControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined checkbox control. See **Procedure Subtypes** on page IV-204 for details. See **CheckBox** for details on creating a checkbox control.

CheckDisplayed

CheckDisplayed [/A/W] *waveName* [, *waveName*]…

The CheckDisplayed operation determines if named waves are displayed or otherwise used in a host window or subwindow.

Flags

/A	Checks all graphs, tables, page layout, panels, and Gizmo windows.
/W=*winName*	Checks only the named window When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

If neither /A nor /W are used, CheckDisplayed checks only the top graph, table, page layout, panel, or Gizmo window.

CheckDisplayed sets a bit in the variable V_flag for each wave that is displayed.

Waves that are not directly displayed in a window can still be used by it. For example, CheckDisplayed returns 1 for the following cases: waves associated with hidden traces, color index waves, waves used to specify error bars, and waves used to draw polygons in page layouts and control panels as well as in graphs.

Example

```
// Checks Graph0 to see if aWave, bWave, and cWave are displayed in it.
// Sets bit 0 of V_flag if aWave is displayed.
// Sets bit 1 of V_flag if bWave is displayed.
// Sets bit 2 of V_flag if cWave is displayed.
CheckDisplayed/W=Graph0 aWave,bWave,cWave
```

See Also

Setting Bit Parameters on page IV-12 for information about bit settings.

CheckName

CheckName(*nameStr*, *objectType* [, *windowNameStr*])

The CheckName function returns a number which indicates if the specified name is legal and unique among objects in the namespace of the specified object type.

In Igor Pro 9.00 or later, you can use the **CreateDataObjectName** function as a replacement for some combination of CheckName, CleanupName, and UniqueName to create names of waves, global variables, and data folders.

Waves, global numeric variables, and global string variables are all in the same namespace and need to be unique only within the data folder containing them. However, they also need to be distinct from names of Igor operations and functions and from names of user-defined procedures.

Data folders are in their own namespace and need to be unique only among other data folders at the same level of the data folder hierarchy.

windowNameStr is optional. If missing, it is taken to be the top graph, panel, layout, or notebook according to the value of *objectType*.

Details

A result of zero indicates that the name is legal and unique within its namespace. Any nonzero result indicates that the name is illegal or not unique. You can use the **CleanupName** and **UniqueName** functions to guarantee legality and uniqueness.

nameStr should contain an unquoted name (i.e., no single quotes for liberal names), such as you might receive from the user through a dialog or control panel.

objectType is one of the following:

The *windowNameStr* argument is used only with *objectTypes* 14, 15, and 16. The *nameStr* is checked for uniqueness only within the named window (other windows might have objects with the given name). If a named window is given but does not exist, any valid *nameStr* is permitted

1: Wave.
2: Reserved.
3: Global numeric variable.
4: Global string variable.
5: XOP target window.
6: Graph window.
7: Table window.
8: Layout window.
9: Control panel window.
10: Notebook window.
11: Data folder.
12: Symbolic path.
13: Picture.
14: Annotation in the named or topmost graph or layout.
15: Control in the named topmost graph or panel.
16: Notebook action character in the named or topmost notebook.

CheckName Thread Safety

As of Igor Pro 8.00, you can call CheckName from an Igor preemptive thread but only if *objectType* is 1 (wave), 3 (global numeric variable), 4, (global string variable), 11 (data folder), or 12 (symbolic path). For any other value of *objectType*, CheckName returns a runtime error.

Examples

```
Variable waveNameIsOK = CheckName(proposedWaveName, 1) == 0
Variable annotationNameIsOK = CheckName("text0", 14, "Graph0") == 0


// Create a valid and unique wave name
Function/S CreateValidAndUniqueWaveName(proposedName)
   String proposedName

   String result = proposedName

   if (CheckName(result,1) != 0)           // 1 for waves
      result = CleanupName(result, 1)      // Make sure it's valid
      result = UniqueName(result, 1, 0)    // Make sure it's unique
   endif

   return result
End
```

See Also

Object Names on page III-501, **Programming with Liberal Names** on page IV-168, **CreateDataObjectName**, **CleanupName**, **UniqueName**

ChildWindowList

ChildWindowList(*hostNameStr*)

The ChildWindowList function returns a string containing a semicolon-separated list of immediate subwindow window names of the specified host window or subwindow.

Parameters

hostNameStr is a string or string expression containing the name of an existing host window or subwindow.

When identifying a subwindow with *hostNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

Error if the host does not exist or if it is not an allowed host type.

See Also

WinList and **WinType** functions.

ChooseColor

ChooseColor [**/A[=a]/C=(*r*,*g*,*b*[,*a*])**]

The ChooseColor operation displays a dialog for choosing a color.

The color initially shown is black unless you specify a different color with /C.

Flags

/A[=*a*]	a=1 shows the alpha (opacity) channel. /A is the same as /A=1. a=0 hides the alpha channel. This is the default setting. The /A flag was added in Igor Pro 7.00.
/C=(*r*,*g*,*b*[,*a*])	Sets the color initially displayed in the dialog. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

Details

ChooseColor sets the variable V_flag to 1 if the user clicks OK in the dialog or to 0 otherwise.

If V_flag is 1 then V_Red, V_Green, V_Blue, and V_Alpha are set to the selected color as integers from 0 to 65535.

A fully opaque color sets V_Alpha=65535. A fully transparent color sets V_Alpha=0.

See Also

ImageTransform **rgb2hsl** and **hsl2rgb**.

CleanupName

CleanupName(*nameStr*, *beLiberal* [, *maxBytes*])

The CleanupName function returns the input name string, possibly altered to make it a legal object name.

The *maxBytes* parameter requires Igor Pro 8.00 or later.

In Igor Pro 9.00 or later, you can use the **CreateDataObjectName** function as a replacement for some combination of CheckName, CleanupName, and UniqueName to create names of waves, global variables, and data folders.

Parameters

nameStr must contain an unquoted (i.e., no single quotes for liberal names) name, such as you might receive from the user through a dialog or control panel.

beLiberal is 0 to use strict name rules or 1 to use liberal name rules. Strict rules allow only letters, digits and the underscore character. Liberal rules allow other characters such as spaces and dots. Liberal names are allowed for waves and data folders only.

maxBytes is the maximum number of bytes allowed in the result. This parameter requires Igor Pro 8.00 or later. *maxBytes* is optional, defaults to 255, and is clipped to the range 1..255.

Prior to Igor Pro 8.00, Igor names were limited to 31 bytes so CleanupName never returned names longer than 31 bytes. In Igor Pro 8.00 or later, names for most types of objects may be up to 255 bytes so CleanupName may return very long names. You may want to use the *maxBytes* parameter to prevent the use of inconveniently-long names.

If *nameStr* includes non-ASCII characters, which in UTF-8 consist of multiple bytes, CleanupName clips the name at a character boundary if clipping is required.

Details

A cleaned up name is not necessarily unique. Call **CheckName** to check for uniqueness or **UniqueName** to ensure uniqueness.

Prior to Igor8, all object names were limited to 31 bytes. Now, for most types of objects, names can be up to 255 bytes. CleanupName always allows up to 255 bytes. Global picture names and notebook ruler names are still limited to 31 bytes so, if you are cleaning up those names, you must test for long names yourself. See **Long Object Names** on page III-502 for details.

If a cleaned up name is liberal, you may need to quote it. See **Programming with Liberal Names** on page IV-168 for details.

Examples

```
String cleanStrVarName = CleanupName(proposedStrVarName, 0)
```

```
// In UTF-8, the "±" character consists of two bytes: 0xC2 and 0XB1
Print CleanupName("±", 1, 1)  // maxBytes=1; returns "" (empty string - 0 bytes)
Print CleanupName("±", 1, 2)  // maxBytes=2; returns "±" (2 bytes)
```

See Also

Object Names on page III-501, **Programming with Liberal Names** on page IV-168, **CreateDataObjectName**, **CheckName**, **UniqueName**

Close

```
Close [/A] fileRefNum
```

The Close operation closes a file previously opened by the **Open** operation or closes all such files if /A is used.

Parameters

fileRefNum is the file reference number of the file to close. This number comes from the Open operation. If /A is used, *fileRefNum* should be omitted.

Flags

/A	Closes all files. Mainly useful for cleaning up after an error during procedure execution occurs so that the normal Close operation is never executed.

CloseHelp

```
CloseHelp [ /ALL /FILE=fileNameStr /NAME=helpNameStr /P=pathName ]
```

The CloseHelp operation closes a help window.

The CloseHelp operation was added in Igor Pro 7.00.

Flags

/ALL	Closes all open help windows.
/FILE=*fileNameStr*	Identifies the help window to close using the help file's location on disk. The file is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.
/NAME=*helpNameStr*	Identifies the help window to close using the window's title as specified by *helpNameStr*. This is the text that appears in the help window title bar.
/P=*pathName*	Specifies the folder to look in for the file specified by /FILE. *pathName* is the name of an existing Igor symbolic path.

Details

You must provide one of the following flags: /ALL, /FILE, /NAME.

See Also

OpenHelp

CloseMovie

```
CloseMovie
```

The CloseMovie operation closes the currently open movie. You must close a movie before you can play it.

Flags

/Z	Suppresses error reporting. If you use /Z, check the V_Flag output variable to see if the operation succeeded.

Output Variables

`V_Flag`	Set to 0 if the operation succeeded or to a non-zero error code. V_Flag is set only if you use the /Z flag.

See Also

Movies on page IV-245, **NewMovie**

CloseProc

`CloseProc /NAME=`*`procNameStr`* `[`***`flags`***`]`

`CloseProc /FILE=`*`fileNameStr`* `[`***`flags`***`]`

The CloseProc operation closes a procedure window. You cannot call CloseProc on the main Procedure window.

CloseProc provides a way to programmatically create and alter procedure files. You might do this in order to make a user-defined menu-bar menu with contents that change.

Note: CloseProc alters procedure windows so it cannot be called while functions or macros are running. If you want to call it from a function or macro, use **Execute/P**.

Warning: If you close a procedure window that has no source file or without specifying a destination file, the window contents will be permanently lost.

Flags

/COMP[=*compile*] Specifies whether procedures should be compiled after closing the procedure window.

compile=1: Compiles procedures (same as /COMP only).

compile=0: Leaves procedures in an uncompiled state.

/D[=*delete*] Specifies whether the procedure file should be deleted after closing the procedure window.

delete=1: Deletes procedure file (same as /D only).

Warning: You cannot recover any file deleted this way.

delete=0: Leaves any associated file unaffected.

/FILE=*fileNameStr* Identifies the procedure window to close using the file name and path to the file given by *fileNameStr*. The string can be just the file name if /P is used to specify a symbolic path name of the enclosing folder. It can be a partial path if /P points to a folder enclosing the start of the partial path. It can also be a full path the file.

/NAME=*procNameStr* Identifies the procedure window to close with the string expression *procNameStr*. This is the same text that appears in the window title. If the procedure window is associated with a file, it will be the file name and extension.

To close a procedure file that is part of an independent module, you must include the independent module name in *procNameStr*. For example:

```
CloseProc /NAME="GraphBrowser.ipf [WM_GrfBrowser]"
```

Note that there is a space after the file name followed by the independent module name in brackets.

/P=*pathName* Specifies the folder to look in for the file specified by /FILE. *pathName* is the name of an existing symbolic path.

/SAVE[=*savePathStr*] Saves the procedure before closing the window. If the flag is used with no argument, it saves any changes to the procedure window to its source file before closing it. If *savePathStr* is present, it must be a full path naming a file in which to save the procedure window contents. The /P flag is not used with *savePathStr* so it must be a full path.

Details

CloseProc cannot be called from a macro or function. Call it from the command line or via Execute/P (see **Operation Queue** on page IV-278).

Specify which window to close using either the /NAME or /FILE flag. You must use one or the other. Usually you would use /NAME, as it is usually more convenient. If by some chance two procedures have the same name, /FILE can be used to distinguish between them.

You cannot call CloseProc on a nonmain procedure window that someone has had the bad taste to call "Procedure".

See Also

Chapter III-13, **Procedure Windows**.

The **Execute/P** operation.

cmplx

cmplx(*realPart*, *imagPart*)

The cmplx function returns a complex number whose real component is *realPart* and whose imaginary component is *imagPart*.

Use this to assign a value to a complex variable or complex wave.

Examples

Assume wave1 is complex. Then:

```
wave1(0) = cmplx(1,2)
```

sets the Y value of wave1 at x=0 such that its real component is 1 and its imaginary component is 2.

Assuming wave2 and wave3 are real, then:

```
wave1 = cmplx(wave2,wave3)
```

sets the real component of wave1 equal to the contents of wave2 and the imaginary component of wave1 equal to the contents of wave3.

You may get unexpected results if the number of points in wave2 or wave3 differs from the number of points in wave1. If wave2 or wave3 are shorter than wave1, the last element of the short wave is copied repeatedly to fill wave1.

See Also

conj, **imag**, **magsqr**, **p2rect**, **r2polar**, and **real** functions.

CmpStr

CmpStr(*str1*, *str2* [, *flags*])

The CmpStr function returns zero if *str1* is equal to *str2* or non-zero otherwise.

Parameters

flags controls the type of comparison that is done. It defaults to 0 and supports the following values:

0:	Case-insensitive text comparison
1:	Case-sensitive text comparison
2:	Binary comparison

Binary comparison (*flags*=2) was added Igor Pro 7.05 and is appropriate if *str1* and *str2* contain binary data, which may contain null bytes (bytes with the value 0), rather than human-readable strings.

Details

If *flags* is omitted, 0, or 1, CmpStr does a text comparison and returns the following values:

-1:	*str1* is alphabetically before *str2*
0:	*str1* and *str2* are equal
1:	*str1* is alphabetically after *str2*

The alphabetic order represented by -1 and 1 is valid only for ASCII text. It is not valid for non-ASCII text, such as text containing accented characters.

If *flags* is 2, CmpStr does a binary comparison and returns the following values:

0:	*str1* and *str2* are equal
1:	*str1* and *str2* are not equal

Usually strings do not contain null bytes. Text comparison treats a null byte as meaning "end of string". Binary comparison does not treat a null byte specially. The example below illustrates the treatment of null bytes.

Example

```
Function TestCmpStr()
    String str1 = "A" + num2char(0) + "B"       // num2char(0) is a null byte
    Print strlen(str1)                          // Prints 3

    String str2 = "A" + num2char(0) + "C"
    Print strlen(str2)// Prints 3

    Print CmpStr(str1,str2,0)                   // Prints 0 meaning "equal"
    Print CmpStr(str1,str2,1)                   // Prints 0 meaning "equal"
    Print CmpStr(str1,str2,2)                   // Prints 1 meaning "unequal"
End
```

The "Print strlen" statements illustrate that strlen does not treat null as meaning end of string.

The results from CmpStr with *flags*=0 and *flags*=1 illustrate that CmpStr treated null as meaning "end of string" and consequently compared only the first byte of *str1* to the first byte of *str2*. Both are "A" so CmpStr returned 0 meaning "equal".

The result from CmpStr with *flags*=2 illustrates that CmpStr did not treat null as meaning "end of string" and consequently compared all three bytes of *str1* to all three bytes of *str2*. The third bytes do not match so CmpStr returned 1 meaning "unequal".

See Also

ReplaceString, **Using Strings** on page IV-172

ColorScale

`ColorScale` [***`flags`***] [, ***`keyword = value,`*** …] [***`axisLabelStr`***]

The ColorScale operation adds a color scale (or “color legend”) annotation to the a graph, Gizmo plot or page layout. For background information, see **Color Scales** on page III-47.

The ability to add colorscales to a Gizmo plot was added in Igor Pro 8.00.

The ColorScale operation can be executed with no flags and no parameters. It acts on the current target window or the active subwindow or on the window or subwindow specified by the /W flag.

When the target is a graph, the color scale represents the colors and values associated with the first image plot that was added to the graph. If there is no image plot in the graph, the color scale represents the first contour plot or first f(z) trace added to the graph, one of which must exist for the command to execute without error when executed without parameters.

When the target is a Gizmo plot, the color scale represents the colors and values associated with the first surface object, preferring the surface fill color table over the surface gridlines color table. If there is no such surface object, the color scale represents the first scatter object using a color table. If there is no such scatter object, the color scale represents the first path object using a color table. If there is no such path object, the color scale represents the first ribbon object using a color table.

Executing ColorScale with no parameters when the target is a page layout displays a color bar as if the `ctab={0,100,Rainbow}` parameters had been specified.

Flags

Use the /W=*winName* flag to specify a specific graph or layout window. When used on the command line or in a Macro or Proc, /W must precede all other flags.

To change a color scale, use the /C/N=*name* flags. Annotations are automatically named “text0”, “text1”, etc. if no name is specified when it is created, and you must use that name to modify an existing annotation or else a new one will be created.

For explanations of all flags see the **TextBox** operation.

Parameters

The following keyword-value pairs are the important parameters to the ColorScale operation, because they specify what object the color scale is representing.

For use with page layouts, the keyword ={*graphName*,…} form is required.

For graphs it is simpler to use the image=*imageInstanceName* form (omitting graphName), though you can use `$""` to mean the top graph, or specify the name of another graph with the long form. See the **Image Plot ColorScale Examples**.

axisLabelStr

Contains the text printed beside the color scale's main axis.

Using escape codes you can change the font, size, style and color of text, create superscripts and subscripts, create dynamically-updated text, insert legend symbols, and apply other effects. See **Annotation Escape Codes** on page III-53 for details.

You can use **AppendText** or **ReplaceText** to modify this axis label string. The default value for axisLabelStr is `""`.

cindex=*cindexMatrixWave*

The colors shown are those in the named color index wave with axis values derived from the wave's X (row) scaling and units.

The image colors are determined by doing a lookup in the specified matrix wave. See the ModifyImage cindex keyword.

contour=*contourInstanceName*

contour={*graphName*,*contourInstanceName*}

The colors show the named contour plot's line colors and associated contour (Z) values and contour data units. All of the image plot's characteristics are represented, including color table, cindex, and fixed colors.

graphName can be just the name of a top-level graph window or a subwindow specification like Panel0#G0. See **Subwindow Syntax** on page III-92 for details on subwindow specifications.

contourFill=*contourInstanceName*

contourFill={*graphName*,*contourInstanceName*}

The colors show the named contour plot's fill colors and associated contour (Z) values and contour data units. All of the image plot's characteristics are represented, including color table, cindex, and fixed colors.

graphName can be just the name of a top-level graph window or a subwindow specification like Panel0#G0. See **Subwindow Syntax** on page III-92 for details on subwindow specifications.

The contourFill keyword was added in Igor Pro 8.00.

ctab={*zMin*,*zMax*,*ctName*, *reverse*}

The color table specified by `ctName` is drawn in the color legend.

zMin and *zMax* set the range of z values to map.

The color table name can be omitted if you want to leave it unchanged.

ctName can be any color table name returned by the CTabList function, such as Grays or Rainbow (see **Image Color Tables** on page II-392) or the name of a 3 column or 4 column color table wave (**Color Table Waves** on page II-399).

A color table wave name supplied to `ctab` must not be the name of a built-in color table (see **CTabList**). A 3 column or 4 column color table wave must have values that range between 0 and 65535. Column 0 is red, 1 is green, and 2 is blue. In column 3 a value of 65535 is fully opaque and 0 is fully transparent.

Set *reverse* to 1 to reverse the color table. Setting it to 0 or omitting it leaves the color table unreversed.

image=*imageInstanceName*

image={*graphName*,*imageInstanceName*}

The colors show the named image plot's colors and associated image (Z) values and image data units. All of the image plot's characteristics are represented, including color table, cindex, lookup wave, eval colors, and NaN transparency. **Note**: only false-color image plots can be used with ColorScale (see **Indexed Color Details** on page II-400).

graphName can be just the name of a top-level graph window or a subwindow specification like Panel0#G0. See **Subwindow Syntax** on page III-92 for details on subwindow specifications.

logLabel=*t*

t is the maximum number of decades before minor tick labels are suppressed. The default value is 3.

The logLabel keyword was added in Igor Pro 7.00.

logTicks=*t*

t=0 means "auto". This is the default value.

If *t* is not 0 then it represents the maximum number of decades before minor ticks are suppressed.

The logLabel keyword was added in Igor Pro 7.00.

lookup=*waveName*

Specifies an optional 1D wave used to modify the mapping of scaled Z values into the color table specified with the ctab keyword. Values should range from 0.0 to 1.0. A linear ramp from 0 to 1 would have no effect while a ramp from 1 to 0 would reverse the image. Used to apply gamma correction to grayscale images or for special effects. Use `lookup=$""` to remove option.

This keyword is not needed with the image keyword, even if the image plot uses a lookup wave. The image plot's lookup wave is used instead of the ColorScale lookup wave.

path=*gizmoObjectSpec*

path={*gizmoName*, *gizmoObjectSpec*}

The colors show the named Gizmo path plot's colors and associated X, Y, or Z values and units. Only the path plot's color table is represented. Other color modes such as color wave or solid colors are not supported.

gizmoName can be just the name of a top-level Gizmo window or a subwindow specification like Panel0#GZ0. See **Subwindow Syntax** on page III-92 for details on subwindow specifications.

gizmoObjectSpec can be just the name of the object like path0, or a colon-separated path to the object in a group like group0:path0. Unlike **ModifyGizmo** setCurrentGroup, *gizmoObjectSpec* does not start with the name of the Gizmo window.

The path keyword was added in Igor Pro 9.00.

ribbon=*gizmoObjectSpec*

ribbon={*gizmoName*, *gizmoObjectSpec*}

The colors show the named Gizmo ribbon plot's colors and associated X, Y, or Z values and units. Only the ribbon plot's color table is represented. Other color modes such as color wave or solid colors are not supported.

See the path keyword for {*gizmoName*, *gizmoObjectSpec*} details.

The ribbon keyword was added in Igor Pro 9.00.

scatter=*gizmoObjectSpec*

scatter={*gizmoName*, *gizmoObjectSpec*}

The colors show the named Gizmo scatter plot's marker colors and associated X, Y, or Z values and units. Only the scatter plot's color table is represented. Other color modes such as color wave or solid colors are not supported.

See the path keyword for {*gizmoName*, *gizmoObjectSpec*} details.

The scatter keyword was added in Igor Pro 9.00.

surface=*gizmoObjectSpec*

surface={*gizmoName*,*gizmoObjectSpec*}

The colors show the named Gizmo surface plot's grid line colors and associated surface X, Y, or Z values and surface matrix units. Unlike image plots, only the surface plot's color table is represented. Other color modes such as color wave or solid colors are not supported.

See the path keyword for {*gizmoName*, *gizmoObjectSpec*} details.

The surface keyword was added in Igor Pro 8.00.

surfaceFill=*gizmoObjectSpec*

surfaceFill={*gizmoName*,*gizmoObjectSpec*}

The colors show the named Gizmo surface plot's fill colors and associated surface X, Y, or Z values and surface matrix units. Unlike image plots, only the surface plot's color table is represented. Other color modes such as color wave or solid colors are not supported.

See the path keyword for {*gizmoName*, *gizmoObjectSpec*} details.

The surfaceFill keyword was added in Igor Pro 8.00.

trace=*traceInstanceName*

trace={*graphName*,*traceInstanceName*}

The colors show the color(s) of the named trace. This is useful when the trace has its color set by a “Z wave” using the `ModifyGraph zColor(`*`traceName`*`)=`... feature. In the Modify Trace Appearance dialog this is selected in the “Set as f(z)” subdialog. The color scale’s main axis shows the range of values in the Z wave, and displays any data units the wave may have.

graphName can be just the name of a top-level graph window or a subwindow specification like Panel0#G0. See **Subwindow Syntax** on page III-92 for details on subwindow specifications.

Size Parameters

The following keyword-value parameters modify the size of the color scale annotation. These keywords are similar to those used by the **Slider** control. The size of the annotation is indirectly controlled by setting the size of the “color bar” and the various axis parameters. The annotation sizes itself to accommodate the color bar, tick labels, and axis labels.

height=*h*
Sets the height of the color bar in points, overriding any heightPct setting. The default height is 75% of the plot area height if the color scale is vertical, or a constant of 15 points if the color scale is horizontal. The default is restored by specifying `height=0`. Specifying a heightPct value resets height to this default.

heightPct=*hpct*
Sets height as a percentage of the graph’s plot area, overriding any height setting. The default height is 75% of the plot area height if the color scale is vertical, or a constant of 15 points if the color scale is horizontal. The default height is restored by setting heightPct=0. Specifying a height value resets heightPct to this default.

side=*s*
Selects on which axis to draw main axis ticks.

s=1: Right of the color bar if vert=1, or below if vert=0.

s=2: Left of the color bar if vert=1, or above if vert=0.

vert=*v*
Specifies color scale orientation.

v=0: Horizontal.

v=1: Vertical (default).

width=*w*
Sets the width of the color bar in points, overriding any widthPct setting. The default width is a constant 15 points if the color scale is vertical, or 75% of the plot area width if the color scale is horizontal. The default is restored by specifying `width=0`. Specifying a widthPct value resets width to this default.

widthPct=*wpct*
Sets width as a percentage of the graph’s plot area, overriding any width setting. The default width is a constant 15 points if the color scale is vertical, or 75% of the plot area width if the color scale is horizontal. The default is restored by setting `widthPct=0`. Specifying a width value resets widthPct to this default.

Color Bar Parameters

The following keyword-value parameters modify the appearance of the color scale color bar.

colorBoxesFrame=*on*
Draws frames surrounding up to 99 swatches of colors in the color bar (*on*=1).

When specifying more than 99 colors in the color bar (such as the Rainbow color table, which has 100 colors), the boxes aren’t framed. Framing color boxes is effective only for small numbers of colors. Set the width of the frame with the frame keyword.

Use *on*=0 to turn off color box frames.

frame=*f*
Specifies the thickness of the frame drawn around the color bar in points (*f* can range from 0 to 5 points). The default is 1 point. Fractional values are permitted.

Turn frames off with *f*=0. Values less that 0.5 do not display on screen, but the thin frame will print.

frameRGB=(*r*,*g*,*b*[,*a*]) or 0
Sets the color of the frame around the color bar. *r*, *g*, and *b* specify the amount of red, green, and blue as integers from 0 to 65535. The frame includes the individual color bar colors when colorBoxesFrame=1.

The frame will use the colorscale foreground color, as set by the /G flag, when frameRGB=0.

Axis Parameters

The following keyword-value parameters modify the appearance of the color scale axes. These keywords are based on the **ModifyGraph** Axis keywords because they modify the main or secondary color scale axes.

axisRange={*zMin*, *zMax*}

Sets the color bar axis range to values specified by *zMin* and *zMax*. Use `*` to use the default axis range for either or both values.

Omit *zMin* or *zMax* to leave that end of the range unchanged. For example, use `{zMin, }` to change *zMin* and leave *zMax* alone, or use `{ ,*}` to set only the axis maximum value to the default value.

dateInfo={*sd*,*tm*,*dt*}

Controls formatting of date/time axes.

sd=0:	Show date in the date&time format.
sd=1:	Suppress date.
sd=2:	Suppress time. The time is always shown if *tm*=2. *sd*=2 requires Igor Pro 9.00 or later.
tm=0:	12 hour (AM/PM) time.
tm=1:	24 hour (military) time.
tm=2:	Elapsed time.
dt=0:	Short dates (2/22/90).
dt=1:	Long dates (Thursday, February 22, 1990).
dt=2:	Abbreviated dates (Thurs, Feb 22, 1990).

These have no effect unless the axis is controlled by a wave with 'dat' data units.

These date formats do not work with dates before 0001-01-01 in which case the axis displays an empty string. You can apply custom date formats using the dateFormat keyword.

To use a custom date format as specified by the dateFormat keyword, you must specify -1 for the *dt* parameter to dateInfo.

For an f(z) color scale:

```
SetScale d, 0,0, "dat", fOfZWave
```

For a contour plot or image plot color scale:

```
SetScale d, 0,0, "dat", ZorXYZorImageWave
```

See **Date/Time Axes** on page II-315 and **Date, Time, and Date&Time Units** on page II-69 for details on how date/time axes work.

font=*fontNameStr*

Name of font as string expression. If the font does not exist, the default font is used. Specifying “default” has the same effect. Unlike ModifyGraph, the *fontNameStr* is evaluated at runtime, and its absence from the system is not an error.

fsize=*s*

Sets the font size in points.

s=0:	Use the graph font size for tick labels and axis labels (default).

fstyle=*fs*

Sets the font style. *fs* is a bitwise parameter with each bit controlling one aspect of the font style for the tick mark labels:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

highTrip=*h*

If the extrema of an axis are between its lowTrip and its highTrip then tick mark labels use fixed point notation. Otherwise they use exponential (scientific or engineering) notation. The default highTrip is 100,000.

lblLatPos=*p*
Sets a lateral offset for the main axis label. This is an offset parallel to the axis. *p* is in points. Positive is down for vertical axes and to the right for horizontal axes. The default is 0.

lblMargin=*m*
Moves the main axis label by *m* points (default is 0) from the normal position. The default value is -5, which brings the axis label closer to the axis. Use more positive values to move the axis label away from the axis.

lblRot=*r*
Rotates the axis label by *r* degrees. *r* is a value from -360 to 360. Rotation is counterclockwise and starts from the label's normal orientation.

log=*l*
Specifies the axis type:

l=0: Linear (default).
l=1: Log base 10.
l=2: Log base 2.

logHTrip=*h*
Same as highTrip but for log axes. The default is 10,000.

logLTrip=*l*
Same as lowTrip but for log axes. The default is 0.0001.

logTicks=*t*
Specifies the maximum number of decades in log axis before minor ticks are suppressed.

lowTrip=*l*
If the axis extrema are between its lowTrip and its highTrip, then tick mark labels use fixed point notation. Otherwise, they use exponential (scientific or engineering) notation. The default lowTrip is 0.1.

minor=*m*
Controls minor tick marks:

m=0: Disables minor ticks (default).
m=1: Enables minor ticks.

notation=*n*
Controls tick label notation:

n=0: Engineering notation (default).
n=1: Scientific notation.

nticks=*n*
Specifies the approximate number of ticks to be distributed along the main axis. Ticks are labelled using the same automatic algorithm used for graph axes. The default is 5.

Set *n*=0 for no ticks.

prescaleExp=*exp*
Multiplies axis range by 10*exp* for tick labeling and *exp* is subtracted from the axis label exponent. In other words, the exponent is moved from the tick labels to the axis label. The default is 0 (no modification). See the discussion in the **ModifyGraph (axes) Details** section.

tickExp=*te*
Controls tick label exponential notation:

te=1: Forces tick labels to exponential notation when labels have units with a prefix.
te=0: Turns off exponential notation.

tickLen=*t* Sets the length of the ticks. *t* is the major tick mark length in points. This value must be between -100 and 50.

t= 0 to 50:	Draws tick marks between the tick labels and the colors box.
t= -1:	Default; auto tick length, equal to 70% of the tick label font size. Draws tick marks between the tick labels and the colors box.
t= -2 to -50:	Draws tick marks crossing the edge of the colors box nearest the tick labels. The actual total tick mark length is -*t*.
t= -100 to -51:	Draws tick marks inside the edge of the colors box nearest the tick labels. Actual tick mark length is -(*t*+50). For example, -58 makes in an inside tick mark that is 8 points long.

tickThick=*t* Sets the tick mark thickness in points (from 0 to 5 points). The default is 1 point. Fractional values are permitted.

t=0:	Turns tick marks off, but not the tick labels.

tickUnit=*tu* Turns on (*tu*=0) or off (*tu*=1) units labels attached to tick marks.

userTicks={*tvWave*,*tlblWave*}

Supplies user defined tick positions and labels for the main axis. *tvWave* contains the numeric tick positions while text wave *tlblWave* contains the corresponding labels.

Overrides normal ticking specified by nticks.

See **User Ticks from Waves** on page II-313 for details.

The tick mark labels can be multiline and use styled text. For more details, see **Fancy Tick Mark Labels** on page II-358.

ZisZ=*z* *z* =1 labels the zero tick mark (if any) with the single digit “0” regardless of the number of digits used for other labels. Default is *z*=0.

Secondary Axis Parameters

axisLabel2=*axisLabelString2*

Axis label for the secondary axis. This axis label is drawn only if userTicks2 is in effect. Text after any `\r` character is ignored, as is the `\r` character. The default is `""`.

lblLatPos2=*p* Sets lateral offset for secondary axis labels. This is an offset parallel to the axis. *p* is in points. Positive is down for vertical axes and to the right for horizontal axes. The default is 0.

lblMargin2=*m* Specifies the distance in points (default 0) to move the secondary axis label from the position that would be normal for a graph. The default is value is -5, which brings the axis label closer to the axis. Use more positive values to move the axis label away from the axis.

lblRot2=*r* Rotates the secondary axis label by *r* degrees counterclockwise starting from the normal label orientation. r is a value from -360 to 360.

userTicks2={*tvWave*,*tlblWave*}

Supplies user defined tick positions and labels for a second axis which is always on the opposite side of the color bar from the main axis. The tick mark labels can be multiline and use styled text. For more details, see **Fancy Tick Mark Labels** on page II-358. This is the only way to draw a second axis.

Image Plot ColorScale Examples

```
Make/O/N=(20,20) img=p*q; NewImage img         // Make and display an image
ColorScale                                     // Create default color scale
// First annotation is text0
ColorScale/C/N=text0 nticks=3,minor=1,"Altitude"
ModifyImage img ctab= {*,*,Relief19,0}         // 19-color color table
ColorScale/C/N=text0 axisRange={100,300}       // Detail for 100-300 range
ColorScale/C/N=text0 colorBoxesFrame=1         // Frame the color boxes
ColorScale/C/N=text0 frameRGB=(65535,0,0)      // Red frame
```

Gizmo Plot ColorScale Example

See the online reference help for ColorScale.

See Also

For all other flags see the **TextBox** and **AppendText** operations.

Color Scales on page III-47, **AnnotationInfo**, **AnnotationList**

Demo

Choose File→Example Experiments→Feature Demos 2→ ColorScale Demo

ColorTab2Wave

ColorTab2Wave *colorTableName*

The ColorTab2Wave operation extracts colors from the built-in color table and places them in an Nx3 matrix of red, green, and blue columns named M_colors. Values are unsigned 16-bit integers and range from 0 to 65535.

N will typically be 100 but may be as little as 9 and as large as 476. Use

```
Variable N= DimSize(M_colors,0)
```

to determine the actual number of colors.

The wave M_colors is created in the current data folder. Red is in column 0, green is in column 1, and blue in column 2.

Parameters

colorTableName can be any of those returned by **CTabList**, such as Grays or Rainbow.

colorTableName can also be `Igor` or `IgorRecent`, to return either the 128 standard or 0-32 user-selected colors from Igor's color menu.

Details

See **Image Color Tables** on page II-392.

Complex

Complex *localName*

Declares a local complex 64-bit double-precision variable in a user-defined function or structure.

Complex is another name for Variable/C. It is available in Igor Pro 7.00 and later.

Concatenate

Concatenate [***type flags***][***flags***] ***waveListStr*, *destWave***

Concatenate [***type flags***][***flags***] **{*wave1*, *wave2*, *wave3*,…}, *destWave***

Concatenate [***type flags***][***flags***] **{*waveWave*}, *destWave***

The Concatenate operation combines data from the source waves into *destWave*, which is created if it does not already exist. If *destWave* does exists and overwrite is not specified, the source waves' data is concatenated with the existing data in the destination wave.

By default the concatenation increases the dimensionality of the destination wave if possible. For example, if you concatenate two 1D waves of the same length you get a 2D wave with two columns. The destination wave is said to be "promoted" to a higher dimensionality.

If you use the /NP (no promotion) flag, the dimensionality of the destination wave is not changed. For example, if you concatenate two 1D waves of the same length using /NP you get a 1D wave whose length is the sum of the lengths of the source waves.

If the source waves are of different lengths, no promotion is done whether /NP is used or not.

Parameters

waveListStr is a string expression containing a list of input wave names separated by semicolons with a semicolon at the end. There is no limit to the number of wave names in *waveListStr*.

The {*wave1*, *wave2*, ...} syntax is limited to 100 waves.

In the {*waveWave*} syntax, waveWave is a single WAVE reference wave containing references to the input waves. This syntax was added in Igor Pro 8.00.

Flags

/DL	Sets dimension labels. For promotion, it uses source wave names as new dimension labels otherwise it uses existing labels.
/FREE	Creates destWave as a free wave (see **Free Waves** on page IV-91). The /FREE flag was added in Igor Pro 8.00.
/KILL	Kills source waves.
/NP	Prevents promotion to higher dimension.
/NP=*dim*	Prevents promotion and appends data along the specified dimension (0= rows, 1= columns, 2=layers, 3=chunks). All dimensions other than the one specified by *dim* must be the same in all waves.
/O	Overwrites *destWave*.

Type Flags *(used only in functions)*

Concatenate also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-73 and **WAVE Reference Type Flags** on page IV-74 for a complete list of type flags and further details.

Details

If *destWave* does not already exist or, if the /O flag is used, *destWave* is created by duplication of the first source wave. Waves are concatenated in order through the list of source waves. If *destWave* exists and the /O flag is not used, then the concatenation starts with *destWave*.

destWave cannot be used in the source wave list.

Source waves must be either all numeric or all text.

If promotion is allowed, the number of low-order dimensions that all waves share in common determines the dimensionality of *destWave* so that the dimensionality of *destWave* will then be one greater. The default behaviors will vary according to the source wave sizes. Concatenating 1D waves that are all the same length will produce a 2D wave, whereas concatenating 1D waves of differing lengths will produce a 1D wave. Similarly, concatenating 2D waves of the same size will produce a 3D wave; but if the 2D source waves have differing numbers of columns then *destWave* will be a 2D wave, or if the 2D waves have differing numbers of rows then *destWave* will be a 1D wave. Concatenating 1D and 2D waves that have the same number of rows will produce a 2D wave, but when the number of rows differs, *destWave* will be a 1D wave. See the examples.

Use the /NP flag to suppress dimension promotion and keep the dimensionality of *destWave* the same as the input waves.

Warning

Under some circumstances, such as in loops in user-defined functions, Concatenate may exhibit unexpected behavior.

When you have a statement like this in a user-defined function:

```
Concatenate/O ..., DestWaveName
```

at compile time, Igor creates an automatic local wave reference variable named DestWaveName. At runtime, if the wave reference variable is NULL, the name is taken to be a literal name and a wave of that name is created in the current data folder.

If the wave reference variable is not NULL, as would occur after the first call to Concatenate in a loop, then the referenced wave is overwritten no matter where it is located.

If your intention is to create or overwrite a wave in the current data folder, you should use one of the following two methods:

```
Concatenate/O ..., $"DestWaveName"
WAVE DestWaveName       // Needed only if you subsequently reference the dest wave
```

or

```
Concatenate/O ....., DestWaveName
// Then after you are finished using DestWaveName...
WAVE DestWaveName=$""
```

Examples

```
// Given the following waves:
Make/N=10 w1,w2,w3
Make/N=11 w4
Make/N=(10,7) m1,m2,m3
Make/N=(10,8) m4
Make/N=(9,8) m5

// Concatenate 1D waves
Concatenate/O {w1,w2,w3},wdest          // wdest is a 10x3 matrix
Concatenate {w1,w2,w3},wdest            // wdest is a 10x6 matrix
Concatenate/NP/O {w1,w2,w3},wdest       // wdest is a 30-point 1D wave
Concatenate/O {w1,w2,w3,w4},wdest       // wdest is a 41-point 1D wave

// Concatenate 2D waves
Concatenate/O {m1,m2,m3},wdest          // wdest is a 10x7x3 volume
Concatenate/NP/O {m1,m2,m3},wdest       // wdest is a 10x21 matrix
Concatenate/O {m1,m2,m3,m4},wdest       // wdest is a 10x29 matrix
Concatenate/O {m4,m5},wdest             // wdest is a 152-point 1D wave
Concatenate/O/NP=0 {m4,m5},wdest        // wdest is a 19x8 matrix

// Concatenate 1D and 2D waves
Concatenate/O {w1,m1},wdest             // wdest is a 10x8 matrix
Concatenate/O {w4,m1},wdest             // wdest is a 81-point 1D wave

// Append rows to 2D wave
Make/O/N=(3,2) m6, m7
Concatenate/NP=0 {m6}, m7               // m7 is a 6x2 matrix

// Append columns to 2D wave
Make/O/N=(3,2) m6, m7
Concatenate/NP=1 {m6}, m7               // m7 is a 3x4 matrix

// Append layer to 2D wave
Make/O/N=(3,2) m6, m7
Concatenate/NP=2 {m6}, m7               // m7 is a 3x2x2 volume
// The last command has the same effect as:
// Concatenate {m6}, m7
// Both versions extend add a third dimension to m7
```

See Also

Duplicate, **Redimension**, **SplitWave**

conj

`conj(`*`z`*`)`

The conj function returns the complex conjugate of the complex value *z*.

See Also

cmplx, **imag**, **magsqr**, **p2rect**, **r2polar**, and **real** functions.

Constant

`Constant kName = literalNumber`

`Constant/C kName = (literalNumberReal, literalNumberImag)`

The Constant declaration defines the number *literalNumber* under the name *kName* for use by other code, such as in a switch construct.

The complex form, using the /C flag to create a complex constant, requires Igor Pro 7.00 or later.

See Also

The **StrConstant** keyword for string types, **Constants** on page IV-51 and **Switch Statements** on page IV-43.

continue

`continue`

The continue flow control keyword returns execution to the beginning of a loop, bypassing the remainder of the loop's code.

See Also

Continue Statement on page IV-48 and **Loops** on page IV-45 for usage details.

ContourInfo

`ContourInfo(graphNameStr, contourWaveNameStr, instanceNumber)`

The ContourInfo function returns a string containing a semicolon-separated list of information about the specified contour plot in the named graph.

Parameters

graphNameStr can be `""` to refer to the top graph.

contourWaveNameStr is a string containing either the name of a wave displayed as a contour plot in the named graph, or a contour instance name (wave name with "#n" appended to distinguish the nth contour plot of the wave in the graph). You might get a contour instance name from the **ContourNameList** function.

If *contourWaveNameStr* contains a wave name, *instanceNumber* identifies which instance you want information about. *instanceNumber* is usually 0 because there is normally only one instance of a wave displayed as a contour plot in a graph. Set *instanceNumber* to 1 for information about the second contour plot of the wave, etc. If *contourWaveNameStr* is `""`, then information is returned on the *instanceNumber*th contour plot in the graph.

If *contourWaveNameStr* contains an instance name, and *instanceNumber* is zero, the instance is taken from *contourWaveNameStr*. If *instanceNumber* is greater than zero, the wave name is extracted from *contourWaveNameStr*, and information is returned concerning the *instanceNumber*th instance of the wave.

Details

The string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

Keyword	Information Following Keyword
AXISFLAGS	Flags used to specify the axes. Usually blank because /L and /B (left and bottom axes) are the defaults.
DATAFORMAT	Either XYZ or Matrix.
LEVELS	A comma-separated list of the contour levels, including the final automatic levels, (or manual or from-wave levels), and the "more levels", all sorted into ascending Z order.
RECREATION	List of keyword commands as used by **ModifyContour** command. The format of these keyword commands is: ***keyword (x)=modifyParameters;***
TRACESFORMAT	The format string used to name the contour traces (see **AppendMatrixContour** or **AppendXYZContour**).

Keyword	Information Following Keyword
XAXIS	X axis name.
XWAVE	X wave name if any, else blank.
XWAVEDF	Full path to the data folder containing the X wave or blank if there is no X wave.
YAXIS	Y axis name.
YWAVE	Y wave name if any, else blank.
YWAVEDF	Full path to the data folder containing the Y wave or blank if there is no Y wave.
ZWAVE	Name of wave containing Z data from which the contour plot was calculated.
ZWAVEDF	Full path to the data folder containing the Z data wave.

The format of the RECREATION information is designed so that you can extract a keyword command from the keyword and colon up to the “`;`”, prepend “`ModifyContour`”, replace the “`x`” with the name of a contour plot (“`data#1`” for instance) and then **Execute** the resultant string as a command.

Examples

The following command lines create a very unlikely contour display. If you did this, you would most likely want to put each contour plot on different axes, and arrange the axes such that they don’t overlap. That would greatly complicate the example.

```
Make/O/N=(20,20) jack
Display;AppendMatrixContour jack
AppendMatrixContour/T/R jack        // Second instance of jack
```

This example accesses the contour information for the second contour plot of the wave “jack” (which has an instance number of 1) displayed in the top graph:

```
Print StringByKey("ZWAVE", ContourInfo("","jack",1))     // prints jack
```

See Also

The **Execute** and **ModifyContour** operations.

ContourNameList

`ContourNameList(`*`graphNameStr`*`, `*`separatorStr`*`)`

The ContourNameList function returns a string containing a list of contours in the graph window or subwindow identified by *graphNameStr*.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

The parameter *separatorStr* should contain a single ASCII character such as “`,`” or “`;`” to separate the names.

A contour name is defined as the name of the wave containing the data from which a contour plot is calculated, with an optional #n suffix that distinguishes between two or more contour plots in the same graph window that have the same wave name. Since the contour name has to be parsed, it is quoted if necessary.

Examples

The following command lines create a very unlikely contour display. If you did this, you would most likely want to put each contour plot on different axes, and arrange the axes such that they don’t overlap. That would greatly complicate the example.

```
Make/O/N=(20,20) jack,'jack # 2';
Display;AppendMatrixContour jack
AppendMatrixContour/T/R jack
AppendMatrixContour 'jack # 2'
AppendMatrixContour/T/R 'jack # 2'
Print ContourNameList("",";")
```

prints `jack;jack#1;'jack # 2';'jack # 2'#1;`

See Also

Another command related to contour plots and waves: **ContourNameToWaveRef**.

For commands referencing other waves in a graph: **TraceNameList**, **WaveRefIndexed**, **XWaveRefFromTrace**, **TraceNameToWaveRef**, **CsrWaveRef**, **CsrXWaveRef**, **ImageNameList**, and **ImageNameToWaveRef**.

ContourNameToWaveRef

`ContourNameToWaveRef(`*`graphNameStr`*`, `*`contourNameStr`*`)`

Returns a wave reference to the wave corresponding to the given contour name in the graph window or subwindow named by *graphNameStr*.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

The contour name is identified by the string in *contourNameStr*, which could be a string determined using ContourNameList. Note that the same contour name can refer to different waves in different graphs, if the waves are in different data folders.

See Also

The **ContourNameList** function.

For a discussion of wave reference functions, see **Wave Reference Functions** on page IV-197.

ContourZ

`ContourZ(`*`graphNameStr`*`, `*`contourInstanceNameStr`*`, `*`x`*`, `*`y`*` [,`*`pointFindingTolerance`*`])`

The ContourZ function returns the interpolated Z value of the named contour plot data displayed in the named graph.

For gridded contour data, ContourZ returns the bilinear interpolation of the four surrounding XYZ values.

For XYZ triplet contour data, ContourZ returns the value interpolated from the three surrounding XYZ values identified by the Delaunay triangulation.

Parameters

graphNameStr can be `""` to specify the topmost graph.

contourNameStr is a string containing either the name of the wave displayed as a contour plot in the named graph, or a contour instance name (wave name with “#n” appended to distinguish the nth contour plot of the wave in the graph). You might get a contour instance name from the **ContourNameList** function.

If *contourNameStr* contains a wave name, *instance* identifies which contour plot of *contourNameStr* you want information about. *instance* is usually 0 because there is normally only one instance of a wave displayed as a contour plot in a graph. Set *instance* to 1 for information about the second contour plot of *contourNameStr*, etc. If *contourNameStr* is `""`, then information is returned on the *instance*th contour plot in the graph.

If *contourNameStr* contains an instance name, and *instance* is zero, the instance is taken from *contourNameStr*. If *instance* is greater than zero, the wave name is extracted from *contourNameStr*, and information is returned concerning the *instance*th instance of the wave.

x and *y* specify the X and Y coordinates of the value to be returned. This may or may not be the location of a data point in the wave selected by *contourNameStr* and *instance*.

Set *pointFindingTolerance* =1e-5 to overcome the effects of perturbation (see the perturbation keyword of the **ModifyContour** operation).

The default value is 1e-15 to account for rounding errors created by the triangulation scaling (see ModifyContour's equalVoronoiDistances keyword), which works well ModifyContour perturbation=0.

A value of 0 would require an exact match between the scaled x/y coordinate and the scaled and possibly perturbed coordinates to return the original z value; that is an unlikely outcome.

Details

For gridded contour data, ContourZ returns NaN if *x* or *y* falls outside the XY domain of the contour data. If *x* and *y* fall on the contour data grid, the corresponding Z value is returned.

For XYZ triplet contour data, ContourZ returns the null value if *x* or *y* falls outside the XY domain of the contour data. You can set the null value to *v* with this command:

```
ModifyContour contourName nullValue=v
```

If *x* and *y* match one of the XYZ triplet values, the corresponding Z value from the triplet usually won't be returned because Igor uses the Watson contouring algorithm which perturbs the x and y values by a small random amount. This also means that normally x and y coordinates on the boundary will return a null value about half the time if perturbation is on and *pointFindingTolerance* is greater than 1e-5.

Examples

Because ContourZ can interpolate the Z value of the contour data at any X and Y coordinates, you can use ContourZ to convert XYZ triplet data into gridded data:

```
// Make example XYZ triplet contour data
Make/O/N=50 wx,wy,wz
wx= enoise(2)          // x = -2 to 2
wy= enoise(2)          // y = -2 to 2
wz= exp(-(wx[p]*wx[p] + wy[p]*wy[p]))       // XY gaussian, z= 0 to 1

// ContourZ requires a displayed contour data set
Display; AppendXYZContour wz vs {wx,wy};DelayUpdate
ModifyContour wz autolevels={*,*,0}         // no contour levels are needed
ModifyContour wz xymarkers=1                // show the X and Y locations

// Set the null (out-of-XY domain) value
ModifyContour wz nullValue=NaN              // default is min(wz) - 1

// Convert to grid: Make matrix that spans X and Y
Make/O/N=(30,30) matrix
SetScale/I x, -2, 2, "", matrix
SetScale/I y, -2, 2, "", matrix
matrix= ContourZ("","wz",0,x,y)             // or = ContourZ("","",0,x,y)
AppendImage matrix
```

See Also

AppendMatrixContour, **AppendXYZContour**, **ModifyContour**, **FindContour**, **zcsr**, **ContourInfo**

References

Watson, David F., *Contouring: A Guide To The Analysis and Display of Spatial Data*, Pergamon, 1992.

ControlBar

ControlBar [***flags***] ***barHeight***

The ControlBar operation sets the height and location of the control bar in a graph.

Parameters

barHeight is in points on Macintosh and pixels or points on Windows, depending on the screen resolution. See **Control Panel Resolution on Windows** on page III-456 for details.

Setting *barHeight* to zero removes the control bar.

Flags

/EXP=*e*	Sets the expansion of the panel control bar area. See the **NewPanel** operation /EXP flag for details. The /EXP flag was added in Igor Pro 9.00.
/L/R/B/T	Designates whether to use the left, right, bottom, or top (default) window edge, respectively, for the control bar location.
/W=*graphName*	Specifies the name of a particular graph containing a control bar.

Details

The control bar is an area at the top of graphs reserved for controls such as buttons, checkboxes and pop-up menus. A line is drawn between this area and the graph area. The control bar may be assigned a separate background color by pressing Control (*Macintosh*) or Ctrl (*Windows*) and clicking in the area, or by right-clicking it (*Windows*), or with the **ModifyGraph** operation. You cannot use draw tools in this area.

For graphs with no controls, you do not need to use this operation.

Use ControlInfo kwControlBar operation to determine the current size of the control bar.

Examples

```
Display myData
ControlBar 35       // 35 pixels high
Button button0,pos={56,8},size={90,20},title="My Button"
```

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Chapter V-1, **ControlInfo**

ControlInfo

ControlInfo [***/W=winName***] ***controlName***

The ControlInfo operation returns information about the state or status of the named control in a graph or control panel window or subwindow.

Flags

/G [=*doGlobal*]	If *doGlobal* is non-zero or absent, the position returned via V_top and V_left is in global screen coordinates rather relative to the window containing the control.
/W=*winName*	Looks for the control in the named graph or panel window or subwindow. If /W is omitted, ControlInfo looks in the top graph or panel window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Parameters

controlName is the name of a control in the window specified by /W or in the top graph or panel window. *controlName* may also be the keyword kwBackgroundColor to set V_Red, V_Green, V_Blue, and V_Alpha, the keyword kwControlBar or kwControlBarBottom to set V_Height, the keyword kwControlBarLeft or kwControlBarRight to set V_Width, or the keyword kwSelectedControl to set S_value and V_flag.

Details

Information for all controls is returned via the following string and numeric variables. Coordinates are returned in **Control Panel Units**.
The kind of control is returned in V_flag as a positive or negative integer. A negative value indicates the control is incomplete or not active. If V_flag is zero, then the named control does not exist. Information returned for specific control types is as follows:

S_recreation Commands to recreate the named control.

S_title Title of the named control.

V_disable Disable state of control:

0: Normal (enabled, visible).

1: Hidden.

2: Disabled, visible.

V_Height, V_Width, V_top, V_left, V_right, V_pos, V_align Dimensions and position of the named control in **Control Panel Units**s.

V_right, V_pos, and V_align were added in Igor Pro 8.00 and depend on whether the align keyword was applied to the control (e.g., Button <name> align=1).

V_align is 1 if the the align keyword was applied to the control or to 0 otherwise.

V_pos is the horizontal coordinate used to position the control. It represents the position of the left end of the control if the align keyword was omitted or of the right end if the align keyword was applied.

V_left and V_right are the the horizontal coordinates of the left and right ends of the control regardless of whether the align keyword was applied or not.

Buttons

V_flag 1

V_value Tick count of last mouse up.

S_UserData Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left, V_right, V_pos, and V_align at the beginning of the Details section.

Chart

V_flag 6 or -6

V_value Current point number.

S_UserData Keyword-packed information string. See **S_value for Chart Details** for more keyword information.

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left, V_right, V_pos, and V_align at the beginning of the Details section.

Checkbox

V_flag 2

V_value 0 if it is deselected or 1 if it is selected.

S_UserData Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left, V_right, V_pos, and V_align at the beginning of the Details section.

CustomControl

V_flag 12

V_value Tick count of last mouse up.

S_UserData Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.

S_value Name of the picture used to define the control appearance.

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left, V_right, V_pos, and V_align at the beginning of the Details section.

GroupBox

V_flag 9

S_value Title text.

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left, V_right, V_pos, and V_align at the beginning of the Details section.

ListBox

V_flag 11

V_value Currently selected row (valid for mode 1 or 2 or modes 5 and 6 when no selWave is used). If no list row is selected, then it is set to -1.

V_selCol Currently selected column (valid for modes 5 and 6 when no selWave is used).

V_horizScroll Number of pixels the list has been scrolled horizontally to the right.

V_vertScroll Number of pixels the list has been scrolled vertically downwards.

V_rowHeight Height of a row in pixels.

V_startRow The current top visible row.

S_columnWidths A comma-separated list of column widths in pixels.

S_dataFolder Full path to listWave (if any).

S_UserData Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.

S_value Name of listWave (if any).

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left, V_right, V_pos, and V_align at the beginning of the Details section.

PopupMenu

V_flag 3 or -3

V_Red, V_Green, V_Blue. V_Alpha For color array pop-up menus, these are the encoded color values.

V_value Current item number (counting from one).

S_UserData Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.

S_value Text of the current item. If PopupMenu is a color array then it contains color values encoded as **RGBA Values**.

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left, V_right, V_pos, and V_align at the beginning of the Details section.

SetVariable

V_flag 5 or -5

V_value Value of the variable. If the SetVariable is used with a string variable, then it is the interpretation of the string as a number, which will be NaN if conversion fails.

S_dataFolder Full path to the variable.

S_UserData Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.

S_value Name of the variable or, if the value was set using _STR: syntax, the string value itself.

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left, V_right, V_pos, and V_align at the beginning of the Details section.

Slider

V_flag 7

V_value Numeric value of the variable.

S_dataFolder Full path to the variable.

S_UserData Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.

S_value Name of the variable.

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left, V_right, V_pos, and V_align at the beginning of the Details section.

TabControl

V_flag 8

V_value Number of the current tab.

S_UserData Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.

S_value Tab text.

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left, V_right, V_pos, and V_align at the beginning of the Details section.

TitleBox

V_flag 10

S_dataFolder Full path if text is from a string variable.

S_value Name if text is from a string variable.

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, and V_left at the beginning of the Details section.

See **TitleBox Positioning** on page V-1040 for details on V_right, V_pos, and V_align.

ValDisplay

V_flag 4 or -4

V_value Displayed value.

S_value Text of expression that ValDisplay evaluates.

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left, V_right, V_pos, and V_align at the beginning of the Details section.

kwBackgroundColor

V_Red, V_Green, V_Blue, V_Alpha	If *controlName* is kwBackgroundColor then this is the color of the control panel background. This color is usually the default user interface background color, as set by the Appearance control panel on the Macintosh or by the Appearance tab of the Display Properties on Windows, until changed by `ModifyPanel cbRGB`.

kwControlBar or kwControlBarTop

V_Height	The height in pixels of the top control bar area in a graph as set by **ControlBar**.

kwControlBarBottom

V_Height	The height in pixels of the bottom control bar area in a graph as set by **ControlBar**/B.

kwControlBarLeft

V_Width	The width in pixels of the left control bar area in a graph as set by **ControlBar**/L.

kwControlBarRight

V_Width	The width in pixels of the right control bar area in a graph as set by **ControlBar**/R.

kwSelectedControl

V_flag	Set to 1 if a control is selected or 0 if not. **SetVariable** and **ListBox** controls can be selected, most other controls can not.
S_value	Name of selected control (if any) or `""`.

S_value for Chart Details

The following applies *only* to the keyword-packed information string returned in S_value for a chart. S_value will consist of a sequence of sections with the format: "*keyword*:*value*;" You can pick a value out of a keyword-packed string using the **NumberByKey** and **StringByKey** functions. Here are the S_value keywords:

Keyword	Type	Meaning
FNAME	string	Name of the FIFO chart is monitoring.
LHSAMP	number	Left hand sample number.
NCHANS	number	Number of channels displayed in chart.
PPSTRIP	number	The chart's points per strip value.
RHSAMP	number	Right hand sample number (same as V_value).

In addition, ControlInfo writes fields to S_value for each channel in the chart. The keyword for the field is a combination of a name and a number that identify the field and the channel to which it refers. For example, if channel 4 is named "Pressure" then the following would appear in the S_value string: "CHNAME4:Pressure". In the following table, the channel's number is represented by #:

Keyword	Type	Meaning
CHCTAB#	number	Channel's color table value as set by Chart ctab keyword.
CHGAIN#	number	Channel's gain value as set by Chart gain keyword.
CHNAME#	string	Name of channel defined by FIFO.
CHOFFSET#	number	Channel's offset value as set by Chart offset keyword.

Examples

```
ControlInfo myChart; Print S_value
```

Prints the following to the history area:

```
FNAME:myFIFO;NCHANS:1;PPSTRIP:1100;RHSAMP:271;LHSAMP:-126229;
```

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

The **ControlInfo** operation for information about the control.

The **GetUserData** function for retrieving named user data.

ControlNameList

`ControlNameList(`*`winNameStr`*` [, `*`listSepStr`*` [, `*`matchStr`*`]])`

The ControlNameList function returns a string containing a list of control names in the graph or panel window or subwindow identified by *winNameStr*.

Parameters

winNameStr can be `""` to refer to the top graph or panel window.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

The optional parameter *listSepStr* should contain a single ASCII character such as "," or ";" to separate the names; the default value is ";".

The optional parameter *matchStr* is some combination of normal characters and the asterisk wildcard character that matches anything. To use *matchStr*, *listSepStr* must also be used. See **StringMatch** for wildcard details.

Only control names that satisfy the match expression are returned. For example, "*_tab0" matches all control names that end with "_tab0". The default is "*", which matches all control names.

Examples

```
NewPanel
Button myButton
Checkbox myCheck
Print ControlNameList("")                    // prints "myButton;myCheck;"
Print ControlNameList("", ";", "*Check")    // prints "myCheck;"
```

See Also

The **ListMatch**, **StringFromList** and **StringMatch** functions, and the **ControlInfo** and **ModifyControlList** operations. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

ControlUpdate

`ControlUpdate [/A/W=`*`winName`*`][`*`controlName`*`]`

The ControlUpdate operation updates the named control or all controls in a window, which can be the top graph or control panel or the named graph or control panel if you use /W.

Flags

/A	Updates all controls in the window. You must omit *controlName*.
/W=*winName*	Specifies the window or subwindow containing the control. If you omit *winName* it will use the top graph or control panel window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

ControlUpdate is useful for forcing a pop-up menu to rebuild, to update a ValDisplay control, or to forcibly accept a SetVariable’s currently-being-edited value.

Normally, a pop-up menu rebuilds only when the user clicks on it. If you set up a pop-up menu so that its contents depend on a global string variable, on a user-defined string function or on an Igor function (e.g., **WaveList**), you may want to force the pop-up menu to be updated at your command.

Usually, a ValDisplay control displays the value of a global variable or of an expression involving a global variable. If the global variable changes, the ValDisplay will automatically update. However, you can create a ValDisplay that displays a value that does not depend on a global variable. For example, it might display the result of an external function. In a case like this, the ValDisplay will not automatically update. You can update it by calling ControlUpdate.

When a SetVariable control is being edited, the text the user types isn't "accepted" (or processed) until the user presses Return or Enter. ControlUpdate effectively causes the named control to act as though the user has pressed one of those keys. If /A is specified, the currently active SetVariable control (if any) is affected this way. The motivation here is that the user may have typed a new value without having yet pressed return, and then may click a button in a different panel which runs a routine that uses the SetVariable value as input. The user expected the typed value to have been accepted but the variable has not yet been set. Calling ControlUpdate/A on the first panel will read the typed value in the variable, avoiding a discrepancy between the visible value of the SetVariable control and the actual value of the variable.

Examples

```
NewPanel;DoWindow/C PanelX
String/G popupList="First;Second;Third"
PopupMenu oneOfThree value=popupList       // popup shows "First"
popupList="1;2;3"                          // popup is unchanged
ControlUpdate/W=PanelX oneOfThree          // popup shows "1"
```

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **ValDisplay** and **WaveList** operations.

ConvertGlobalStringTextEncoding

ConvertGlobalStringTextEncoding [flags] originalTextEncoding, newTextEncoding, [string , string, ...]

The ConvertGlobalStringTextEncoding operation converts the contents of the specified global string variables or the contents of all global string variables in the data folder specified with the /DF flag from one text encoding to another.

The ConvertGlobalStringTextEncoding operation was added in Igor Pro 7.00.

By default, the contents of global string variables containing control characters (codes less than 32) other than carriage return (13), linefeed (10), and tab (9) are not converted. To convert the contents of global string variables containing control characters, you must include the /SKIP=0 flag.

Also by default, in Igor Pro 9.00 or later, when converting to UTF-8, strings that are already valid as UTF-8 are skipped.

In Igor Pro 7 or later, the contents of all global string variables are assumed to be encoded as UTF-8. You should convert the text encoding of a global string variable if, for example, you have global string variables in experiments created prior to Igor7 and they contain non-ASCII characters. For those strings to be displayed and interpreted correctly, their contents need to be converted to UTF-8.

Most users will have no need to worry about the text encoding of Igor global string variables since most global string variables do not contain non-ASCII text. You should not use this operation unless you have a thorough understanding of text encoding issues or are instructed to use it by someone who has a thorough understanding.

See **String Variable Text Encodings** on page III-478 for essential background information.

ConvertGlobalStringTextEncoding can work on a list of specific global string variables or on all of the global string variables in a data folder (/DF flag). When working on a data folder, it can work on just the data folder itself or recursively on sub-data folders as well.

Conversion does not change the characters that make up text - it merely changes the numeric codes used to represent those characters.

Parameters

originalTextEncoding specifies the original (current) text encoding used by the specified global string variables. See **Text Encoding Names and Codes** on page III-490 for a list of codes. It will typically be 2 (MacRoman), 3 (Windows-1252) or 4 (Shift JIS), depending on the system on which the string variables were created.

newTextEncoding specifies the output text encoding. See **Text Encoding Names and Codes** on page III-490 for a list of codes. It will typically be 1 which stands for UTF-8.

string , *string* , ... is a list of targeted global string variables. Only the name of string variables, not a path specification plus the name, is allowed. Therefore, the string variables must be in the current data folder. The list is optional and must be omitted if you use the /DF flag. Using both the /DF flag and a list of string variables is treated as an error. Use of local string variables in this list is also an error. Use **ConvertTextEncoding** to convert local string variables.

Flags

/CONV={*errorMode* [, *diagnosticsFlags*]}

errorMode determines how ConvertGlobalStringTextEncoding behaves if the conversion can not be done because the text can not be mapped to the specified text encoding. This occurs if the string variable's original text is not valid in the specified *originalTextEncoding* or if it contains characters that can not be represented in the specified *newTextEncoding*.

errorMode takes one of these values:

1:	Generate error. SetWaveTextEncoding returns an error to Igor.
2:	Use a substitute character for any unmappable characters. The substitute character for Unicode is the Unicode replacement character, U+FFFD. For most non-Unicode text encodings it is either control-Z or a question mark.
3:	Skip unmappable input characters. Any unmappable characters will be missing in the output.
4:	Use escape sequences representing any unmappable characters or invalid source text. If the source text is valid in the source text encoding but can not be represented in the destination text encoding, unmappable characters are replaced with \uXXXX where XXXX specifies the UTF-16 code point of the unmappable character in hexadecimal. If the conversion can not be done because the source text is not valid in the source text encoding, invalid bytes are replaced with \xXX where XX specifies the value of the invalid byte in hexadecimal.

diagnosticsFlags is an optional bitwise parameter defined as follows:

Bit 0:	Emit diagnostic message if text conversion succeeds.
Bit 1:	Emit diagnostic message if text conversion fails.
Bit 2:	Emit diagnostic message if text conversion is skipped.
Bit 3:	Emit summary diagnostic message.

All other bits are reserved for future use.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

diagnosticsFlags defaults to 6 (bits 1 and 2 set) if the /DF flag is not present and to 14 (bits 1, 2 and 3 set) if the /DF flag is present.

ConvertGlobalStringTextEncoding skips text conversion if the global string variable's contents are detected as binary and you omit /SKIP=0.

/DF={*dfr, recurse, excludedDFR*}

dfr is a reference to a data folder. ConvertGlobalStringTextEncoding operates on all global string variables in the specified data folder. If *dfr* is null ($"") ConvertGlobalStringTextEncoding acts as if /DF was omitted.

If you use the /DF flag, you must omit the optional string variable list.

If *recurse* is 1, ConvertGlobalStringTextEncoding works recursively on all sub-data folders. Otherwise it affects only the data folder referenced by *dfr*.

excludedDFR is an optional reference to a data folder to be skipped by ConvertGlobalStringTextEncoding. For example, this command converts the string data for all global string variables in all data folders except for root:Packages and its sub-data folders:

```
ConvertGlobalStringTextEncoding /DF={root:,1,root:Packages} 4, 1
```

If *excludedDFR* is null ($"") ConvertGlobalStringTextEncoding acts as if *excludedDFR* was omitted and no data folders are excluded.

/SKIP=*skip* Skips conversion of a string variable as follows:

skip=0:	Do not skip conversion.
skip=1:	Skip conversion of string variables containing binary data.
skip=2:	When converting to UTF-8, skip conversion of strings that are already valid as UTF-8. This includes strings that contain only ASCII characters.
skip=3:	Skips both strings containing binary data and, when converting to UTF-8, strings that are already valid as UTF-8. This is the default behavior if /SKIP is omitted.

/SKIP=2 and /SKIP=3 require Igor Pro 9.00 or later.

/Z[=z] Prevents procedure execution from aborting if ConvertGlobalStringTextEncoding generates an error. Use /Z or the equivalent, /Z=1, if you want to handle errors in your procedures rather than having execution abort.

/Z does not suppress invalid parameter errors. It suppresses only errors in doing text encoding reinterpretation or conversion.

Details

For general background information on text encodings, see **Text Encodings** on page III-459.

For background information on string variable text encodings, see **String Variable Text Encodings** on page III-478.

Using ConvertGlobalStringTextEncoding

ConvertGlobalStringTextEncoding is used to change the numeric codes representing the text - i.e., to convert the content to a different text encoding. Its main use is to convert Igor Pro 6 global string variables from whatever text encoding they use to UTF-8. UTF-8 is a form of Unicode which is more modern but is not backward compatible with Igor Pro 6. Igor Pro 7 and later assume that string variables are encoded as UTF-8.

For example, if you have global string variables from Igor Pro 6 that are encoded in Japanese (Shift JIS), you should convert them to UTF-8. Otherwise you will get errors or garbled text when you print or display the strings. This also applies to western text containing non-ASCII characters encoded as MacRoman or Windows-1252.

If you have an Igor6 experiment containing non-ASCII string variables encoded as, for example, MacRoman, and you add some non-ASCII string variables in Igor7 or later, you now have a mix of MacRoman and UTF-8 non-ASCII string variables. In Igor9 or later, when you are converting to UTF-8, by default ConvertGlobalStringTextEncoding skips conversion of strings that are already valid as UTF-8. This allows you to convert the MacRoman string variables to UTF-8 without messing up the string variables that are already UTF-8.

Output Variables

The ConvertGlobalStringTextEncoding operation returns information in the following variables:

`V_numConversionsSucceeded` V_numConversionsSucceeded is set to the number of successful text conversions.

`V_numConversionsFailed` V_numConversionsFailed is set to the number of unsuccessful text conversions.

`V_numConversionsSkipped` V_numConversionsSkipped is set to the number of skipped text conversions. Text conversion is skipped if the string variable contains binary data and the /SKIP=0 flag is omitted.

Text conversion is skipped by default if the string variable contains binary data. Text conversion is also skipped by default if *newTextEncoding* is 1 (UTF-8) and the string variable is already valid as UTF-8. See the /SKIP flag for details.

Examples

```
// In the following examples 1 means UTF-8, 4 means Shift JIS.

// Convert specific strings' content from Shift JIS to UTF-8
ConvertGlobalStringTextEncoding 4, 1, string0, string1

// Convert all strings' content from Shift JIS to UTF-8
ConvertGlobalStringTextEncoding /DF={root:,1} 4, 1

// Same as before but exclude the root:Packages data folder
ConvertGlobalStringTextEncoding /DF={root:,1,root:Packages} 4, 1

// Convert all strings’ content from Shift JIS to UTF-8 except strings containing binary
ConvertGlobalStringTextEncoding /DF={root:,1}/SKIP=0 4, 1
```

See Also

Text Encodings on page III-459, **String Variable Text Encodings** on page III-478, **Text Encoding Names and Codes** on page III-490

ConvertTextEncoding, **SetWaveTextEncoding**

ConvertTextEncoding

ConvertTextEncoding(*sourceTextStr*, *sourceTextEncoding*, *destTextEncoding*, *mapErrorMode*, *options*)

ConvertTextEncoding converts text from one text encoding to another.

The ConvertTextEncoding function was added in Igor Pro 7.00.

All text in memory is assumed to be in UTF-8 format except for text stored in waves which can be stored in any text encoding. You might want to convert text from UTF-8 to Windows-1252 (Windows Western European), for example, to export it to a program that expects Windows-1252.

You might have text already loaded into Igor that you know to be in Windows-1252. To display it correctly, you need to convert it to UTF-8.

You can also use ConvertTextEncoding to test if text is valid in a given text encoding, by specifying the same text encoding for *sourceTextEncoding* and *destTextEncoding*.

Parameters

sourceTextStr is the text that you want to convert.

sourceTextEncoding specifies the source text encoding.

destTextEncoding specifies the output text encoding.

See **Text Encoding Names and Codes** on page III-490 for a list of acceptable text encoding codes.

mapErrorMode determines what happens if an input character can not be mapped to the output text encoding because the character does not exist in the output text encoding. It takes one of these values:

options is a bitwise parameter which defaults to 0 and with the bits defined as follows:

1: Generate error. The function returns "" and generates an error.

2: Return a substitute character for the unmappable character. The substitute character for Unicode is the Unicode replacement character, U+FFFD. For most non-Unicode text encodings it is either control-Z or a question mark.

3: Skip unmappable input character.

4: Return escape sequences representing unmappable characters and invalid source text.

If the source text is valid in the source text encoding but can not be represented in the destination text encoding, unmappable characters are replaced with \uXXXX where XXXX specifies the UTF-16 code point of the unmappable character in hexadecimal. The DemoUnmappable example function below illustrates this.

If the conversion can not be done because the source text is not valid in the source text encoding, invalid bytes are replaced with \xXX where XX specifies the value of the invalid byte in hexadecimal. The DemoInvalid example function below illustrates this.

If mapErrorMode is 2, 3 or 4, the function does not return an error in the event of an unmappable character.

Bit 0: If cleared, in the event of a text conversion error, a null string is returned and an error is generated. Use this if you want to abort procedure execution if an error occurs.

If set, in the event of a text conversion error, a null string is returned but no error is generated. Use this if you want to detect and handle a text conversion error. You can test for null using strlen as shown in the example below.

Bit 1: If cleared (default), null bytes in *sourceTextStr* are considered invalid and ConvertTextEncoding returns an error. If set, null bytes are considered valid.

Bit 2: If cleared (default) and *sourceTextEncoding* and *destTextEncoding* are the same, ConvertTextEncoding attempts to do the conversion anyway. If *sourceTextStr* is invalid in the specified text encoding, the issue is handled according to *mapErrorMode*. This allows you to check the validity of text whose text encoding you think you know, by passing 1 for *mapErrorMode* and 5 for *options*. Use **strlen** to test if the returned string is null, indicating that *sourceTextStr* is not valid in the specified text encoding.

If set and *sourceTextEncoding* and *destTextEncoding* are the same, ConvertTextEncoding merely returns *sourceTextStr* without doing any conversion.

All other bits are reserved and must be cleared.

Details

ConvertTextEncoding returns a null result string if *sourceTextEncoding* or *destTextEncoding* are not valid text encoding codes or if a text conversion error occurs. You can test for a null string using strlen which returns NaN if the string is null.

If bit 0 of the *options* parameter is cleared, Igor generates an error which halts procedure execution. If it is set, Igor generates no error and you should test for null and attempt to handle the error, as illustrated by the example below.

A text conversion error occurs if *mapErrorMode* is 1 and the source text contains one or more characters that are not mappable to the destination text encoding. A text conversion error also occurs if the source text contains a sequence of bytes that is not valid in the source text encoding.

The "binary" text encoding (255) is not a real text encoding. If either *sourceTextEncoding* or *destTextEncoding* are binary (255), ConvertTextEncoding does no conversion and just returns *sourceTextStr* unchanged.

See **Text Encodings** on page III-459 for further details.

Example

In reading these examples, keep in mind that Igor converts escape codes such as “\u8C4A”, when they appear in literal text, to the corresponding UTF-8 characters. See **Unicode Escape Sequences in Strings** on page IV-15 for details.

```
Function DemoConvertTextEncoding()
    // Get text encoding codes for text the text encodings used below
    Variable teUTF8 = TextEncodingCode("UTF-8")
    Variable teWindows1252 = TextEncodingCode("Windows-1252")
    Variable teShiftJIS = TextEncodingCode("ShiftJIS")

    // Convert from Windows-1252 to UTF-8
    String source = "Division sign: " + num2char(0xF7)
    String result = ConvertTextEncoding(source, teWindows1252, teUTF8, 1, 0)
    Print result

    // Convert unmappable character from UTF-8 to Windows-1252
    // \u8C4A is an escape sequence representing a Japanese character in Unicode
    // for which there is no corresponding character in Windows-1252

    // Demonstrate mapErrorMode = 1 (fail unmappable character)
    source = "Unmappable character causes failure: {\u8C4A}"
    // Pass 1 for options parameter to tell Igor to ignore error and let us handle it
    result = ConvertTextEncoding(source, teUTF8, teWindows1252, 1, 1)
    Variable len = strlen(result)     // Will be NaN if conversion failed
    if (NumType(len) == 2)
        Print "Conversion failed (as expected). Result is NULL."
        // You could cope with this error by trying again with the mapErrorMode
        // parameter set to 2, 3 or 4.
    else
        // We should not get here
        Print "Conversion succeeded (should not happen)."
        Print result
    endif

    // Demonstrate mapErrorMode = 2 (substitute for unmappable character)
    source = "Unmappable character replaced by question mark: {\u8C4A}"
    result = ConvertTextEncoding(source, teUTF8, teWindows1252, 2, 0)
    Print result                      // Prints "?" in place of unmappable character

    // Demonstrate mapErrorMode = 3 (skip unmappable character)
    source = "Unmappable character skipped: {\u8C4A}"
    result = ConvertTextEncoding(source, teUTF8, teWindows1252, 3, 0)
    Print result                      // Skips unmappable character

    // Demonstrate mapErrorMode = 4 (insert escape sequence for unmappable character)
    source = "Unmappable character replaced by escape sequence: {\u8C4A}"
    result = ConvertTextEncoding(source, teUTF8, teWindows1252, 4, 0)
    Print result        // Unmappable character represented as escape sequence

    // Demonstrate mapErrorMode = 4 (insert escape sequence for unmappable character)
    source = "Unmappable character replaced by escape sequence: {\u8C4A}"
    // First convert UTF-8 to Shift_JIS (Japanese). This will succeed.
    result = ConvertTextEncoding(source, teUTF8, teShiftJIS, 1, 0)
    // Next convert Shift_JIS (Japanese) to Windows-1252. The character can not
    // be mapped and is replaced by an escape sequence.
    result = ConvertTextEncoding(result, teShiftJIS, teWindows1252, 4, 0)
    Print result        // Unmappable character represented as escape sequence
End

// Demo unmappable character
// In this example, the source text is valid but not representable in destination
// text encoding. Because we pass 4 for the mapErrorMode parameter, ConvertTextEncoding
// uses an escape sequenceto represent the unmappable text.
Function DemoUnmappable()
    String input = "\u2135"// Alef symbol - available in UTF-8 but not in MacRoman
    String output = ConvertTextEncoding(input, 1, 2, 4, 0)
    Print output            // Prints "\u2135"
End

// Demo invalid input text
// In this example, the source text is invalid in the source text encoding.
// Because we pass 4 for the mapErrorMode parameter, ConvertTextEncoding uses an escape
    sequences
// to represent the invalid text.
Function DemoInvalidInput()
    String input = "\x8E"   // Represents "é" in MacRoman but is not valid in UTF-8
    String output = ConvertTextEncoding(input, 1, 2, 4, 0)
    Print output            // Prints "\x8E"
End
```

See Also

Text Encodings on page III-459, **Text Encoding Names and Codes** on page III-490

TextEncodingCode, **TextEncodingName**, **SetWaveTextEncoding**

ConvertGlobalStringTextEncoding, **String Variable Text Encoding Error Example** on page III-479

ConvexHull

`convexHull` `[`***`flags`***`]`***`xwave, ywave`***

`convexHull` `[`***`flags`***`]` ***`tripletWave`***

The ConvexHull operation calculates the convex hull in either 2 or 3 dimensions. The dimensionality is deduced from the input wave(s). If the input consists of two 1D waves of the same length, the number of dimensions is assumed to be 2. If the input consists of a single triplet wave (a wave of 3 columns), then the number of dimensions is 3.

In 2D cases the operation calculates the convex hull and produces the result in a pair of x and y waves, W_XHull and W_YHull.

In 3D cases the operation calculates the convex hull and stores it in a triplet wave M_Hull that describes ordered facets of the convex hull.

ConvexHull returns an error if the input waves have fewer than 3 data points.

Flags

/C	(2D convex hull only) adds the first point to the end of the W_XHull and W_YHull waves so that the first and the last points are the same.
/E	(3D case only) if you use this flag the operation also creates a wave that lists the indices of the vertices which are not part of the convex hull, i.e., vertices which are interior to the hull. The output is in the wave W_HullExcluded.
/I	(3D convex hull only) use this flag to get the corresponding index of the vertex as the fourth column in the M_Hull wave.
/S	(3D convex hull only) use this flag if you want the resulting M_Hull to have NaN lines separating each triangle.
/T=*tolerance*	(3D case only) default tolerance for measuring if a point is inside or outside the convex hull is $1.0\text{x}10^{-20}$. You can use any other positive value.
/V	(3D case only) if you use this flag the operation also creates a wave containing the output in a list of vertex indices. The wave M_HullVertices contains a row per triangle where each entry on a row corresponds to the index of the input vertex.
/Z	No error reporting.

Examples

```
Make/O/N=33 xxx=gnoise(5),yyy=gnoise(7)
Convexhull/c xxx,yyy
Display W_Yhull vs W_Xhull
Appendtograph yyy vs xxx
ModifyGraph mode(yyy)=3,marker(yyy)=8,rgb(W_YHull)=(0,15872,65280)
```

See Also

Triangulate3D

Convolve

`Convolve` `[`**`/A/C`**`]` ***`srcWaveName, destWaveName`*** `[,` ***`destWaveName`***`]`…

The Convolve operation convolves *srcWaveName* with each destination wave, putting the result of each convolution in the corresponding destination wave.

Convolve is not multidimensional aware. Some multidimensional convolutions are covered by the **MatrixConvolve**, **MatrixFilter**, and **MatrixOp** operations

Flags

/A	Acausal linear convolution.
/C	Circular convolution.

Details

Convolve performs linear convolution unless the /C or /A flag is used. See the diagrams in the examples below.

Depending on the type of convolution, the destination waves' lengths may increase. *srcWaveName* is not altered unless it also appears as a destination wave.

If *srcWaveName* is real-valued, each destination wave must be real-valued, and if *srcWaveName* is complex, each destination wave must be complex, too. Double and single precision waves may be freely intermixed; calculations are performed in the higher precision.

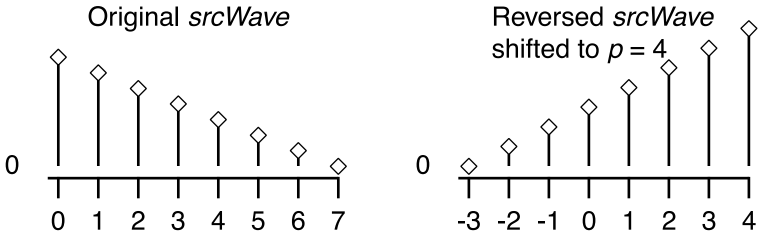
The linear convolution equation is:

$$destWaveOut[p] = \sum_{m=0}^{N-1} destWaveIn[m] \cdot srcWave[p-m]$$

where *N* is the number of points in the longer of *destWaveIn* and *srcWave*. For circular convolution, the index [*p* -*m*] is wrapped around when it exceeds the range of [0,numpnts(*srcWave*)-1]. For acausal convolution, when [*p* -*m*] exceeds the range a zero value is substituted for *srcWave* [*p* -*m*]. Similar operations are applied to *destWaveIn* [*m*].

Another way of looking at this equation is that, for all *p*, *destWaveOut*[*p*] equals the sum of the point-by-point products from 0 to *p* of the destination wave and an end-to-end reversed copy of the source wave that has been shifted to the right by *p*.

The following diagram shows the reversed/shifted *srcWave* that would be combined with *destWaveIn*. The points numbered 0 through 4 of the reversed *srcWave* would be multiplied with *destWaveIn*[0…4] and summed to produce *destWaveOut*[4]:



For linear and acausal convolution, the destination wave is first zero-padded by one less than the length of the source wave. This prevents the "wrap-around" effect that occurs in circular convolution. The zero-padded points are removed after acausal convolution, and retained after linear convolution. The X scaling of the waves is ignored.

The convolutions are performed by transforming the source and destination waves with the Fast Fourier Transform, multiplying them in the frequency domain, and then inverse-transforming them into the destination wave(s).

The convolution is performed in segments if the resulting wave has more than 256 points and the destination wave has twice as many points as the source wave. For acausal convolution, the length of the resulting wave is considered to be (numpnts(*srcWaveName*) +numpnts(*destWaveName*)-1) for this calculation.

Applications

The usual application of convolution is to compute the response of a linear system defined by its impulse response to an input signal. *srcWaveName* would contain the impulse response, and the destination wave would initially contain the input signal. After the Convolve operation has completed, the destination wave contains the output signal.

Use linear convolution when the source wave contains an impulse response (or filter coefficients) where the first point of *srcWave* corresponds to no delay (*t* = 0).

Use circular convolution for the case where the data in *srcWaveName* and *destWaveName* are considered to endlessly repeat (or “wrap around” from the end back to the start), which means no zero padding is needed.

Use acausal convolution when the source wave contains an impulse response where the middle point of *srcWave* corresponds to no delay (*t* = 0).

See Also

Convolution on page III-284 for illustrated examples. **MatrixOp**.

References

A very complete explanation of circular and linear convolution can be found in sections 2.23 and 2.24 of Rabiner and Gold, *Theory and Application of Digital Signal Processing*, Prentice Hall, 1975.

CopyDimLabels

`CopyDimLabels [`*`flags`*`]` ***`srcWave`*****`,`** ***`destWave`*****`, [`*`destWave`*`]...`**

The CopyDimLabels operation copies dimension labels from the source wave to the destination wave or waves.

CopyDimLabels was added in Igor Pro 8.00.

Support for multiple destination waves was added in Igor Pro 9.00.

Flags

In the following flags, *dim* is 0 for the rows dimension, 1 for the columns dimension, 2 for the layers dimension, and 3 for the chunks dimension.

/ROWS=*dim*	Copies the row dimension labels of *srcWave* into the *destWave* dimension specified by *dim*.
/COLS=*dim*	Copies the column dimension labels of *srcWave* into the *destWave* dimension specified by *dim*.
/LAYR=*dim*	Copies the layer dimension labels of *srcWave* into the *destWave* dimension specified by *dim*.
/CHNK=*dim*	Copies the chunk dimension labels of *srcWave* into the *destWave* dimension specified by *dim*.

Details

If you omit all flags, CopyDimLabels copies all dimension labels in *srcWave* to the corresponding dimension labels of *destWave* for dimensions that exist in *destWave*.

You can use /ROWS, /COLS, /LAYR and /CHNK to copy dimension labels from any existing source wave dimension into any existing destination wave dimension. For example, to copy the column dimension labels of wave1 into the row dimension labels of wave2 use:

```
CopyDimLabels /COLS=0 wave1, wave2
```

To copy the column dimension labels of wave1 into the layer dimension labels of wave2 and the row dimension labels of wave1 into the column dimension labels of wave 2 use:

```
CopyDimLabels /COLS=2 /ROWS=1 wave1, wave2
```

If the source dimension has N elements and the destination dimension has M>N elements then only the first N dimension labels are set in the destination. The remaining dimension labels in the destination are unchanged.

If the source dimension has N elements and the destination dimension has M<N elements then only the first M dimension labels are copied from the source to the destination.

It is an error to attempt to copy dimension labels to a non-existent dimension in the destination.

See Also

FindDimLabel, **GetDimLabel**, **SetDimLabel**, **Dimension Labels** on page II-93

CopyFile

`CopyFile [`*`flags`*`][`*`srcFileStr`*`] [as `*`destFileOrFolderStr`*`]`

The CopyFile operation copies a file on disk.

Parameters

srcFileStr can be a full path to the file to be copied (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*.

If Igor can not determine the location of the source file from *srcFileStr* and *pathName*, it displays a dialog allowing you to specify the source file.

destFileOrFolderStr is interpreted as the name of (or path to) an existing folder when /D is specified, otherwise it is interpreted as the name of (or path to) a possibly existing file.

If *destFileOrFolderStr* is a partial path, it is relative to the folder associated with pathName.

If /D is specified, the source file is copied inside the folder using the source file's name.

If Igor can not determine the location of the destination file from *pathName*, *srcFileStr*, and *destFileOrFolderStr*, it displays a Save File dialog allowing you to specify the destination file (and folder).

If you use a full or partial path for either *srcFileStr* or *destFileOrFolderStr*, see **Path Separators** on page III-451 for details on forming the path.

Folder paths should not end with single Path Separators. See the **Details** section for **MoveFolder**.

Flags

/D Interprets *destFileOrFolderStr* as the name of (or path to) an existing folder (or "directory"). Without /D, *destFileOrFolderStr* is interpreted as the name of (or path to) a file.

If *destFileOrFolderStr* is not a full path to a folder, it is relative to the folder associated with pathName.

/I [=*i*] Specifies the level of user interactivity.

/I=0: Interactive only if one or *srcFileStr* or *destFileOrFolderStr* is not specified or if the source file is missing. (Same as if /I was not specified.)

/I=1: Interactive even if *srcFileStr* is specified and the source file exists.

/I=2: Interactive even if *destFileOrFolderStr* is specified.

/I=3: Interactive even if *srcFileStr* is specified, the source file exists, and *destFileOrFolderStr* is specified. Same as /I only.

/M=*messageStr* Specifies the prompt message in the Open File dialog. If /S is not used, then *messageStr* will be used for both Open File and for Save File dialogs.

/O Overwrites any existing destination file.

/P=*pathName* Specifies the folder to look in for the source file, and the folder into which the file is copied. *pathName* is the name of an existing symbolic path.

Using /P means that both *srcFileStr* and *destFileOrFolderStr* must be either simple file or folder names, or paths relative to the folder specified by *pathName*.

/S=*saveMessageStr* Specifies the prompt message in the Save File dialog.

/Z [=*z*] Prevents procedure execution from aborting if it attempts to copy a file that does not exist. Use /Z if you want to handle this case in your procedures rather than aborting execution.

/Z=0: Same as no /Z.

/Z=1: Copies a file only if it exists. /Z alone has the same effect as /Z=1.

/Z=2: Copies a file if it exists or displays a dialog if it does not exist.

Variables

The CopyFile operation returns information in the following variables:

`V_flag`	Set to zero if the file was copied, to -1 if the user cancelled either the Open File or Save File dialogs, and to some nonzero value if an error occurred, such as the specified file does not exist.
`S_fileName`	Stores the full path to the file that was copied. If an error occurred or if the user cancelled, it is set to an empty string.
`S_path`	Stores the full path to the file copy. If an error occurred or if the user cancelled, it is set to an empty string.

Examples

Copy a file within the same folder using a new name:

```
CopyFile/P=myPath "afile.txt" as "destFile.txt"
```

Copy a file into subfolder using the original name (using /P):

```
CopyFile/D/P=myPath "afile.txt" as ":subfolder"
Print S_Path     // prints "Macintosh HD:folder:subfolder:afile.txt"
```

Copy file into subfolder using the original name (using full paths):

```
CopyFile/D "Macintosh HD:folder:afile.txt" as "Server:archive"
```

Copy a file from one folder to another, assigning the copy a new name:

```
CopyFile "Macintosh HD:folder:afile.txt" as "Server:archive:destFile.txt"
```

Copy user-selected file in any folder as destFile.txt in myPath folder (prompt to save even if destFile.txt doesn't exist):

```
CopyFile/I=2/P=myPath as "destFile.txt"
```

Copy user-selected file in any folder as destFile.txt in any folder:

```
CopyFile as "destFile.txt"
```

See Also

The **Open**, **MoveFile**, **DeleteFile**, and **CopyFolder** operations. The **IndexedFile** function. **Symbolic Paths** on page II-22.

CopyFolder

CopyFolder [***flags***][***srcFolderStr***] [**as** ***destFolderStr***]

The CopyFolder operation copies a folder (and its contents) on disk.

Warning: *The CopyFolder command can destroy data* by overwriting another folder and contents!

When overwriting an existing folder on disk, CopyFolder will do so only if permission is granted by the user. The default behavior is to display a dialog asking for permission. The user can alter this behavior via the Miscellaneous Settings dialog's Misc category.

If permission is denied, the folder will not be copied and V_Flag will return 1088 (Command is disabled) or 1275 (You denied permission to overwrite a folder). Command execution will cease unless the /Z flag is specified.

Parameters

srcFolderStr can be a full path to the folder to be copied (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a folder inside the folder associated with *pathName*.

If Igor can not determine the location of the folder from *srcFolderStr* and *pathName*, it displays a dialog allowing you to specify the source folder.

If /P=*pathName* is given, but *srcFolderStr* is not, then the folder associated with *pathName* is copied.

destFolderStr can be a full path to the output (destination) folder (in which case /P is not needed), or a partial path relative to the folder associated with *pathName*.

An error is returned if the destination folder would be inside the source folder.

If Igor can not determine the location of the destination folder from *destFolderStr* and *pathName*, it displays a dialog allowing you to specify or create the destination folder.

If you use a full or partial path for either folder, see **Path Separators** on page III-451 for details on forming the path.

Flags

/D Interprets *destFolderStr* as the name of (or path to) an existing folder (or directory) to copy the source folder into. Without /D, *destFolderStr* is interpreted as the name of (or path to) the copied folder.

If *destFolderStr* is not a full path to a folder, it is relative to the folder associated with *pathName*.

/I [=*i*] Specifies the level of user interactivity.

/I=0:	Interactive only if the source or destination folder is not specified or if the source folder is missing. (Same as if /I was not specified.)
/I=1:	Interactive even if the source folder is specified and it exists.
/I=2:	Interactive even if *destFolderStr* is specified.
/I=3:	Interactive even if the source folder is specified, the source folder exists, and *destFolderStr* is specified. Same as /I only.

/M=*messageStr* Specifies the prompt message in the Select (source) Folder dialog. If /S is not used, then *messageStr* will be used for the Select Folder dialog and for the Create Folder dialog.

/O Overwrite existing destination folder, if any.

On Macintosh, a Macintosh-style overwrite-move is performed in which the source folder completely replaces the destination folder.

On Windows, a Windows-style mix-in move is performed in which the contents of the source folder are moved into the destination folder, replacing any same-named files but leaving other files in place.

/P=*pathName* Specifies the folder to look in for the source folder. *pathName* is the name of an existing symbolic path.

If *srcFolderStr* is not specified, the folder associated with *pathName* is copied.

Using /P means that *srcFolderStr* (if specified) and *destFolderStr* must be either simple folder names or paths relative to the folder specified by *pathName*.

/S=*saveMessageStr* Specifies the prompt message in the Create Folder dialog.

/Z [=z] Prevents procedure execution from aborting if it attempts to copy a file that does not exist. Use /Z if you want to handle this case in your procedures rather than aborting execution.

/Z=0:	Same as no /Z.
/Z=1:	Copies a folder only if it exists. /Z alone has the same effect as /Z=1.
/Z=2:	Copies a folder if it exists or displays a dialog if it does not exist.

Variables

The CopyFolder operation returns information in the following variables:

`V_flag` Set to zero if the folder was copied, to -1 if the user cancelled either the Select Folder or Create Folder dialogs, and to some nonzero value if an error occurred, such as the specified file does not exist.

`S_fileName` Stores the full path to the folder that was copied, with a trailing colon. If an error occurred or if the user cancelled, it is set to an empty string.

`S_path` Stores the full path to the folder copy, with a trailing colon. If an error occurred or if the user cancelled, it is set to an empty string.

Details

You can use only /P=*pathName* (without *srcFolderStr*) to specify the source folder to be copied.

Folder paths should not end with single Path Separators. See the **Details** section for **MoveFolder**.

See Also

Open, **MoveFile**, **DeleteFile**, **MoveFolder**, **NewPath**, and **IndexedDir** operations, and **Symbolic Paths** on page II-22.

CopyScales

`CopyScales [/I/P] srcWaveName, waveName [, waveName]…`

The CopyScales operation copies the x, y, z, and t scaling, x, y, z, and t units, the data Full Scale and data units from *srcWaveName* to the other waves.

Flags

/I	Copies the x, y, z, and t scaling in inclusive format.
/P	Copies the x, y, z, and t scaling in slope/intercept format (x0, dx format).

Details

Normally the x, y, z, and t (dimension) scaling is copied in min/max format. However, if you use /P, the dimension scaling is copied in slope/intercept format so that if *srcWaveName* and the other waves have differing dimension size (number of points if the wave is a 1D wave), then their dimension values will still match for the points they have in common. Similarly, /I uses the inclusive variant of the min/max format. See **SetScale** for a discussion of these dimension scaling formats.

If a wave has only one point, /I mode reverts to /P mode.

CopyScales copies scales only for those dimensions that *srcWaveName* and *waveName* have in common.

See Also

x, **y**, **z**, and **t** scaling functions.

Correlate

`Correlate [/AUTO/C/NODC] srcWaveName, destWaveName [, destWaveName]…`

The Correlate operation correlates *srcWaveName* with each destination wave, putting the result of each correlation in the corresponding destination wave.

Flags

/AUTO Auto-correlation scaling. This forces the X scaling of the destination wave's center point to be x=0, and divides the destination wave by the center point's value so that the center value is exactly 1.0.

If srcWaveName and destWaveName do not have the same number of points, this flag is ignored.

/AUTO is not compatible with /C.

/C Circular correlation. (See **Compatibility Note**.)

/NODC Removes the mean from the source and destination waves before computing the correlations. Removing the mean results in the un-normalized auto- or cross-covariance.

"DC" is an abbrevation of "direct current", an electronics term for the non-varying average value component of a signal.

Details

Note: To compute a single-value correlation number use the **StatsCorrelation** function which returns the Pearson's correlation coefficient of two same-length waves.

Correlate performs linear correlation unless the /C flag is used.

Depending on the type of correlation, the length of the destination may increase. *srcWaveName* is not altered unless it also appears as a destination wave.

If the source wave is real-valued, each destination wave must be real-valued and if the source wave is complex, each destination wave must be complex, too. Double and single precision waves may be freely intermixed; calculations are performed in the higher precision.

The linear correlation equation is:

$$destWaveOut[p] = \sum_{m=0}^{N-1} srcWave[m] \cdot destWaveIn[p+m]$$

where N is the number of points in the longer of *destWaveIn* and *srcWave*.

For circular correlation, the index [p +m] is wrapped around when it exceeds the range of `[0,numpnts(destWaveIn)-1]`. For linear correlation, when [p +m] exceeds the range a zero value is substituted for *destWaveIn*[p +m]. When m exceeds `numpnts(srcWave)-1`, 0 is used instead of *srcWave*[m].

Comparing this with the **Convolve** operation, which is the linear convolution:

$$destWaveOut[p] = \sum_{m=0}^{N-1} destWaveIn[m] \cdot srcWave[p-m]$$

you can see that the only difference is that for correlation the source wave is *not* reversed before shifting and combining with the destination wave.

The Correlate operation is not multidimensional aware. For details, see **Analysis on Multidimensional Waves** on page II-95 and in particular **Analysis on Multidimensional Waves** on page II-95.

Compatibility Note

Prior to Igor Pro 5, Correlate/C scaled and rotated the results improperly (the result was often rotated left by one and the X scaling was entirely negative).

Now the destination wave's X scaling is unaltered and it does not rotate the result. You can force the old behavior for compatibility with old procedures that depend on the old behavior by setting `root:V_oldCorrelationScaling=1`.

A better way to get identical Correlate/C results with all versions of Igor Pro is to use this code, which rotates the result so that x=0 is always the first point in *destWave*, no matter which Igor Pro version runs this code (currently, it doesn't change anything and runs extremely quickly because it does no rotation):

```
Correlate/C srcWave, destWave
Variable pointAtXEqualZero= x2pnt(destWave,0)     // 0 for Igor Pro 5
Rotate -pointAtXEqualZero,destWave
SetScale/P x, 0, DimDelta(destWave,0), "", destWave
```

Applications

A common application of correlation is to measure the similarity of two input signals as they are shifted by one another.

Often it is desirable to normalize the correlation result to 1.0 at the maximum value where the two inputs are most similar. To normalize *destWaveOut*, compute the RMS values of the input waves and the number of points in each wave:

```
WaveStats/Q srcWave
Variable srcRMS = V_rms
Variable srcLen = numpnts(srcWave)

WaveStats/Q destWave
Variable destRMS = V_rms
Variable destLen = numpnts(destWave)

Correlate srcWave, destWave                 // overwrites destWave

// now normalize to max of 1.0
destWave /= (srcRMS * sqrt(srcLen) * destRMS * sqrt(destLen))
```

Another common application is using autocorrelation (where *srcWaveName* and *destWaveName* are the same) to determine Power Spectral Density. In this case it better to use the **DSPPeriodogram** operation which provides more options.

See Also

Convolution on page III-284 and **Correlation** on page III-286 for illustrated examples. See the **Convolve** operation for algorithm implementation details, which are identical except for the lack of source wave reversal, and the lack of the /A (acausal) flag.

The **MatrixOp**, **StatsCorrelation**, **StatsCircularCorrelationTest**, **StatsLinearCorrelationTest**, and **DSPPeriodogram** operations.

References

An explanation of autocorrelation and Power Spectral Density (PSD) can be found in Chapter 12 of Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

WaveMetrics provides Igor Technical Note 006, “DSP Support Macros” that computes the PSD with options such as windowing and segmenting. See the Technical Notes folder. Some of the techniques discussed there are available as Igor procedure files in the “WaveMetrics Procedures:Analysis:” folder.

Wikipedia: http://en.wikipedia.org/wiki/Correlation

Wikipedia: http://en.wikipedia.org/wiki/Cross_covariance

Wikipedia: http://en.wikipedia.org/wiki/Autocorrelation_function

cos

`cos(angle)`

The cos function returns the cosine of *angle* which is in radians.

In complex expressions, *angle* is complex, and `cos(angle)` returns a complex value:

$$\cos(x+iy)=\cos(x)\cosh(y)-i\sin(x)\sinh(y).$$

See Also

acos, **sin**, **tan**, **sec**, **csc**, **cot**

cosh

`cosh(num)`

The cosh function returns the hyperbolic cosine of *num*:

$$\cosh(x)=\frac{e^x+e^{-x}}{2}.$$

In complex expressions, *num* is complex, and `cosh(num)` returns a complex value.

See Also

sinh, **tanh**, **coth**

CosIntegral

`CosIntegral(z)`

The CosIntegral(*z*) function returns the cosine integral of *z*.

If *z* is real, a real value is returned. If *z* is complex then a complex value is returned.

The CosIntegral function was added in Igor Pro 7.00.

Details

The cosine integral is defined by

$$Ci(z) = \gamma + \ln(z) + \int_0^z \frac{\cos(t) - 1}{t} dt, \qquad \left(|\arg(z)| < \pi\right)$$

where γ is the Euler-Mascheroni constant 0.5772156649015328606065.

IGOR computes the CosIntegral using the expression:

$$Ci(z) = -\frac{Z^2}{4} \, {}_2F_3\left(1,1;2,2,\frac{3}{2};-\frac{z^2}{4}\right) + \ln(z) + \gamma,$$

References

Abramowitz, M., and I.A. Stegun, "Handbook of Mathematical Functions", Dover, New York, 1972. Chapter 5.

See Also

SinIntegral, **ExpIntegralE1**, **hyperGPFQ**

cot

`cot(`*`angle`*`)`

The cot function returns the cotangent of *angle* which is in radians.

In complex expressions, *angle* is complex, and cot(*angle*) returns a complex value.

See Also

sin, **cos**, **tan**, **sec**, **csc**

coth

`coth(`*`num`*`)`

The coth function returns the hyperbolic cotangent of *num*:

$$\coth(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}}.$$

In complex expressions, *num* is complex, and coth(*num*) returns a complex value.

See Also

sinh, **cosh**, **tanh**

CountObjects

`CountObjects(`*`sourceFolderStr`*`, `*`objectType`*`)`

The CountObjects function returns the number of objects of the specified type in the data folder specified by the string expression.

CountObjectsDFR is preferred.

Parameters

sourceFolderStr can be either `":"` or `""` to specify the current data folder. You can also use a full or partial data folder path. *objectType* should be one of the following values:

1	Waves
2	Numeric variables
3	String variables
4	Data folders

See Also

Chapter II-8, **Data Folders**, and the **GetIndexedObjName** function.

CountObjectsDFR

`CountObjectsDFR(dfr,objectType)`

The CountObjectsDFR function returns the number of objects of the specified type in the data folder specified by the data folder reference *dfr*.

CountObjectsDFR is the same as CountObjects except the first parameter, *dfr*, is a data folder reference instead of a string containing a path.

Parameters

dfr is a data folder reference.

objectType is one of the following values:

1	Waves
2	Numeric variables
3	String variables
4	Data folders

See Also

Data Folders on page II-107, **Data Folder References** on page IV-78, **Built-in DFREF Functions** on page IV-81, **GetIndexedObjNameDFR**

cpowi

`cpowi(num, ipow)`

This function is obsolete as the exponentiation operator ^ handles complex expressions with any combination of real, integer and complex arguments. See **Operators** on page IV-6. The cpowi function returns a complex number resulting from raising complex *num* to integer-valued power *ipow*. *ipow* can be positive or negative, but if it is not an integer cpowi returns (NaN, NaN).

CreateAliasShortcut

`CreateAliasShortcut [flags][targetFileDirStr] [as aliasFileStr]`

The CreateAliasShortcut operation creates an alias (*Macintosh*) or shortcut (*Windows*) file on disk. The alias can point to either a file or a folder. The file or folder pointed to is called the “target” of the alias or shortcut.

Parameters

targetFileDirStr can be a full path to the file or folder to make an alias or shortcut for, a partial path relative to the folder associated with /P=*pathName*, or the name of a file or folder in the folder associated with *pathName*.

If Igor can not determine the location of the file or folder from *targetFileDirStr* and *pathName*, it displays a dialog allowing you to specify a target file. Use /D to select a folder as the alias target, instead.

aliasFileStr can be a full path to the created alias file, a partial path relative to the folder associated with *pathName* if specified, or the name of a file in the folder associated with *pathName*.

If Igor can not determine the location of the alias or shortcut file from *aliasFileStr* and *pathName*, it displays a File Save dialog allowing you to create the file.

If you use a full or partial path for either *targetFileDirStr* or *aliasFileStr*, see **Path Separators** on page III-451 for details on forming the path.

Folder paths should not end with single path separators. See the **MoveFolder Details** section.

Flags

/D	Uses the Select Folder dialog rather than Open File dialog when *targetFileDirStr* is not fully specified.

/I [=*i*] Specifies the level of user interactivity.

/I=0: Interactive only if one or *targetFileDirStr* or *aliasFileStr* is not specified or if the target file is missing. (Same as if /I was not specified.)

/I=1: Interactive even if *targetFileDirStr* is fully specified and the target file exists.

/I=2: Interactive even if *targetFileDirStr* is specified.

/I=3: Interactive even if *targetFileDirStr* is specified and the target file exists. Same as /I only.

/M=*messageStr* Specifies the prompt message in the Open File or Select Folder dialog. If /S is not specified, then *messageStr* will be used for Open File (or Select Folder) and for Save File dialogs.

/O Overwrites any existing file with the alias or shortcut file.

/P=*pathName* Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/S=*saveMessageStr* Specifies the prompt message in the Save File dialog when creating the alias or shortcut file.

/Z[=*z*] Prevents procedure execution from aborting the procedure tries to create an alias or shortcut for a file or folder that does not exist. Use /Z if you want to handle this case in your procedures rather than aborting execution.

/Z=0: Same as no /Z.

/Z=1: Creates an alias to a file or folder only if it exists. /Z alone has the same effect as /Z=1.

/Z=2: Creates an alias to a file or folder only if it exists and displays a dialog if it does not exist.

Variables

The CreateAliasShortcut operation returns information in the following variables:

`V_flag` Status output:

0 Created an alias or shortcut file.

1 User cancelled any of the Open File, Select Folder, or Save File dialogs.

Other: An error occurred, such as the target file does not exist.

`S_fileName` Full path to the target file or folder. If an error occurred or if the user cancelled, it is an empty string.

`S_path` Full path to the created alias or shortcut file. If an error occurred or if the user cancelled, it is an empty string.

Examples

Create a shortcut (*Windows*) to the current experiment, on the desktop:

```
String target= Igorinfo(1)+".pxp" // experiments are usually .pxp on Windows
CreateAliasShortcut/O/P=home target as "C:WINDOWS:Desktop:"+target
```

Create an alias (*Macintosh*) to the VDT XOP in the Igor Extensions folder:

```
String target= ":More Extensions:Data Acquisition:VDT"
CreateAliasShortcut/O/P=Igor target as ":Igor Extensions:VDT alias"
```

Create an alias to the “HD 2” disk. Put the alias on the desktop:

```
CreateAliasShortcut/D/O "HD 2" as "HD:Desktop Folder:Alias to HD 2"
```

See Also

Symbolic Paths on page II-22.

The **Open**, **MoveFile**, **DeleteFile**, and **GetFileFolderInfo** operations. The **IgorInfo** and **ParseFilePath** functions.

CreateBrowser

CreateBrowser [/M] [***keyword = value*** [, ***keyword = value*** …]]

The CreateBrowser operation creates a data browser window.

Documentation for the CreateBrowser operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "CreateBrowser"
```

CreateDataObjectName

CreateDataObjectName(*dfr*, nameInStr, *objectType*, *suffixNum*, *options*)

The CreateDataObjectName function returns a name suitable for use for a new object of the type specified by objectType. It can replace a combination of **CleanupName** and **UniqueName**.

CreateDataObjectName was added in Igor Pro 9.00 or later.

Parameters

dfr is a data folder reference for the data folder in which the objects are to be created. Pass : for the current data folder.

nameInStr must contain an unquoted (i.e., no single quotes for liberal names) name, such as you might receive from the user through a dialog or control panel.

objectType is one of the following:

1:	Wave
3:	Numeric variable
4:	String variable
11:	Data folder

suffixNum is a value used in generating a series of names from a base name when allowing overwriting. For other uses, pass 0 for *suffixNum*. See ***Generating a Series of Names from a Base Name*** below.

options is a bitwise parameter with the bits defined as follows:

Bit 0: Be liberal.

If cleared, CreateDataObjectName always returns a standard object name. If set, it returns a liberal object name if nameInStr is liberal. See **Object Names** on page III-501 for a discussion of standard and liberal object names.

If objectType is 3 (numeric variable) or 4 (string variable), the output name will not be liberal, even if this bit is set as Igor allows only wave and data folder names to be liberal.

Bit 1: Allow overwrite.

If cleared, CreateDataObjectName returns a name that is unique in the namespace of the type of object specified by objectType. If set, CreateDataObjectName returns a name that may be in use in that namespace.

Waves, numeric variables and string variables are in the same namespace and so must have unique names within a given data folder. Data folders are in their own namespace and so their names can be the same as the names of waves, numeric variables and string variables.

Bit 2: Input name is a base name.

If cleared, *nameInStr* is taken to be a proposed object name that CreateDataObjectName cleans up (i.e., makes legal). If the name is in use and allow overwrite is not specified, CreateDataObjectName makes the name unique by appending one or more digits.

If set, *nameInStr* is taken to be a proposed base name for a series of objects and the output name always has at least one digit appended. CreateDataObjectName cleans the name up and then appends one or more digits. If allow overwrite is not specified, the appended digits are chosen to return a unique name. If allow overwrite is specified, the appended digits represent suffixNum whether there is an existing object with the resulting name or not.

For an example using a base name, see ***Generating a Series of Names from a Base Name*** next.

Generating a Series of Names from a Base Name

If you are creating a series of waves, such as when loading a data file containing multiple columns, it is sometimes convenient to generate names of the form <baseName><number>, for example, wave0, wave1, wave2. To do this, you pass the base name as the nameInStr parameter and set bit 2 (input name is base name) of the options parameter.

If you disallow creation of names that are in use for existing objects (bit 1 of options cleared), CreateDataObjectName appends a number to the base name such that the output name is not in use for an object in the namespace of the type of object specified by objectType. In this case, suffixNum is not used and you should pass 0 for it.

If you allow creation of names that are in use for existing objects (bit 1 of options set), CreateDataObjectName appends the digits representing suffixNum to the base name to create the output name. In this case, you must pass the desired suffix number in each call to CreateDataObjectName.

Here is an example demonstrating this technique:

```
Function/S CreateSeriesOfWavesWithBaseName(dfr, baseName, beLiberal, allowOverwrite,
    numWavesToCreate)
    DFREF dfr                          // : for current data folder
    String baseName
    int beLiberal
    int allowOverwrite
    int numWavesToCreate

    int options = 4                    // inNameIsBaseName
    if (beLiberal)
        options += 1
    endif
    if (allowOverwrite)
        options += 2
    endif

    int suffixNum = 0
    String list = ""

    int i
    for(i=0; i<numWavesToCreate; i+=1)
        String outName = CreateDataObjectName(dfr, baseName, 1, suffixNum, options)
        Make/O/N=(100) dfr:$outName
        list += outName + ";"
        if (allowOverwrite)
            suffixNum += 1             // suffixNum for next call
        endif
    endfor

    return list
End
```

With overwrite disallowed, you must create the object each time through the loop because that is the only way that CreateDataObjectName can determine that the wave created in a previous iteration exists.

With overwrite allowed, you do not need to create the object each time through the loop although that is normally what you want.

See Also

Object Names on page III-501, **Programming with Liberal Names** on page IV-168, **CheckName**, **CleanupName**, **UniqueName**

CreationDate

CreationDate(*waveName*)

Returns creation date of wave as an Igor date/time value, which is the number of seconds from 1/1/1904.

The returned value is valid for waves created with Igor Pro 3.0 or later. For waves created in earlier versions, it returns 0.

See Also

ModDate.

Cross

Cross [**/DEST=destWave /FREE /T /Z**] ***vectorA*, *vectorB*** [, ***vectorC***]

The Cross operation computes the cross products *vectorA* x *vectorB* and *vectorA* x (*vectorB* x *vectorC*). Each vector is a 1D real wave containing 3 rows. Stores the result in the wave W_Cross in the current data folder.

Flags

/DEST=*destWave*	Stores the cross product in the wave specified by *destWave*. The destination wave is overwritten if it exists. The destination wave must be different from the input waves. The operation creates a wave reference for the destination wave if called in a user-defined function. See **Automatic Creation of WAVE References** on page IV-72 for details. If you omit /DEST, the operation stores the result in the wave W_Cross in the current data folder. Requires Igor7 or later.
/FREE	When used with /DEST, the destination wave is created as a free wave. See **Free Waves** on page IV-91 for details on free waves. /FREE is allowed in user-defined functions only. Requires Igor7 or later.
/T	Stores output in a row instead of a column in W_Cross.
/Z	Generates no errors for any unsuitable inputs.

csc

csc(*angle*)

The csc function returns the cosecant of *angle* which is in radians.

$$\csc(x) = \frac{1}{\sin(x)}.$$

In complex expressions, *angle* is complex, and csc(*angle*) returns a complex value.

See Also

sin, **cos**, **tan**, **sec**, **cot**

csch

csch(*x*)

The csch function returns the hyperbolic cosecant of *x*.

$$\mathrm{csch}(x) = \frac{1}{\sinh(x)} = \frac{2}{e^{x} - e^{-x}}.$$

In complex expressions, *x* is complex, and csch(*x*) returns a complex value.

See Also

cosh, **tanh**, **coth**, **sech**

CsrInfo

`CsrInfo(`*`cursorName`*` [, `*`graphNameStr`*`])`

The CsrInfo function returns a keyword-value pair list of information about the specified cursor (*cursorName* is A through J) in the top graph or graph specified by *graphNameStr*. It returns `""` if the cursor is not in the graph.

Details

The returned string contains information about the cursor in the following format:

```
TNAME:traceName; ISFREE:freeNum;POINT:xPointNumber;[YPOINT:yPointNumber;]
   RECREATION:command;
```

The *traceName* value is the name of the graph trace or image to which it is attached or which supplies the x (and y) values even if the cursor isn't attached to it.

If TNAME is empty, fields POINT, ISFREE, and YPOINT are not present.

The *freeNum* value is 1 if the cursor is not attached to anything, 0 if attached to a trace or image.

The POINT value is the same value **pcsr** returns.

The YPOINT keyword and value are present only when the cursor is attached to a two-dimensional item such as an image, contour, or waterfall plot or when the cursor is free. Its value is the same as returned by **qcsr**.

If cursor is free, POINT and YPOINT values are fractional relative positions (see description in the **Cursor** command).

The RECREATION keyword contains the Cursor commands (including /W) necessary to regenerate the current settings.

Examples

```
Variable aExists= strlen(CsrInfo(A)) > 0    // A is a name, not a string
Variable bIsFree= NumberByKey("ISFREE",CsrInfo(B,"Graph0"))
```

See Also

Programming With Cursors on page II-321.

Cursors — Moving Cursor Calls Function on page IV-339.

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

CsrWave

`CsrWave(`*`cursorName`*` [, `*`graphNameStr`*` [, `*`wantTraceName`*`]])`

The CsrWave function returns a string containing the name of the wave the specified cursor (A through J) is on in the top (or named) graph. If the optional *wantTraceName* is nonzero, the trace name is returned. A trace name is the wave name with optional instance notation (see **ModifyGraph (traces)**).

Details

The name of a wave by itself is not sufficient to identify the wave because it does not specify what data folder contains the wave. Thus, if you are calling CsrWave for the purpose of passing the wave name to other procedures, you should use the **CsrWaveRef** function instead. Use CsrWave if you want the name of the wave to use in an annotation or a notebook.

Examples

```
String waveCursorAIsOn = CsrWave(A)            // not CsrWave("A")
String waveCursorBIsOn = CsrWave(B,"Graph0")   // in specified graph
String traceCursorBIsOn = CsrWave(B,"",1)      // trace name in top graph
```

See Also

Programming With Cursors on page II-321.

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

CsrWaveRef

CsrWaveRef(*cursorName* [, *graphNameStr*])

The CsrWaveRef function returns a wave reference to the wave the specified cursor (A through J) is on in the top (or named) graph.

Details

The wave reference can be used anywhere Igor is expecting the name of a wave (not a string containing the name of a wave).

CsrWaveRef should be used in place of the CsrWave() string function to work properly with data folders.

Examples

```
Print CsrWaveRef(A)[50]              // not CsrWaveRef("A")
Print CsrWaveRef(B,"Graph0")[50]     // in specified graph
```

See Also

Programming With Cursors on page II-321.

Wave Reference Functions on page IV-197.

CsrXWave

CsrXWave(*cursorName* [, *graphNameStr*])

The CsrXWave function returns a string containing the name of the wave supplying the X coordinates for an XY plot of the Y wave the specified cursor (A through J) is attached to in the top (or named) graph.

Details

CsrXWave returns an empty string ("") if the wave the cursor is on is not plotted versus another wave providing the X coordinates (that is, if the wave was not plotted with a command such as `Display theWave vs anotherWave`).

The name of a wave by itself is not sufficient to identify the wave because it does not specify what data folder contains the wave. Thus, if you are calling CsrXWave for the purpose of passing the wave name to other Igor procedures, you should use the **CsrXWaveRef** function instead. Use CsrXWave if you want the name of the wave to use in an annotation or a notebook.

Examples

```
Display ywave vs xwave
```

ywave supplies the Y coordinates and xwave supplies the X coordinates for this XY plot.

```
Cursor A ywave,0
Print CsrXWave(A)       // prints xwave
```

See Also

Programming With Cursors on page II-321.

CsrXWaveRef

CsrXWaveRef(*cursorName* [, *graphNameStr*])

The CsrXWaveRef function returns a wave reference to the wave supplying the X coordinates for an XY plot of the Y wave the specified cursor (A through J) is attached to in the top (or named) graph.

Details

The wave reference can be used anywhere Igor is expecting the name of a wave (not a string containing the name of a wave).

CsrXWaveRef returns a null reference (see **WaveExists**) if the wave the cursor is on is not plotted versus another wave providing the X coordinates (that is, if the wave was not plotted with a command such as `Display theWave vs anotherWave`). CsrXWaveRef should be used in place of the CsrXWave string function to work properly with data folders.

Examples

```
Display ywave vs xwave
```

ywave supplies the Y coordinates and xwave supplies the X coordinates for this XY plot.

```
Cursor A ywave,0
Print CsrXWaveRef(A)[50]      // prints value of xwave at point #50
```

See Also

Programming With Cursors on page II-321.

Wave Reference Functions on page IV-197.

CTabList

`CTabList()`

The CTabList function returns a semicolon-separated list of the names of built-in color tables. It is useful for creating pop-up menus in control panels.

Color tables available through Igor Pro 4:

Grays	Rainbow	YellowHot	BlueHot	BlueRedGreen
RedWhiteBlue	PlanetEarth	Terrain		

Additional color tables added for Igor Pro 5:

Grays256	Rainbow256	YellowHot256	BlueHot256	BlueRedGreen256
RedWhiteBlue256	PlanetEarth256	Terrain256	Grays16	Rainbow16
Red	Green	Blue	Cyan	Magenta
Yellow	Copper	Gold	CyanMagenta	RedWhiteGreen
BlueBlackRed	Geo	Geo32	LandAndSea	LandAndSea8
Relief	Relief19	PastelsMap	PastelsMap20	Bathymetry9
BlackBody	Spectrum	SpectrumBlack	Cycles	Fiddle
Pastels				

Additional color tables added for Igor Pro 6:

RainbowCycle	Rainbow4Cycles	GreenMagenta16	dBZ14	dBZ21
Web216	BlueGreenOrange	BrownViolet	ColdWarm	Mocha
VioletOrangeYellow	SeaLandAndFire			

Additional color tables added for Igor Pro 6.2:

Mud	Classification

Additional color tables added for Igor Pro 9:

Turbo

See Also

Image Color Tables on page II-392, **Color Table Waves** on page II-399, **ColorTab2Wave**

CtrlBackground

`CtrlBackground` `[`***`key`*** `[`**`=`** ***`value`***`]]`...

The CtrlBackground operation controls the unnamed background task.

CtrlBackground works only with the unnamed background task. New code should used named background tasks instead. See **Background Tasks** on page IV-319 for details.

Parameters

dialogsOK=1 *or* 0 — If 1, your task will be allowed to run while an Igor dialog is present. This can potentially cause crashes unless your task is well-behaved.

noBurst=1 *or* 0 — Normally (or noBurst=0), your task will be called at maximum rate if a delay causes normal run times to be missed. Using noBurst=1, will suppress this burst catch up mode.

period=*deltaTicks* Sets the minimum number of ticks that must pass between invocations of the background task.

start[=*startTicks*] Starts the background task (designated by SetBackground) when the tick count reaches *startTicks*. If you omit *startTicks* the task starts immediately.

stop Stops the background task.

See Also

The **BackgroundInfo**, **SetBackground**, **CtrlNamedBackground**, **KillBackground**, and **SetProcessSleep** operations, and **Background Tasks** on page IV-319.

CtrlNamedBackground

`CtrlNamedBackground` ***`taskName`*****`,`** ***`keyword`*** **`=`** ***`value`*** `[,` ***`keyword`*** **`=`** ***`value`*** …]

The CtrlNamedBackground operation creates and controls named background tasks.

We recommend that you see **Background Tasks** on page IV-319 for an orientation before working with background tasks.

Parameters

taskName *taskName* is the name of the background task or _all_ to control all named background tasks. You can use any valid standard Igor object name as the background task name.

burst [= *b*] Enable burst catch up mode (off by default, *b*=0). When on (*b*=1), the task is called at the maximum rate if a delay misses normal run times.

dialogsOK [= *d*] Use dialogsOK=1 to allow the background task to run when a dialog window is active. By default, dialogsOK=0 is in effect. See **Background Tasks and Dialogs** on page IV-321 for details.

kill [= *k*] Stops and releases task memory for reuse (*k*=1; default) or continues (*k*=0).

noEvent=*mode* Controls Igor's event processing while a background task function is executing. The noEvents keyword was added in Igor Pro 8.04.

If your background task function uses a large fraction of available processor time, you may find that it is difficult to type on the command line or anywhere else. That is because Igor normally discards events that arrive while a user-defined function is running, causing events to be lost. This is the default mode of operation and corresponds to noEvents=0.

Using noEvents=1 causes Igor to postpone event handling until after the background task function returns which improves the reliability of the user interface. A consequence of using noEvents=1 is that you can not abort a rogue background task function using **User Abort Key Combinations**.

The noEvents keyword applies to all background tasks, including the unnamed background task. You should use _all_ for the taskName parameter.

mode defaults to 0 when Igor starts. The mode set using noEvents remains in effect until Igor quits.

period=*deltaTicks* Sets the minimum number of ticks (*deltaTicks*) that must pass between background task invocations. *deltaTicks* is truncated to an integer and clipped to a value greater than zero. See **Background Task Period** on page IV-320 for details.

proc=*funcName* Specifies name of a background user function (see **Details**).

start [=*startTicks*] Starts when the tick count reaches *startTicks*. A task starts immediately without *startTicks*.

status Returns background task information in the S_info string variable.

stop [= *s*] Stops the background task (*s*=1; default) or continues (*s*=0).

Details

The user function you specify via the proc keyword must have the following format:

```
Function myFunc(s)
   STRUCT WMBackgroundStruct &s
   …
```

The members of the `WMBackgroundStruct` are:

Base `WMBackgroundStruct` Structure Members

Member	Description
`char name[MAX_OBJ_NAME+1]`	Background task name.
`uint32 curRunTicks`	Tick count when task was called.
`int32 started`	TRUE when CtrlNamedBackground start is issued. You may clear or set to desired value.
`uint32 nextRunTicks`	Precomputed value for next run but user functions may change this.

You may also specify a user function that takes a user-defined STRUCT as long as the first elements of the structure match the `WMBackgroundStruct` or, preferably, if the first element is an instance of `WMBackgroundStruct`. Use the started field to determine when to initialize the additional fields. Your structure may not include any String, WAVE, NVAR, DFREF or other fields that reference memory that is not part of the structure itself.

If you specify a user-defined structure that matches the first fields rather than containing an instance of `WMBackgroundStruct`, then your function will fail if, in the future, the size of the built-in structure changes. The value of `MAX_OBJ_NAME` is 255 but this may also change. It was changed from 31 to 255 in Igor Pro 8.00.

Your function should return zero unless it wants to stop in which case it should return 1.

You can call CtrlNamedBackground within your background function. You can even switch to a different function if desired.

Use the status keyword to obtain background task information via the S_info variable, which has the format:

`NAME:`*`name`*`;PROC:`*`fname`*`;RUN:`*`r`*`;PERIOD:`*`p`*`;NEXT:`*`n`*`;QUIT:`*`q`*`;FUNCERR:`*`e`*`;`

When parsing S_info, do not rely on the number of key-value pairs or their order. RUN, QUIT, and FUNCERR values are 1 or 0, NEXT is the tick count for the next firing of the task. QUIT is set to 1 when your function returns a nonzero value and FUNCERR is set to 1 if your function could not be used for some reason.

See Also

See **Background Tasks** on page IV-319 for examples.

Demos

Choose File→Example Experiments→Programming→ Background Task Demo.

CtrlFIFO

`CtrlFIFO` ***`FIFOName`*** `[,` ***`key = value`***`]…`

The CtrlFIFO operation controls various aspects of the named FIFO.

Parameters

close	Closes the FIFO’s output or review file (if any).
deltaT=*dt*	Documents the data acquisition rate.
doffset=*dataOffset*	Used only with rdfile. Offset to data. If not provided offset is zero.
dsize=*dataSize*	Used only with rdfile. Size of data in bytes. If not provided, then data size is assumed to be the remainder of file. If this assumption is not valid then unexpected results may be observed.
flush	New data in FIFO is flushed to disk immediately.

file=*oRefNum*	File reference number for the FIFO's output file. You obtain this reference number from the **Open** operation used to create the file.
note=*noteStr*	Stores the note string in the file header. It is limited to 255 bytes.
rdfile=*rRefNum*	Like rfile but for review of raw data (use Open/R command). Channel data must match raw data in file. Offset from start of file to start of data can be provided using doffset given in same command. If data does not extend all the way to the end of the file, then the number of bytes of data can be provided using dsize in the same command.
rfile=*rRefNum*	File reference number for the FIFO's review file. Use a review file when you are using a FIFO to review existing data. Obtain the reference number from the Open/R operation used to open the file. File may be either unified header/data or a split format where the header contains the name of a file containing the raw data.
size=*s*	Sets number of chunks in the FIFO. The default is 10000. A chunk of data consists of a single data point from each of the FIFO's channels.
start	Starts the FIFO running by setting the time/date in the FIFO header, writing the header to the output file and marking the FIFO active.
stop	Stops the FIFO by flushing data to disk and marking the FIFO as inactive.
swap	Used only with rdfile. Indicates that the raw data file requires byte-swapping when it is read. This would be the case if you are running on a Macintosh, reading a binary file from a PC, or vice versa.

Details

Once start has been issued, the FIFO can accept no further commands except stop.

The FIFO must be in the valid state for you to access its data (using a chart control or using the **FIFO2Wave** operation). When you create a FIFO, using **NewFIFO**, it is initially invalid. It becomes valid when you issue the start command via the CtrlFIFO operation. It remains valid until you change a FIFO parameter using CtrlFIFO.

FIFOs are used for data acquisition.

See Also

The **NewFIFO** and **FIFO2Wave** operations, and **FIFOs and Charts** on page IV-313.

Cursor

```
Cursor [flags] cursorName traceName x_value
Cursor /F[flags] cursorName traceName x_value, y_value
Cursor /K[/W=graphName] cursorName
Cursor /I[/F][flags] cursorName imageName x_value, y_value
Cursor /M[flags] cursorName
```

The Cursor operation moves the cursor specified by *cursorName* onto the named trace at the point whose X value is *x_value*. or the coordinates of an image pixel or free cursor position at *x_value* and *y_value*.

Parameters

cursorName is one of ten cursors A through J.

Flags

/A=*a*	Activates (*a*=1) or deactivates (*a*=0) the cursor. Active cursors move with arrow keys or the cursor panel.
/C=(*r*,*g*,*b*[,*a*])	Sets the cursor color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

/DF=*format*

Sets the format to use when displaying date/time data in the Graph Info Panel (see **Info Panel and Cursors** on page II-319).

The /DF flag was added in Igor Pro 9.00.

The values for format are:

0: Compact format: YYMMDD HHMM.

1: Compact format with seconds added: YYMMDD HHMMSS. The seconds portion may optionally show fractions of seconds - see the /SDGT flag below.

2: Date and Time using a more readable format, the same format you get on a graph axis if you select the "short date" format. Time is formatted as HH:MM.

3: Date and Time with seconds added. Time is formatted as HH:MM:SS. The seconds portion may optionally show fractions of seconds - see the /SDGT flag below.

4: Time without the date. Time is formatted as HH:MM:SS. May optionally show fractions of seconds - see the /SDGT flag below.

/DGTS=*nd*

Sets the number of digits precision to use when a cursor value is displayed in the Graph Info Panel (see **Info Panel and Cursors** on page II-319). The number of digits is set by *nd* and must be a value from 1 to 15.

The /DGTS flag was added in Igor Pro 9.00.

/F

Cursor roams free. The trace or image provides the axis pair that defines x and y coordinates for the setting and readout. Use /P to set in relative coordinates, where 0,0 is the top left corner of the rectangle defined by the axes and 1,1 is the right bottom corner.

/H=*h*

Specifies crosshairs on cursors.

h =0: Full crosshairs off.

h =1: Full crosshairs on.

h =2: Vertical hairline.

h =3: Horizontal hairline.

/I

Places cursor on specified image.

/K

Removes the named cursor from the top graph.

/L=*lStyle*

Line style for crosshairs (full or small).

lStyle=0: Solid lines.

lStyle=1: Alternating color dash.

/M

Modifies properties without having to specify trace or image coordinates. Does not work with the /F or /I flags.

/N=*noKill*

Determines if the cursor is removed ("killed") if the user drags it outside of the plot area:

noKill=0: Remove the cursor (default).

noKill=1: Do not remove the cursor.

/NUML=*n*

Used in conjunction with /H when *h* is non-zero. Sets the number of crosshair lines to draw. *n* must be between 1 and 3. When *n* is greater than 1, the line separation is set by the /T=*t* flag. If *n* = 2 or 3 and *t* is less than 3, the line appears as if *n* is 1. If *n* = 3 and *t* is less than 5, the appearance reverts to *n* = 2. Lines are symmetrically disposed around the cursor position. When *n* = 3, *t* sets the separation of the outer pair of lines.

/NUML was added in Igor Pro 7.00.

/P Interpret *x_value* as a point number rather than an X value. If the cursor is on a trace representing a subrange of a wave, the point numbers are “trace” point numbers. See **Details** below.

When used with the /I flag, *x_value* and *y_value* are row and column numbers.

When used with the /F flag, *x_value* and *y_value* are relative graph coordinates (0-1).

/S=*s* Sets cursor style.

s=0:	Original square or circle.
s=1:	Small crosshair with letter.
s=2:	Small crosshair without letter.

/SDGT=*nd* Set the number of places to the right of the decimal point to be displayed in the Graph Info Panel (see **Info Panel and Cursors** on page II-319) when the display is in one of the date/time modes that includes seconds, or if the corresponding axis is showing elapsed time. *nd* is a value from 0 to 6.

The /SDGT flag was added in Igor Pro 9.00.

/T=*t* Sets the thickness of crosshair lines for /H when *h* is non-zero. If /NUML sets the number of lines greater than 1 then /T sets the separation of the outer pair of lines.

t is the line thickness or separation distance in units of pixels. The default is /T=1.

The form /T={*mode*, *t1*, *t2*} provides finer control.

/T was added in Igor Pro 7.00.

/T={*mode*,*t1*,*t2*} Sets the thickness of crosshair lines for /H when h is non-zero. If /NUML sets the number of lines greater than 1 then /T sets the separation of the outer pair of lines.

If *mode*=1 then *t1* and *t2* are in units of screen pixels. *t1* is the vertical line thickness or separation distance and *t2* is the horizontal line thickness or separation distance.

The default crosshair appearance is equivalent to /T={1,1,1}.

If *mode*=0 then *t1* and *t2* are in units of axis coordinates and consequently track changes in axis range and graph size. Normally *t1* is the vertical line thickness or separation distance and *t2* is the horizontal line thickness or separation distance but they are swapped if the trace or graph is in swap XY mode.

/T was added in Igor Pro 7.00.

/W=*graphName* Specifies a particular named graph window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *graphName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

Usually *traceName* is the same as the name of the wave displayed by that trace, but it could be a name in instance notation. See **ModifyGraph (traces)** and **Instance Notation** on page IV-20 for discussions of trace names and instance notation.

A string containing *traceName* can be used with the $ operator to specify the trace name.

x_value is an X value in terms of the X scaling of the wave displayed by *traceName*. If *traceName* is graphed as an XY pair, then *x_value* is *not* the same as the X axis coordinate. Since the X scaling is ignored when displaying an XY pair in a graph, we recommend you use the /P flag and use a point number for *x_value*.

cursorName is a name, *not* a string.

To get a cursor readout, choose ShowInfo from the Graph menu.

If a cursor is attached to a trace that represents a subrange of a wave, the /P flag causes *x_value* to be interpreted as a trace point number, not as a wave point number. For instance, if the trace was created by the command

```
Display yWave[4,25;3]
```

then *x_value*=0 references trace point 0 which corresponds to wave point 4, and *x_value*=1 references trace point 1 which corresponds to wave point 7.

Moving a cursor in a macro or function does not immediately erase the old cursor. DoUpdate has to be explicitly called.

Examples

```
Display myWave                          // X coordinates from X scaling of myWave
Cursor A, myWave, leftx(myWave)         //cursor A on first point of myWave

AppendToGraph yWave vs xWave            //X coordinates from xWave, not X scaling
Cursor/P B,yWave,numpnts(yWave)-1       //cursor B on last point of yWave
DoUpdate                                // erase any old A or B cursors
```

See Also

Programming With Cursors on page II-321 and the **DoUpdate** operation.

CursorStyle

`CursorStyle`

CursorStyle is a procedure subtype keyword that puts the name of the procedure in the “Style function” submenu of the Cursor Info pop-up menu. It is automatically used when Igor creates a cursor style function. To create a cursor style function, choose “Save style function” in the “Style function” submenu of the Cursor Info pop-up menu.

See also **Programming With Cursors** on page II-321.

CurveFit

`CurveFit` [***flags***] ***fitType***, [**kwCWave=*coefWaveName*,**] ***waveName*** [***flag parameters***]

The CurveFit operation fits one of several built-in functions to your data (for user-defined fits, see the **FuncFit** operation). When with CurveFit and built-in fit functions, automatic initial guesses will provide a good starting point in most cases.

The results of the fit are returned in a wave, by default W_coef. In addition, the results are put into the system variables K0, K1 … K*n* but the use of the system variables is limited and considered obsolete

As explained under **Fitting with Complex-Valued Functions** on page III-248, a user-defined fitting function can return complex values. A complex fitting function can be written to require a complex coefficient wave in which case the system variables are not supported.

You can specify your own wave for the coefficient wave instead of W_coef using the kwCWave keyword.

Virtually all waves specified to the CurveFit operation can be a sub-range of a larger wave using the same sub-range syntax as the Display operation uses for graphing. See **Wave Subrange Details** on page V-132.

See Chapter III-8, **Curve Fitting** for detailed information including the use of the Curve Fit dialog.

CurveFit operation parameters are grouped in the following categories: flags, fit type, parameters (kwCWave=*coefWaveName* and *waveName*), and flag parameters. The sections below correspond to these categories. Note that flags must precede the fit type and flag parameters must follow *waveName*.

Flags

/B=*pointsPerCycle* Used when *type* is sin; *pointsPerCycle* is the estimated number of data points per sine wave cycle. This helps provide initial guesses for the fit. You may need to try a few different values on either side of your estimated points/cycle.

/C Makes constraint matrix and vector. This only applies if you use the /C=*constraintSpec* parameter to specify constraints (see below). Creates the M_FitConstraint matrix and the W_FitConstraint vector. For more information, see **Fitting with Constraints** on page III-227.

/G Use values in variables K0, K1 … K*n* as starting guesses for a fit. If you specify a coefficient wave with the kwCWave keyword, the starting guesses will be read from the coefficient wave.

/H="*hhh…*"

Specifies coefficients to hold constant.

h is 1 for coefficients to hold, 0 for coefficients vary.

For example, `/H="100"` holds K0 constant, varies K1 and K2.

/K={*constants*}

Sets values of constants (not fit coefficients) in certain fitting functions. For instance, the exp_XOffset function contains an X offset constant. Built-in functions will set the constant automatically, but the automatic value can be overridden using this flag.

constants is a list of constant values, e.g., /K={1,2,3}. The length of the list must match the number of constants used by the chosen fit function.

This flag is not currently supported by the Curve Fit dialog. Use the To Cmd button and add the flag on the command line.

/L=*destLen*

Sets the length of the wave created by the AutoTrace feature, that is, /D without destination wave (see the /D parameter below, in the section Flag Parameters). The length of the wave fit_*waveName* will be set to *destLen*. This keyword also sets the lengths of waves created for confidence and prediction bands.

/M

Generates the covariance matrix, the waves CM_K*n*, where *n* is from 0 (for K0) to the number of coefficients minus one.

/M=*doMat*

Generates the covariance matrix. If *doMat* =2, the covariance matrix is put into a 2D matrix wave called M_Covar. If *doMat* =1 or is missing, the covariance matrix is generated as the 1D waves CM_K*n*, where *n* is from 0 (for K0) to the number of coefficients minus one. If *doMat* =0, the covariance matrix is not generated. *doMat* =1 is included for compatibility with previous versions; it is better to use *doMat* =2.

CurveFit does not generate the covariance matrix for line fits. Instead it reports the coefficient of correlation via the output variable V_rab.

/N[=*updateMode*]

Controls updating of windows during a curve fitting operation. Updating windows after each iteration can slow curve fitting down significantly.

updateMode = 0: An update is done every iteration so that you can monitor the progress of a fit. An update is performed when the fit finishes.

updateMode = 1: Suppresses updates after each fit iteration.If you use the /X flag, a screen update is done to ensure that the X range is accurate. /N is the same as /N=1. This is the default update mode.

updateMode = 2: Suppresses all updates including when using the /X flag. In a user-defined function, you must ensure that your graph has been updated by calling **DoUpdate** if you use the /X flag. This mode requires Igor Pro 9.00 or later.

See **Curve Fitting Screen Updates** for details.

/NTHR = *nthreads*

This flag is accepted but is obsolete and does nothing. See **Curve Fitting with Multiple Processors** on page III-249 for further information.

/O

Only generates initial guesses; doesn't actually do the fit.

Unless /ODR=2, this flag is ignored when used with linear fit types (line, poly, poly_XOffset, and poly2D). The **FuncFit** operation also ignores this flag.

/ODR=*fitMethod*

Selects a fitting method. Values for *fitMethod* are:

0: Default Levenberg-Marquardt least-squares method using old code.

1: Trust-region Levenberg-Marquardt ordinary least-squares method implemented using ODRPACK95 code. See **Curve Fitting References** on page III-267.

2: Trust-region Levenberg-Marquardt least orthogonal distance method implemented using ODRPACK95 code. This method is appropriate for fitting when there are measurement errors in the independent variables, sometimes called “errors in variables fitting”, “random regressor models,” or “measurement error models”.

3: Implicit fit. No dependent variable is specified; instead fitting attempts to adjust the fit coefficients such that the fit function returns zero for all dependent variables.

Implicit fitting will be of almost no use with the built-in fitting functions. Instead, use **FuncFit** and a user-defined fit function designed for an implicit fit.

/Q[=*quiet*]

If quiet = 1, prevents results from being printed in history. /Q is the same as /Q=1.

/TBOX = *textboxSpec*

If the top graph displays the fitted data, the /TBOX flag adds an annotation to the graph or updates the annotation if it already exists. Graphs other than the top graph are not affected.

The textbox contains a customizable set of information about the fit. The *textboxSpec* is a bitfield to select various elements to be included in the textbox:

1	Title “Curve Fit Results”
2	Date
4	Time
8	Fit Type (Least Squares, ODR, etc.)
16	Fit function name
32	Model Wave, the autodestination wave (includes a symbol for the trace if appropriate)
64	Y Wave, with trace symbol
128	X Wave
256	Coefficient value report
512	Include errors in the coefficient value report

Request inclusion of various parts by adding up the values for each part you want.

textboxSpec defaults to all bits on.

Setting *textboxSpec* to zero removes the textbox.

/X[=*fullRange*]

/X or /X=1 sets the X scaling of the auto-trace destination wave to match the appropriate X axis on the graph when the Y data wave is on the top graph. This is useful when you want to extrapolate the curve outside the range of X data being fit.

If you omit /X or specify /X=0, the X scaling of the auto-trace destination wave is set to the range of X values being fit.

/W=*wait*

Specifies behavior for the curve fit results window.

wait=1: Wait till user clicks OK button before dismissing curve fit results window. This is the default behavior from the command line or dialog.

wait=0: Display the curve fit window but do not wait for the user to click the OK button.

wait=2: Do not display the curve fit results window at all. This is the default for fits run from a procedure.

Fit Types

fitType is one of the built-in curve fit function types:

gauss

Gaussian peak: $y = K_0 + K_1 \exp\left[-\left(\frac{x - K_2}{K_3}\right)^2\right]$.

lor

Lorentzian peak: $y = K_0 + \frac{K_1}{(x - K_2)^2 + K_3}$.

Voight

Fits a Voigt peak, a convolution of a Lorentzian and a Gaussian peak. By changing the ratio of the widths of the Lorentzian and Gaussian peak components, the shape can grade between a Lorentzian shape and a Gaussian shape:

y = VoigtPeak(W_coef, x).

exp

Exponential: $y = K_0 + K_1 \exp(-K_2 x)$.

dblexp

Double exponential: $y = K_0 + K_1 \exp(-K_2 x) + K_3 \exp(-K_4 x)$.

exp_XOffset

Exponential: $y = K_0 + K_1 \exp(-(x - x_0)/K_2)$.

x_0 is a constant; by default it is set to the minimum x value involved in the fit. Inclusion of x_0 prevents problems with floating-point roundoff errors that can afflict the exp function.

dblexp_XOffset

Double exponential: $y = K_0 + K_1 \exp(-(x - x_0)/K_2) + K_3 \exp(-(x - x_0)/K_{4(2)})$.

x_0 is a constant; by default it is set to the minimum x value involved in the fit. Inclusion of x_0 prevents problems with floating-point roundoff errors that can afflict the exp function.

dblexp_peak

Double exponential with amplitudes of opposite sign, making a peak:

$$y = K_0 + K_1 \cdot \left\{-exp\left[\frac{-(x - K_4)}{K_2}\right] + exp\left[\frac{-(x - K_4)}{K_3}\right]\right\}$$

sin

Sinusoid: $y = K_0 + K_1 \sin(K_2 x + K_3)$.

line

Line: $y = K_0 + K_1 x$.

poly *n*

Polynomial: $y = k_0 + K_1 x + K_2 x^2 + \ldots$.

n is from 1 to 20. *n* is the number of terms or the degree plus one. Prior to Igor Pro 9.00, the minimum accepted value for *n* was 3.

n = 1 is equivalent to finding the mean of the Y values, and *n* = 2 fits a line to the data set, but doesn't output the special statistical information you get if you use the line fit type.

poly_XOffset n

Polynomial: y = K_0 + K_1*(x - x_0) + K_2*(x - x_0)^2 +...

n is from 1 to 20. *n* is the number of terms or the degree plus one. Prior to Igor Pro 9.00, the minimum accepted value for *n* was 3.

n = 1 is equivalent to finding the mean of the Y values, and *n* = 2 fits a line to the data set, but doesn't output the special statistical information you get if you use the line fit type.

x_0 is a constant; by default it is set to the minimum X value involved in the fit. Inclusion of x_0 prevents problems with floating-point roundoff errors when you have large values of X in your data set.

hillequation

Hill's Equation: $y = K_0 + \frac{(K_1 - K_0)}{1 + (K_3/x)^{K_2}}$.

This is a sigmoidal function. X values must be greater than 0.

sigmoid

$y = K_0 + \frac{K_1}{1 + \exp(K_2 - x/K_3)}$.

power

Power law: $y = K_0 + K_1 x^{K_2}$. X values must be greater than 0.

log

Log: $y = K_0 + K_1 \log(x)$. X values must be greater than 0.

lognormal

Log normal: $y = K_0 + K_1 \exp\left[-\left(\frac{\ln(x/K_2)}{K_3}\right)^2\right]$. X values must be greater than 0.

gauss2D

2D Gaussian: $z = K_0 + K_1 \exp\left[\frac{-1}{2(1 - K_6^2)}\left(\left(\frac{x - K_2}{K_3}\right)^2 + \left(\frac{y - K_4}{K_5}\right)^2 - \frac{2K_6(x - K_2)(y - K_4)}{K_3 K_5}\right)\right]$.

The cross-correlation coefficient (K_6) must be between -1 and 1. This coefficient is automatically constrained to lie in that range. If you are confident that the correlation is zero, it may greatly speed the fit to hold it at zero.

poly2D *n*

Two-dimensional polynomial: $z = K_0 + K_1 x + K_2 y + K_3 x^2 + K_4 xy + K_5 y^2 + \ldots$.

where *n* is the degree of the polynomial. All terms up to degree *n* are included, including cross terms. For instance, degree 3 terms are x^3, x^2y, xy^2, and y^3.

For more details on the built-in fit functions, see **Built-in Curve Fitting Functions** on page III-206.

Parameters

kwCWave=*coefWaveName* specifies an optional coefficient wave. If present, the specified coefficient wave is set to the final coefficients determined by the curve fit. If absent, a wave named W_coef is created and is set to the final coefficients determined by the curve fit.

If you use kwCWave=*coefWaveName* and you include the /G flag, initial guesses are taken from the specified coefficient wave.

waveName is the wave containing the Y data to be fit to the selected function *type*. You can fit to a subrange of the wave by supplying (*startX*,*endX*) after the wave name. Though not shown in the syntax description, you can also specify the subrange in points by supplying [*startP*,*endP*] after the wave name. See **Wave Subrange Details** on page V-132 for more information on subranges of waves in curve fitting.

If you are using one of the two-dimensional fit functions (gauss2D or poly2D) either *waveName* must name a matrix wave or you must supply a list of X waves via the /X flag.

Flag Parameters

These flag parameters must follow *waveName*.

/A=*appendResid*

appendResid =1 (default) appends the automatically-generated residual to the graph and *appendResid* =0 prevents appending (see /R[=*residwaveName*]). With *appendResid* =0, the wave is generated and filled with residual values, but not appended to the graph.

/AD[=*doAutoDest*]

If *doAutoDest* is 1, it is the same as /D alone. /AD is the same as /AD=1.

/C=*constraintSpec* Applies linear constraints during curve fitting. Constraints can be in the form of a text wave containing constraint expressions (/C=*textWaveName*) or a suitable matrix and vector (/C={*constraintMatrix*, *constraintVector*}). See **Fitting with Constraints** on page III-227. **Note**: Constraints are not available for the built-in line, poly and poly2D fit functions. To apply constraints to these fit functions you must create a user-defined fit function.

/D [=*destwaveName*] *destwaveName* is evaluated based on the equation resulting from the fit. *destwaveName* must have the same length as *waveName*.

If only /D is specified, an automatically-named wave is created. The name is based on the *waveName* with "fit_" as a prefix. This automatically named wave will be appended (if necessary) to the top graph if *waveName* is graphed there. The X scaling of the fit_ wave is set from the range of x data used during the fit.

By default the length of the automatically-created wave is 200 points (or 2 points for a straight line fit). This can be changed with the /L flag.

If *waveName* is a 1D wave displayed on a logarithmic X axis, Igor also creates an X wave with values exponentially spaced. The name is based on *waveName* with "fitX_" as a prefix.

If you fit to a user-defined function that returns complex values, the destination wave will also be complex.

/F={*confLevel*, *confType* [, *confStyleKey* [, *waveName*…]]}

Calculates confidence intervals for a confidence level of *confLevel*. The value of *confLevel* must be between 0 and 1 corresponding to confidence levels of 0 to 100 per cent.

confType selects what to calculate:

1:	Confidence bands for the model.
2:	Prediction bands for the model.
4:	Confidence intervals for the fit coefficients.

These values can be added together to select multiple options. That is, to select both a confidence band and fit coefficient confidence intervals, set *confType* to 5.

Confidence and prediction bands can be shown as waves contouring a given confidence level (use "Contour" for *confStyleKey*) or as error bars (use "ErrorBar" for *confStyleKey*). The default is Contour.

If no waves are specified, waves to contain the results are automatically generated and appended to the top graph (if the top graph contains the fitted data). See **Confidence Band Details** for details on the waves for confidence bands.

Note: Confidence bands and prediction bands are not available for multivariate curve fits.

/I [=*weightType*] If *weightType* is 1, the weighting wave (see /W parameter) contains standard deviations. If *weightType* is 0, the weighting wave contains reciprocal of the standard deviation. If the /I parameter is not present, the default is /I=0.

/M=*maskWaveName* Specifies that you want to use the wave named *maskWaveName* to select points to be fit. The mask wave must match the dependent variable wave in number of points and dimensions. Setting a point in the mask wave to zero or NaN (blank in a table) eliminates that point from the fit. The mask wave must be real even when fitting complex data.

/R [=*residwaveName*]

Calculates elements of *residwaveName* by subtracting model values from the data values. *residwaveName* must have the same length as *waveName*.

If only /R is specified, an automatically-named wave is created with the same number of points as *waveName*. The name is based on *waveName* with “Res_” as a prefix. If a new wave needs to be created, it is automatically filled with NaN, but if the wave already exists it is simply re-used.

During fitting the residual is computed only for points that actually participate in the fit. If you fit to a subrange or use a mask wave to fit to specific points, only those points are stored. See **Residuals Using Auto Trace** on page III-219 for details.

The automatically created residual wave is appended (if necessary) to the top graph if *waveName* is graphed there. The residual wave is appended to a new free axis named by prepending “Res_” to the name of the vertical axis used for plotting *waveName*. To the extent possible, the new free axis is formatted nicely.

If the graph containing the data to be fit has very complex formatting, you may not wish to automatically append the residual to the graph. In this case, use /A=0.

If you fit to a user-defined function that returns complex values, the residual wave will also be complex.

/AR=*doAutoResid*

If *doAutoResid* is 1, it is the same as /R alone. /AR is the same as /AR=1.

/W=*wghtwaveName*

wghtwaveName contains weighting values applied during the fit, and must have the same length as *waveName*. These weighting values can be either the reciprocal of the standard errors, or the standard errors. See the /I parameter above for details.

If you fit to a user-defined function that returns complex values, the weighting wave must also be complex.

/X=*xwaveName*

The X values for the data to fit come from *xwaveName*, which must have the same length and type as *waveName*.

If you are fitting to one of the two-dimensional fit functions and *waveName* is a matrix wave, *xwaveName* supplies independent variable data for the X dimension. In this case, *xwaveName* must name a 1D wave with the same number of rows as *waveName*.

When fitting to a complex-valued fit function, the X waves must be real or complex consistent with the X input or parameters declared by your fitting function.

/X={*xwave1*, *xwave2*}

For fitting to one of the two-dimensional fit functions when *waveName* is a 1D wave. *xwave1* and *xwave2* must have the same length as *waveName*.

/Y=*ywaveName*

For fitting using one of the 2D fit functions if *waveName* is a matrix wave. *ywaveName* must be a 1D wave with length equal to the number of columns in *waveName*.

/NWOK

Allowed in user-defined functions only. When present, certain waves may be set to null wave references. Passing a null wave reference to CurveFit is normally treated as an error. By using /NWOK, you are telling CurveFit that a null wave reference is not an error but rather signifies that the corresponding flag should be ignored. This makes it easier to write function code that calls CurveFit with optional waves.

The waves affected are the X wave or waves (/X), weight wave (/W), mask wave (/M) and constraint text wave (/C). The destination wave (/D=wave) and residual wave (/R=wave) are also affected, but the situation is more complicated because of the dual use of /D and /R to mean "do autodestination" and "do autoresidual". See /AR and /AD.

If you don't need the choice, it is better not to include this flag, as it disables useful error messages when a mistake or run-time situation causes a wave to be missing unexpectedly.

Note: To work properly this flag must be the last one in the command.

Flag Parameters for Nonzero /ODR

/XW=*xWeightWave*

/XW={*xWeight1*, *xWeight2*}

/ODR=2 or 3 only.

Specifies weighting values for the independent variables using *xWeightWave*, which must have the same length as *waveName*. When fitting to one of the multivariate fit functions such as poly2D or Gauss2D, you must supply a weight wave for each independent variable using the second form.

Weighting values can be either the reciprocal of the standard errors, or the standard errors. The choice of standard error or reciprocal standard error must be the same for both /W and /XW. See /I for details.

When fitting to a complex-valued fit function, the X weighting waves must be real or complex consistent with the X parameters declared by your fitting function.

/XHLD=*holdWave*

/XHLD={*holdWave1*, *holdWave2*}

/ODR=2 or 3 only.

Specifies a wave or waves to hold the values of the independent variables fixed during orthogonal distance regression. The waves must match the input X data; a one in a wave element fixes the value of the corresponding X value.

/CMAG=*scaleWave* Specifies a wave that indicates the expected scale of the fit coefficients at the solution. If different coefficients have very different orders of magnitude of expected values, this can improve the efficiency and accuracy of the fit.

/XD=*xDestWave*

/XD={*xDestWave1*, *xDestWave2*}

/ODR=2 or 3 only.

Specifies a wave or waves to receive the fitted values of the independent variables during a least orthogonal distance regression.

When fitting to a complex-valued fit function, the X destination waves must be real or complex consistent with the X parameters declared by your fitting function.

/XR=*xResidWave*

/XR={*xResidWave1*, *xResidWave2*}

/ODR=2 or 3 only.

Specifies a wave or waves to receive the differences between fitted values of the independent variables and the starting values during a least orthogonal distance regression. That is, they will be filled with the X residuals.

When fitting to a complex-valued fit function, the X residual waves must be real or complex consistent with the X parameters declared by your fitting function.

Details

CurveFit gets initial guesses from the K*n* system variables when user guesses (/G) are specified, unless a coefficient wave is specified using the kwCWave keyword. Final curve fit parameters are written into a wave name W_coef, unless you specify a coefficient wave with the kwCWave keyword.

Other output waves are M_Covar (see the /M flag), M_FitConstraint and W_FitConstraint (see /C parameter and **Fitting with Constraints** on page III-227) and W_sigma.

For compatibility with earlier versions of Igor, the parameters are also stored in the system variables Kn. This can be a source of confusion. We suggest you think of W_coef as the **output** coefficients and Kn as **input** coefficients that get overwritten.

Other output waves are M_Covar (see the /M flag), M_FitConstraint and W_FitConstraint (see /C parameter and **Fitting with Constraints** on page III-227), W_sigma. If you have selected coefficient confidence limits using the /F parameter, a wave called W_ParamConfidenceInterval is created with the confidence intervals for the fit coefficients.

CurveFit stores other curve fitting statistics in variables whose names begin with "V_". CurveFit also looks for certain V_ variables which you can use to modify its behavior. These are discussed in **Special Variables for Curve Fitting** on page III-232.

When fitting with /ODR=nonzero, fitting with constraints is limited to simple "bound constraints." That is, you can constrain a fit coefficient to be greater than or less than some value. Constraints involving combinations of fit coefficients are supported only with /ODR=0. The constraints are entered in the same way, using an expression like `K0>1`.

Wave Subrange Details

Almost any wave you specify to CurveFit can be a subrange of a wave. The syntax for wave subranges is the same as for the Display command (see **Subrange Display Syntax** on page II-321 for details). However, the Display command allows only one dimension to have a range (multiple elements from the dimension); if a multidimensional wave is appropriate for CurveFit, you may use a range for more than one dimension.

Some waves must have the same number of points as other waves. For instance, a one-dimensional Y wave must have the same number of points as any X waves. Thus, if you use a subrange for an X wave, the number of points in the subrange must match the number of points being used in the Y wave (but see **Subrange Backward Compatibility** on page V-133 for a complication to this rule).

A common use of wave subranges might be to package all your data into a single multicolumn wave, along with the residuals and model values. For a univariate fit, you might need X and Y waves, plus a destination (model) wave and a residual wave. You can achieve all of that using a 4 column wave. For example:

```
Make/D/N=(100, 4) Data
... fill column zero with X data and column one with Y data ...
CurveFit poly 3, Data[][1] /X=Data[][0]/D=Data[][2]/R=Data[][3]
```

Note that because all the waves are full columns from a single multicolumn wave, the number of points is guaranteed to be the same.

The number of points used for X waves (*xwaveName* or {*xwave1*, *xwave2*, …}), weighting wave (*wghtwaveName*), mask wave (`maskWaveName`), destination wave (*destwaveName*) and residual wave (*residwaveName*) must be the same as the number of points used for the Y wave (*waveName*). If you specify your own confidence band waves (/F flag) they must match the Y wave; you cannot use subranges with confidence band waves. If you set /ODR = nonzero, the X weight, hold, destination and residuals waves must match the Y wave.

The total number of points in each wave does not need to match other waves, just the number of points in the specified subrange.

When fitting to a univariate fit function (that includes almost all the fit types) the Y wave must have effectively one dimension. That means the Y wave must either be a 1D wave, or it must have a subrange that makes the data being used one dimensional. For instance:

```
Make/N=(100,100) Ydata          // 2D wave
CurveFit gauss Ydata[][0]       // OK- a single column is one-dimensional
CurveFit gauss Ydata[2][]       // OK- s single row is one-dimensional
CurveFit gauss Ydata            // not OK- Ydata is two-dimensional
CurveFit gauss Ydata[][0,1]     // not OK- two columns makes 2D subrange
```

When fitting a multivariate function (**poly2D** or **Gauss2D**) you have the choice of making the Y data either one-dimensional or two-dimensional. If it is one-dimensional, then you must be fitting XYZ (or Y,X1,X2) triplets. In that case, you must provide a one-dimensional Y wave and two one-dimensional X waves, or 2 columns from a multicolumn wave. For instance:

These are OK:

```
Make/N=(100,3) myData
CurveFit Gauss2D myData[][0] /X={myData[][1],myData[][2]}
CurveFit Gauss2D myData[][0] /X=myData[][1,2]
```

These are not OK:

```
CurveFit Gauss2D myData /X={myData[][1],myData[][2]}// 2D Y wave with 1D X waves
CurveFit Gauss2D myData[][0] /X=myData            // too many X columns
```

If you use a 2D Y wave, the X1 and X2 data can come from the grid positions and the Y wave's X and Y index scaling, or you can use one-dimensional waves or wave subranges to specify the X1 and X2 positions of the grid:

```
Make/N=(20,30) yData
CurveFit Gauss2D yData        //OK- 2D Y data, X1 and X2 from scaling
Make/N=20 x1Data
Make/N=30 x2Data
// OK: 2D effective Y data, matching 1D X and Y flags
CurveFit Gauss2D yData[0,9][0,19] /X=x1Data[0,9]/Y=x2data[10,29]
// OK: effective 2D Y data
Make/N=(10,20,3) Y data
CurveFit Gauss2D yData[][][0]
```

There are, of course, lots of possible combinations, too numerous to enumerate.

Subrange Backward Compatibility

Historically, a Y wave could have a subrange. The same subrange applied to all other waves. For backward compatibility, if you use a subrange with the Y wave only, and other waves lack a subrange, these other waves must have either: 1) The same total number of points as the total number of points in the Y wave in which case the Y wave subrange will be applied; or 2) The same total number of points as the Y wave's subrange.

In addition, the Y wave can take a subrange in parentheses to indicate that the subrange refers to the Y wave's scaled indices (X scaling). If you use parentheses to specify an X range, you must satisfy the old subrange rules: All waves must have the same number of points. Subrange is allowed for the Y wave only. The Y wave subrange is applied to all other waves.

Confidence Band Details

Automatic generation of confidence and prediction bands occurs if the /F={…} parameter is used with no wave names. One to four waves are generated, or you can specify one to four wave names yourself depending on the *confKind* and *confStyle* settings.

Waves auto-generated by /F={*confLevel*, *confKind*, *confStyle*}:

confKind	*confStyle*	**What You Get**	**Auto Wave Names**
1	"Contour"	upper and lower confidence contours	UC_*dataName,* LC_*dataName*
2	"Contour"	upper and lower prediction contours	UP_*dataName,* LP_*dataName*
3	"Contour"	upper and lower confidence contours and prediction contours	UC_*dataName,* LC_*dataName,* UP_*dataName,* LP_*dataName*
1	"ErrorBar"	confidence interval wave	CI_*dataName*
2	"ErrorBar"	prediction interval wave	PI_*dataName*
3	"ErrorBar"	confidence and prediction interval waves	CI_*dataName,* PI_*dataName*

Note that *confKind* may have 4 added to it if you want coefficient confidence limits calculated as well.

The contour waves are appended to the top graph as traces if the data wave is displayed in the top graph. The wave names have *dataName* replaced with the name of the wave containing the Y data for the fit.

Waves you must supply for /F={*confLevel, confKind, confStyle, wave, wave…*}:

confKind	*confStyle*	**You Supply**
1	"Contour"	2 waves to receive upper and lower confidence contours.
2	"Contour"	2 waves to receive upper and lower prediction contours.
3	"Contour"	4 waves to receive upper and lower confidence and upper and lower prediction contours.
1	"ErrorBar"	1 wave to receive values of confidence band width.
2	"ErrorBar"	1 wave to receive values of prediction band width.
3	"ErrorBar"	2 waves to receive values of confidence and prediction band widths.

The waves you supply must have the same number of points as the dependent variable data wave. The band intervals will be calculated at the X values of the input data. These waves are not automatically appended to a graph; it is expected that you will display the contour waves as traces or use the error bar waves to make error bars on the model fit wave.

Residual Details

Residuals are calculated only for elements corresponding to elements of waveName that are included in the fit. Thus, you can calculate residuals automatically for a piecewise fit done in several steps.

The automatic residual wave will be appended to the top graph if the graph displays the Y data. It is appended to a new free axis positioned directly above the axis used to display the Y data, making a stacked graph. Other axes are shortened as necessary to make room for the new axis. You can alter the axis formatting later. See **Creating Stacked Plots** on page II-324 for details.

While Igor will go to some lengths to make a nicely formatted stacked graph, the changes made to the graph formatting may be undesirable in certain cases. Use /A=0 to suppress the automatic append to the graph. The automatic residual wave will be created and filled with residual values, but not appended to the graph.

Curve Fitting Screen Updates

A screen update redraws windows displaying data that has changed since the previous update, if any. Updates can take a significant amount of time, so the /N flag allows you to control them during a curve fitting operation.

Igor historically performed a screen update after each curve fitting iteration so that a graph showing the fit destination would display the latest trial solution after each iteration. This was default behavior, equivalent to /N=0.

As processors became faster, updating after every iteration became less useful because, in most cases, the time between iterations is short and the entire fit finishes quickly. Consequently, for Igor Pro 7.00, we changed the default to update only at the end of a fit, equivalent to /N=1.

There are some potential pitfalls to suppressing screen updates.

If you use the /X flag to tell Igor to extrapolate the fit curve to the entire X range of a graph, and if your procedures alter the X axis range, you need to call DoUpdate before calling CurveFit or FuncFit to allow Igor to finalize the change to the axis range.

If you call CurveFit or FuncFit from a user-defined function and use the /D flag, which turns on the auto-destination feature, then the fit curve will not update until all running functions in the call chain return. If it is important that the graph displaying the destination wave reflect the fit result before functions return, you must call DoUpdate.

See Also

Inputs and Outputs for Built-In Fits on page III-212 and **Special Variables for Curve Fitting** on page III-232 as well as **Accessing Variables Used by Igor Operations** on page IV-123.

When fitting to a user-specified function, see **FuncFit**. For multivariate user-specified fitting functions, see **FuncFit** and **FuncFitMD**. See **Confidence Bands and Coefficient Confidence Intervals** on page III-223 for a detailed description of confidence and prediction bands.

References

An explanation of the Levenberg-Marquardt nonlinear least squares optimization can be found in Chapter 14.4 of Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

CustomControl

CustomControl [**/Z**] ***ctrlName*** [***keyword* = *value*** [, ***keyword* = *value*** …]]

The CustomControl operation creates or modifies a custom control in the target window. A CustomControl starts out as a generic button, but you can customize both its appearance and its action.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the CustomControl to be created or changed. See **Button** for standard default parameters.

The following keyword=value parameters are supported:

align=*alignment*
Sets the alignment mode of the control. The alignment mode controls the interpretation of the *leftOrRight* parameter to the pos keyword. The align keyword was added in Igor Pro 8.00.

If *alignment*=0 (default), *leftOrRight* specifies the position of the left end of the control and the left end position remains fixed if the control size is changed.

If *alignment*=1, *leftOrRight* specifies the position of the right end of the control and the right end position remains fixed if the control size is changed.

fColor=(*r*,*g*,*b*[,*a*])
Sets color of the button only when picture is not used and frame=1.

r, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

focusRing=*fr*
Enables or disables the drawing of a rectangle indicating keyboard focus:

fr=0: Focus rectangle will not be drawn.
fr=1: Focus rectangle will be drawn (default).

On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.

frame=*f*
Sets frame style used only when picture is not used:

f=0: No frame (only the title is drawn).
f=1: Default, a button is drawn with a centered title. Set fColor to something other than black to colorize the button.
f=2: Simple box.
f=3: 3D sunken frame. On Macintosh, when "native GUI appearance" is enabled for the control, the frame is filled with the proper operating system color.
f=4: 3D raised frame.
f=5: Text well.

help={*helpStr*}
Sets the help for the control.

helpStr is limited to 1970 bytes (255 in Igor Pro 8 and before).

You can insert a line break by putting "\r" in a quoted string.

labelBack=(*r*,*g*,*b*[,*a*]) or 0

Sets background color for the control only when a picture is not used and frame is not 1 and is not 3 on Macintosh.

r, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. If not set (or labelBack=0), then background is transparent (not erased).

mode=*m*
Notifies the control that something has happened. Can be used for any purpose. See **Details** discussion of the `kCCE_mode` event.

noproc
Specifies that no procedure will execute when clicking the custom control.

picture= *pict*
Uses the named Proc Pictures to draw the control. The picture is taken to be three side-by-side frames, which show the control appearance in the normal state, when the mouse is down, and in the disabled state.

The control action function can overwrite the picture number using the picture={*pict*,*n*} syntax.

The picture size overrides the size keyword.

picture={*pict*,*n*} Uses the specified Proc Picture to draw the control. The picture is *n* side-by-side frames instead of the default three frames.

pos={*leftOrRight*,*top*} Sets the position in **Control Panel Units** of the top/left corner of the control if its alignment mode is 0 or the top/right corner of the control if its alignment mode is 1. See the align keyword above for details.

pos+={*dx*,*dy*} Offsets the position of the control in **Control Panel Units**.

proc=*procName* Specifies the name of the action function for the control. The function must not kill the control or the window.

size={*width*,*height*} Sets size of the control in **Control Panel Units** but only when not using a Proc Picture.

title=*titleStr* Specifies text that appears in the control.

Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.

userdata(*UDName*)=*UDStr*

Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

value=*varName* Sets the numeric variable, string variable, or wave that is associated with the control. With a wave, specify a point using the standard bracket notation with either a point number (`value=awave[4]`) or a row label (`value=awave[%alabel]`).

valueColor=(*r*,*g*,*b*[, *a*]) Sets initial color of the title for the button drawn only when picture is not used and frame=1.

r, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

To further change the color of the title text, use escape sequences as described for title=*titleStr*.

Flags

/Z No error reporting.

Details

When you create a custom control, your action procedure gets information about the state of the control using the `WMCustomControlAction` structure. See **WMCustomControlAction** for details on the `WMCustomControlAction` structure.

Although the return value is not currently used, action procedures should always return zero.

When Igor calls your action procedure for the `kCCE_draw` event, the basic button picture (custom or default) will already have been drawn. You can use standard drawing commands such as **DrawLine** to draw on top of the basic picture. Unlike the normal situation when drawing commands merely add to a drawing operation list which only later is drawn, `kCCE_draw` event drawing commands are executed directly. The coordinate system, which you can not change, is **Control Panel Units** with (0,0) being the top left corner of the control. Most drawing commands are legal but because of the immediate nature of drawing, the /A (append) flag of **DrawPoly** is not allowed.

The `kCCE_mode` event, which Igor sends when you execute CustomControl with the mode keyword, can be used for any purpose, but it mainly serves as a notification to the control that something has happened. For example, to send information to a control, you can set a named (or the unnamed) userdata and then set the `mode` to indicate that the control should examine the userdata. For this signaling purpose, you should use a `mode` value of 0 because this value will not become part of the recreation macro.

The kCCE_frame event is sent just before drawing one of the *pict* frames, as set by the picture parameter. On input, the curFrame field is set to 0 (normal or mouse down outside button), to 1 (mouse down in button), or to 2 (disable). You may modify curFrame as desired but your value will be clipped to a valid value.

When you specify a *pict* with the picture parameter, you will get a kCCE_drawOSBM event when that *pict* is drawn into an offscreen bitmap. Once it is created, all updates use the offscreen bitmap until you specify a new picture parameter. Thus the custom drawing done at this event is static, unlike drawing done during the kCCE_draw event, which can be different each time the control is drawn. Because the *pict* can be contain multiple side-by-side frames, the width of the offscreen bitmap is the width derived from the ctrlRect field multiplied by the number of frames.

Because the action function is called in the middle of various control events, it must not kill the control or the window. Doing so will almost certainly cause a crash.

CustomControl Action Procedure

The action procedure for a CustomControl control can takes a predefined structure WMCustomControlAction as a parameter to the function:

```
Function ActionProcName(s)
   STRUCT WMCheckboxAction& s
   …
   return 0
End
```

See **WMCustomControlAction** for details on the WMCustomControlAction structure.

Although the return value is not currently used, action procedures should always return zero.

The action procedure may have drawing commands such as **SetDrawEnv**, **DrawRect**, etc. These commands include a /W flag that directs their action to a particular window or subwindow. However, in a CustomControl action procedure, the /W flag is ignored and all drawing commands are directed to the window or subwindow containing the control.

Because the action procedure runs during a drawing operation that cannot be interrupted without crashing Igor, the debugger cannot be invoked while it is running. Consequently breakpoints set in the function are ignored. Use **Debugging With Print Statements** on page IV-212 instead.

Examples

See **Creating Custom Controls** on page III-424 for some examples of custom controls.

For a demonstration of custom controls, choose File→Example Experiments→Feature Demos 2→Custom Control Demo.

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

The **ControlInfo** operation for information about the control.

The **GetUserData** function for retrieving named user data.

Proc Pictures on page IV-56.

The **TextBox**, **DrawPoly** and **DefaultGUIControls** operations.

CWT

CWT [***flags***] ***srcWave***

The CWT operation computes the continuous wavelet transform (CWT) of a 1D real-valued input wave (*srcWave*). The input can be of any numeric type. The computed CWT is stored in the wave M_CWT in the current data folder. M_CWT is a double precision 2D wave which, depending on your choice of mother wavelet and output format, may also be complex. The dimensionality of M_CWT is determined by the specifications of offsets and scales. The operation sets the variable V_flag to zero if successful or to a nonzero number if it fails for any reason.

Flags

/ENDM=*method* Selects the method used to handle the two ends of the data array with direct integration (/M=1).

method=0: Padded on both sides by zeros.

method=1: Reflected at both the start and end.

method=2: Entered with cyclical repetition.

/FSCL Use correction factor to the wave scaling of the second dimension of the output wave so that the numbers are more closely related to Fourier wavelength. See **References** for more information on the calculation of these correction factors. This flag does not affect the output from the Haar wavelet.

/M=*method* Specifies the CWT computation method.

method=0: Fast method uses FFT (default).

method=1: Slower method using direct integration.

You should mostly use the more efficient FFT method. The direct method should be reserved to situations where the FFT is not producing optimal results. Theoretically, when the FFT method fails, the direct method should also be fairly inaccurate, e.g., in the case of undersampled signal. The main advantage in the direct method is that you can use it to investigate edge effects.

/OSCW CWT creates an output scaing wave named W_CWTScaling that can be used as the Y wave in an image plot. For example:

```
Display; AppendImage M_CWT vs {*,W_CWTScaling}
```

If you omit /OSCW, CWT creates W_CWTScaling only if you include /SMP2=4.

/OSCW was added in Igor Pro 9.00.

/OUT=*format* Sets the format of the output wave M_CWT:

format=1: Complex.

format=2: Real valued.

format=4: Real and contains the magnitude.

Depending on the method of calculation and the choice of mother wavelet, the "native" output of the transform may be real or complex. You can force the output to have a desired format using this flag.

/Q Quiet mode; no results printed to the history.

/R1={*startOffset*, *delta1*, *numOffsets*

Specifies offsets for the CWT. Offsets are the first dimension in a CWT. Normally you will calculate the CWT for the full range of offsets implied by *srcWave* so you will not need to use this flag. However, when using the slow method, this flag restricts the output range of offsets and save some computation time. *startOffset* (integer) is the point number of the first offset in *srcWave*. *delta1* is the interval between two consecutive CWT offsets. It is expressed in terms of the number *srcWave* points. *numOffsets* is the number of offsets for which the CWT is computed.

By default *startOffset*=0, *delta1*=1, and *numOffsets* is the number of points in *srcWave*. If you want to specify just the *startOffset* and *delta1*, you can set *numOffsets*=0 to use the same number of points as the source wave.

/R2={*startScale*, *scaleStepSize*, *numScales*}

Specifies the range of scales for the CWT is computation. Scales are the second dimension in the output wave. Note however that there are limitations on the minimum and maximum scales having to do with the sampling of your data. Because there is a rough correspondence between a Fourier spatial frequency and CWT scale it should be understood that there is also a maximum theoretical scale. This is obvious if you compute the CWT using an FFT but it also applies to the slow method. If you specify a range outside the allowed limits, the corresponding CWT values are set to NaN.

Use NaN if you want to use the default value for any parameter.

The default value for *startScale* is determined by sampling of the source wave and the wavelet parameter or order.

At a minimum you must specify either *scaleStepSize* or *numScales*.

/SMP1=*offsetMode* Determines computation of consecutive offsets. Currently supporting only *offsetMode*=1 for linear, user-provided partial offset limits (see /R1 flag): *`val1=startOffset+numOffsets*delta1`*.

/SMP2=*scaleMode* Determines computation of consecutive scales. *scaleMode* is 1 by default if you specify the /R2 flag.

format=1: Linear:
```
theScale = startScale+index*scaleStepSize
```

format=2: Auto:
```
theScale = numScales * startScale/(index+1)
```

format=4: Power of 2 scaling interval:
```
theScale = startScale*2.^(index*scaleStepSize)
```

When using *scaleMode*=4 the operation saves the consecutive scale values in the wave W_CWTScaling. Note also that if you use *scaleMode*=4 without specifying a corresponding /R2 flag, the default *scaleStepSize* of 1 and 64 scale values gives rise to scale values that quickly exceed the allowed limits.

(See /R2 flag for details about the different parameters used in the equations above.)

/SUBM Subtracts the mean of the input before performing the transform. /SUBM was added in Igor Pro 9.00.

/SW2=*sWave* Provides specific scale values at which the transform is evaluated. Use instead of /R2 flag. It is your responsibility to make sure that the entries in the wave are appropriate for the sampling density of *srcWave*.

/WBI1={*Wavelet* [, *order*]}

Specifies the built-in wavelet (mother) function. *Wavelet* is the name of a wavelet function: Morlet (default), MorletC (complex), Haar, MexHat, DOG, and Paul.

Morlet:

$$\Psi_0(x) = \frac{1}{\pi^{1/4}} \cos(\omega x) \exp\left(-\frac{x^2}{2}\right).$$

By default, ω=5. Use the /WPR1 flag to specify other values for ω.

MorletC:

$$\Psi_0(x) = \frac{1}{\pi^{1/4}} \exp(i\omega x) \exp\left(-\frac{x^2}{2}\right).$$

By default, ω=5. Use the /WPR1 flag to specify other values for ω.

Haar:

$$\Psi_0(x) = \begin{cases} 1 & 0 \le x < 0.5 \\ -1 & 0.5 \le x < 1 \end{cases}.$$

DOG:

$$\Psi_0(x) = \frac{(-1)^{m+1}}{\sqrt{\Gamma\left(m + \frac{1}{2}\right)}} \frac{d^m}{dx^m}\left(\exp\left(-\frac{x^2}{2}\right)\right).$$

MexHat: Special case of DOG with *m*=2.

Paul:

$$\Psi_0(m,x) = \frac{2^m i^m m!}{\sqrt{\pi(2m)!}} (1 - ix)^{-(m+1)}.$$

order applies to DOG and Paul wavelets only and specifies *m*, the particular member of the wavelet family.

The default wavelet is the Morlet.

/WPR1={*param1*} *param1* is a wavelet-specific parameter for the wavelet function selected by /WBI1. For example, use /WPR1={6} to change the Morlet frequency from the default (5).

/Z No error reporting. If an error occurs, sets V_flag to -1 but does not halt function execution.

Details

The CWT can be computed directly from its defining integral or by taking advantage of the fact that the integral represents a convolution which in turn can be calculated efficiently using the fast Fourier transform (FFT).

When using the FFT method one encounters the typical sampling problems and edge effects. Edge effects are also evident when using the slow method but they only significant in high scales.

From sampling considerations it can be shown that the maximum frequency of a discrete input signal is 1/2dt where dt is the time interval between two samples. It follows that the smallest CWT scale is 2dt and the largest scale is Ndt where N is the total number of samples in the input wave.

The transform in M_CWT is saved with the wave scaling. *startOffset* and *delta1* are used for the X-scaling. Both *startOffset* and *delta1* are either specified by the /R1 flag or copied from *srcWave*. The Y-scaling of M_CWT depends on your choice of /SMP2. If the CWT scaling is linear then the wave scaling is based on *startScale* and *scaleStepSize*. If you are using power of 2 scaling interval then the Y wave scaling of M_CWT has a *start*=0 and *delta*=1 and the wave W_CWTScaling contains the actual scale values for each column of M_CWT. Note that W_CWTScaling has one extra data point to make it suitable for display using an operation like:

```
AppendImage M_CWT vs {*, W_CWTScaling}
```

We have encountered two different definitions for the Morlet wavelet in the literature. The first is a complex function (MorletC) and the second is real (Morlet). Instead of choosing one of these definitions we implemented both so you may choose the appropriate wavelet.

See Also

For discrete wavelet transforms use the **DWT** operation. The **WignerTransform** and **FFT** operations.

For further discussion and examples see **Continuous Wavelet Transform** on page III-282.

References

Torrence, C., and G.P. Compo, A Practical Guide to Wavelet Analysis, *Bulletin of the American Meteorological Society*, *79*, 61-78, 1998.

The Torrence and Compo paper is also online at:
`<http://paos.colorado.edu/research/wavelets/>`.

DataFolderDir

`DataFolderDir(`*`mode`*` [, `*`dfr`*` ])`

The DataFolderDir function returns a string containing a listing of some or all of the objects contained in the current data folder or in the data folder referenced by *dfr*.

Parameters

mode is a bitwise flag for each type of object. Use -1 for all types. Use a sum of the bit values for multiple types.

Desired Type	Bit Number	Bit Value
All		-1
Data folders	0	1
Waves	1	2
Numeric variables	2	4
String variables	3	8

dfr is a data folder reference.

Details

The returned string has the following format:

1. FOLDERS:*name,name,…*;<CR>
2. WAVES:*name,name,…*;<CR>
3. VARIABLES:*name,name,…*;<CR>
4. STRINGS:*name,name,…*;<CR>

Where <CR> represents the carriage return character.

Tip

This function is mostly useful during debugging, used in a **Print** command. For finding the contents of a data folder programmatically, it will be easier to use the functions **CountObjects** and **GetIndexedObjName**.

Examples

```
Print DataFolderDir(8+4)       // prints variables and strings
Print DataFolderDir(-1)        // prints all objects
```

See Also

Chapter II-8, **Data Folders**.

Setting Bit Parameters on page IV-12 for information about bit settings.

DataFolderExists

`DataFolderExists(`*`dataFolderNameStr`*`)`

The DataFolderExists function returns the truth that the specified data folder exists.

dataFolderNameStr can bea a full path or partial path relative to the current data folder.

If *dataFolderNameStr* is empty (“”), DataFolderExists returns 1 because, for historical reasons, an empty data folder path is taken to refer to the current data folder.

See Also

Chapter II-8, **Data Folders**.

DataFolderList

`DataFolderList(`*`matchStr`*`, `*`separatorStr`*` [, `*`dfr`*` ] )`

The DataFolderList function returns a string containing a list of data folder names selected based on the *matchStr* parameter. The data folders listed are all within the current data folder or the folder specified by *dfr*.

The DataFolderList function was added in Igor Pro 9.00.

Details

For a data folder name to appear in the output string, it must match *matchStr*. *separatorStr* is appended to each data folder name as the output string is generated.

The name of each data folder is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all data folder names.
"xyz"	Matches data folder name xyz only.
"*xyz"	Matches data folder names which end with xyz.
"xyz*"	Matches data folder names which begin with xyz.
"*xyz*"	Matches data folder names which contain xyz.
"abc*xyz"	Matches data folder names which begin with abc and end with xyz.

matchStr may begin with the "!" character to return items that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches data folder names which do not end with xyz.

The "!" character is considered to be a normal character if it appears anywhere else.

dfr is an optional data folder reference: a data folder name, an absolute or relative data folder path, or a reference returned by, for example, **GetDataFolderDFR**.

The returned list contains data folder names only, without data folder paths. Use **GetDataFolder** to get the data folder path prefix.

Liberal data folder names are quoted if necessary; see **Liberal Object Names** on page III-501.

Examples

```
NewDataFolder/O aSubDataFolder
NewDataFolder/O 'Another quoted subfolder'

// Print the list of data folders in the current data folder whose names begin with a or A
Print DataFolderList("a*",";")
  aSubDataFolder;'Another quoted subfolder';

// Print the list of all data folders in the root:Packages data folder.
Print DataFolderList("*",";", root:Packages)
  WM_MedianXY;WM_WaveSelectorList;WindowCoordinates;
```

See Also

Data Folders on page II-107, **GetDataFolderDFR**, **GetIndexedObjName**, **StringList**, **StringFromList**, **VariableList**, **WaveList**, **PossiblyQuoteName**

DataFolderRefChanges

DataFolderRefChanges(*dfr*, *changeType*)

The DataFolderRefChanges function returns the number of changes to the data folder specified by the data folder reference *dfr*.

Use DataFolderRefChanges to decide when to update something that depends on the data folder's state by comparing the current number of changes to the change count when your previous update was done. DataFolderRefChanges is not thread-safe so this must be done in the main thread.

Changes to the data folder and its contents are tracked by incrementing a count specific to kind of change. The *changeType* parameter select one of those counts.

The DataFolderRefChanges function was added in Igor Pro 9.00.

Parameters

dfr is a data folder reference.

changeType is one of the following values:

0:	Changes to any to the data folder's waves, numeric variables, strings, or child data folders.
1:	Changes to the data folder's waves.
2:	Changes to the data folder's numeric variables.
3:	Changes to the data folder's string variables.
4:	Changes to the data folder's child data folders.

Details

The definition of a "change" depends on the object.

For data folders, a "change" is when a child data folder is created, killed, or renamed.

For waves, a "change" is when a wave is created, killed, renamed, locked, or modified.

For string or numeric values, a "change" is when they are created, killed, renamed, or modified.

The change counts are reset to 0 when the experiment is reopened.

Examples

```
Variable oldWaveChanges = DataFolderRefChanges(GetDataFolderDFR(),1)
Make/O/N=10 aWave = p
Variable newWaveChanges = DataFolderRefChanges(GetDataFolderDFR(),1)
Variable dif = newWaveChanges - oldWaveChanges
Print dif        // Prints 2: 1 for Make, 1 for aWave=p
```

See Also

Data Folders on page II-107, **Data Folder References** on page IV-78, **Built-in DFREF Functions** on page IV-81.

DataFolderRefsEqual

DataFolderRefsEqual(*dfr1*, *dfr2*)

The DataFolderRefsEqual function returns the truth the two data folder references are the same.

See Also

Data Folders on page II-107, **Data Folder References** on page IV-78, **Built-in DFREF Functions** on page IV-81.

DataFolderRefStatus

DataFolderRefStatus(*dfr*)

The DataFolderRefStatus function returns the status of a data folder reference.

Details

DataFolderRefStatus returns zero if the data folder reference is invalid or non-zero if it is valid.

DataFolderRefStatus returns a bitwise result with bit 0 indicating if the reference is valid and bit 1 indicating if the reference data folder is free. Therefore the returned values are:

0:	The data folder reference is invalid.
1:	The data folder reference refers to a regular global data folder.
3:	The data folder reference refers to a free data folder.

A data folder reference is invalid if it was never assigned a value or if it is assigned an invalid value. For example:

```
DFREF dfr                                // dfr is invalid
DFREF dfr = root:                        // dfr is valid
DFREF dfr = root:NonExistentDataFolder   // dfr is invalid
DFREF dfr = root:ExistingDataFolder      // dfr is valid
KillDataFolder dfr                       // dfr is invalid
```

You should use DataFolderRefStatus to test any DFREF variables that might not be valid, such as after assigning a reference when you are not sure that the referenced data folder exists. For historical reasons, if you use an invalid DFREF variable it will often act like root.

See Also

Data Folders on page II-107, **Data Folder References** on page IV-78, **Built-in DFREF Functions** on page IV-81.

dateToJulian

dateToJulian(*year*, *month*, *day*)

The dateToJulian function returns the Julian day number for the specified date. The Julian day starts at noon. Use negative number for BC years and positive numbers for AD years. To exclude any ambiguity, there is no year zero in this calendar. For general orientation, Julian day 2450000 corresponds to October 9, 1995.

See Also

The **JulianToDate** function.

For more information about the Julian calendar see:
<http://www.tondering.dk/claus/calendar.html>.

date

date()

The date function returns a string containing the current date.

Formatting of dates depends on your operating system and on your preferences entered in the Date & Time control panel (*Macintosh*) or the Regional Settings control panel (*Windows*).

Examples

```
Print date()        // Prints Mon, Mar 15, 1993
```

See Also

The **Secs2Date**, **Secs2Time**, and **time** functions.

date2secs

date2secs(*year*, *month*, *day*)

The date2secs function returns the number of seconds from midnight on 1/1/1904 to the specified date.

The month and day parameters are one-based, so these series start at one.

Date2Secs is limited to the range -32768-01-01 to 32767-12-31. For dates outside that range, it returns NaN. It also returns NaN if the year is 0 because Igor uses the Gregorian calendar in which there is no year 0.

If *year*, *month*, and *day* are all -1 then date2secs returns the offset in seconds from the local time to the UTC (Universal Time Coordinate) time.

Examples

```
Print Secs2Date(date2secs(1993,3,15),1)        // Ides of March, 1993
```

Prints the following, depending on your system's date settings, in the history area:

```
  Monday, March 15, 1993
```

This next example sets the X scaling of a wave to 1 day per point, starting January 1, 1993:

```
Make/N=125 myData = 100 + gnoise(50)
SetScale/P x,date2secs(1993,1,1),24*60*60,"dat",myData
Display myData;ModifyGraph mode=5
```

See Also

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-85.

The **Secs2Date**, **Secs2Time**, and **time** functions.

DateTime

DateTime

The DateTime function returns number of seconds from 1/1/1904 to current local date and time.

To get the UTC date and time, subtract Date2Secs(-1,-1,-1) from the value returned by DateTime.

Unlike most Igor functions, DateTime is used without parentheses.

Examples

```
Variable localNow = DateTime
```

See Also

The **Secs2Date**, **Secs2Time** and **time** functions.

dawson

dawson(*x*)

The dawson function returns the value of the Dawson integral:

$$F(x) = \exp\left(-x^2\right)\int_0^x \exp\left(t^2\right)dt.$$

If x is real, dawson returns a real result. If x is complex, dawson returns a complex result.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 298 pp., Dover, New York, 1972.

The code used to implement the Dawson integral was written by Steven G. Johnson of MIT. See http://ab-initio.mit.edu/Faddeva

Debugger

Debugger

The Debugger operation breaks into the debugger if it is enabled.

See Also

The Debugger on page IV-212 and the **DebuggerOptions** operation.

DebuggerOptions

DebuggerOptions [**enable=*en*, debugOnAbort=*doa*, debugOnError=*doe*, NVAR_SVAR_WAVE_Checking=*nvwc***]

The DebuggerOptions operation programmatically changes the user-level debugger settings. These are the same three settings that are available in the Procedure menu (and the debugger source pane contextual menu)

Parameters

All parameters are optional. If none are specified, no action is taken, but the output variables are still set.

enable=*en* Turns the debugger on (*en*=1) or off (*en*=0).

If the debugger is disabled then the other settings are cleared even if other settings are on.

debugOnAbort=*doa* Turns Debugging On Abort on or off.

doa=0:	Disables Debugging On Abort.
doa=1:	Enables Debugging On Abort and also enables the debugger (implies enable=1).

The Debug on Abort feature was added in Igor Pro 9.00. See **Debugging on Abort** on page IV-214 for details.

debugOnError=*doe* Turns Debugging On Error on or off.

doe=0:	Disables Debugging On Error.
doe=1:	Enables Debugging On Error and also enables the debugger (implies enable=1).

See **Debugging on Error** on page IV-213 for details.

NVAR_SVAR_WAVE_Checking=*nvwc*

Turns NVAR, SVAR, and WAVE checking on or off.

nvwc=0:	Disables “NVAR SVAR WAVE Checking”. See **Accessing Global Variables and Waves** on page IV-65 for more details.
nvwc=1:	Enables this checking and also enables the debugger (implies enable=1).

Output Variables

DebuggerOptions sets the following variables to indicate the Debugger settings that are in effect *after* the command is executed. A value of zero means the setting is off, nonzero means the setting is on.

```
V_enable, V_debugOnError, V_debugOnAbort, V_NVAR_SVAR_WAVE_Checking
```

See Also

The Debugger on page IV-212 and the **Debugger** operation.

default

default:

The default flow control keyword is used in switch and strswitch statements. When none of the case labels in the switch or strswitch match the evaluation expression, execution will continue with code following the default label, if it is present.

See Also

Switch Statements on page IV-43.

DefaultFont

DefaultFont [**/U**] "***fontName***"

The DefaultFont operation sets the default font to be used in graphs for axis labels, tick mark labels and annotations, and in page layouts for annotations.

Parameters

"fontName" should be a font name, optionally in quotes. The quotes are not required if the font name is one word.

Flags

/U	Updates existing graphs and page layouts immediately to use the new default font.

DefaultGUIControls

DefaultGUIControls [**/Mac/W=*winName*/Win**] [***appearance***]

The DefaultGUIControls operation changes the appearance of user-defined controls.

Note: The recommended way to change the appearance of user-defined controls is to use the Miscellaneous Settings dialog's Native GUI Appearance for Controls checkbox in the Compatibility tab, which is equivalent to `DefaultGUIControls native` when checked, and to `DefaultGUIControls os9` when unchecked.

Use `DefaultGUIControls/W=`*`winName`* to override that setting for individual windows.

Parameters

appearance may be one of the following:

native	Creates standard-looking controls for the current computer platform. This is the default value.
os9	Igor Pro 5 appearance (quasi-Macintosh OS 9 controls that look the same on Macintosh and Windows).
default	Inherits the window appearance from either a parent window or the experiment-wide default (only valid with /W).

Flags

/Mac	Changes the appearance of controls only on Macintosh, and it affects the experiment whenever it is used on Macintosh.
/W=*winName*	Affects the named window or subwindow. When omitted, sets an experiment-wide default. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.
/Win	Changes the appearance of controls only on Windows, and it affects the experiment whenever it is used on Windows.

Details

If *appearance* is not specified, nothing is changed. The current value for appearance is returned in S_value.

If *appearance* is specified the previous appearance value for the window- or experiment-wide default is returned in S_value.

With /W, the control appearance applies only to the specified window (Graph or Panel). If it is not used, then the settings are global to experiments on the current computer. **Tip**: Use `/W=#` to refer to the current active subwindow.

The /Mac and /Win flags specify the affected computer platform. If the current platform other than specified, then the settings are not used, but (if native or OS9) are remembered for use in window recreation macros or experiment recreation. This means you can create an experiment that with different appearances depending on the current platform.

If neither /Mac nor /Win are used, it is implied by the current platform. To set native appearance on both platforms, use two commands:

```
DefaultGUIControls/W=Panel0/Mac native
DefaultGUIControls/W=Panel0/Win native
```

Note: The setting for DefaultGUIControls without /W is not stored in the experiment file; it is a user preference set by the Miscellaneous Settings dialog's Native GUI Appearance for Controls checkbox in the Compatibility tab. If you use DefaultGUIControls native or DefaultGUIControls os9 commands, the checkbox will not show the current state of the experiment-wide setting. Clicking Save Settings in the Miscellaneous Settings dialog will overwrite the DefaultGUIControls setting (but not the per-window settings).

In addition to the experiment-wide appearance setting and the window-specific appearance setting, an individual control's appearance can be set with the appropriate control command's appearance keyword (or a ModifyControl appearance keyword). A control-specific appearance setting overrides a window-specific appearance, which in turn overrides the experiment-wide appearance setting.

Although meant to be used before controls are created, calling DefaultGUIControls will update all open windows.

DefaultGUIControls does not change control fonts or font sizes, which means you can create controls that look "native-ish" without having to readjust their positions to avoid avoid shifting or overlap. However, the smooth font rendering that the Native GUI uses on Macintosh does change the length of text slightly, so some shifting will occur that affects mostly controls that were aligned on their right sides.

The native appearance affects the way that controls are drawn in **TabControl** and **GroupBox** controls.

TabControl Background Details

Unlike the os9 appearance which draws only an outline to define the tab region (leaving the center alone) the native tab appearance fills the tab region. Fortunately, TabControls are drawn before all other kinds of controls which allows enclosed controls to be drawn on top of a tab control regardless of the order in which the buttons are defined in the window recreation macro.

However the drawing order of native TabControls does matter: the top-most TabControls draws over other TabControls. (The top-most TabControl is listed last in the window recreation macro.) The os9 appearance allows a smaller (nested) TabControl to be underneath the later (enclosing) TabControl because tabs normally aren't filled. Converting these tabs to native appearance will cause nested tab to be hidden.

To fix the drawing order problem in an existing panel, turn on the drawing tools, select the arrow tool, right-click the enclosing TabControl, and choose Send to Back to correct this situation. If the TabControl itself is inside another TabControl, select that enclosing TabControl and also choose Send to Back, etc.

To fix the window recreation macro or function that created the panel, arrange the enclosing TabControl commands to execute before the commands that create the enclosed TabControls.

A natively-drawn TabControl draws any drawing objects that are entirely enclosed by the tab region so that it behaves the same as an os9 unfilled TabControl with drawing objects inside.

Groupbox Control Background Details

GroupBox controls, unlike TabControls, are not drawn before all other controls, so the drawing order always matters if the GroupBox specifies a background (fill) color and it contains other controls.

You may find that enabling native appearance hides some controls inside the GroupBox. They are probably underneath (before) the GroupBox in the drawing order.

To fix this in an existing panel, turn on the drawing tools, right-click on the GroupBox and choose Send to Back. To fix the window recreation macro or function that created the panel, arrange the GroupBox commands to execute before the commands that create the enclosed controls.

A natively-drawn GroupBox draws any drawing objects that are entirely enclosed by the box; an os9 filled GroupBox does not.

See Also

The **DefaultGUIFont**, **ModifyControl**, **Button**, **GroupBox**, and **TabControl** operations.

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

DefaultGUIFont

`DefaultGUIFont` `[`**`/W=`*`winName`* `/Mac/Win`**`]` ***`group`* `= {`*`fNameStr`*`,`*`fSize`*`,`*`fStyle`*`}`** `[,…]`

The DefaultGUIFont operation changes the default font for user-defined controls and other Graphical User Interface elements.

Parameters

fNameStr is the name of a font, *fSize* is the font size, and *fStyle* is a bitwise parameter with each bit controlling one aspect of the font style. See **Button** for details about these parameters.

group may be one of the following:

all	All controls
button	Button and default CustomControl
checkbox	CheckBox controls
tabcontrol	TabControl controls
popup	Affects the icon (not the title) of a PopupMenu control. The text in the popped state is set by the system and can not be changed. The title of a PopupMenu is affected by the all group but the icon text is not.
panel	Draw text in a panel.
graph	Overlay graphs. Size is used only if `ModifyGraph gfSize= -1`; style is not used.
table	Overlay tables.

Flags

/Mac	Changes control fonts only on Macintosh, and it affects the experiment whenever it is used on Macintosh.
/W=*winName*	Affects the named window or subwindow. When omitted, sets an experiment-wide default. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.
/Win	Changes control fonts only on Windows, and it affects the experiment whenever it is used on Windows.

Details

Although designed to be used before controls are created, calling DefaultGUIFont will update all affected windows with controls. This makes it easy to experiment with fonts. Keep in mind that fonts can cause compatibility problems when moving between machines or platforms.

The /Mac and /Win flags indicate the platform on which the fonts are to be used. If the current platform is not the one specified then the settings are not used but are remembered for use in window recreation macros or experiment recreation. This allows a user to create an experiment that will use different fonts depending on the current platform.

If the /W flag is used then the font settings apply only to the specified window (Graph or Panel.) If the /W flag is not used, then the settings are global to the experiment. Tip: Use `/W=#` to refer to the current active subwindow.

fNameStr may be an empty string (`""`) to clear a group. Setting the font name to `"_IgorSmall"`, `"_IgorMedium"`, or `"_IgorLarge"` will use Igor's own defaults. The standard defaults for controls are the equivalent to setting all to `"_IgorSmall"`, tabcontrol to `"_IgorMedium"`, and button to `"_IgorLarge"`. Use a *fSize* of zero to also get the standard default for size. On Windows, the three default fonts and sizes are all the same.

Although designed to be used before controls are created, calling DefaultGUIFont will update all affected windows with controls. This makes it easy to experiment with fonts. Keep in mind that fonts can cause compatibility problems when moving between machines or platforms.

To read back settings, use `DefaultGUIFont [/W=winName/Mac/Win/OVR] group` to return the current font name in S_name, the size in V_value, and the style in V_flag. With /OVR or if /Mac or /Win is not current, it returns only override values. Otherwise, values include Igor built-in defaults. If S_name is zero length, values are not defined.

Default Fonts and Sizes

The standard defaults for controls is the equivalent to setting all to `"_IgorSmall"`, tabcontrol to `"_IgorMedium"`, and button to `"_IgorLarge"`. Use a *fSize* of zero to also get the standard default for size. On Windows, the three default fonts and sizes are all the same.

	Macintosh		Windows	
Control	**Font**	**Font Size**	**Font**	**Font Size**
Button	Lucida Grande	13	MS Shell Dlg*	12
Checkbox	Geneva	9	MS Shell Dlg	12
GroupBox	Geneva	9	MS Shell Dlg	12
ListBox	Geneva	9	MS Shell Dlg	12
PopupMenu†	Geneva	9	MS Shell Dlg	12
SetVariable	Geneva	9	MS Shell Dlg	12
Slider	Geneva	9	MS Shell Dlg	12
TabControl	Geneva	12	MS Shell Dlg	12
TitleBox	Geneva	9	MS Shell Dlg	12
ValDisplay	Geneva	9	MS Shell Dlg	12

* MS Shell Dlg is a "virtual font name" which maps to Tahoma on Windows XP, to MS Sans Serif on Windows 7, and to Segoe UI on Windows 8 and Windows 10.

† On Macintosh, the PopupMenu font is Geneva 9 for the title and Lucida Grande 12 for the popup menu itself. On Windows, both fonts are MS Shell Dlg 12.

Examples

```
DefaultGUIFont/Mac all={"Zapf Chancery",12,0},panel={"geneva",12,3}
DefaultGUIFont/Win all={"Century Gothic",12,0},panel={"arial",12,3}
NewPanel
Button b0
DrawText 40,43,"Some text"
```

See Also

The **DefaultGUIControls** operation. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Window Position Coordinates on page III-455 and **Points Versus Pixels** on page III-455 for explanations of how font sizes in panels are interpreted for various screen resolutions.

Demos

Choose File→Example Experiments→Feature Demos 2→All Controls Demo.

DefaultTab

`#pragma DefaultTab={`*`mode`*`,`*`widthInPoints`*`,`*`widthInSpaces`*`}`

The DefaultTab pragma allows you to specify default tab settings by entering a pragma statement in a procedure file. Specifying tab widths in spaces rather than points provides better results if the procedure window font or font size is changed.

The DefaultTab pragma was added in Igor Pro 9.00. It is ignored by earlier versions of Igor.

See **Procedure Window Default Tabs** on page III-405 for details.

DefaultTextEncoding

DefaultTextEncoding [encoding=*textEncoding*, overrideDefault=*override*]

The DefaultTextEncoding operation programmatically changes the default text encoding and experiment text encoding override settings. These settings, which are discussed under **The Default Text Encoding** on page III-465, are also accessible via the Misc→Text Encoding→Default Text Encoding menu.

DefaultTextEncoding is rarely needed because typically you will change the default text encoding manually using the menu, if at all.

The DefaultTextEncoding operation was added in Igor Pro 7.00.

Parameters

All parameters are optional. If none are specified, no action is taken, but the output variables are still set.

encoding=*textEncoding*

textEncoding specifies the new default text encoding. See **Text Encoding Names and Codes** on page III-490 for a list of codes.

Pass 0 to set the default text encoding to the equivalent of selecting "Western" from the Default Text Encoding submenu.

The value 255, corresponding to the binary text encoding type, is treated as an invalid value for the *textEncoding* parameter.

overrideDefault=*override*

Turns overriding of the experiment's text encoding off or on.

0: Turns override off

1: Turns override on

Details

The default text encoding affects Igor’s behavior when opening a file whose text encoding is unknown. See **The Default Text Encoding** on page III-465 for details.

The experiment text encoding affects how files are loaded during experiment loading only. The override setting allows you to override the experiment text encoding stored in the experiment file. Normally you will not need to do this. See **The Default Text Encoding** on page III-465 for further discussion.

You may occasionally find it necessary to change the default text encoding because an Igor operation lacks a /ENCG flag that allows you to specify the text encoding and instead uses the current default text encoding. In such cases it is a good idea to save the original default text encoding, change it as necessary, and then change it back to the original text encoding. The example below demonstrates this technique.

Output Variables

DefaultTextEncoding sets the following output variables to indicate the settings that are in effect after the command executes:

`V_defaultTextEncoding`	A text encoding code.
`V_overrideExperiment`	A value of zero means the setting is off, nonzero means the setting is on.

Example

```
Function DemoDefaultTextEncoding()
   // Store the original default text encoding
   DefaultTextEncoding
   Variable originalTextEncoding = V_defaultTextEncoding

   // Set new default text encoding
   DefaultTextEncoding encoding = 3     // 3= Windows-1252

   [ Do something that depends on the default text encoding ]

   // Restore the original default text encoding
   DefaultTextEncoding encoding = originalTextEncoding
End
```

See Also

The Default Text Encoding on page III-465, **Text Encoding Names and Codes** on page III-490

defined

`defined(`*`symbol`*`)`

The defined function returns 1 if the symbol is defined 0 if the symbol is not defined.

symbol is a symbol possibly created by a #define statement or by SetIgorOption poundDefine=*symbol*.

symbol is a name, not a string. However you can use $ to convert a string expression to a name.

Details

The defined function can be used in three ways:

Outside of a procedure using a #if statement

Inside a procedure using a #if statement

Inside a procedure using an if statement

For example:

```
#define DEBUG

#if defined(DEBUG)                  // Outside of a function with #if
   Constant kSomeConstant = 100
#else
   Constant kSomeConstant = 50
#endif

Function Test1()                    // Inside a function with #if
   #if defined(DEBUG)
      Print "Debugging"
   #else
      Print "Not debugging"
   #endif
End

Function Test1()                    // Inside a function with if
   if (defined(DEBUG))
      Print "Debugging"
   else
      Print "Not debugging"
   endif
End
```

In these examples, we could have just as well used #ifdef instead of the defined function. For logical combinations of conditions however, only defined will do:

```
#if (defined(SYMBOL1) && defined(SYMBOL2)
   . . .
#endif
```

When used in a procedure window, defined(symbol) returns 1 if symbol is defined at the time the line is compiled. In a given procedure file, only the following symbols are visible:

Symbols defined earlier in that procedure file *

Symbols defined in the built-in procedure window †

Predefined symbols (see **Predefined Global Symbols** on page IV-110)

Symbols defined by **SetIgorOption** poundDefine=*symbol*

* When used in the body of a procedure, as opposed to outside of a procedure, a symbol defined anywhere in a given procedure window is visible. However, to avoid depending on this confusing exception, you should define all symbols before they are referenced in a procedure file.

† Symbols defined in the built-in procedure window are not available to independent modules.

When the defined function is used from the command line, only symbols defined in the built-in procedure window, predefined symbols, and symbols defined using SetIgorOption are visible.

See Also

#define, **Conditional Compilation** on page IV-108, **Predefined Global Symbols** on page IV-110

DefineGuide

`DefineGuide [/W= winName] newGuideName = {[guideName1, val [, guideName2]]} [,…]`

The DefineGuide operation creates or overwrites a user-defined guide line in the target or named window or subwindow. Guide lines help with the positioning of subwindows in a host window.

Parameters

newGuideName is the name for the newly created guide. When it is the name of an existing user-defined guide, the guide will be moved to the new position.

guideName1, *guideName2*, etc., must be the names of existing guides.

The meaning of *val* depends on the form of the command syntax. When using only one guide name, *val* is an absolute distance offset from to the guide. The directionality of *val* is to the right or below the guide for positive values. The units of measure are points except in panels where they are in **Control Panel Units**. When using two guide names, *val* is the fractional distance between the two guides.

Flags

/W=*winName*	Defines guides in the named window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The names for the built-in guides are as defined in the following table:

	Left	**Right**	**Top**	**Bottom**
Host Window Frame	FL	FR	FT	FB
Host Graph Rectangle	GL	GR	GT	GB
Inner Graph Plot Rectangle	PL	PR	PT	PB
Layout Margin Rectangle	ML	MR	MT	MB

The frame guides apply to all window and subwindow types. The graph rectangle and plot rectangle guide types apply only to graph windows and subwindows. The layout margin rectangle guide types only apply to layout windows.

To delete a guide use `guideName={}`.

See Also

The **Display**, **Edit**, **NewPanel**, **NewImage**, and **NewWaterfall** operations.

The **GuideInfo** function.

DelayUpdate

`DelayUpdate`

The DelayUpdate operation delays the updating of graphs and tables while executing a macro.

Details

Use DelayUpdate at the end of a line in a macro if you want the next line in the macro to run before graphs or tables are updated.

This has no effect in user-defined functions. During execution of a user-defined function, windows update only when you explicitly call the **DoUpdate** operation.

See Also

The **DoUpdate**, **PauseUpdate**, and **ResumeUpdate** operations.

DeleteAnnotations

DeleteAnnotations [*flags*] [tagOffscreen, tagTraceHidden, invisible,
offsetOffscreen, tooSmall[=*size*]]

The DeleteAnnotations operation lists, in the S_name output variable, and optionally deletes annotations that are hidden for reasons specified by the flags and keywords.

The operation affects the window or subwindow specified by the /W flag or, if /W is omitted, the active window or subwindow.

Do not use DeleteAnnotations to progammatically delete a specific, single annotation. Instead use:

```
TextBox/W=winName/K/N=annotationName
```

The /LIST flag limits the action to only listing, instead of deleting, the annotations.

The DeleteAnnotations operation was added in Igor Pro 7.00.

Keywords

The keywords identify annotations based on the reasons for their being hidden:

invisible — Deletes or lists annotations hidden with /V=0.

offsetOffscreen — Deletes or lists annotations that are offscreen, usually because of excessive /X and /Y offsets.

tagOffscreen — Deletes or lists tags hidden because they are attached to trace points that are offscreen. This affects trace tags, axis tags, and image tags if their "if offscreen" setting, as set in the Position tab of the Modify Annotation dialog, is set to "hide the tag".

tagTraceHidden — Deletes or lists tags hidden because the tagged trace is hidden.

tooSmall [=*size*] — Deletes or lists annotations whose height or width is *size* points or smaller. *size* is expressed in points and defaults to 8. This is useful for deleting annotations that are too small to see or to double-click.

Flags

/A — All annotations, whether hidden or not, are listed or deleted. All keywords are ignored.

/LIST — Specifies that annotations identified by the other parameters are to be listed in the S_name output variable but not deleted.

/W=*winName* — Annotations in the named window or subwindow are considered. When omitted, annotations in the active window or subwindow are considered.

When identifying a subwindow with winName, see Subwindow Syntax for details on forming the window hierarchy.

Output Variables

`S_name` — A semicolon-separated list of the annotations that match the criteria set by the keywords and flags.

`V_flag` — Set to the number of annotations deleted or listed.

Examples

```
Function DeleteAnnotationsInWin(win)
   String win          // Specifies a top-level window or a subwindow

   // Handle specified top-level window or subwindow
   DeleteAnnotations/W=$win/A
   Variable numDeleted = V_Flag

   // Now handle subwindows, if any
   String children = ChildWindowList(win)
   Variable n = ItemsInList(children)
   Variable i
```

```
	for(i=0; i<n; i+=1)
		String child = StringFromList(i, children)
		numDeleted += DeleteAnnotationsInWin(child) // Recurse
	endfor

	return numDeleted
End
```

See Also

TextBox, **StringFromList**, **AnnotationList**

DeleteFile

DeleteFile [***flags***] [***fileNameStr***]

The DeleteFile operation deletes a file on disk.

Parameters

fileNameStr can be a full path to the file to be deleted (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*.

If Igor can not locate the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file to be deleted.

If you use a full or partial path for either file, see **Path Separators** on page III-451 for details on forming the path.

Flags

/I	Interactive mode displays the Open File dialog even if *fileNameStr* is specified and the file exists.
/M=*messageStr*	Specifies the prompt message for the Open File dialog.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/Z[=z]	Prevents procedure execution from aborting if it attempts to delete a file that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort. /Z=0: Same as no /Z. /Z=1: Deletes a file only if it exists. /Z alone has the same effect as /Z=1. /Z=2: Deletes a file if it exists or displays a dialog if it does not exist.

Variables

The DeleteFile operation returns information in the following variables:

`V_flag`	Set to zero if the file was deleted, to -1 if the user cancelled the Open File dialog, and to some nonzero value if an error occurred, such as the specified file does not exist.
`S_path`	Stores the full path to the file that was deleted. If an error occurred or if the user cancelled, it is set to an empty string.

See Also

DeleteFolder, **MoveFile**, **CopyFile**, **NewPath**, and **Symbolic Paths** on page II-22.

DeleteFolder

```
DeleteFolder [flags] [folderNameStr]
```

The DeleteFolder operation deletes a disk folder and all of its contents.

Warning: *The DeleteFolder command destroys data!* The deleted folder and the contents are not moved to the Trash or Recycle Bin.

DeleteFolder will delete a folder only if permission is granted by the user. The default behavior is to display a dialog asking for permission. The user can alter this behavior via the Miscellaneous Settings dialog's Misc category.

If permission is denied, the folder will not be deleted and V_Flag will return 1088 (Command is disabled) or 1276 (You denied permission to delete a folder). Command execution will cease unless the /Z flag is specified.

Parameters

folderNameStr can be a full path to the folder to be deleted, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a folder within the folder associated with *pathName*.

If Igor can not determine the location of the folder from *folderNameStr* and *pathName*, it displays a Select Folder dialog allowing you to specify the folder to be deleted.

If /P=*pathName* is given, but *folderNameStr* is not, then the folder associated with *pathName* is deleted.

If you use a full or partial path for either folder, see **Path Separators** on page III-451 for details on forming the path.

Folder paths should not end with single Path Separators. See the **MoveFolder Details** section.

Flags

/I	Interactive mode displays a Select Folder dialog even if *folderNameStr* is specified and the folder exists.
/M=*messageStr*	Specifies the prompt message for the Select Folder dialog.
/P=*pathName*	Specifies the folder to look in for the folder. *pathName* is the name of an existing symbolic path.
/Z[=z]	Prevents procedure execution from aborting if it attempts to delete a folder that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

/Z=0:	Same as no /Z.
/Z=1:	Deletes a folder only if it exists. /Z alone has the same effect as /Z=1.
/Z=2:	Deletes a folder if it exists or displays a dialog if it does not exist.

Variables

The DeleteFolder operation returns information in the following variables:

`V_flag`	Set to zero if the folder was deleted, to -1 if the user cancelled the Select Folder dialog, and to some nonzero value if an error occurred, such as the specified folder does not exist.
S_path	Stores the full path to the folder that was deleted, with a trailing colon. If an error occurred or if the user cancelled, it is set to an empty string.

Details

You can use only /P=*pathName* (without *folderNameStr*) to specify the source folder to be deleted.

Folder paths should not end with single Path Separators. See the **Details** section for **MoveFolder**.

See Also

The **DeleteFile**, **MoveFolder**, **CopyFolder**, **NewPath**, and **IndexedDir** operations. **Symbolic Paths** on page II-22.

DeletePoints

DeletePoints [**/M=*dim***] ***startElement*, *numElements*, *waveName***
[, ***waveName***]…

The DeletePoints operation deletes *numElements* elements from the named waves starting from element *startElement*.

Flags

/M=*dim* — *dim* specifies the dimension from which elements are to be deleted. Values are:

0:	Rows.
1:	Columns.
2:	Layers.
3:	Chunks.

If /M is omitted, DeletePoints deletes from the rows dimension.

Details

A wave may have any number of points, including zero. Removing all elements from any dimension removes all points from the wave, leaving a 1D wave with zero points.

Except for the case of removing all elements, DeletePoints does not change the dimensionality of a wave. Use **Redimension** for that.

See Also

The **Redimension** operation.

deltax

deltax(*waveName*)

The deltax function returns the named wave's dx value. deltax works with 1D waves only.

Details

This is equal to the difference of the X value of point 1 minus the X value of point 0.

See Also

The **leftx** and **rightx** functions.

When working with multidimensional waves, use the **DimDelta** function.

For an explanation of waves and wave scaling, see **Changing Dimension and Data Scaling** on page II-68.

DFREF

DFREF *localName* [= *path* or *dfr*], [*localName1* [= *path* or *dfr*]]

DFREF is used to define a local data folder reference variable or input parameter in a user-defined function.

The syntax of the DFREF is:

```
DFREF localName [= path or dfr ][, localName1 [= path or dfr ]]...
```

where *dfr* stands for "data folder reference". The optional assignment part is used only in the body of a function, not in a parameter declaration.

Unlike the **WAVE** reference, a DFREF in the body without the assignment part does not do any lookup. It simply creates a variable whose value is null.

Examples

```
Function Test(dfr)
    DFREF dfr

    Variable dfrStatus = DataFolderRefStatus(dfr)

    if (dfrStatus == 0)
        Print "Invalid data folder reference"
        return -1
    endif
```

```
   if (dfrStatus & 2)            // Bit 1 set means free data folder
      Print "Data folder reference refers to a free data folder"
   endif

   if (dfrStatus == 1)
      Print "Data folder reference refers a global data folder"
      DFREF dfSav = GetDataFolderDFR()
      Print GetDataFolder(1)     // Print data folder path
      SetDataFolder dfSav
   endif

   Make/O dfr:jack=sin(x/8)      // Make a wave in the referenced data folder

   return 0
End
```

See Also

For information on programming with data folder references, see **Data Folder References** on page IV-78.

Differentiate

Differentiate [***type flags***][***flags***] ***yWaveA*** [**/X =** ***xWaveA***]
   [**/D =** ***destWaveA***][**,** ***yWaveB*** [**/X =** ***xWaveB***][**/D =** ***destWaveB***][**,** …]]

The Differentiate operation calculates the 1D numerical derivative of a wave.

Differentiate is multi-dimension-aware in the sense that it computes a 1D differentiation along the dimension specified by the /DIM flag or along the rows dimension if you omit /DIM.

Complex waves have their real and imaginary components differentiated individually.

Flags

/DIM=*d*	Specifies the wave dimension along which to differentiate when *yWave* is multidimensional. *d*=-1: Treats entire wave as 1D (default). *d*=0: Differentiates along rows. *d*=1: Differentiates along columns. *d*=2: Differentiates along layers. *d*=3: Differentiates along rows. For example, for a 2D wave, /DIM=0 differentiates each row and /DIM=1 differentiates each column.
/EP=*e*	Controls end point handling. *e*=0: Replaces undefined points with an approximation (default). *e*=1: Deletes the point(s).
/METH=*m*	Sets the differentiation method. *m*=0: Central difference (default). *m*=1: Forward difference. *m*=2: Backward difference.
/P	Forces point scaling.

Type Flags (*used only in functions*)

Differentiate also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-73 and **WAVE Reference Type Flags** on page IV-74 for a complete list of type flags and further details.

For example, when the input (and output) waves are complex, the output wave will be complex. To get the Igor compiler to create a complex output wave reference, use the /C type flag with /D=destwave:

```
Make/O/C cInput=cmplx(sin(p/8), cos(p/8))
Make/O/C/N=0 cOutput
Differentiate/C cInput /D=cOutput
```

Wave Parameters

Note:	***All*** wave parameters must follow *yWave* in the command. All wave parameter flags and type flags must appear immediately after the operation name.
/D=*destWave*	Specifies the name of the wave to hold the differentiated data. It creates *destWave* if it does not already exist or overwrites it if it exists.
/X=*xWave*	Specifies the name of the corresponding X wave.

Details

If the optional /D = *destWave* flag is omitted, then the wave is differentiated in place overwriting the original data.

When using a method that deletes points (/EP=1) with a multidimensional wave, deletion is not done if no dimension is specified.

When using an X wave, the X wave must match the Y wave data type (excluding the complex type flag) and it must be 1D with the number points matching the size of the dimension being differentiated. X waves are not used with integer source waves.

`Differentiate/METH=1/EP=1` is the inverse of `Integrate/METH=2`, but `Integrate/METH=2` is the inverse of `Differentiate/METH=1/EP=1` only if the original first data point is added to the output wave.

Differentiate applied to an XY pair of waves does not check the ordering of the X values and doesn't care about it. However, it is usually the case that your X values should be monotonic. If your X values are not monotonic, you should be aware that the X values will be taken from your X wave in the order they are found, which will result in random X intervals for the X differences. It is usually best to sort the X and Y waves using **Sort**.

See Also

The **Integrate** operation.

digamma

`digamma(`*`x`*`)`

The digamma function returns the digamma, or psi function of *x*. This is the logarithmic derivative of the gamma function:

$$\Psi(z) \equiv \frac{d}{dz}\ln\left(\Gamma(z)\right) = \frac{\Gamma'(z)}{\Gamma(z)}.$$

In complex expressions, *x* is complex, and `digamma(x)` returns a complex value.

Limited testing indicates that the accuracy is approximately 1 part in 10^{16}, at least for moderately-sized values of *x*.

Dilogarithm

`Dilogarithm(`*`z`*`)`

Returns the Dilogarithm function for real or complex argument *z*. The dilogarithm is a special case of the polylogarithm defined by

$$Li_2(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^2}.$$

The dilogarithm function was added in Igor Pro 7.00.

See Also

zeta

Reference

Wood, D.C. (June 1992). "The Computation of Polylogarithms. Technical Report 15-92". Canterbury, UK: University of Kent Computing Laboratory.

The function based on an algorithm by Didier Clamond.

DimDelta

`DimDelta(`*`waveName`*`, `*`dimNumber`*`)`

The DimDelta function returns the scale factor delta of the given dimension.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers and 3 for chunks.

If *dimNumber*=0 this is identical to `deltax(`*`waveName`*`)`.

See Also

DimOffset, **DimSize**, **SetScale**, **WaveUnits**, **ScaleToIndex**

For an explanation of waves and wave scaling, see **Changing Dimension and Data Scaling** on page II-68.

DimOffset

`DimOffset(`*`waveName`*`, `*`dimNumber`*`)`

The DimOffset function returns the scaling offset of the given dimension.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers, and 3 for chunks.

If *dimNumber*=0 this is identical to `leftx(`*`waveName`*`)`.

See Also

DimDelta, **DimSize**, **SetScale**, **WaveUnits**, **ScaleToIndex**

For an explanation of waves and wave scaling, see **Changing Dimension and Data Scaling** on page II-68.

DimSize

`DimSize(`*`waveName`*`, `*`dimNumber`*`)`

The DimSize function returns the size of the given dimension.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers, and 3 for chunks.

For a 1D wave, `DimSize(`*`waveName`*`,0)` is identical to `numpnts(`*`waveName`*`)`.

See Also

DimDelta, **DimOffset**, **SetScale**, **WaveUnits**

Dir

`Dir` `[`***`dataFolderSpec`***`]`

The Dir operation returns a listing of all the objects in the specified data folder.

Parameters

If you omit *dataFolderSpec* then the current data folder is used.

If present, *dataFolderSpec* can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

Details

The format of the printed information is the same as the format used by the string function **DataFolderDir**. Igor programmers may find it more convenient to use **CountObjects** and **GetIndexedObjName**.

Usually it is easier to use the Data Browser (Data menu). However, **Dir** is useful when you want to copy a name into the command line or when you want to document the current state of the folder in the history.

See Also

Chapter II-8, **Data Folders**.

Display

Display [***flags***] [***waveName*** [, ***waveName***]…[**vs** ***xwaveName***]]
[**as** ***titleStr***]

The Display operation creates a new graph window or subwindow, and appends the named waves, if any. Waves are displayed as 1D traces.

By default, waves are plotted versus the left and bottom axes. Use the /L, /B, /R, and /T flags to plot the waves against other axes.

Parameters

Up to 100 *waveName*s may be specified, subject to the 2500 byte command line length limit. If no wave names are specified, a blank graph is created and the axis flags are ignored.

If you specify "vs *xwaveName*", the Y values of the named waves are plotted versus the Y values of *xwaveName*. If you don't specify "vs *xwaveName*", the Y values of each *waveName* are plotted versus its own X values.

If *xwaveName* is a text wave or the special keyword '_labels_', the resulting plot is a category plot. Each element of *waveName* is plotted by default in bars mode (`ModifyGraph mode=5`) against a category labeled with the text of the corresponding element of *xwaveName* or the text of the dimension labels of the first Y wave..

The Y waves for a category plot should have point scaling (see **Changing Dimension and Data Scaling** on page II-68); this is how category plots were intended to work. However, if all the Y waves have the same scaling, it will work correctly.

titleStr is a string expression containing the graph's title. If not specified, Igor will provide one which identifies the waves displayed in the graph.

Subsets of data, including individual rows or columns from a matrix, may be specified using **Subrange Display Syntax** on page II-321.

You can provide a custom name for a trace by appending /TN=traceName to the waveName specification. This is useful when displaying waves with the same name but from different data folders. See **User-defined Trace Names** on page IV-89 for more information.

Flags

/B[=*axisName*] Plots X coordinates versus the standard or named bottom axis.

/FG=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the frame guide to which the outer frame of the subwindow is attached inside the host window.

The standard frame guide names are `FL`, `FR`, `FT`, and `FB`, for the left, right, top, and bottom frame guides, respectively, or user-defined guide names as defined by the host. Use `*` to specify a default guide name.

Guides may override the numeric positioning set by /W.

/HIDE=*h* Hides (h = 1) or shows (h = 0, default) the window.

/HOST=*hcSpec* Embeds the new graph in the host window or subwindow specified by *hcSpec*.

When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/I Specifies that /W coordinates are in inches.

/K=*k* Specifies window behavior when the user attempts to close it.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.
k=3:	Hides the window.

If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.

/L[=*axisName*] Plots Y coordinates versus the standard or named left axis.

/M — Specifies that /W coordinates are in centimeters.

/N=*name* — Requests that the created graph have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen graph name.

/NCAT — In Igor Pro 6.37 or later, allows subsequent appending of a category trace to a numeric plot. See for details.

/PG=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the inner plot rectangle of the graph subwindow inside its host window.

The standard plot rectangle guide names are `PL`, `PR`, `PT`, and `PB`, for the left, right, top, and bottom plot rectangle guides, respectively, or user-defined guide names as defined by the host. Use `*` to specify a default guide name.

Guides may override the numeric positioning set by /W.

/R[=*axisName*] — Plots Y coordinates versus the standard or named right axis.

/T[=*axisName*] — Plots Y coordinates versus the standard or named top axis.

/TN=*traceName* — Allows you to provide a custom trace name for a trace. This is useful when displaying waves with the same name but from different data folders. See **User-defined Trace Names** on page IV-89 for details.

/W=(*left*,*top*,*right*,*bottom*)

Gives the graph a specific location and size on the screen. Coordinates for /W are in points unless /I or /M are specified before /W.

When used with the /HOST flag, the specified location coordinates of the sides can have one of two possible meanings:

1: When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.

2: When any value is greater than 1, coordinates are taken to be fixed locations measured in points, or **Control Panel Units** for control panel hosts, relative to the top left corner of the host frame.

When the subwindow position is fully specified using guides (using the /HOST, /FG, or /PG flags), the /W flag may still be used although it is not needed.

Details

If /N is not used, Display automatically assigns to the graph a name of the form “Graph*n*”, where *n* is some integer. In a function or macro, the assigned name is stored in the S_name string. This is the name you can use to refer to the graph from a procedure. Use the **RenameWindow** operation to rename the graph.

Examples

To make a contour plot, use:

```
Display; AppendMatrixContour waveName
```

or

```
Display; AppendXYZContour waveName
```

To display an image, use:

```
Display; AppendImage waveName
```

or

```
NewImage waveName
```

See Also

The **AppendToGraph** operation.

The operations **AppendImage**, **AppendMatrixContour**, **AppendXYZContour**, and **NewImage**. For more information on Category Plots, see Chapter II-14, **Category Plots**.

The operations **ModifyGraph**, **ModifyContour**, and **ModifyImage** for changing the characteristics of graphs.

The **DoWindow** operation for changing aspects of the graph window.

DisplayHelpTopic

`DisplayHelpTopic [/K=`*`k`*` /Z] `*`TopicString`*

The DisplayHelpTopic operation displays a help topic as if a help link had been clicked in an Igor help file.

Parameters

TopicString is string expression containing the topic. It may be in one of three forms: <topic name>, <subtopic name>, <topic name>[<subtopic name>]. These forms are illustrated by the examples.

Make sure that your topic string is specific to minimize the likelihood that Igor will find the topic in a help file other than the one you intended. To avoid this problem, it is best to use the <topic name>[<subtopic name>] form if possible.

Flags

/K=*k* Determines when the help file is closed.

k=0: Leaves the help file open indefinitely (default). Use this if the help topic may be of interest in any experiment.

k=1: If the found topic is in a closed help file, the help file closes with the current experiment. Use this if the help topic is tightly associated with the current experiment.

/Z Ignore errors. If /Z is used, DisplayHelpTopic sets V_flag to 0 if the help topic was found or to a nonzero error code if it was not found. V_flag is set only when /Z is used.

Details

DisplayHelpTopic first searches for the specified topic in the open help files. If the topic is not found, it then searches all help files in the Igor Pro folder and subfolders.

If the topic is still not found, it then searches all help files in the current experiment's home folder, but not in subfolders. This puts a help file that is specific to a particular experiment in the experiment's home folder.

If the topic is still not found and if DisplayHelpTopic was called from a procedure and if the procedure resides in a stand-alone file on disk (i.e., it is not in the built-in procedure window or in a packed procedure file), Igor then searches all help files in the procedure file's folder, but not in subfolders. This puts a help file that is specific to a particular set of procedures in the same folder as the procedure file.

If Igor finds the topic, it displays it. If Igor can not find the topic, it displays an error message, unless /Z is used.

Examples

```
// This example uses the topic only.
DisplayHelpTopic "Modifying Traces"

// This example uses the subtopic only.
DisplayHelpTopic "Markers"

// This example uses the topic[subtopic] form.
DisplayHelpTopic "Modifying Traces[Markers]"
```

See Also

Chapter II-1, **Getting Help** for information about Igor help files and formats.

DisplayProcedure

`DisplayProcedure [flags] [`*`functionOrMacroNameStr`*`]`

The DisplayProcedure operation displays the named function, macro or line by bringing the procedure window it is defined in to the front with the function, macro or line highlighted.

Parameters

functionOrMacroNameStr is a string expression containing the name of the function or macro to display. If you omit *functionOrMacroNameStr* then you must use /W or /L.

functionOrMacroNameStr may be a simple name or may include independent module and/or module name prefixes to display static functions.

If you use /L to display a particular line then you must omit *functionOrMacroNameStr*.

To display a procedure window without changing its scrolling or selection, use /W and omit *functionOrMacroNameStr*.

Flags

/B=*winTitleOrName* Brings up the procedure window just behind the window with this name or title.

/L=*lineNum* If /W is specified, *lineNum* is a zero-based line number in the specified window.

If /W is not specified, *lineNum* is a "global" line number. Each procedure window line has a unique global line number as if all of the procedure files were concatenated into one big file. The order of concatenation of files can change when procedures are recompiled.

If you use /L then you must omit *functionOrMacroNameStr*.

/W=*procWinTitle* Searches in the procedure window with this title.

procWinTitle is a name, not a string, so you construct /W like this:

```
/W=$"New Polar Graph.ipf"
```

If you omit /W, DisplayProcedure searches all open (nonindependent module) procedure windows.

Details

If a procedure window has syntax errors that prevent Igor from determining where functions and macros start and end, then DisplayProcedure may not be able to locate the procedure.

winTitleOrName is not a string; it is a name. To position the found procedure window behind a window whose title has a space in the name, use the $ operator as in the second example, below.

If *winTitleOrName* does not match any window, then the found procedure window is placed behind the top target window.

lineNum is a zero-based line number: 0 is the first line of the window. Because each line of a procedure window is a paragraph, line numbers and paragraph numbers are the same. You can use the Procedure→Info menu item to show a selection's starting and ending paragraph/line number.

procWinTitle is also a name. Use `/W=$"New Polar Graph.ipf"` to search for the function or macro in only that procedure file.

Don't specify both *functionOrMacroNameStr* and /L=*lineNum* as this is ambiguous and not allowed.

Advanced Details

If `SetIgorOption IndependentModuleDev=1`, *procWinTitle* can also be a title followed by a space and, in brackets, an independent module name. In such cases searches for the function or macro are in the specified procedure window and independent module. (See **Independent Modules** on page IV-238 for independent module details.)

For example, if any procedure file contains these statements:

```
#pragma IndependentModule=myIM
#include <Axis Utilities>
```

The command

```
DisplayProcedure/W=$"Axis Utilities.ipf [myIM]" "HVAxisList"
```

opens the procedure window that contains the `HVAxisList` function, which is in the Axis Utilities.ipf file and the independent module `myIM`. The command uses the `$""` syntax because space and bracket characters interfere with command parsing.

Similarly, if `SetIgorOption IndependentModuleDev=1` then *functionOrMacroNameStr* may also contain an independent module prefix followed by the # character. The preceding command can be rewritten as:

```
DisplayProcedure/W=$"Axis Utilities.ipf" "myIM#HVAxisList"
```

or more simply

```
DisplayProcedure "myIM#HVAxisList"
```

You can use the same syntax to display a static function in a non-independent module procedure file using a module name instead of (or in addition to) the independent module name.s

procWinTitle can also be just an independent module name in brackets to display all procedure windows that belong to the named independent module and define the specified function:

```
DisplayProcedure/W=$"[myIM]" "HVAxisList"
```

Examples

```
DisplayProcedure "Graph0"
DisplayProcedure/B=Panel0 "MyOwnUserFunction"
DisplayProcedure/W=Procedure         // Shows the main Procedure window
DisplayProcedure/W=Procedure/L=5     // Shows line 5 (the sixth line)
DisplayProcedure/W=$"Wave Lists.ipf"
DisplayProcedure "moduleName#myStaticFunctionName"
SetIgorOption IndependentModuleDev=1
DisplayProcedure "WMGP#GizmoBoxAxes#DrawAxis"
```

See Also

Independent Modules on page IV-238

HideProcedures, **DoWindow**

ProcedureText, **ProcedureVersion**, **ModifyProcedure**

MacroList, **FunctionList**

do-while

```
do
   <loop body>
while(<expression>)
```

A do-while loop executes *loop body* until *expression* evaluates as FALSE (zero) or until a break statement is executed.

See Also

Do-While Loop on page IV-45 and **break** for more usage details.

DoAlert

`DoAlert [/T=titleStr] alertType, promptStr`

The DoAlert operation displays an alert dialog and waits for user to click button.

Parameters

alertType=*t*	Controls the type of alert dialog:	
	t=0:	Dialog with an OK button.
	t=1:	Dialog with Yes button and No buttons.
	t=2:	Dialog with Yes, No, and Cancel buttons.
promptStr	Specifies the text that is displayed in the alert dialog.	

Flags

/T=*titleStr*	Changes the title of the dialog window from the default title.

Details

DoAlert sets the variable V_flag as follows:

1:	Yes clicked.
2:	No clicked.
3:	Cancel clicked.

See Also

The **Abort** operation.

DoIgorMenu

```
DoIgorMenu [/C /OVRD] MenuNameStr, MenuItemStr
```

The DoIgorMenu operation allows an Igor programmer to invoke Igor's built-in menu items. This is useful for bringing up Igor's built-in dialogs under program control.

Parameters

MenuNameStr	The name of an Igor menu or submenu, like "File", "Graph", or "Load Waves".
MenuItemStr	The text of an Igor menu item, like "Copy" (in the Edit menu) or "New Graph" (in the Windows menu) or "Load Igor Binary" (in the Load Waves submenu). If you include /C, *MenuItemStr* can be "".

Flags

Using both the /C and the /OVRD flag in one command is not permitted.

/C	Just Checking. The menu item is not invoked, but V_flag is set to 1 if the item was enabled or to 0 if it was not enabled. In Igor Pro 9.01 and later, if *MenuItemStr* is "", then V_Flag, V_isInvisible, and S_value pertain to the menu instead of a menu item.
/OVRD	Tells Igor to skip checks that it normally does before executing the menu command specified by *MenuNameStr* and *MenuItemStr*. You are responsible for ensuring that the menu command you are invoking is appropriate under conditions existing at runtime. The main use for the /OVRD flag is to allow an advanced programmer to invoke a menu command for a menu that is currently hidden when dealing with subwindows. For example, if you have a graph subwindow in a control panel which is in operate mode, the Graph menu is not visible in the menu bar. Normally the user could not invoke an item, such as Modify Trace Appearance. /OVRD allows you to invoke the menu command, but it is up to you to verify that it is appropriate. In the Modify Trace Appearance example, you should invoke the menu command only if the active window or subwindow is a graph that contains at least one trace. /OVRD was added in Igor Pro 7.00.

Details

All menu names and menu item text are in English to ensure that code developed for a localized version of Igor Pro will run on all versions. Note that no trailing "…" is used in *MenuItemStr*.

V_flag is set to 1 if the corresponding menu item was enabled, which usually means the menu item was successfully selected. Otherwise V_flag is 0. V_flag does not reflect the success or failure of the resulting dialog, if any.

V_isInvisible, added in Igor Pro 7, is set to 1 if the corresponding menu item was invisible. This is fairly rare - in most cases V_isInvisible will be set to 0. Invisible menu items are items like File→Adopt All which are hidden unless the user presses the shift key while summoning the menu. This is different from menus items hidden by HideIgorMenus for which V_isInvisible is set to 0.

S_value, added in Igor Pro 9.00, is set to the translated menu item text. This can be useful for always-enabled menu items that toggle between to states like Show Tools and Hide Tools.

If the menu item selection displays a dialog that generates a command, clicking the Do It button executes the command immediately without using the command line as if Execute/Z operation had been used. Clicking the To Cmd Line button appends the command to the command line rather than inserting the command at the front.

The DoIgorMenu operation will not attempt to select a menu during curve fitting or while a dynamic menu item's function is running. Doubtless there are other times during which using DoIgorMenu would be unwise.

The text of some items in the File menu changes depending on the type of the active window. In these cases you must pass generic text as the *MenuItemStr* parameter. Use “Save Window”, “Save Window As”, “Save Window Copy”, “Adopt Window”, and “Revert Window” instead of “Save Notebook” or “Save Procedure”, etc. Use “Page Setup” instead of “Page Setup For All Graphs”, etc. Use “Print” instead of “Print Graph”, etc.

The Edit→Insert File menu item was previously named Insert Text. For compatibility, you can specify either “Insert File” or “Insert Text” as *MenuItemStr* to invoke this item.

See Also

SetIgorMenuMode, **ShowIgorMenus**, **HideIgorMenus**, **Execute**

The SetIgorMenuModeProc.ipf WaveMetrics procedure file contains SetIgorMenuMode commands for every menu and menu item. You can load it using

```
#include <SetIgorMenuModeProc>
```

DoPrompt

`DoPrompt` `[`**`/HELP=`*`helpStr`***`]` ***`dialogTitleStr`***`,` ***`variable`*** `[,` ***`variable`***`]`…

The DoPrompt statement in a function invokes the simple input dialog. A DoPrompt specifies the title for the simple input dialog and which input variables are to be included in the dialog.

Flags

/HELP=*helpStr*	Sets the help topic or help text that appears when the dialog’s Help button is pressed. *helpStr* can be a help topic and subtopic such as is used by `DisplayHelpTopic/K=1` *`helpStr`*, or it can be text (255 characters max) that is displayed in a subdialog just as if `DoAlert 0,` *`helpStr`* had been called, or *helpStr* can be `""` to remove the Help button.

Parameters

variable is the name of a dialog input variable, which can be real or complex numeric local variable or local string variable, defined by a Prompt statement. You can specify as many as 10 variables.

dialogTitleStr is a string or string expression containing the text for the title of the simple input dialog.

Details

Prompt statements are required to define what variables are to be used and the text for any string expression to accompany or describe the input variable in the dialog. When a DoPrompt variable is missing a Prompt statement, you will get a compilation error. Pop-up string data can not be continued across multiple lines as can be done using Prompt in macros. See **Prompt** for further usage details.

Prompt statements for the input variables used by DoPrompt must come before the DoPrompt statement itself, otherwise, they may be used anywhere within the body of a function. The variables are not required to be input parameters for the function (as is the case for Prompt in macros) and they may be declared within the function body. DoPrompt can accept as many as 10 variables.

Functions can use multiple DoPrompt statements, and Prompt statements can be reused or redefined.

When the user clicks the Cancel button, any new input parameter values are not stored in the variables.

DoPrompt sets the variable V_flag as follows:

0:	Continue button clicked.
1:	Cancel button clicked.

See Also

The Simple Input Dialog on page IV-144, the **Prompt** keyword, and **DisplayHelpTopic**.

Double

`double` ***`localName`***

Declares a local 64-bit double-precision variable in a user-defined function or structure.

Double is another name for Variable. It is available in Igor Pro 7 and later.

DoUpdate

`DoUpdate` `[`***`/E=e /W=targWin /SPIN=ticks`***` ]`

The DoUpdate operation updates windows and dependent objects.

Flags

/E=*e*	Used with /W, /E=1 marks window as a progress window that can accept mouse events while user code is executing. Currently, only control panel windows can be used as a progress window.
/W=*targWin*	Updates only the specified window. Does not update dependencies or do any other updating. Currently, only graph and panel windows honor the /W flag. V_Flag is set to the truth the window exists. See **Progress Windows** on page IV-156 for other values for V_Flag.
/SPIN=*ticks*	Sets the delay between the start of a control procedure and the spinning beachball. *ticks* is the delay in ticks (60th of a second.) Unless used with the /W flag, /SPIN just sets the delay and an update is not done.

Details

Call DoUpdate from an Igor procedure to force Igor to update any objects that need updating. Igor updates any windows that need to be updated and also any objects (string variables, numeric variables, waves, controls) that depend on other objects that have changed since the last update. Page layout windows may not be immediately updated. For more information on page layout updates, see **Automatic Updating of Layout Objects** on page II-487.

Igor performs updates automatically if:

- No user-procedure is running.
- An interpreted procedure (Macro, Proc, Window type procedures) is running and PauseUpdate or DelayUpdate is not in effect.

Igor does not perform an automatic DoUpdate while a user-defined function is running. You can call DoUpdate from a user-defined function to force an update.

See Also

The **DelayUpdate**, **PauseUpdate**, and **ResumeUpdate** operations, **Progress Windows** on page IV-156.

DoWindow

`DoWindow` `[`***`flags`***`] [`***`windowName`***`]`

The DoWindow operation controls various window parameters and aspects. There are additional forms for DoWindow when the /S or /T flags are used; see the following DoWindow entries.

DoWindow does not support **Subwindow Syntax**.

Parameters

windowName is the name of a top-level graph, table, page layout, notebook, panel, Gizmo, camera, or XOP target window. *windowName* can not be a subwindow path.

A window's name is *not* the same as its title. The title is shown in the window's title bar. The name is used to manipulate the window from Igor commands. You can check both the name and the title using the Window Control dialog (in the Arrange submenu of the Window menu).

Flags

/B[=*bname*]
Moves the specified window to the back (to the bottom of desktop) or behind window *bname*.

/C
Changes the name of the target window to the specified name. The specified name must not be used for any other object except that it can be the name of an existing window macro.

/C/N
Changes the target window name and creates a new window macro for it. However, /N does nothing if a macro or function is running. /N is not applicable to notebooks.

/D
Deletes the file associated with window, if any (for notebooks only).

/F
Brings the window with the given name to the front (top of desktop).

/H
Specifies the command window as the target of the operation. When using /H, *windowName* must not be specified and only the /B and /HIDE flags are honored.

Use /H to bring the command window to the front (top of desktop).

Use /H/B to send the command window to the bottom of the desktop.

Use /H/HIDE to hide or show the command window.

/HIDE=*h*
Sets hidden state of a window.

h=0: Visible.
h=1: Hidden.
h=?: Sets the variable V_flag as follows:
0: The window does not exist.
1: The window is visible.
2: The window is hidden.

You can also read the hidden state using **GetWindow** and set it using **SetWindow**.

/K
Kills the window with the given name.

We recommend using **KillWindow** instead of DoWindow/K.

/N
Creates a new window macro for the window with the given name. However, /N does nothing if a macro or function is running. /N is not applicable to notebooks.

/R
Replaces (updates) the window macro for the named window or creates it if it does not yet exist. However, /R does nothing if a macro or function is running. /R is not applicable to notebooks.

/R/K
Replaces (updates) the window macro for the named window or creates it if it does not yet exist and then kills the window. However, /R does nothing if a macro or function is running. /R is not applicable to notebooks.

/W=*targWin*
Designates *targWin* as the target window; it also requires that you specify *windowName*. Use this mainly with floating panels, which are always on top. You can use a subwindow specification of an external subwindow only with the /T flag or without any flags.

Details

DoWindow sets the variable V_flag to 1 if there was a window with the specified name after DoWindow executed, to 0 if there was no such window, or to 2 if the window is hidden.

You can call DoWindow with a *windowName* and no flags to check if a window exists without altering the window. A better method is to use **WinType** which supports subwindows.

When used with the /N flag, *windowName* must not conflict with the name of any other object. When used with the /C flag, *windowName* must not conflict with the name of any other object except that it can be the name of an existing window macro.

The /R and /N flags do nothing when executed while a macro or function is running. This is necessary because changing procedures while they are executing causes unpredictable and undesirable results. However you can use the Execute/P operation to cause the DoWindow command to be executed after procedures are finished running. For example:

```
Function SaveWindowMacro(windowName)
   String windowName                    // "" for top graph or table

   if (strlen(windowName) == 0)
      windowName = WinName(0, 3)        // Name of top graph or table
   endif

   String cmd
   sprintf cmd, "DoWindow/R %s", windowName
   Execute/P cmd
End
```

You can use the /D flag in conjunction with the /K flag to kill a notebook window and delete its associated file, if any. /D has no effect on any other type of window and has no effect if the /K flag is not present.

Examples

```
DoWindow Graph0            // Set V_flag to 1 if Graph0 window exists.
DoWindow/F Graph0          // Make Graph0 the top/target window.
DoWindow/C MyGraph         // Target window (Graph0) renamed MyGraph.
DoWindow/H/B               // Put the command window in back.
DoWindow/D/K Notebook2     // Kill Notebook2, delete its file.
```

See Also

RenameWindow, **MoveWindow**, **MoveSubwindow**, **SetActiveSubwindow**, **KillWindow**

HideProcedures, **IgorInfo**

DoWindow/T

DoWindow /T windowName, windowTitleStr

The DoWindow/T operation sets the window title for the named window to the specified title.

Details

The title is shown in the window's title bar, and listed in the appropriate Windows submenu. The window *name* is still used to manipulate the window, so, for example, the window name (if *windowName* is a graph or table) is listed in the New Layout dialog; not the title.

You can check both the name and the title using the Window Control dialog (in the Control submenu of the Windows menu).

windowName is the name of the window or a special keyword, kwTopWin or kwFrame.

If *windowName* is kwTopWin, DoWindow retitles the top target window.

If *windowName* is kwFrame, DoWindow retitles the "frame" or "application" window that Igor has only under Windows. This is the window that contains Igor's menus and status bar. On Macintosh, kwFrame is allowed, but the command does nothing.

The Window Control dialog does not support kwFrame. The frame title persists until Igor quits or until it is restored as shown in the example. Setting *windowTitleStr* to "" will restore the normal frame title.

Examples

```
DoWindow/T MyGraph, "My Really Neat Graph"
DoWindow/T kwFrame, "My Igor-based Application"
DoWindow/T kwFrame, ""            // restore normal frame title
```

DoWindow/S

`DoWindow /N/S=`*`styleMacroName windowName`*

`DoWindow /R/S=`*`styleMacroName windowName`*

The DoWindow/S operation creates a new “style macro” for the named window, using the specified style macro name. Does not create or replace the window macro for the specified window.

Flags

/N/S=*styleMacroName*	Creates a new style macro with the given name based on the named window.
/R/S=*styleMacroName*	Creates or replaces the style macro with the given name based on the named window.

Details

The /R or /N flag must appear before the /S flag.

If the /S flag is present, the DoWindow operations does *not* create or replace the window macro for the specified window.

The /R and /N flags do nothing when executed while a macro or function is running. This is necessary because changing procedures while they are executing causes unpredictable and undesirable results.

DoXOPIdle

`DoXOPIdle`

The DoXOPIdle operation sends an IDLE event to all open XOPs. This operation is very specialized. Generally, only the author of an XOP will need to use this operation.

Details

Some XOPs (External OPeration code modules) require IDLE events to perform certain tasks.

Igor does not automatically send IDLE events to XOPs while an Igor program is running. You can call DoXOPIdle from a user-defined program to force Igor to send the event.

DPSS

`DPSS [`*`flags`*`] `*`numPoints`*`, `*`numWindows`*

The DPSS operation generates Slepian's Discrete Prolate Spheroidal Sequences.

The DPSS operation was added in Igor Pro 7.00.

Flags

/DEST=*destWave*	Saves the DPSS in a wave specified by *destWave*. The destination wave is overwritten if it exists. Creates a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-72 for details. If you omit /DEST the operation saves the result in the wave M_DPSS in the current data folder.
/EV=*evWave*	Saves the first numWindows eigenvalues in a wave specified by *evWave*. The eigenvalues are computed for a symmetric tridiagonal matrix. They are real, positive and close to 1. They can be used to estimate bias in multitaper calculations.
/FREE	Creates output waves as free waves. /FREE is permitted in user-defined functions only, not from the command line or in macros. If you use /FREE then *destWave*, *evWave* and *sumsWave* must be simple names, not paths. See **Free Waves** on page IV-91 for details on free waves.

/NW=nw Specifies the time-bandwidth product. This value should typically be in the range [2,6]. Given a time-bandwidth product nw it is recommended to use no more than 2*nw tapers in order to maximize variance efficiency. The default value of the time-bandwidth product is 3.

/DTPS=*sumsWave* Saves the sums of the generated DPSS windows in a wave specified by *sumsWave*.

/Q Suppress printing information in the history.

/Z Suppress errors. The variable V_Flag is set to 0 if successful and to -1 otherwise.

Details

DPSS generates Slepian's Discrete Prolate Spheroidal Sequences in a 2D double-precision wave of dimensions *numPoints* by *numWindows*.

If you do omit /DEST the operation creates the output wave M_DPSS in the current data folder. The sequences/tapers are arranged as columns in the output wave.

Examples

```
DPSS/DEST=dpss5 1000,5
Display dpss5[][0],dpss5[][1],dpss5[][2],dpss5[][3],dpss5[][4]
ModifyGraph rgb(dpss5#1)=(0,65535,0),rgb(dpss5#2)=(1,16019,65535)
ModifyGraph rgb(dpss5#3)=(65535,0,52428),rgb(dpss5#4)=(0,0,0)

// Different sequences are orthogonal
MatrixOp/o aa=col(dpss5,1)*col(dpss5,4)
Integrate/METH=1 aa/D=W_INT
Print W_INT[numpnts(W_INT)-1]
```

See Also

MultiTaperPSD, **WindowFunction**, **ImageWindow**, **Hanning**

References

D. Slepian, "Prolate spheroidal wave functions, Fourier analysis and uncertainty -- V: The discrete case.", Bell Syst. Tech J., vol 57 pp. 1317-1430, May 1978.

DrawAction

DrawAction [**/L=*layerName*/W=*winName***] ***keyword* = *value*** [, ***keyword* = *value*** …]

The DrawAction operation deletes, inserts, and reads back a named drawing object group or the entire draw layer.

Parameters

DrawAction accepts multiple *keyword* = *value* parameters on one line.

beginInsert [=*index*] Inserts draw commands before or at *index* position or at position specified by getgroup or delete parameters; position otherwise is zero.

commands [=*start*,*stop*] Stores commands in S_recreation for draw objects between *start* and *stop* index values, range defined by getgroup, or entire layer otherwise.

delete [=*start*,*stop*] Deletes draw objects between *start* and *stop* index values, range defined by getgroup, or entire layer otherwise.

extractOutline [=*start*,*stop*] Stores polygon outline between *start* and *stop* index values, range defined by getgroup, or entire layer otherwise. Waves W_PolyX and W_PolyY contain coordinates with NaN separators. V_npnts contains the number of objects, V_startPos contains the starting index value and V_endPos contains the ending index value. Coordinates are for the first object encountered.

endInsert Terminates insert mode.

getgroup=*name* Stores first and last index of named group in V_startPos and V_endPos. Use `_all_` to specify the entire layer. Sets V_flag to truth group exists.

Flags

/L=*layerName*   Specifies the drawing layer on which to act. *layerName* is one of the drawing layers as specified in **SetDrawLayer**.

/W=*winName*   Sets the named window or subwindow for drawing. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

Commands stored in S_recreation are the same as those that would be generated for the range of objects in the recreation macro for the window but also have comment lines preceding each object of the form:

```
// ;ITEMNO:n;
```

where *n* is the item number of the draw object.

Examples

Create a drawing with a named group:

```
NewPanel /W=(455,124,936,413)
SetDrawEnv fillfgc= (65535,0,0)
DrawRect 58,45,132,103
SetDrawEnv gstart,gname= fred
SetDrawEnv fillfgc= (65535,43690,0)
DrawRect 79,62,154,120
SetDrawEnv arrow= 1
DrawLine 139,70,219,70
SetDrawEnv gstop
SetDrawEnv fillfgc= (0,65535,65535)
DrawRect 95,77,175,138
SetDrawEnv fillfgc= (0,0,65535)
DrawRect 111,91,191,156
```

Get and print commands for the “fred” group:

```
DrawAction getgroup=fred,commands
Print S_recreation
```

prints:

```
// ;ITEMNO:2;
SetDrawEnv gstart,gname= fred
// ;ITEMNO:3;
SetDrawEnv fillfgc= (65535,43690,0)
// ;ITEMNO:4;
DrawRect 79,62,154,120
// ;ITEMNO:5;
SetDrawEnv arrow= 1
// ;ITEMNO:6;
DrawLine 139,70,219,70
// ;ITEMNO:7;
SetDrawEnv gstop
```

Replace group fred (the orange rectangle and the arrow) with a different object. First delete the group and enter insert mode:

```
DrawAction getgroup=fred, delete, begininsert
```

Next draw the replacement:

```
SetDrawEnv gstart,gname= fred
SetDrawEnv fillfgc= (65535,65535,0)
DrawOval 82,62,161,123
SetDrawEnv gstop
```

Lastly exit insert mode:

```
DrawAction endinsert
```

See Also

The **SetDrawEnv** operation and Chapter III-3, **Drawing**.

DrawArc

DrawArc [**/W=*winName*/X/Y**] ***xOrg*, *yOrg*, *arcRadius*, *startAngle*, *stopAngle***

The DrawArc operation draws a circular counterclockwise arc with center at *xOrg* and *yOrg*.

Parameters

(*xOrg*, *yOrg*) defines the center point for the arc in the currently active coordinate system.

Angles are measured in degrees increasing in a counterclockwise direction. The *startAngle* specifies the starting angle for the arc and *stopAngle* specifies the end. If *stopAngle* is equal to *startAngle*, 360° is added to *stopAngle*. Thus, a circle can be drawn using *startAngle* = *stopAngle*.

The *arcRadius* is the radial distance measured in points from (*xOrg*, *yOrg*).

Flags

/W=*winName* — Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/X — Measures *arcRadius* using the current X-coordinate system. If /Y is also used, the arc may be elliptical.

/Y — Measures *arcRadius* using the current Y-coordinate system. If /X is also used, the arc may be elliptical.

Details

Arcs honor the current dash pattern and arrowhead setting in the same way as polygons and Beziers. In fact, arcs are implemented using Bezier curves.

Normally, you would create arcs programmatically. If you need to sketch an arc-like object, you should probably use a Bezier curve because it is more flexible and easier to adjust. However, there is one handy feature of arcs that make them useful for manual drawing: the origin can be in any of the supported coordinate systems and the radius is in points.

To draw an arc interactively, see **Arcs and Circles** on page III-64 for instructions.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv** and **SetDrawLayer** operations.

The **DrawBezier**, **DrawOval** and **DrawAction** operations.

DrawBezier

DrawBezier [**/W=*winName* /ABS**] ***xOrg*, *yOrg*, *hScaling*, *vScaling*, {x_0,y_0,x_1,y_1 …}**

DrawBezier [**/W=*winName* /ABS**] ***xOrg*, *yOrg*, *hScaling*, *vScaling*, *xWaveName*, *yWaveName***

DrawBezier/A [**/W=*winName***] **{x_n, y_n, x_{n+1}, y_{n+1} …}**

The DrawBezier operation draws a Bezier curve with first point of the curve positioned at *xOrg* and *yOrg*.

Parameters

(*xOrg*, *yOrg*) defines the starting point for the Bezier curve in the currently active coordinate system.

hScaling and *vScaling* set the horizontal and vertical scale factors about the origin, with 1 meaning 100%.

The *xWaveName*, *yWaveName* version of DrawBezier gets data from the named X and Y waves. This connection is maintained so that any changes to either wave will result in updates to the Bezier curve.

To use the version of DrawBezier that takes a literal list of vertices, you place as many vertices as you like on the first line and then use as many /A versions as necessary to define all the vertices.

Flags

/A	Appends the given vertices to the currently open Bezier (freshly drawn or current selection).
/ABS	Suppresses the default subtraction of the first point from the rest of the data.
/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

Data waves defining Bezier curves must have 1+3**n* data points. Every third data point is an anchor point and lies on the curve; intervening points are control points that define the direction of the curve relative to the adjacent anchor point.

Normally, you should create and edit a Bezier curve using drawing tools, and not calculate values. See **Polygon Tool** on page III-64 and **Editing a Bezier Curve** on page III-70 for instructions.

You can include the /ABS flag to suppress the default subtraction of the first point.

To change just the origin and scale without respecifying the data use:

```
DrawBezier xOrg, yOrg, hScaling, vScaling,{}
```

It is possible to separate a polygon into segments by adding coordinate pairs that are NaN. For details, see **Segmented Bezier Curves** on page III-71.

Example

For an example using Bezier curves, see **Segmented Bezier Curves** on page III-71.

See Also

Chapter III-3, **Drawing**.

Polygon Tool on page III-64 for discussion on creating Beziers. **DrawPoly and DrawBezier Operations** on page III-75 and the **SetDrawEnv** and **SetDrawLayer** operations.

DrawArc, **DrawPoly**, **DrawAction**, **BezierToPolygon**

DrawLine

DrawLine [/W=*winName*] *x*$_0$, *y*$_0$, *x*$_1$, *y*$_1$

The DrawLine operation draws a line in the target graph, layout or control panel from (*x0,y0*) to (*x1,y1*).

Flags

/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The coordinate system as well as the line's thickness, color, dash pattern and other properties are determined by the current drawing environment. The line is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawOval

`DrawOval [/W=winName] left, top, right, bottom`

The DrawOval operation draws an oval in the target graph, layout or control panel within the rectangle defined by *left*, *top*, *right,* and *bottom*.

Flags

/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The coordinate system as well as the oval's thickness, color, dash pattern and other properties are determined by the current drawing environment. The oval is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawPICT

`DrawPICT [/W=winName][/RABS] left, top, hScaling, vScaling, pictName`

The DrawPICT operation draws the named picture in the target graph, layout or control panel. The *left* and *top* parameters set the position of the top/left corner of the picture. *hScaling* and *vScaling* set the horizontal and vertical scale factors with 1 meaning 100%.

Flags

/RABS	Draws the named picture using absolute scaling. In this mode, it draws the picture in the rectangle defined by *left* and *top* for point (x0,y0), and by *hScaling* and *vScaling* for point (x1,y1), respectively.
/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The coordinate system for the left and top parameters is determined by the current drawing environment. The PICT is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawPoly

```
DrawPoly [/W=winName /ABS] xorg, yorg, hScaling, vScaling, xWaveName, yWaveName
DrawPoly [/W=winName /ABS] xorg, yorg, hScaling, vScaling, {x0,y0,x1,y1 …}
DrawPoly/A [/W=winName] {xn, yn, xn+1, yn+1 …}
```

The DrawPoly operation draws a polygon in the target graph, layout or control panel.

Parameters

(*xorg*, *yorg*) defines the starting point for the polygon in the currently active coordinate system.

hScaling and *vScaling* set the horizontal and vertical scale factors, with 1 meaning 100%.

The *xWaveName*, *yWaveName* version of DrawPoly gets data from those X and Y waves. This connection is maintained so that changes to either wave will update the polygon.

The DrawPoly operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

To use the version of DrawPoly that takes a literal list of vertices, you place as many vertices as you like on the first line and then use as many /A versions as necessary to define all the vertices.

Flags

/A	Appends the given vertices to the currently open polygon (freshly drawn or current selection).
/ABS	Suppresses the default subtraction of the first point from the rest of the data.
/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

Because *xorg* and *yorg* define the location of the starting vertex of the poly, adding or subtracting a constant from the vertices will have no effect. The first XY pair in the {*x0, y0, x1, y1,…*} vertex list will appear at (*xorg,yorg*) *regardless* of the value of *x0* and *y0*. *x0* and *y0* merely serve to set a reference point for the list of vertices. Subsequent vertices are relative to (*x0,y0*).

To keep your mental health intact, we recommend that you specify (*x0,y0*) as (0,0) so that all the following vertices are offsets from that origin. Then (*xorg,yorg*) sets the position of the polygon and all of the vertices in the list are relative to that origin.

An alternate method is to use the same values for (*x0,y0*) as for (*xorg,yorg*) if you consider the vertices to be “absolute” coordinates.

You can include the /ABS flag to suppress the subtraction of the first point.

To change just the origin and scale of the currently open polygon — without having to respecify the data — use:

```
DrawPoly xorg, yorg, hScaling, vScaling,{}
```

The coordinate system as well as the polygon’s thickness, color, dash pattern and other properties are determined by the current drawing environment. The polygon is drawn in the current draw layer for the window, as determined by SetDrawLayer.

It is possible to separate a polygon into segments by adding coordinate pairs that are NaN. For details, see **Segmented Polygons** on page III-70.

Examples

Here are some commands to draw some small triangles using absolute drawing coordinates (see **SetDrawEnv**).

```
Display             // make a new empty graph
//Draw one triangle, starting at 50,50 at 100% scaling
SetDrawEnv xcoord= abs,ycoord= abs
DrawPoly 50,50,1,1, {0,0,10,10,-10,10,0,0}
//Draw second triangle below and to the right, same size and shape
SetDrawEnv xcoord= abs,ycoord= abs
DrawPoly 100,100,1,1, {0,0,10,10,-10,10,0,0}
```

For another example using polygons, see **Segmented Polygons** on page III-70.

See Also

DrawPoly and DrawBezier Operations on page III-75

SetDrawEnv, **SetDrawLayer**, **DrawBezier**, **DrawAction**, **PolygonOp**.

DrawRect

`DrawRect [/W=`*`winName`*`]` ***`left, top, right, bottom`***

The DrawRect operation draws a rectangle in the target graph, layout or control panel within the rectangle defined by *left*, *top*, *right*, and *bottom*.

Flags

/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The coordinate system as well as the rectangle's thickness, color, dash pattern and other properties are determined by the current drawing environment. The rectangle is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

SetDrawEnv, **SetDrawLayer**, **DrawAction**, **BezierToPolygon**

DrawRRect

`DrawRRect [/W=`*`winName`*`]` ***`left, top, right, bottom`***

The DrawRRect operation draws a rounded rectangle in the target graph, layout or control panel within the rectangle defined by *left*, *top*, *right*, and *bottom*.

Flags

/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The coordinate system as well as the rectangle's rounding, thickness, color, dash pattern and other properties are determined by the current drawing environment. The rounded rectangle is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawText

`DrawText [/W=`*`winName`*`]` ***x_0, y_0, `textStr`***

The DrawText operation draws the specified text in the target graph, layout or control panel. The position of the text is determined by (*x0*, *y0*) along with the current textxjust, textyjust and textrot settings as set by **SetDrawEnv**.

Flags

/W=*winName* Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The coordinate system as well as the text's font, size, style and other properties are determined by the current drawing environment. The text is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawUserShape

DrawUserShape [***/W=winName /MO=options***] *x0*, *y0*, *x1*, *y1*, *userFuncName*, *textString*, *privateString*

The DrawUserShape operation is similar to built-in drawing operations except the shape is defined by a user-defined function which is executed when the shape is drawn.

DrawUserShape was added in Igor Pro 7.00.

Parameters

For rectangular shapes (*x0*,*y0*) defines the top left corner while (*x1*,*y1*) defines the lower right corner in the currently active coordinate system. For line-like shapes, (*x0*,*y0*) specifies the start of the line while(*x1*,*y1*) specifies the end.

userFuncName specifies your user-defined function that uses built-in drawing operations to define the shape and provides information about it to Igor.

textString specifies text to be drawn over the shape by Igor. Pass "" if you do not need text. Using escape codes you can change the font, size, style, and color of the text. See **Annotation Escape Codes** on page III-53 or details.

privateString is text or binary data used by the user-defined function for any purpose. It is typically used to maintain state information and is saved in recreation macros using a method that supports binary. Pass "" if you do not need this.

To support modifying an existing shape, each of the last three parameters above can be `_NoChange_`. The *x0*, *y0*, *x1*, *y1* parameters can be all zero for no change. Using `_NoChange_` when there is no existing shape is an error. To target an existing shape, it must be selected by the drawing tools.

Flags

/W=*winName* Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/M=*options* An integer that Igor passes to your user-defined drawing function for use for any purpose.

Details

The user-defined function must have the following form and structure parameter.

```
Function MyUserShape(s) : DrawUserShape // Optional ": DrawUserShape" indicates
                                        // that this function should be added
   STRUCT WMDrawUserShapeStruct &s      // to the menu of shapes in the drawing palette

   Variable returnValue= 1
```

```
	StrSwitch(s.action)
		case "draw":
			Print "Put DrawXXX commands here"
			break
		case "drawSelected":
			Print "Put Draw commands when selected (may be same as normalDraw) here"
			break
		case "hitTest":
			Print "Put code to test if mouse is over a control point here"
			// Set doSetCursor and optionally cursorCode if so"
			// Set returnValue to zero if not over control point or, if in operate
			// mode and you don't want to start a button operation.
			break
		case "mouseDown":
			if( s.operateMode )
				Print "Mouse down in button mode."
			else
				// Code similar or same as hitTest.
				Print "User is starting a drag of a control point."
			endif
			// Set returnValue to zero if if no action or redraw needed
			break
		case "mouseMove":
			if(s.operateMode)
				Print "Roll-over mode and mouse is inside shape."
			else
				Print "Mouse has been captured and user is dragging a control point."
			endif
			// Set returnValue to zero if if no action or redraw needed
			break
		case "mouseUp":
			if(s.operateMode)
				Print "Mouse button released."
			else
				Print "User finished dragging control point."
			endif
			break
		case "getInfo":
			Print "Igor is requesting info about this shape."
			s.textString= "category for shape"
			s.privateString= "shape name"
			s.options= 0// Or set bits 0, 1 and/or 2
			break
	EndSwitch

	return returnValue
End
```

WMDrawUserShapeStruct is a built-in structure with the following members:

`WMDrawUserShapeStruct` Structure Members

Member	Description
`char action[32]`	Input: Specifies what action is requested.
`Int32 options`	Input: Value from /MO flag. Output: When action is `getInfo`, set bits as follows: Set bit 0 if the shape should behave like a simple line. When resizing end-points, you will get live updates. Set bit 1 if the shape is to act like a button; you will get mouse down in normal operate mode. Set bit 2 to get roll-over action. You will get `hitTest` action and if 1 is returned, the mouse will be captured.
`Int32 operateMode`	Input: If 0, the shape is being edited; if 1, normal operate mode (only if options bit 1 or 2 was set during `getInfo`).
`PointF mouseLoc`	Input: The location of the mouse in normalized coordinates.

`WMDrawUserShapeStruct` Structure Members

Member	Description
`Int32 doSetCursor`	Output: If action is `hitTest`, set true to use the following cursor number. Also used for `mouseMoved` in rollover mode.
`Int32 cursorCode`	Output: If action is `hitTest` and `doSetCursor` is set, then set this to the desired Igor cursor number.
`double x0,y0,x1,y1`	Input: Coordinates of the enclosing rectangle of the shape.
`RectF objectR`	Input: Coordinates of the enclosing rectangle of the shape in device units.
`char winName[MAX_HostChildSpec+1]`	Full path to host subwindow
`// Information about the coordinate system`	
`Rect drawRect`	Draw rect in device coordinates
`Rect plotRect`	In a graph, this is the plot area
`Rect axisRect`	In a graph, this is the plot area including axis standoff
`char xcName[MAX_OBJ_NAME+1]`	Name of X coordinate system, may be axis name
`char ycName[MAX_OBJ_NAME+1]`	Name of Y coordinate system, may be axis name
`double angle`	Input: Rotation angle, use when displaying text
`String textString`	Input: Use or ignore; special output for `getInfo`
`String privateString`	Input and output: Maintained by Igor but defined by user function; may be binary; special output for `getInfo`

The constants used to size the char arrays, `MAX_HostChildSpec` and `MAX_OBJ_NAME`, are defined internally in Igor and are subject to change in future versions.

The drawing commands you provide for the `draw` and `drawSelected` actions must use normalized coordinates ranging from (0,0) to (1,1). For example, to draw a rectangle you would use

```
DrawRect 0,0,1,1
```

Generally, you will not set drawing environment parameters such as color or fill mode but will leave that to the user of your shape just as they would for built-in shapes such as a Rectangle.

Drawing commands such as **SetDrawEnv**, **DrawRect**, etc., include a /W flag that directs their action to a particular window or subwindow. However, when called from a DrawUserShape function, the /W flag is ignored and all drawing commands are directed to the window or subwindow containing the shape.

You will get the `drawSelected` action when the user has selected your shape with the arrow draw tool. For simple shapes, this can use the same code as the draw action but if you want the shape to be editable, you can draw control points such as a yellow dot. You would then provide code in the `hitTest` and `mouseDown` actions that determines if the user has moved over or clicked on a control point. For an example of this, see the FatArrows procedure file in the User Draw Shapes demo experiment.

Your function should respond to the `getInfo` action by setting the textString field to a category name and the privateString field to a shape name. A category name might be "Arrows" and a shape name might be "Right Arrow". These names are used to populate the menus you get when right-clicking the user shapes icon in the drawing tools palette. You can also set bits in the options field to enable certain special actions. For examples of these, see the LineCallout and button procedure windows in the User Draw Shapes demo experiment.

Because the user-defined function runs during a drawing operation that cannot be interrupted without crashing Igor, the debugger cannot be invoked while it is running. Consequently breakpoints set in the function are ignored. Use **Debugging With Print Statements** on page IV-212 instead.

To support button-like shapes in graphs and layouts, an additional drawing layer named Overlay was added in Igor Pro 7.00. This layer is drawn over the top of everything else and is not printed or exported. Objects in the Overlay draw layer can update without the need to redraw the entire window as would be the case if one of the previously existing layers were used. For consistency, this layer is also available in control panels.

See Also

Chapter III-3, **Drawing**.

SetDrawEnv, **SetDrawLayer**, **DrawBezier**, **DrawPoly**, **DrawAction**

DSPDetrend

`DSPDetrend` `[`***`flags`***`]` ***`srcWave`***

The DSPDetrend operation removes from *srcWave* a trend defined by the best fit of the specified function to the data in *srcWave*.

Flags

/A	Subtracts the average of *srcWave* before performing any fitting. Added in Igor Pro 7.00.
/F= *function*	*function* is the name of a built-in curve fitting function: gauss, lor, exp, dblexp, sin, line, poly (requires /P flag), hillEquation, sigmoid, power, lognormal, poly2d (requires /P flag), gauss2d. If *function* is unspecified, the defaults are line if *srcWave* is 1D or poly2d if *srcWave* is 2D.
/M=*maskWave*	Detrending will only affect points that are nonzero in *maskWave*. Note that *maskWave* must have the same dimensionality as *srcWave*.
/P=*n*	Specifies polynomial order for poly or poly2d functions (see **CurveFit** for details). When used with the 1D poly function *n* specifies the number of terms in the polynomial. By default *n*=3 for the 1D case and *n*=1 for poly2d.
/Q	Quiet mode; no error reporting.

Details

DSPDetrend sets V_flag to zero when the operation succeeds, otherwise it will be set to -1 or will contain an error code from the curve fitting routines. Results are saved in the wave W_Detrend (for 1D input) or M_Detrend (for 2D input) in the current data folder. If a wave by that name already exists in the current data folder it will be overwritten.

See Also

CurveFit for more information about V_FitQuitReason and the built-in fitting functions.

DSPPeriodogram

`DSPPeriodogram` `[`***`flags`***`]` ***`srcWave`*** `[`***`srcWave2`***`]`

The DSPPeriodogram operation calculates the periodogram, cross-spectral density or the degree of coherence of the input waves. The result of the operation is stored in the wave W_Periodogram in the current data folder or in the wave that you specify using the /DEST flag.

To compute the cross-spectral density or the degree of coherence, you need to specify the second wave using the optional *srcWave2* parameter. In this case, W_Periodogram will be complex and the /DB and /DBR flags do not apply.

Flags

/DB	Expresses results in dB using the maximum value as reference.
/DBR=*ref*	Express the results in dB using the specified *ref* value.
/COHR	Computes the degree of coherence. This flag applies when the input consists of two waves.

/DEST=*destWave* Specifies the output wave created by the operation.

The /DEST flag was added in Igor Pro 8.00.

It is an error to specify the same wave as both *srcWave* and *destWave*.

When used in a function, the DSPPeriodogram operation by default creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-72 for details.

/DLSG When computing the periodogram, cross-spectral density or the degree of coherence using multiple segments the operation by default pads the last segment with zeros as necessary. If you specify this flag, an incomplete last segment is dropped and not included in the calculation.

/DTRD Detrends segments by subtracting the linear regression of each segment before multiplication by the window function. /DTRD affects segments and is not compatible with /NODC=1. /DTRD was added in Igor Pro 8.00.

/NODC=*val* Suppresses the DC term:

val=1: Removes the DC by subtracting the average value of the signal before processing and before applying any window function (see /Win below).

val=2: Suppresses the DC term by setting it equal to the second term in the FFT array.

val=0: Computes the DC term using the FFT (default).

/NOR=*N* Sets the normalization, N, in the periodogram equation. By default, it is the number of data points times the square norm of the window function (if any).

N=0 or 1: Skips default normalization.

Any other value of *N* is used as the only normalization.

/PARS Sets the normalization to satisfy Parseval's theorem even when using a window function.

The /PARS flag was added in Igor Pro 8.00. It overrides the /NOR flag.

See **Normalization Satisfying Parseval's Theorem** on page V-185 for further information.

/Q Quiet mode; suppresses printing in the history area.

/SEGN={*ptsPerSegment*, *overlapPts*}

Use this flag to compute the periodogram, cross-spectral density or degree of coherence by averaging over multiple segments taken from the input waves. The size of each interval is *ptsPerSegment*. *overlapPts* determines the number of points at the end of each interval that are included in the next segment.

/R=[*startPt*, *endPt*] Calculates the periodogram for a limited range of the wave. *startPt* and *endPt* are expressed in terms of point numbers in *srcWave*.

/R=(*startX*, *endX*) Calculates the periodogram for a limited range of the wave. *startX* and *endX* are expressed in terms of x-values. Note that this option will convert your x-specifications to point numbers and some roundoff may occur.

/WIN=*windowKind* Specifies the window type. If you omit the /W flag, DSPPeriodogram uses a rectangular window for the full wave or the range of data selected by the /R flag.

Choices for *windowKind* are:

Bartlett, Blackman367, Blackman361, Blackman492, Blackman474, Cos1, Cos2, Cos3, Cos4, Hamming, Hanning, KaiserBessel20, KaiserBessel25, KaiserBessel30, Parzen, Poisson2, Poisson3, Poisson4, and Riemann.

See **FFT** for window equations and details.

/Z Do not report errors. When an error occurs, V_flag is set to -1.

Details

The default periodogram is defined as

$$Periodogram = \frac{|F(s)|^2}{N},$$

where F (s) is the Fourier transform of the signal s and N is the number of points.

In most practical situations you need to account for using a window function (when computing the Fourier transform) which takes the form

$$Periodogram = \frac{|F(s \cdot w)|^2}{N_p N_w},$$

where w is the window function, N_p is the number of points and N_w is the normalization of the window function.

If you compute the periodogram by subdividing the signal into multiple segments (with any overlap) and averaging the results over all segments, the expression for the periodogram is

$$Periodogram = \frac{\sum_{i=1}^{M} |F(s_i \cdot w)|^2}{MN_s N_w},$$

where si is the ith segment s, Ns is the number of points per segment and M is the number of segments.

When calculating the cross-spectral density (csd) of two waves s_1 and s_2, the operation results in a complex valued wave

$$csd = \frac{F(s_A)[F(s_B)]^*}{N},$$

which contains the normalized product of the Fourier transform of the first wave S_A with the complex conjugate of the Fourier transform of the second wave S_B. The extension of the csd calculation to segment averaging has the form

$$csd = \frac{\sum_{i=0}^{M} F(s_{Ai})[F(s_{Bi})]^*}{MN_s N_w},$$

where S_{Ai} is the ith segment of the first wave, M is the number of segments and N_s is the number of points in a segment.

The degree of coherence is a normalized version of the cross-spectral density. It is given by

$$\gamma = \frac{\sum_{i=0}^{M} F(s_{Ai})[F(s_{Bi})]^*}{\sqrt{\sum_{i=0}^{M} F(s_{Ai})[F(s_{Ai})]^* \sum_{i=0}^{M} F(s_{Bi})[F(s_{Bi})]^*}}.$$

The bias in the degree of coherence is calculated using the approximation

$$B = \frac{1}{M}\left[1-|\gamma|^2\right]^2.$$

The bias is stored in the wave W_Bias.

If you use the /SEGN flag the actual number of segments is reported in the variable V_numSegments.

Note that DSPPeriodogram does not test the dimensionality of the wave; it treats the wave as 1D. When you compute the cross-spectral density or the degree of coherence the number-type, dimensionality and the scaling of the two waves must agree.

Normalization Satisfying Parseval's Theorem

After executing DSPPeriodogram with the /PARS flag, you can check that the normalization satisfies Parseval's theorem using this function:

```
Function CheckNormalization(srcWave, periodogramWave)
   Wave srcWave                        // A real valued time series
   Wave periodogramWave                // e.g., W_Periodogram

   Duplicate/FREE periodogramWave,wp    // Preserve original
   wp[0]/=2                            // Correct the 0 bin
   wp[numpnts(wp)-1] /=2               // Correct the Nyquist bin
   MatrixOP/FREE w2=magsqr(srcWave)/numPoints(srcWave)
   Print sum(wp), sum(w2)              // Parseval: These should be equal
End
```

See Also

The **ImageWindow** operation for 2D windowing applications. **FFT** for window equations and details.

The **Hanning**, **LombPeriodogram** and **MatrixOp** operations.

References

For more information about the use of window functions see:

Harris, F.J., On the use of windows for harmonic analysis with the discrete Fourier Transform, *Proc, IEEE, 66*, 51-83, 1978.

G.C. Carter, C.H. Knapp and A.H. Nuttall, The Estimation of the Magnitude-squared Coherence Function Via Overlapped Fast Fourier Transform Processing, *IEEE Trans. Audio and Electroacoustics*, V. AU-21, (4) 1973.

Duplicate

Duplicate [***flags***][***type flags***] ***srcWaveName***, ***destWaveName*** [, ***destWaveName***]…

The Duplicate operation creates new waves, the names of which are specified by *destWaveName*s and the contents, data type and scaling of which are identical to *srcWaveName*.

Parameters

srcWaveName must be the name of an existing wave.

The *destWaveName*s should be wave names not currently in use unless the /O flag is used to overwrite existing waves.

Flags

/FREE[=*nm*]
Creates a free wave. Allowed only in functions and only if a simple name or wave reference structure field is specified.

See **Free Waves** on page IV-91 for further discussion.

If *nm* is present and non-zero, then waveName is used as the name for the free wave, overriding the default name '_free_'. The ability to specify the name of a free wave was added in Igor Pro 9.00 as a debugging aid - see **Free Wave Names** on page IV-95 and **Wave Tracking** on page IV-207 for details.

/O
Overwrites existing waves with the same name as *destWaveName*.

/R=(*startX*,*endX*)
Specifies an X range in the source wave from which the destination wave is created.

See Details for further discussion of /R.

/R=(*startX*,*endX*)(*startY*,*endY*)

Specifies both X and Y range. Further dimensions are constructed analogously.

See Details for further discussion of /R.

/R=[*startP*,*endP*]
Specifies a row range in the source wave from which the destination wave is created. Further dimensions are constructed just like the scaled dimension ranges.

See Details for further discussion of /R.

/RMD=[*firstRow*,*lastRow*][*firstColumn*,*lastColumn*][*firstLayer*,*lastlayer*][*firstChunk*,*lastChunk*]

Designates a contiguous range of data in the source wave to which the operation is to be applied. This flag was added in Igor Pro 7.00.

You can include all higher dimensions by leaving off the corresponding brackets. For example:

```
/RMD=[firstRow,lastRow]
```

includes all available columns, layers and chunks.

You can use empty brackets to include all of a given dimension. For example:

```
/RMD=[][firstColumn,lastColumn]
```

means "all rows from column A to column B".

You can use a * to specify the end of any dimension. For example:

```
/RMD=[firstRow,*]
```

means "from firstRow through the last row".

Type Flags *(used only in functions)*

When used in user-defined functions, Duplicate can also take the /B, /C, /D, /I, /S, /U, /W, /T, /DF and /WAVE flags. This does not affect the result of the Duplicate operation - these flags are used only to identify what kind of wave is expected at runtime.

This information is used if, later in the function, you create a wave assignment statement using a duplicated wave as the destination:

```
Function DupIt(wv)
   Wave/C wv                  //complex wave

   Duplicate/O/C wv,dupWv     //tell Igor that dupWv is complex
   dupWv[0]=cmplx(5.0,1.0)    //no error, because dupWv known complex
   …
```

If Duplicate did not have the /C flag, Igor would complain with a “function not available for this number type” message when it tried to compile the assignment of dupWv to the result of the cmplx function.

These type flags do not need to be used except when it needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-73 and **WAVE Reference Type Flags** on page IV-74 for a complete list of type flags and further details.

Details

If /R is omitted, the entire wave is duplicated.

In the range specifications used with /R, a `*` for the end means duplicate to the end. You can also simply leave out the end specification. To include all of a given dimension, use `/R=[]`. If you leave off higher dimensions, all those dimensions are duplicated. That is, `/R=[1,5]` for a 2D wave is equivalent to `/R=[1,5][]`.

The destination wave will always be unlocked, even if the source wave was locked.

Warning:

Under some circumstances, such as in loops in user-defined functions, Duplicate may exhibit undesired behavior. When you use

```
Duplicate/O srcWave, DestWaveName
```

in a user-defined function, it creates a local WAVE variable named *DestWaveName* at compile time. At runtime, if the WAVE variable is NULL, it creates a wave of the same name in the current data folder. If, however, the WAVE variable is not NULL, as it would be in a loop, then the referenced wave will be overwritten no matter where it is located. If the desired behavior is to create (or overwrite) a wave in the current data folder, you should use one of the following two methods:

```
Duplicate/O srcWave, $"DestWaveName"
WAVE DestWaveName   // only if you need to reference dest wave
```

or

```
Duplicate/O srcWave, DestWaveName
// then after you are finished using DestWaveName…
WAVE DestWaveName=$""
```

See Also

Rename, **Concatenate**, **SplitWave**

DuplicateDataFolder

`DuplicateDataFolder [ /O=`*`options`*` /Z ] `*`sourceDataFolderSpec`*`, `*`destDataFolderSpec`*

The DuplicateDataFolder operation makes a copy of the source data folder and everything in it and places the copy at the specified location with the specified name.

Parameters

sourceDataFolderSpec can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name, or a data folder reference (DFREF) in a user-defined function.

destDataFolderSpec can be just a data folder name, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name. If just a data folder name is used then the new data folder is created in the current data folder, or a data folder reference (DFREF) in a user-defined function.

Depending on the syntax you use, DuplicateDataFolder may overwrite the data folder specified by destDataFolderSpec or it may copy the source data folder into the data folder specified by destDataFolderSpec. See **DuplicateDataFolder Destination** on page V-188 below for details.

Flags

/O=*options*	Overwrites the destination data folder if it already exists. The /O flag was added in Igor Pro 8.00. options=0: Attempting to overwrite a data folder generates an error, as if you omitted /O. options=1: Completely overwrites the destination data folder. If the destination data folder exists, DuplicateDataFolder first deletes it if possible. If it can not be deleted, for example because it contains a wave that is in use, DuplicateDataFolder generates an error. If the deletion succeeds, DuplicateDataFolder then copies the source to the destination.

options=2: Merges the source data folder into the destination data folder. If an item in the source data folder exists in the destination data folder, DuplicateDataFolder overwrites it. Otherwise it copies it. Items in the destination data folder that do not exist in the source data folder remain in the destination data folder.

options=3: Merges the source data folder into the destination data folder and then deletes items that are not in the source data folder if possible. If an item in the source data folder does not exist in the destination data folder, DuplicateDataFolder attempts to delete it from the destination data folder. If it can not be deleted because it is in use, no error is generated.

/Z Errors are not fatal - if an error occurs, procedure execution continues. You can check the V_flag output variable to see if an error occurred.

Details

An error is issued if the destination is contained within the source data folder.

A common use for a data folder reference variable is to create a copy of a data folder inside a free data folder:

```
DFREF dest = NewFreeDataFolder()
DuplicateDataFolder source, dest     // Copy of source in a free data folder
```

DuplicateDataFolder Destination

Depending on the syntax you use, DuplicateDataFolder may overwrite the data folder specified by destDataFolderSpec or it may copy the source data folder into the data folder specified by destDataFolderSpec. To explain this, we will assume that you have executed these commands:

```
KillDataFolder/Z root:
NewDataFolder/O root:DataFolderA
NewDataFolder/O root:DataFolderA:SubDataFolderA
NewDataFolder/O root:DataFolderB
```

This gives you a data hierarchy like this:

```
root
    DataFolderA
        SubDataFolderA
    DataFolderB
```

The following commands illustrate what DuplicateDataFolder does for a given syntax starting with that data hierarchy.

```
// 1. Literal dest without trailing colon:
// Overwrites DataFolderB with copy of DataFolderA named DataFolderB
DuplicateDataFolder/O=1 root:DataFolderA, root:DataFolderB

// 2. Literal dest with trailing colon:
// Copies DataFolderA into DataFolderB
DuplicateDataFolder/O=1 root:DataFolderA, root:DataFolderB:

// 3. Literal dest with explicit dest child name:
// Copies DataFolderA into DataFolderB with name Child
DuplicateDataFolder/O=1 root:DataFolderA, root:DataFolderB:Child

// 4. DFREF dest without trailing colon:
// Copies DataFolderA into DataFolderB
DFREF dest = root:DataFolderB
DuplicateDataFolder/O=1 root:DataFolderA, dest

// 5. DFREF dest with trailing colon:
// Generates error
DFREF dest = root:DataFolderB
DuplicateDataFolder/O=1 root:DataFolderA, dest:// Error - trailing colon not allowed

// 6. DFREF dest with explicit dest child name:
// Copies DataFolderA into DataFolderB with name Child
DFREF dest = root:DataFolderB
DuplicateDataFolder/O=1 root:DataFolderA, dest:Child
```

Output Variables

DuplicateDataFolder sets the following output variable:

`V_flag`	0 if the operation succeeded, -1 if the destination data folder already existed, or a non-zero error code. The V_flag output variable was added in Igor Pro 8.00.

Examples

```
DuplicateDataFolder root:DF0, root:DF0Copy // Create a copy of DF0 named DF0Copy
```

See Also

MoveDataFolder, **Data Folders** on page II-107, **Data Folder References** on page IV-78, **Free Data Folders** on page IV-96

DWT

`DWT` `[`***`flags`***`]` ***`srcWaveName, destWaveName`***

The DWT operation performs discrete wavelet transform on the input wave *srcWaveName*. The operation works on one or more dimensions only as long as the number of elements in each dimension is a power of 2 or when the /P flag is specified

Flags

/D Denoises the source wave. Performs the specified wavelet transform in the forward direction. It then zeros all transform coefficients whose magnitude fall below a given percentage (specified by the /V flag) of the maximum magnitude of the transform. It then performs the inverse transform placing the result in *destWaveName*. The /I flag is incompatible with the /D flag.

/I Perform the inverse wavelet transform. The /S and /D flags are incompatible with the /I flag.

/N=*num* Specifies the number of wavelet coefficients. See /T flag for supported combinations.

/P=*num* Controls padding:

num=1: Adds zero padding to the end of the dimension up to nearest power of 2 when the number of data elements in a given dimension of *srcWaveName* is not a power of 2.

num=2: Uses zero padding to compute the transform, but the resulting wave is truncated to the length of the input wave.

/S Smooths the source wave. This performs the specified wavelet transform in the forward direction. It then zeros all transform coefficients except those between 0 and the cut-off value (specified in % by /V flag). It then performs the inverse transform placing the result in *destWaveName*. The /I flag is incompatible with the /S flag.

/T=*type* Performs the wavelet transform specified by *type*. The following table gives the transform name with the *type* code for the transform and the allowed values of the *num* parameter used with the /N flag. “NA” means that the /N flag is not applicable to the corresponding transform.

Wavelet Transform	***type***	***num***
Daubechies	1 (default)	4, 6, 8, 10, 12, 20
Haar	2	NA
Battle-Lemarie	4	NA
Burt-Adelson	8	NA
Coifman	16	2, 4, 6
Pseudo-Coifman	32	NA
splines	64	1 (2-2), 2 (2-4), 3 (3-3), 4 (3-7)

/V=*value* Specifies the degree of smoothing with the /S and /D flags only.

For /S, *value* gives the cutoff as a percentage of data points above which coefficients are set to zero. For /D, *value* specifies the percentage of the maximum magnitude of the transform such that coefficients smaller than this value are set to zero.

Details

If *destWaveName* exists, DWT overwrites it; if it does not exist, DWT creates it.

When used in a function, the DWT operation automatically creates a wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-72 for details.

If *destWaveName* is not specified, the DWT operation stores the results in W_DWT for 1D waves and M_DWT for higher dimensions.

When working with 1D waves, the transform results are packed such that the higher half of each array contains the detail components and the lower half contains the smooth components and each successive scale is packed in the lower elements. For example, if the source wave contains 128 points then the lowest scale results are stored in elements 64-127, the next scale (power of 2) are stored from 32-63, the following scale from 16-31 etc.

Example

```
Make/O/N=1024 testData=sin(x/100)+gnoise(0.05)
DWT /S/N=20/V=25 testData, smoothedData
```

See Also

For continuous wavelet transforms use the **CWT** operation. See the **FFT** operation.

For further discussion and examples see **Discrete Wavelet Transform** on page III-283.

e

e

The e function returns the base of the natural logarithm system (2.7182818…).

EdgeStats

EdgeStats [*flags*] *waveName*

The EdgeStats operation produces simple statistics on a region of a wave that is expected to contain a single edge. If more than one edge exists, EdgeStats works on the first one found.

Flags

/A=*avgPts*	Determines *startLevel* and *endLevel* automatically by averaging *avgPts* points at centered at *startX* and *endX*. Default is /A=1.
/B=*box*	Sets box size for sliding average. This should be an odd number. If /B=*box* is omitted or *box* equals 1, no averaging is done.

/F=*frac*	Specifies levels 1, 2 and 3 as a fraction of (*endLevel*-*startLevel*):
	level1 = *frac** (*endLevel*-*startLevel*) + *startLevel*
	level2 = 0.5 * (*endLevel*-*startLevel*) + *startLevel*
	level3 = (1-*frac*) * (*endLevel*-*startLevel*) + *startLevel*
	The default value for *frac* is 0.1 which makes level1 the 10% level, level2 the 50% level and level3 the 90% level.
	frac must be between 0 and 0.5.

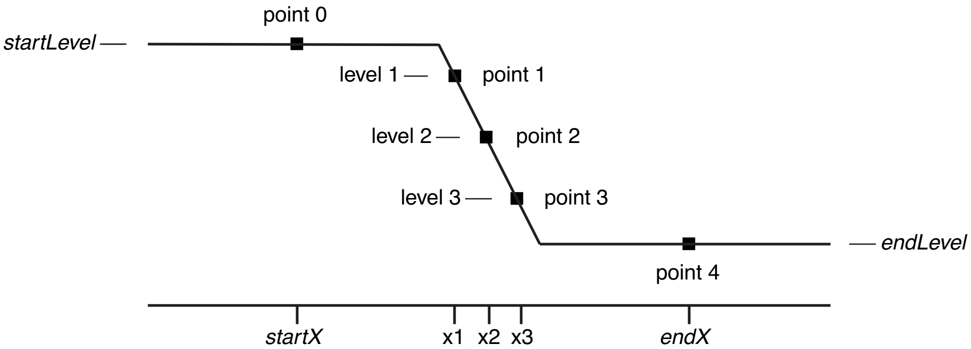
/L=(*startLevel*, *endLevel*)	Sets *startLevel* and *endLevel* explicitly. If omitted, they are determined automatically. See /A.
/P	Output edge locations (see **Details**) are returned as point numbers. If /P is omitted, edge locations are returned as X values.
/Q	Prevents results from being printed in history and prevents error if edge is not found.
/R=(*startX*,*endX*)	Specifies an X range of the wave to search. You may exchange *startX* and *endX* to reverse the search direction.

/R=[*startP*,*endP*] Specifies a point range of the wave to search. You may exchange *startP* and *endP* to reverse the search direction. If /R is omitted, the entire wave is searched.

/T=*dx* Forces search in two directions for a possibly more accurate result. *dx* controls where the second search starts.

Details

The /B=*box*, /T=*dx*, /P, and /Q flags behave the same as for the **FindLevel** operation.



EdgeStats considers a region of the input wave between two X locations, called *startX* and *endX*. *startX* and *endX* are set by the /R=(*startX*,*endX*) flag. If this flag is missing, *startX* and *endX* default to the start and end of the entire wave. *startX* can be greater than *endX* so that the search for an edge can proceed from the "right" to the "left".

The diagram above shows the default search direction, from the "left" (lower point numbers) of the wave toward the "right" (higher point numbers).

The *startLevel* and *endLevel* values define the base levels of the edge. You can explicitly set these levels with the /L=(*startLevel*, *endLevel*) flag or you can let EdgeStats find the base levels for you by using the /A=*avgPts* flag which averages points around *startX* and *endX*.

Given *startLevel* and *endLevel* and a *frac* value (see the /F=*frac* flag) EdgeStats defines level1, level2 and level3 as shown in the diagram above. With the default *frac* value of 0.1, level1 is the 10% point, level2 is the 50% point and level3 is the 90% point.

With these levels defined, EdgeStats searches the wave from *startX* to *endX* looking for level2. Having found it, it then searches for level1 and level3. It returns results via variables described below.

EdgeStats sets the following variables:

`V_flag`	0: All three level crossings were found. 1: One or two level crossings were found. 2: No level crossings were found.
`V_EdgeLoc1`	X location of level1.
`V_EdgeLoc2`	X location of level2.
`V_EdgeLoc3`	X location of level3.
`V_EdgeLvl0`	*startLevel* value.
`V_EdgeLvl1`	level1 value.
`V_EdgeLvl2`	level2 value.
`V_EdgeLvl3`	level3 value.
`V_EdgeLvl4`	*endLevel* value.
`V_EdgeAmp4_0`	Edge amplitude (*endLevel* - *startLevel*).
`V_EdgeDLoc3_1`	Edge width (x distance between point 1 and point 3).
`V_EdgeSlope3_1`	Edge slope (straight line slope from point 1 and point 3).

These X locations and distances are in terms of the X scaling of the named wave unless you use the /P flag, in which case they are in terms of point number.

The EdgeStats operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

See Also

The **FindLevel** operation for use of the /B=*box*, /T=*dx*, /P and /Q flags, and **PulseStats**.

Edit

`Edit` `[`***`flags`***`] [`***`columnSpec`***` [, `***`columnSpec`***`]…][`**`as`**` `***`titleStr`***`]`

The Edit operation creates a table window or subwindow containing the specified columns.

Parameters

columnSpec is usually just the name of a wave. If no *columnSpec*s are given, Edit creates an empty table.

Column specifications are wave names optionally followed by one of the suffixes:

Suffix	Meaning
.i	Index values.
.l	Dimension labels.
.d	Data values.
.id	Index and data values.
.ld	Dimension labels and data values.

If the wave is complex, the wave names may be followed by .real or .imag suffixes. However, as of Igor Pro 3.0, both the real and imaginary columns are added to the table together — you can not add one without the other — so using these suffixes is discouraged.

Historical Note: Prior to Igor Pro 3.0, only 1D waves were supported. We called index values "X values" and used the suffix ".x" instead of ".i". We called data values "Y values" and used the suffix ".y" instead of ".d". For backward compatibility, Igor accepts ".x" in place of ".i" and ".y" in place of ".d".

titleStr is a string expression containing the table's title. If not specified, Igor will provide one which identifies the columns displayed in the table.

Flags

/FG=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the frame guide to which the outer frame of the subwindow is attached inside the host window.

The standard frame guide names are `FL`, `FR`, `FT`, and `FB`, for the left, right, top, and bottom frame guides, respectively, or user-defined guide names as defined by the host. Use `*` to specify a default guide name.

Guides may override the numeric positioning set by /W.

/HIDE=*h* Hides (h = 1) or shows (h = 0, default) the window.

/HOST=*hcSpec* Embeds the new table in the host window or subwindow specified by *hcSpec*.

When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/I Specifies that /W coordinates are in inches.

/K=*k* Specifies window behavior when the user attempts to close it.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.
k=3:	Hides the window.

If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.

/M Specifies that /W coordinates are in centimeters.

/N=*name* Requests that the created table have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen table name.

/W=(*left*,*top*,*right*,*bottom*)

Gives the table a specific location and size on the screen. Coordinates for /W are in points unless /I or /M are specified before /W.

When used with the /HOST flag, the specified location coordinates of the sides can have one of two possible meanings:

When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.

When any value is greater than 1, coordinates are taken to be fixed locations measured in points, or **Control Panel Units** for control panel hosts, relative to the top left corner of the host frame.

Details

You can not change dimension index values shown in a table. Use the Change Wave Scaling dialog or the **SetScale** operation.

If /N is not used, Edit automatically assigns to the table window a name of the form “Table*n*”, where *n* is some integer. In a function or macro, the assigned name is stored in the S_name string. This is the name you can use to refer to the table from a procedure. Use the **RenameWindow** operation to rename the graph.

Examples

These examples assume that the waves are 1D.

```
Edit myWave,otherWave      // 2 columns: data values from each wave
Edit myWave.id             // 2 columns: x and data values
Edit cmplxWave             // 2 columns: real and imaginary data values
Edit cmplxWave.i           // One column: x values
```

The following examples illustrates the use of column name suffixes in procedures when the name of the wave is in a string variable.

```
Macro TestEdit()
   String w = "wave0"
   Edit $w                 // edit data values
   Edit $w.i               // show index values
   Edit $w.id              // index and data values
End
```

Note that the suffix, if any, must not be stored in the string. In a user-defined function, the syntax would be slightly different:

```
Function TestEditFunction()
   Wave w = $"wave0"
   Edit w                  // no $, because w is name, not string
   Edit w.i                // show index values
   Edit w.id               // index and data values
End
```

See Also

The **DoWindow** operation. For a description of how tables are used, see Chapter II-12, **Tables**.

ei

`ei(x)`

The ei function returns the value of the exponential integral *Ei*(*x*):

$$Ei(x) = \quad P\int_{-\infty}^{x} \frac{e^{t}}{t} dt \qquad (x > 0),$$

where *P* denotes the principal value of the integral.

See Also

The **expInt** function.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 228 pp., Dover, New York, 1972.

EllipticE

The ellipticE function returns the complete elliptic integral of the second kind,

$$E(k) = \int_{0}^{1} \frac{\sqrt{1-k^2t^2}\,dt}{\sqrt{1-t^2}},$$

with

$$|k| \le 1.$$

See Also

EllipticK, **GeometricMean**, **JacobiCn**, **JacobiSn**

EllipticK

`EllipticK(x)`

The EllipticK function returns the complete elliptic integral of the first kind,

$$K(k) = \int_{0}^{1} \frac{dt}{\sqrt{\left(1-t^2\right)\left(1-k^2t^2\right)}},$$

with

$$0 \le k \le 1.$$

See Also

EllipticE, **GeometricMean**

End

`End`

The End keyword marks the end of a macro, user function, or user menu definition.

See Also

The **Function** and **Macro** keywords.

EndMacro

`EndMacro`

The EndMacro keyword marks the end of a macro. You can also use End to end a macro.

See Also

The **Macro** and **Window** keywords.

EndStructure

`EndStructure`

The EndStructure keyword marks the end of a Structure definition.

See Also

The **Structure** keyword.

endtry

`endtry`

The endtry flow control keyword marks the end of a try-catch-endtry flow control construct.

See Also

The **try-catch-endtry** flow control statement for details.

enoise

`enoise(`*`num`*` [, `*`RNG`*`])`

The enoise function returns a random value drawn from a uniform distribution having a range of [-*num*, *num*).

enoise returns a complex result if *num* is complex or if it is used in a complex expression. See **Use of enoise With Complex Numbers** on page V-196.

The random number generator is initialized using a seed derived from the system clock when you start Igor. This almost guarantees that you will never get the same sequence twice. If you want repeatable “random” numbers, use **SetRandomSeed**.

The optional parameter *RNG* selects one of three pseudo-random number generators.

If you omit the RNG parameter, enoise uses RNG number 3, named "Xoshiro256**". This random number generator was added in Igor Pro 9.00 and is recommended for all new code. In earlier versions of Igor, the default was 1 (Linear Congruential Generator).

The available random number generators are:

RNG	**Description**
1	Linear Congruential Generator by L'Ecuyer with added Bayes-Durham shuffle. The algorithm is described in *Numerical Recipes* as the function `ran2()`. This option has nearly 23^2 distinct values and the sequence of random numbers has a period in excess of 10^{18}. This RNG is provided for backward compatibility only. New code should use RNG=3 (Xoshiro256**).
2	Mersenne Twister by Matsumoto and Nishimura. It is claimed to have better distribution properties and period of 2^{19937}-1. See Matsumoto, M., and T. Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Trans. on Modeling and Computer Simulation*, *8*, 3-30, 1998. This RNG is provided for backward compatibility only. New code should use RNG=3 (Xoshiro256**).
3	Xoshiro256** by David Blackman and Sebastiano Vigna. It has a period of 2^{256}-1, is 4 dimensionally equidistributed and passes all statistical tests at the time of writing. For technical details, see "Scrambled linear pseudorandom number generators", 2019 (http://vigna.di.unimi.it/ftp/papers/ScrambledLinear.pdf).

Use of enoise With Complex Numbers

enoise returns a complex result if *num* is complex or if it is used in a complex expression.

```
Function DemoENoiseComplex()
   // enoise(rv) in a complex expression returns a complex result
   // and is equivalent to cmplx(enoise(rv),enoise(rv))
   Variable rv = 1                  // A real variable
   SetRandomSeed 1
   Variable/C cv1 = enoise(rv)      // Real parameter, complex result
   SetRandomSeed 1
   Variable/C cv2 = cmplx(enoise(rv),enoise(rv))     // Equivalent
   Print cv1, cv2

   // enoise(cv) is equivalent to cmplx(enoise(real(cv)),enoise(imag(cv)))
   Variable/C cv = cmplx(1,1)       // A complex variable
   SetRandomSeed 1
   Variable/C cv3 = enoise(cv)      // Complex parameter, complex result
   SetRandomSeed 1
   Variable/C cv4 = cmplx(enoise(real(cv)),enoise(imag(cv)))   // Equivalent
   Print cv3, cv4
End
```

Example

```
// Generate uniformly-distributed integers on the interval [from,to] with from<to
Function RandomInt(from, to)
   Variable from, to
   Variable amp = to - from
   return floor(from + mod(abs(enoise(100*amp)), amp+1))
End
```

See Also

SetRandomSeed, **gnoise**, **Noise Functions**, **Statistics**

EqualWaves

`EqualWaves(`*`waveA`*`, `*`waveB`*`, `*`selector`*` [, `*`tolerance`*`])`

TheEqualWaves function compares *waveA* to *waveB*. Each wave can be of any data type. It returns 1 for equality and zero otherwise.

Use the *selector* parameter to determine which aspects of the wave are compared. You can add *selector* values to test more than one field at a time or pass -1 to compare all aspects.

selector	**Field Compared**
1	Wave data
2	Wave data type
4	Wave scaling
8	Data units
16	Dimension units
32	Dimension labels
64	Wave note
128	Wave lock state
256	Data full scale
512	Dimension sizes

If you use the selectors for wave data, wave scaling, dimension units, dimension labels or dimension sizes, EqualWaves will return zero if the waves have unequal dimension sizes. The other selectors do not require equal dimension sizes.

Details

If you are testing for equality of wave data and if the *tolerance* is specified, it must be a positive number. The function returns 1 for equality if the data satisfies:

$$\sum_i \left(waveA[i] - waveB[i]\right)^2 < tolerance.$$

If *tolerance* is not specified, it defaults to 10^{-8}.

If *tolerance* is set to zero and *selector* is set to 1 then the data in the two waves undergo a binary comparison (byte-by-byte).

If *tolerance* is non-zero then the presence of NaNs at a given point in both waves does not contribute to the sum shown in the equation above when both waves contain NaNs at the same point. A NaN entry that is present in only one of the waves is sufficient to flag inequality. Similarly, INF entries are excluded from the tolerance calculation when they appear in both waves at the same position and have the same signs.

If you are comparing wave data (selector =1) and both waves contain zero points, the function returns 1.

The EqualWaves() function comparison of all text fields is case-sensitive.

See Also

The **MatrixOp** operation equal keyword.

erf

erf(*num* [, *accuracy*])

The erf function returns the error function of *num*.

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt.$$

Optionally, *accuracy* can be used to specify the desired fractional accuracy.

In complex expressions the error function is

$$erf(z) = \frac{2z}{\sqrt{\pi}} {}_1F_1\left(\frac{1}{2};\frac{3}{2};-z^2\right),$$

where

$$ {}_1F_1\left(\frac{1}{2};\frac{3}{2};-z^2\right)$$

is the confluent hypergeometric function of the first kind HyperG1F1. In this case the accuracy parameter is ignored.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of (f_{actual} - $f_{returned}$)/f_{actual} be less than 10^{-7}.

For backwards compatibility, in the absence of *accuracy* an alternate calculation method is used that achieves fractional accuracy better than about 2×10^{-7}.

If *accuracy* is present, erf can achieve fractional accuracy better than 8×10^{-16} for *num* as small as 10^{-3}. For smaller *num* fractional accuracy is better than 5×10^{-15}.

Higher accuracy takes somewhat longer to calculate. With *accuracy* set to 10^{-16} erfc takes about 50% more time than with *accuracy* set to 10^{-7}.

See Also

The **erfc**, **erfcw**, **dawson**, **inverseErf**, and **inverseErfc** functions.

erfc

`erfc(`*`num`*` [, `*`accuracy`*`])`

The erfc function returns the complementary error function of *num* (erfc(x) = 1 - erf(x)). Optionally, *accuracy* can be used to specify the desired fractional accuracy.

In complex expressions the complementary error function is

$$erfc(z) = 1 - erfc(z) = 1 - \frac{2z}{\sqrt{\pi}} {}_1F_1(\frac{1}{2},\frac{3}{2},-z^2) \text{ where } {}_1F_1(\frac{1}{2},\frac{3}{2},-z^2)$$

is the confluent hypergeometric function of the first kind HyperG1F1. In this case the accuracy parameter is ignored.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of (f_{actual} - $f_{returned}$)/f_{actual} be less than 10^{-7}.

For backwards compatibility, in the absence of *accuracy* an alternate calculation method is used that achieves fractional accuracy better than 2×10^{-7}.

If *accuracy* is present, erfc can achieve fractional accuracy better than 2×10^{-16} for *num* up to 1. From *num* = 1 to 10 fractional accuracy is better than 2×10^{-15}.

Higher accuracy takes somewhat longer to calculate. With *accuracy* set to 10^{-16} erfc takes about 50% more time than with *accuracy* set to 10^{-7}.

See Also

erf, **erfcw**, **erfcx**, **inverseErfc**, **dawson**

erfcw

`erfcw(z)`

The erfcw is a complex form of the error function defined by

$$\operatorname{erfcw}(z) = \exp[-z^2]\operatorname{erfc}(-iz),$$

where

$$\operatorname{erfc}(z) = \frac{2}{\sqrt{\pi}}\int_z^\infty \exp[-t^2]dt.$$

The function is computed with accuracy of 0.5e-10. It is particularly useful for large |z| where the computation of erfc(*z*) starts encountering numerical instability.

References

1. http://en.wikipedia.org/wiki/Error_function

2. W. Gautschi, "*Efficient Computation of the Complex Error Function*", SIAM J. Numer. Anal. Vol. 7, No. 1, March 1970.

See Also

erf, **erfc**, **erfcx**, **inverseErfc**, **dawson**

erfcx

erfcx(*num*)

The erfcx function returns the scaled complementary error function of num:

```
f(x) = exp(x2)*erfc(x)
```

Details

This implementation is for real numbers only. If you need complex input and output, use:

```
erfcx = Faddeeva(ix)
```

This method has poorer accuracy for negative X and a more limited region of numerical validity than this real implementation.

Computing erfcx using the mathematical definition has a very limited range in which it is accurate, or even valid.

References

Our implementation comes from **this Stack Overflow post**.

The code in that post is based on this paper:

M. M. Shepherd and J. G. Laframboise, "*Chebyshev Approximation of (1 + 2 x) exp(x2) erfc x in 0 ≤ x < ∞.*", Mathematics of Computation, Volume 36, No. 153, January 1981, pp. 249-253

See Also

erf, **erfc**, **erfcw**, **inverseErfc**, **dawson**, **Faddeeva**

ErrorBars

ErrorBars [***flags***] ***traceName***, ***mode*** [***errorSpecification***]

The ErrorBars operation adds or removes error bars to or from the named trace in the specified graph and modifies error bar settings.

The “error bars” are lines that extend from each data point to “caps”. The length of the line (or “bar”) is usually used to bracket a measured value by the amount of uncertainty, or “error” in the measurement.

Parameters

traceName is the name of a trace on a graph (see **Trace Names** on page II-282).

A string containing *traceName* can be used with the $ operator to specify *traceName*.

mode is one of the following keywords:

OFF No error bars.

X Horizontal error bars only.

Y Vertical error bars only.

XY Horizontal and vertical error bars.

BOX [=*fillColor*] Box error bars, optionally with fill color expressed as (r,g,b) or (r,g,b,a). If no color is specified, boxes are not filled. The *fillColor* parameter was added in Igor Pro 8.00.

ELLIPSE={mode, p, alpha}

Plots error ellipses.

You must provide error values via a three-column wave using the ewave=*ew* error specification described below.

mode=0: *ew* contains the standard deviation in X, the standard deviation in Y, and the correlation between X and Y.

mode=1: *ew* contains the variance in X, the variance in Y, and the covariance of X and Y.

p is the probability level represented by the error ellipses. *p*=0.6837, *p*=0.95, and *p*=0.997 respresent one, two, and three standard deviations respectively. Larger values of *p* result in larger ellipses corresponding to higher confidence levels.

alpha is an opacity value in the range of 0 (fully transparent) to 65535 (fully opaque). See **Error Ellipse Color** on page II-305 for details.

The ellipse mode was added in Igor Pro 9.00.

See **Error Ellipses** on page II-305 for further discussion.

NOCHANGE This mode allows you to programmatically change aspects of error bars using flags (see ***Flags*** below) without affecting the mode or error specification.

The NOCHANGE mode was added in Igor Pro 9.00.

SHADE={*options*, *fillMode*, *fgColor*, *bkColor* [, *negFillMode*, *negFgColor*, *negBkColor*]}

SHADE was added in Igor Pro 7.00.

options is reserved for future use and must be zero.

fillMode sets the fill pattern.

n=0:	No fill.
n=1:	Erase.
n=2:	Solid black.
n=3:	75% gray.
n=4:	50% gray.
n=5:	25% gray.
n>=6:	See **Fill Patterns** on page III-498.

fgColor is (r,g,b) or (r,g,b,a). If all zeros including alpha, i.e., (0,0,0,0), then the actual color will be the trace color with an alpha of 20000.

bkColor is used for patterns only and can be simply (0,0,0) for solid fills.

negFillMode is the same as *fillMode* but for negative error shading.

negFgColor is the same as fgColor but for negative error shading.

negBkColor is the same as bkColor but for negative error shading.

The *errorSpecification* , described below, affects only the Y amplitude of error shading.

The X values of the trace to which shading is applied must be monotonic. Results with non-monotonic X values are undefined.

See **Error Shading** on page II-305 for more information and examples. See **Color Blending** on page III-498 for information on the alpha color parameter.

Error Specification Parameters

For any mode other than OFF and NOCHANGE, you must provide an error specification which consists of a keyword followed by zero or more parameters:

pct=*value*

value is the length of an error bar expressed as a percentage of the trace X or Y value.

sqrt

The error bar length is the square root of the trace X or Y value.

const=*value*

value is the length of an error bar independent of the trace X or Y value.

ewave=ew

ew is a three-column wave with error ellipse parameters. Each row represents information for one trace data point. The interpretation of the columns of ew depends on the mode parameter provided with the ELLIPSE keyword:

mode=0: *ew* contains the standard deviation in X, the standard deviation in Y, and the correlation between X and Y.

mode=1: *ew* contains the variance in X, the variance in Y, and the covariance of X and Y.

ew may include a subrange specification as long as it results in effectively a 2D wave with three columns and a row for each trace data point. See **Subrange Display Syntax** on page II-321.

See **Error Ellipses** on page II-305 for further discussion.

wave=(*w1*,*w2*)

Each error bar length is obtained from the corresponding point of wave *w1* for negative bars and from wave *w2* for positive bars.

If you omit *w1* no negative bars are drawn. If you omit *w2* no positive bars are drawn.

You can use subrange syntax for *w1* and *w2*. See **Subrange Display Syntax** on page II-321.

mulWave = *ew*

Each error bar length is computed from a multiplier taken from the corresponding point of a wave *ew*. If your trace wave is tw, the error bar lengths are computed as:

positive error bar length = tw * (ew - 1)

negative error bar length = tw * (1 - 1/ew)

This is appropriate if you have computed geometric mean and standard deviation, as would be appropriate for errors with lognormal distribution. Such error bars appear symmetrical on a log axis.

The values in *e* must be greater than or equal to 1. Values less than 1 result in the error bar not be displayed for that data point. An error value of exactly 1 results in a zero-length error bar.

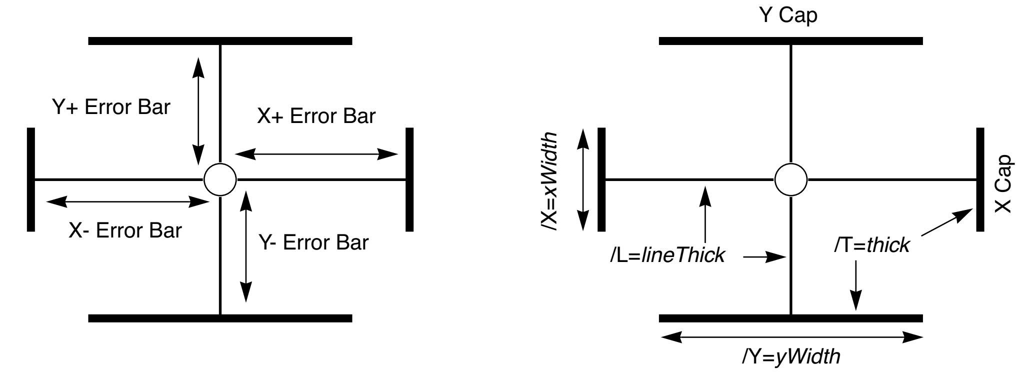
The mulWave keyword was added in Igor Pro 9.00.

nochange

Don't change *errorSpecification* from existing values. nochange allows you to, for instance, change the fill color for the box mode without having to repeat *errorSpecification* for the X and Y errors. nochange was in Igor Pro 8.00.

See the ***Examples*** below for examples using *mode* and *errorSpecification*.

mode and *errorSpecification* control only the lengths of the horizontal and vertical lines (the “bars”) to the “caps”. All other sizes and thicknesses are controlled by the flags.



Sizes controlled by *mode* and *errorSpecification*

Sizes controlled by flag values

XY *mode* Error Bars

Flags

/CLIP=*clip*

/CLIP={*clipH*, *clipV*}

Sets a distance in points by which error bars are allowed to extend beyond the plot area. If you use the first form, the vertical and horizontal values are both set to clip. Use the second form to set horizontal and vertical clipping independently. clipH and clipV default to 2 points. Values may be negative to restrict error bar drawing to an area inside the plot area. /CLIP was added in Igor Pro 9.00.

/L=*lineThick*

Specifies the thickness of both the X and Y error bars drawn from the point on the wave to the caps. If you use the ELLIPSE keyword to draw error ellipses, lineThick sets the thickness of the ellipse outline.

/RGB=*strokeColor*

Sets the color of the lines used to draw the error bars. *strokeColor* is expressed as (r,g,b) or (r,g,b,a). The default color, used if you omit /RGB, is the same as the color of the trace to which the error bars are attached. /RGB was added in Igor Pro 8.00.

If you use the ELLIPSE keyword, strokeColor sets the color of the ellipse outline.

/T=*thick*

Specifies the thickness of both the X and Y error bar “caps”. In box mode, /T sets the thickness of the box outline.

/W=*winName*

Changes error bars in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/X=*xWidth*

Specifies the width (height, actually) of the caps to the left or right of the point.

/Y=*yWidth*

Specifies the width of the caps above or below the point.

The thicknesses and widths are in units of points. The thickness parameters need not be integers. Although only integral thicknesses can be displayed exactly on a standard resolution screen, nonintegral thicknesses are produced properly on high resolution devices. Use /T=0 to completely suppress the caps and /L=0 to completely suppress the lines to the caps.

Details

If a wave specifying error values for *traceName* is shorter than the wave displayed by *traceName* then the last value of the error wave is used for the unavailable points. If a point in an error wave contains NaN (Not a Number) then the half-bar associated with that point is not shown.

Examples

```
// X 10% of wave1, Y is 5% of wave1
ErrorBars wave1, XY pct=10, pct=5
```

```
// X error bars only, square root of wave1
ErrorBars wave1,X sqrt
```

```
// Y error bars only, constant error value = 4.3
ErrorBars wave1,Y const=4.3

// Error box, 10% in horizontal direction, 5% in vertical direction
ErrorBars wave1,BOX pct=10,pct=5

// Error box filled with blue color having 50% alpha (transparency)
// 10% in horizontal direction, 5% in vertical direction
ErrorBars wave1,BOX=(0,0,65535,32767) pct=10,pct=5

// Change the error box fill to red color having 50% alpha
// without changing the way the errors are computed.
ErrorBars wave1,BOX=(65535,0,0,32767) nochange, nochange

// Y error bars only, wave w1 has errors for Y+ bars
// wave w2 has errors for Y- bars
ErrorBars wave1,Y wave=(w1,w2)

// Y error bars only, same wave for both Y+ and Y-.
// Overrides the trace color to make the error bars black.
ErrorBars/RGB=(0,0,0) wave1,Y wave=(w1,w1)

// Y error bars only, no Y+ error bars, wave w2 has errors for Y- bars
ErrorBars wave1,Y wave=(,w2)

// Turns error bars for wave1 off
ErrorBars wave1,OFF
```

Error Ellipse Example

See **Error Ellipse Example** on page II-306.

See Also

Error Bars on page II-304, **Trace Names** on page II-282, **Programming With Trace Names** on page IV-87.

EstimatePeakSizes

EstimatePeakSizes [/B=*baseWave*] [/X=*xWave*] [/E=*bothEdgesWave*] *edgePct*, *maxWidth*, *box*, *npks*, *peakCentersWave*, *peakWave*, *peakAmplitudesWave*, *peakWidthsWave*

The EstimatePeakSizes operation estimates the amplitudes and widths of peaks whose estimated centers are given.

The EstimatePeakSizes operation is used primarily by the Igor Technical Note #20 and its variants.

Parameters

edgePct is the percentage of peak height at which the edge is detected, relative to the baseline. It must be between 1 and 99, and is usually 50.

maxWidth is the maximum width that will be returned in *peakWidthsWave*, in X coordinates.

box is the number of peak values included in the sliding average when smoothing *peakWave* and *baseWave*. If you specify an even number, the next-higher odd number is used.

npks is the number of peaks whose sizes are to be estimated. It must be at least 1.

peakCentersWave must contain the point numbers of the centers of the peaks and have a length of at least npks. The peak sizes are estimated by starting the search for the peak edges from these peak centers. The i-th peak center must be stored in *peakCentersWave*[i] where i ranges from 0 to *npks*-1. The peak center values in *peakCentersWave* must be monotonically increasing or decreasing.

peakWave is the input wave containing the peaks.

peakAmplitudesWave is an output wave that will contain the baseline-corrected peak amplitudes of the peaks. It must have a length of at least *npks*. The i-th peak amplitude is stored in *peakAmplitudesWave*[i].

peakWidthsWave is an output wave that will contain the widths of the peaks in X coordinates. It must have a length of at least *npks*. The i-th peak width is stored in *peakWidthsWave*[i].

Flags

/B=*baseWave*	*baseWave* is subtracted from *peakWave* to compute the derived data which is searched for edges. It must be the same length as *peakWave*.

/E=*bothEdgesWave*	*bothEdgesWave* specifies an already-existing output wave. It must have a length of at least *npks**2. The point coordinates of the i-th peak edges are stored in *bothEdgesWave*[i*2] and *bothEdgesWave*[i*2+1]. If the X values increase with point number, *bothEdgesWave*[i*2] will be greater than *bothEdgesWave*[i*2+1], which may defy expectations.
/X=*xWave*	*xWave* supplies the X coordinates for the corresponding points in *peakWave* and *baseWave*. It must be of the same length as *peakWave* and must be monotonically increasing or decreasing.

See Also

FindPeak

Execute

Execute [**/Z**] ***cmdStr***

The Execute operation executes the contents of *cmdStr* as if it had been typed in the command line.

The most common use of Execute is to call a macro or an external operation from a user-defined function. This is necessary because Igor does not allow you to make such calls directly.

When the /Z flag is used, an error code is placed in V_flag.The error code will be -1 if a missing parameter style macro is called and the user clicks Quit Macro, or zero if there was no error.

Flags

/Z	Errors are not fatal and error dialogs are suppressed.

Details

Because the command line and command buffer are limited to 2500 bytes on a single line, *cmdStr* is likewise limited to a maximum of 2500 executable bytes.

Do not reference local variables in *cmdStr*. The command is not executed in the local environment provided by a macro or user-defined function.

Execute can accept a string expression containing a macro. The string must start with Macro, Proc, or Window, and must follow the normal rules for macros. All lines must be terminated with carriage returns including the last line. The name of the macro is not important but must exist. Errors will be reported except when using the /Z flag, which will assign V_Flag a nonzero number in an error condition.

Examples

It is a good idea to compose the command to be executed in a local string variable and then pass that string to the Execute operation. This prints the string to the history for debugging:

```
String cmd
sprintf cmd, "GBLoadWave/P=%s/S=%d \"%s\"", pathName, skipCount, fileName
Print cmd       // For debugging
Execute cmd
```

```
// Execute with a macro:
Execute "Macro junk(a,b)\rvariable a=1,b=2\r\rprint \"hello from macro\",a,b\rEnd\r"
```

See Also

The Execute Operation on page IV-201 for other uses.

Execute/P

Execute/P [**/Q/Z**] ***cmdStr***

Execute/P is similar to Execute except the command string, *cmdStr*, is not immediately executed but rather is posted to an operation queue. Items in the operation queue execute only when nothing else is happening. Macros and functions must not be running and the command line must be empty.

Flags

/Q Command is not printed in the command line or history area.

/Z No error reporting.

See Also

Operation Queue on page IV-278 for more details on using Execute/P with the operation queue.

ExecuteScriptText

`ExecuteScriptText [`*`flags`*`] `*`textStr`*

The ExecuteScriptText operation passes your text to Apple’s scripting subsystem for compilation and execution or to the Windows command line.

Text produced by the script, including error messages, is returned in the string output variable S_value. On Windows, S_value is set only if you include the /B (execute in background) flag.

Parameters

textStr must contain a valid AppleScript program or Windows command line.

Flags

/B Execute the script in the background task, i.e., leaving Igor as the active application.

If you omit /B on Windows, the program invoked by the script becomes the active program. If you include /B, Igor remains the active program.

/B is ignored on Macintosh and the script is always executed in the background.

/W=*waitTime* waitTime is the maximum time in seconds to wait before ExecuteScriptText returns. /W is accepted on any platform but is supported only on Windows.

/UNQ Removes any leading and trailing double-quote characters from S_value. /UNQ is useful on Macintosh only and has no effect on Windows.

The /UNQ flag was added in Igor Pro 7.00.

/Z Include /Z if you want to handle errors in your code. Omit it if you want ExecuteScriptText to report errors to Igor. See ***ExecuteScriptText Error Handling*** below for details.

ExecuteScriptText Error Handling

If you omit the /Z flag, if the script returns an error, ExecuteScriptText returns the error to Igor and procedure execution stops.

If you include the /Z flag then ExecuteScriptText sets the V_flag output variable to a nonzero value if the script returned an error or to zero if no error occurred. ExecuteScriptText does not report the error to Igor so procedure execution continues.

ExecuteScriptText on Macintosh

Text generated by the script, including error messages, is returned in the S_value output string variable.

Prior to Igor Pro 8.00, S_value containing a text value was typically quoted. This does not appear to be the case starting with Igor Pro 8 which uses more up-to-date system calls. We recommend that you use the /UNQ flag with Igor 7 and later so that you can reliably expect S_value to be unquoted.

The /B flag (execute in background) is ignored.

The /W flag is ignored and ExecuteScriptText does not return until the script is finished.

Prior to Igor Pro 8.00, the current directory was always / and the PATH environment variable was the system default, not the user default (usually PATH=/usr/bin:/bin:/usr/sbin:/sbin.). Now neither of these is guaranteed to be the case. You should not make any assumptions about the current directory or the value of any environment variable. Use full paths to files and programs whenever you need to provide a path.

Macintosh Examples

```
// Macintosh: Convert file.PICT to file.GIF
String script = "tell application \"clip2gif\" "
script += "to save file \"HD:file.PICT\"\r"
ExecuteScriptText/Z script

// Macintosh: Execute a Unix shell command
Function/S DemoUnixShellCommand()
   // Paths must be POSIX paths (using /).
   // Paths containing spaces or other nonstandard characters
   // must be single-quoted. See Apple Techical Note TN2065 for
   // more on shell scripting via AppleScript.
   String unixCmd
   unixCmd = "ls '/Applications/Igor Pro 8 Folder'"

   String igorCmd

   sprintf igorCmd, "do shell script \"%s\"", unixCmd
   Print igorCmd                     // For debugging only

   ExecuteScriptText/UNQ igorCmd
   Print S_value                     // For debugging only
   return S_value
End
```

ExecuteScriptText on Windows

textStr contains the full path to a batch file or the name of an executable file with optional Windows-style path and optional arguments:

```
[path]executableName  [.exe]  [arg1 ]...
```

If you omit path and executableName ends with ".exe" or with no extension, Igor locates the executable by searching first the registry and then along the PATH environment variable. If not found, then the Igor Pro Folder directory is assumed. Here is an example of a command consisting of just the name of an executable:

```
ExecuteScriptText "calc"             // calc.exe, calculator
```

When calling a batch file or other non-*.exe file, supply the full path. If the path (or file name) contains spaces, you must quote the path:

```
ExecuteScriptText "\"C:\\Program Files\\my.bat\""
```

The outer double-quotes are consumed by ExecuteScriptText. The inner double quotes, represented by the \" escape sequence, remain in the command passed to the operating system.

Use the /B flag to run the command in the background, keeping Igor as the active application. Omit /B to allow the program launched by the command to become the active program. When executing a batch file, /B prevents the DOS window from appearing.

If you include /B, output from the command is returned via the S_value output string variable. If you omit /B, S_value is set to "".

The /W=waitTime flag provides a way to time out if a command takes too long. It is useful when calling non-GUI programs only. For GUI programs, ExecuteScriptText returns as soon as the GUI program starts processing events, regardless of the value specified for waitTime.

For non-GUI programs, if you specify a positive value for waitTime, ExecuteScriptText waits up to that many seconds for the command to complete. If the command fails to complete within that period of time, ExecuteScriptText returns an error. If you omit the /W flag or if you pass zero for waitTime, ExecuteScriptText waits until the command completes no matter how long it takes. If the script being executed creates a command window, via cmd.exe, for example, the Windows process completes only when the command window closes, not after the script commands execute.

Windows Examples

```
// Windows: Execute a DOS command in the background
ExecuteScriptText/B "hostname"; Print S_value

// Windows: Open MatLab in background
ExecuteScriptText/B "C:\\Matlab\\bin\\matlab.exe myFile.m"

// Windows: Pass a script to Windows Script Host
ExecuteScriptText/W=5 "WScript.exe \"C:\\Test Script.vbs\""
```

```
// Windows: Execute a batch file and leave the command window open
ExecuteScriptText "cmd.exe /K \"C:\\mybatch.bat\""

// Windows: Execute a DOS command and get output, if any
// ExecuteDOSCommand(command, maxSecondsToWait)
// Executes a DOS command and returns any output text as the function result.
// Returns "" if the DOS command returns no text.
// maxSecondsToWait is the maximum number of seconds to wait for DOS to finish
// the command. If it takes longer than that, an error is generated.
// This function creates files in the Igor Pro User Folder:
//     IgorBatch.bat              Holds command that DOS is to execute
//     IgorBatchOutput.txt        Holds output generated by DOS command, if any
// Example:
//     Print ExecuteDOSCommand("echo %PATH%", 3)
Function/S ExecuteDOSCommand(command, maxSecondsToWait)
   String command                    // e.g., "echo %PATH%"
   Variable maxSecondsToWait         // Error if DOS takes longer than this

   String quoteStr = "\""

   // Get path to batch file in "Igor Pro User Files"
   String dirPath = SpecialDirPath("Igor Pro User Files", 0, 0, 0)
   dirPath = ParseFilePath(5, dirPath, "\\", 0, 0        )  // Convert to Windows path
   String batchFilePath = dirPath + "IgorBatch.bat"
   String batchOutputFilePath = dirPath + "IgorBatchOutput.txt"

   DeleteFile/Z batchOutputFilePath

   // Write DOS command to batch file
   String dosCommand = command + " > " + quoteStr + batchOutputFilePath + quoteStr
   Variable refNum
   Open refNum as batchFilePath
   FBinWrite refNum, dosCommand
   Close refNum

   // Execute batch file
   // The DOS command must complete in the number of seconds specified via /W
   // /C means cmd.exe quits after executing the command
   String text
   sprintf text, "cmd.exe /C \"%s\"", batchFilePath
   ExecuteScriptText/W=(maxSecondsToWait) text

   // Get output
   String result = ""
   Open/R/Z refNum as batchOutputFilePath
   if (V_flag != 0)
      result = ""
      // result = "<No output was generated by batch file>"    // For debugging
   else
      // Read contents of batch file into string
      FStatus refNum
      Variable numBytesInFile = V_logEOF
      result = PadString("", numBytesInFile, 0x20)
      FBinRead refNum, result
      Close refNum
   endif

   return result
End
```

See Also

See **AppleScript** on page IV-263.

exists

exists(*objNameStr*)

The exists function returns a number which indicates if *objNameStr* contains the name of an Igor object, function or operation.

Details

objNameStr can optionally include a full or partial path to the object. If the name does not include a path, exists checks for waves, strings and variables in the current data folder.

objNameStr can optionally include a module name or independent module name prefix such as "ProcGlobal#" to check for the existence of functions. This works for macros as well.

The return values are:

0:	Name not in use, or does not conflict with a wave, numeric variable or string variable in the specified data folder.
1:	Name of a wave in the specified data folder.
2:	Name of a numeric or string variable in the specified data folder.
3:	Function name.
4:	Operation name.
5:	Macro name.
6:	User-defined function name.

exists is not aware of local variables or parameters in user-defined functions, however it is aware of local variables and parameters in macros.

objNameStr is a string or string expression, *not* a name.

Examples

```
// Prints 2 if V_flag exists as a global variable in the current data folder:
Print exists("V_Flag")
```

```
// Prints 5 if a macro named Graph0 exists.
Print exists("ProcGlobal#Graph0")
```

See Also

The **DataFolderExists** and **WaveExists** functions and the **WinType** operation.

exp

exp(*num*)

The exp function returns $e^{num.}$ In complex expressions, *num* is complex, and exp(*num*) returns a complex value.

ExperimentInfo

ExperimentInfo [/Q[=*quiet*]] [keyword=*value* [, keyword=*value* ...]]

The ExperimentInfo operation returns information about the current Igor experiment. The information is returned in the output variable V_Flag and the output string variable S_Value, as described below.

ExperimentInfo was added in Igor Pro 8.00.

To get the name of the current experiment, use **IgorInfo**(1).

Flags

/Q[=*quiet*] If you omit /Q or pass 0 for quiet , ExperimentInfo prints information to the history area of the command window.

If you specify /Q or /Q=1, ExperimentInfo prints nothing to the history area. Information is still returned via V_Flag and S_Value.

Keywords

getRequiredIgorVersion Returns the version of Igor required to open the current experiment via V_Flag. Returns a message explaining the requirement via S_Value.

See **Getting the Required Igor Version** on page V-210 for details.

getLongNameUsage[={*dfRef* , *mask*, *options*, *length*}

Returns information about long object names used in the current experiment via V_Flag and S_Value. This may be of interest because experiments that use long object names (longer than 31 bytes) require Igor Pro 8.00 or later.

V_Flag is set to the number of long names found.

S_Value is set to a description of the long names found. If you omit /Q or specify /Q=0, the description is also printed to the history area of the command window.

All of the parameters are optional. However, if you specify any of them, you must specify all of them.

dfRef specifies the data folder to start from and defaults to root:. It is a full or partial path to the starting data folder. You can specify root: or * for *dfRef* to get the default. You can specify : to get the current data folder.

getLongNameUsage reports on the contents of the data folder specified by *dfRef* . Consequently, the specified data folder itself is not reported even if its name is long.

mask specifies what types of long object names you want to know about. It is a bitwise parameter defined as follows:

Bit 0:	Wave names
Bit 1:	Variable names
Bit 2:	Data folder names
Bit 3:	Target window names
Bit 4:	Symbolic path names

mask defaults to 31 which specifies all of the above.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

options controls various aspects of the returned information. It is a bitwise parameter defined as follows:

Bit 0:	Produce text formatted for reading by humans
Bit 1:	Include headers if bit 0 is set
Bit 2:	Use full paths for waves, variables and data folders
Bit 3:	Use full paths for waves, variables and data folders

options defaults to 15.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

length specifies the length in bytes greater than which a name must be to be included in the output. It defaults to 31. If you specify 0 for *length* then waves are listed regardless of the length of their name.

See **Getting Long Name Usage Info** on page V-210 for details.

getLongDimensionLabelUsage[={*dfRef* , *mask*, *options*, *length*}

Returns information about waves with long dimension labels in the current experiment via V_Flag and S_Value. This may be of interest because such experiments require Igor Pro 8.00 or later.

V_Flag is set to the number of waves with long dimension labels found.

S_Value is set to a description of the waves found. If you omit /Q or specify /Q=0, the description is also printed to the history area of the command window.

All of the parameters are optional. However, if you specify any of them, you must specify all of them.

dfRef specifies the data folder to start from and defaults to root:. It is a full or partial path to the starting data folder. You can specify root: or * for *dfRef* to get the default. You can specify : to get the current data folder.

mask determines if data folders are listed along with waves with long dimension labels to provide context. It is a bitwise parameter defined as follows:

Bit 2:	Include data folder names

All other bits are ignored.

mask defaults to 4 which means that bit 2 is set and data folders are listed, but they are listed only if bit 0 of the options parameter is also set.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

options controls various aspects of the returned information. It is a bitwise parameter defined as follows:

Bit 0:	Produce text formatted for reading by humans
Bit 1:	Include headers if bit 0 is set
Bit 2:	Use full paths for waves, variables and data folders
Bit 3:	Use full paths for waves, variables and data folders

options defaults to 15.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

length specifies the length in bytes greater than which a dimension label must be for the wave be included in the output. It defaults to 31. If you specify 0 for *length* then waves are listed regardless of the length of their dimension labels.

Getting the Required Igor Version

The getRequiredIgorVersion keyword returns the version of Igor required to open the current experiment via V_Flag. If no particular version is required, it returns 0. If V_Flag is non-zero, S_Value contains an explanation of why a particular version of Igor is required.

As used here, the concept of the version of Igor required to open the current experiment is very narrow. If getRequiredIgorVersion returns a non-zero value via V_Flag, this means that the current experiment can not be opened at all in versions of Igor earlier than specified by V_Flag. Currently the only situation that getRequiredIgorVersion flags is the use of long object names, specifically long wave, variable, data folder, target window, and symbolic path names, or long wave dimension labels. If the experiment uses a long name or long dimension label, the experiment can not be opened in versions of Igor prior to 8.00, and getRequiredIgorVersion sets V_Flag to 8.00. See **Long Object Names** on page III-502 and **Long Dimension Labels** on page II-94 for background information.

If getRequiredIgorVersion returns zero, it means that the current experiment can be opened in earlier versions of Igor. However, it does not guarantee that the experiment will open without errors. For example, if you use a function or operation added in Igor Pro 8.00 in a procedure, you can open the experiment in earlier versions, but you will get a compile error when you do so. getRequiredIgorVersion does not flag the use of all Igor features that require a particular version. It flags only the specific situation explained in the previous paragraph.

Getting Long Name Usage Info

For background information on long object names, see **Long Object Names** on page III-502.

Experiments that use long object names (longer than 31 bytes), require Igor Pro 8.00 or later. You can use the ExperimentInfo operation with the getLongNameUsage keyword to determine what long object names, if any, the current experiment uses. This will help if you want to modify the experiment so it can be opened in an earlier version of Igor.

The getLongNameUsage keyword reports information about long wave, variable, data folder, target window, and symbolic path names. It does not report information about long axis, annotation, control, special character, or XOP names or about long dimension labels.

getLongNameUsage returns information via the V_Flag and S_Value output variables and, if you omit /Q or specify /Q=0, it prints information to the history area of the command window.

In human-readable mode (bit 0 of options parameter set), if you include data folders in the output (bit 2 of mask parameter set), ExperimentInfo generates output for each data folder, whether its name is long or not, and uses indentation. This allows you to see any long names in the context of the entire data folder hierarchy.

If human-readable mode is off, ExperimentInfo generates output for a given data folder only if its name is long. This mode is intended for use by code rather than reading by humans. In this mode, the hierarchy (i.e., which data object exists in which data folder) is discernible only if you call for full paths.

If you specify a starting data folder using the *dfRef* parameter, and if human-readable mode is off, ExperimentInfo does not generate output for the starting data folder name, even if it is long. This is because ExperimentInfo generates output for the contents of the starting data folder.

Liberal names are quoted if you specify full paths (bit 1 of options parameter set) but unquoted if you request simple names only. The reason for not quoting simple names is explained under **Accessing Global Variables and Waves Using Liberal Names** on page IV-68.

GetLongNameUsage Examples

```
// Display long names for all data objects (waves, variables, data folders)
// and for target windows an symbolic paths.
ExperimentInfo getLongNameUsage

// Print number of long names uses and nothing else
ExperimentInfo/Q getLongNameUsage; Print V_Flag

// More specialized tests - paste the following into the procedure window
// of an experiment that uses long names and execute DemoGetLongNamesUsage()

Constant kLongNamesLength = 31
Constant kHumanMask = 1              // Human mode
Constant kHeadersMask = 2            // Include headers
Constant kFullPathsMask = 4          // Use full paths
Constant kRecursiveMask = 8          // Recursive

Function DemoGetLongNamesUsage()
   Print "=== Data folders, human, names only ==="
   ExperimentInfo getLongNameUsage={*, 4, kHumanMask|kRecursiveMask, kLongNamesLength}
   Printf "\r"

   Print "=== Data folders, human, full paths ==="
   ExperimentInfo getLongNameUsage={*, 4, kHumanMask|kFullPathsMask | kRecursiveMask,
   kLongNamesLength}
   Printf "\r"

   Print "=== Data folders, non-human, names only ==="
   ExperimentInfo getLongNameUsage={*, 4, 0|kRecursiveMask, kLongNamesLength}
   Printf "\r"

   Print "=== Data folders, non-human, full paths ==="
   ExperimentInfo getLongNameUsage={*, 4, kFullPathsMask|kRecursiveMask,
   kLongNamesLength}
   Printf "\r"

   Print "=== All data objects, human, names only ==="
   ExperimentInfo getLongNameUsage={*, 7, kHumanMask|kRecursiveMask, kLongNamesLength}
   Printf "\r"

   Print "=== All data objects, human, full paths ==="
   ExperimentInfo getLongNameUsage={*, 7, kHumanMask|kFullPathsMask | kRecursiveMask,
   kLongNamesLength}
   Printf "\r"

   Print "=== All data objects, non-human, names only ==="
   ExperimentInfo getLongNameUsage={*, 7, 0|kRecursiveMask, kLongNamesLength}
   Printf "\r"

   Print "=== All data objects, non-human, full paths ==="
   ExperimentInfo getLongNameUsage={*, 7, kFullPathsMask|kRecursiveMask,
   kLongNamesLength}
   Printf "\r"
   Print "=== All data objects, human, headers, names only ==="
   int flags = kHumanMask | kHeadersMask | kRecursiveMask
   ExperimentInfo getLongNameUsage={*, 7, flags, kLongNamesLength}
   // Printf "\r"      // Skip this because headers mode prints trailing blank line
End
```

See Also

ExperimentModified, **IgorInfo**, **Long Object Names** on page III-502

ExperimentModified

`ExperimentModified [`*`newModifiedState`*`]`

The ExperimentModified operation gets and optionally sets the modified (save) state of the current experiment.

Use this command to prevent Igor from asking you to save the current experiment after you have made changes you do not need to save or, conversely, to force Igor to ask about saving the experiment even though Igor would not normally do so.

The variable V_flag is always set to the experiment-modified state that was in effect before the ExperimentModified command executed: 1 for modified, 0 for not modified.

Parameters

If *newModifiedState* is present, it sets the experiment-modified state as follows:

newModifiedState = 0:	Igor will not ask to save the experiment before quitting or opening another experiment, and the Save Experiment menu item will be disabled.
newModifiedState = 1:	Igor will ask to save the experiment before quitting or opening another experiment, and the Save Experiment menu item will be enabled.

If *newModifiedState* is omitted, the state of experiment-modified state is not changed.

Details

Executing ExperimentModified 0 on the command line will not work because the command will be echoed to the history area, marking the experiment as modifed. Use the command in a function or macro that does not echo text to the history area.

Examples

The /Q flag is vital: it suppresses printing into the history area which would mark the experiment as modified again.

```
Menu "File"
   "Mark Experiment Modified",/Q,ExperimentModified 1    // Enables "Save Experiment"
   "Mark Experiment Saved",/Q,ExperimentModified 0       // Disables "Save Experiment"
End
```

See Also

SaveExperiment, **ExperimentInfo**, **Menu Definition Syntax** on page IV-126.

expInt

`expInt(`*`n`*`, `*`x`*`)`

The expInt function returns the value of the exponential integral $E_n(x)$:

$$E_n(x) = \quad P\int_1^\infty \frac{e^{-xt}}{t^n}dt \qquad (x > 0; n = 0, 1, 2\ldots).$$

See Also

ei, **ExpIntegralE1**

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

ExpIntegralE1

`ExpIntegralE1(z)`

The ExpIntegralE1(*z*) function returns the exponential integral of *z*.

If *z* is real, a real value is returned. If *z* is complex then a complex value is returned.

The ExpIntegralE1 function was added in Igor Pro 7.00.

Details

The exponential integral is defined by

$$E_1(z) = \int_z^{\infty} \frac{e^{-t}}{t} dt, \qquad (|\arg(z)| < \pi).$$

References

Abramowitz, M., and I.A. Stegun, "Handbook of Mathematical Functions", Dover, New York, 1972. Chapter 5.

See Also

expInt, **CosIntegral**, **SinIntegral**, **hyperGPFQ**

expNoise

`expNoise(`*`b`*`)`

The expNoise function returns a pseudo-random value from an exponential distribution whose average and standard deviation are *b* and the probability distribution function is

$$f(x) = \frac{1}{b}\exp\left(-\frac{x}{b}\right).$$

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable “random” numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-390.

Chapter III-12, **Statistics** for a function and operation overview.

ExportGizmo

`ExportGizmo [`*`flags`*`]` ***`keyword`*** **`[=`*`value`*`]`**

The ExportGizmo operation is obsolete but is still partially supported for partial backward compatibility.

You can export Gizmo graphics using File→Save Graphics which generates a **SavePICT** command. The ExportGizmo operation is only partially supported. It can export to the clipboard or to an Igor wave and it can print but it can no longer export to a file. Use SavePICT instead.

Documentation for the ExportGizmo operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "ExportGizmo"
```

Extract

`Extract [`*`type flags`*`][/INDX/O]` ***`srcWave, destWave, LogicalExpression`***

The Extract operation finds data in *srcWave* wherever *LogicalExpression* evaluates to TRUE and stores the matching data sequentially in *destWave*, which will be created if it does not already exist.

Parameters

srcWave is the name of a wave.

destWave is the name of a new or existing wave that will contain the result.

LogicalExpression can use any comparison or logical operator in the expression.

Flags

/FREE Creates a free destWave (see **Free Waves** on page IV-91).

/FREE is allowed only in functions and only if a simple name or wave reference structure field is specified for *destWave*.

/INDX Stores the index in *destWave* instead of data from *srcWave*.

/O Allows *destWave* to be the same as *srcWave* (overwrite source).

Type Flags *(used only in functions)*

Extract also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when it needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-73 and **WAVE Reference Type Flags** on page IV-74 for a complete list of type flags and further details.

Details

srcWave may be of any type including text.

destWave has the same type as *srcWave*, but it is always one dimensional. With /INDX, the *destWave* type is set to unsigned 32-bit integer and the values represent a linear index into *srcWave* regardless of its dimensionality.

Example

```
Make/O source= x
Extract/O source,dest,source>10 && source<20
print dest
```

Prints the following to the history area:

```
  dest[0]= {11,12,13,14,15,16,17,18,19}
```

See Also

The **Duplicate** operation.

factorial

`factorial(`*`n`*`)`

The factorial function returns *n*!, where *n* is assumed to be a positive integer.

Note that while factorial is an integer-valued function, a double-precision number has 53 bits for the mantissa. This means that numbers over 2^{52} will be accurate to about one part in about $2x10^{16}$. Values of *n* greater than 170 result in overflow and return Inf.

If you encounter overflow you can use the **APMath** operation to obtain the result. For example:

```
• Print factorial(1000)
  inf
• APMath/V result = factorial(1000)
  4.02387260077093773543702433923003985719374864210715E+2567
```

Faddeeva

`Faddeeva(`*`z`*`)`

Faddeeva computes the Faddeeva function, also commonly denoted as "w", using an approximation that has, as described by the author, "accuracy is typically at at least 13 significant digits". Both the input and the output are complex numbers.

The Faddeeva function was added in Igor Pro 8.00.

The Faddeeva function is the basis of the Voigt function, implemented in Igor as **VoigtFunc**. VoigtFunc(X, Y) = real(Faddeeva(cmplx(X, Y)).

References

The code used to compute the VoigtFunc was written by Steven G. Johnson of MIT. You can learn more about it at http://ab-initio.mit.edu/Faddeeva.

See Also

VoigtPeak, **VoigtFunc**, **Built-in Curve Fitting Functions** on page III-206

FakeData

FakeData(*waveName*)

The FakeData function puts fake data in the named wave, which must be single-precision float. This is useful for testing things that require changing data before you have the source for the eventual real data. FakeData can be useful in a background task expression.

The FakeData function is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

Examples

```
Make/N=200 wave0; Display wave0
SetBackground FakeData(wave0)        // define background task
CtrlBackground period=60, start      // start background task
// observe the graph for a while
CtrlBackground stop                  // stop the background task
```

FastGaussTransform

FastGaussTransform [***flags***] ***srcLocationsWave***, ***srcWeightsWave***

The FastGaussTransform operation implements an efficient algorithm for evaluating the discrete Gauss transform, which is given by

$$G(y_j) = \sum_{i=0}^{N-1} q_i \exp\left(-\frac{\left\| y_j - x_i \right\|^2}{h} \right),$$

where ***G*** is an M-dimensional vector, ***y*** is an N-dimensional vector representing the observation position, {$\boldsymbol{q}_i$} are the M-dimensional weights, {$\boldsymbol{x}_i$} are N-dimensional vectors representing source locations, and *h* is the Gaussian width. The wave M_FGT contains the output in the current data folder.

Flags

/AERR=*aprxErr*	Sets the approximate error, which determines how many terms of the Taylor expansion of the Gaussian are used by the calculation. Default value is 1e-5.
/WDTH=*h*	Sets the Gaussian width. Default value is 1.
/OUTW=*locWave*	Specifies the locations at which the output is computed. *locWave* must have the same number of columns as *srcLocationsWave*. The other /OUT flags are mutually exclusive; you should use only one at any time.
/OUT1={*x1*,*nx*,*x2*}	
/OUT2={*x1*,*nx*,*x2*,*y1*,*ny*,*y2*}	
/OUT3={*x1*,*nx*,*x2*,*y1*,*ny*,*y2*,*z1*,*nz*,*z2*}	
	Specifies gridded output of the required dimension. In each case you set the starting and ending values together with the number of intervals in that dimension. You cannot specify an output that does not match the dimensions of the input source.
/Q	No results printed in the history area.
/RX=*rx*	Sets the maximum radius of any cluster. The clustering algorithm terminates when the maximum radius is less than *rx*. Without /RX, the maximum radius is the same as the maximum radius encountered.
/RY=*ry*	Sets the upper bound for the distance between an observation point and a cluster center for which the cluster contributes to the transform value. The default value for *ry* is 4 times the Gaussian width as specified by the /WDTH flag.

/TET=*nTerms* Sets the number of terms in the Taylor expansion. Use /TET to set the number of terms and bypass the default error estimate, which is estimated from the approximate error value (/AERR).

/Z No error reporting.

Details

The discrete Gauss transform can be computed as a direct sum. An exact calculation is practical only for moderate number of sources and observation points and for low spatial dimensionality. With increasing dimensionality and increasing number of sources it is more efficient to take advantage of some properties of the Gaussian function. The FastGaussianTransform operation does so in two ways: It first arranges the sources in N-dimensional spatial clusters so that it is not necessary to compute the contributions of all source points that belong to remote clusters (see **FPClustering**). The second component of the algorithm is an approximation that factorizes the sum into a factor that depends only on source points and a factor that depends only on observation points. The factor that depends only on source points is computed only once while the factor that depends on observation points is evaluated once for each observation point.

The trade-off between computation efficiency and accuracy can be adjusted using multiple parameters. By default, the operation calculates the number of terms it needs to use in the Taylor expansion of the Gaussian. You can modify the default approximate error value using /AERR or you can directly set the number of terms in the expansion using /TET.

FastGaussianTransform supports calculations in dimensions that may exceed the maximum allowed wave dimensionality. *srcLocationsWave* must be a 2D, real-valued single- or double-precision wave in which each row corresponds to a single source position and columns represent the components in each dimension (e.g., a triplet wave would represent 3D source locations). *srcWeightsWave* must have the same number of rows as *srcLocationsWave* and it must be a real-valued single- or double-precision wave. In most applications *srcWeightsWave* will have a single column so that the output ***G*** will be scalar. However, if *srcWeightsWave* has multiple columns than ***G*** is a vector. This can be handy if you need to test multiple sets of coefficients at one time. If you specify observation points using /OUTW then *locWave* must have the same number of columns as *srcLocationsWave* (the number of rows in the output is arbitrary). The operation does not support wave scaling.

See Also

The **CWT**, **FFT**, **ImageInterpolate**, **Loess**, and **FPClustering** operations.

References

Yang, C., R. Duraiswami, and L. Davis, Efficient Kernel Machines Using the Improved Fast Gauss Transform, *Advances in Neural Information Processing Systems 16*, 2004.

FastOp

FastOp *destWave* = *prod1* [**± *prod2*** [**± *prod3***]]

FastOp *destWave* += *prod1* [**± *prod2***]

The FastOp operation can be used to get improved speed out of certain wave assignment statements. The syntax was designed so that you can simply insert `FastOp` in front of any wave assignment statement that meets the syntax requirements. The += syntax was added in Igor Pro 9.00.

Parameters

destWave An existing destination wave for the assignment expression. An error will be reported at runtime if the waves are not all the same length or number type.

prod1, *prod2*, *prod3* Products with the following formats:

constexpr * wave1 * wave2*

or

*constexpr * wave1 / wave2*

constexpr may be a literal numeric constant or a constant expression in parentheses. Such expressions are evaluated only once.

Any component in a prod expression may be omitted.

Flags

/C	The /C flag is obsolete in Igor Pro 9.00 and later and is ignored. For floating point waves, the type of expressions, real or complex, is determined by the type of the destination wave. In Igor Pro 8 and before /C is required when using a complex destination wave. FastOp supports real and complex for floating point waves but only real for integer waves.

Details

FastOp supports real and complex for floating point waves. It supports real only for integer waves.

If your waves are complex, make sure to use WAVE/C when declaring wave references in user-defined functions.

Certain combinations, listed in the following table, are evaluated using faster optimized code rather than more general but slower generic code. SP means "single-precision floating point" and DP means "double-precision floating point".

Statement	Optimized Data Types
dest = *0*	All
dest = *waveA*	All
dest += *C0* * *waveA*	SP and DP, real and complex
dest = *dest* + *C0* * *waveA*	SP and DP, real and complex
dest = *dest* + *waveA* * *C0*	SP and DP, real and complex
dest = *C0*	SP and DP and integer, real only
dest = *waveA* +*C1*	SP and DP and integer, real only
dest = *C0* * *waveA* + *C1*	SP and DP, real only
dest = *waveA* + *waveB* + *C2*	Integer only, real only

In the statements above, pluses may be minuses and the trailing constant (*C0, C1, C2*) may be omitted.

Note: Except for the optimized cases listed above which use integer calculations for integer waves, integer results are evaluated using double precision intermediate values. This limits precision for 64-bit integer waves to 53 bits.

Speedups generally range from 10 to 40 times faster than the equivalent statement with FastOp removed. The speedup is dependent on the computer and on the length of the waves with the greatest improvement occurring for waves with 1000 to 100,000 points.

Examples

Valid expressions:

```
FastOp dest = 3
FastOp dest = waveA + waveB
FastOp dest = 0.5*waveA + 0.5*waveB
FastOp dest = waveA*waveB
FastOp dest = (2*3)*waveA + 6
FastOp dest = (locvar)*waveA
FastOp dest += waveA + waveB
```

Expressions that are **not** valid:

```
FastOp dest = 3*4
FastOp dest = (waveA + waveB)
FastOp dest = waveA*0.5 + 0.5*waveB
FastOp dest = waveA*waveB/2
FastOp dest = 2*3*waveA + 6
FastOp dest = locvar*waveA
FastOp dest += waveA + waveB + waveC
```

See Also

MatrixOp, **MatrixMultiply** and **MatrixMultiplyAdd** for additional efficient matrix operations.

MultiThread

faverage

`faverage(`*`waveName`*` [, `*`x1`*`, `*`x2`*`])`

The faverage function returns the trapezoidal average value of the named wave from x=*x1* to x=*x2*.

If your data are in the form of an XY pair of waves, see **faverageXY**.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If *x1* or *x2* are not within the X range of *waveName*, faverage limits them to the nearest X range limit of *waveName*.

faverage returns the area divided by (*x2*-*x1*). In other words, the X scaling of *waveName* is eliminated when computing the average.

If any Y values in the specified X range are NaN, faverage returns NaN.

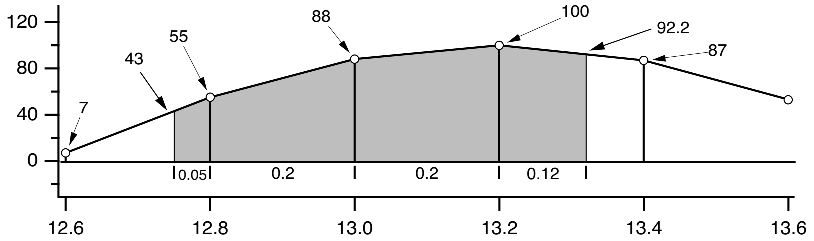
Unlike the **area** function, reversing the order of *x1* and *x2* does *not* change the sign of the returned value.

The faverage function is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

The faverage function returns a complex result for a complex inpt wave. The real part of the result is the average of the real components in the input wave and the imaginary part of the result is the average of the imaginary components.

Examples

Comparison of area, faverage and mean functions over interval (12.75,13.32)



area(wave,12.75,13.32)	= 0.05 · (43+55) / 2
	+ 0.20 · (55+88) / 2
	+ 0.20 · (88+100) / 2
	+ 0.12 · (100+92.2) / 2
	= 47.082
		// first trapezoid
		// second trapezoid
		// third trapezoid
		// fourth trapezoid

faverage(wave,12.75,13.32)	= area(wave,12.75,13.32) / (13.32-12.75) = 47.082/0.57 = 82.6	
mean(wave,12.75,13.32)	= (55+88+100+87)/4 = 82.5	

See Also

Integrate, **area**, **areaXY**, **faverageXY** and **Poly2D Example 3**

faverageXY

`faverageXY(`*`XWaveName`*`, `*`YWaveName`*` [, `*`x1`*`, `*`x2`*`])`

The faverageXY function returns the trapezoidal average value of *YWaveName* from x=*x1* to x=*x2*, using X values from *XWaveName*.

This function operates identically to faverage, except that it uses an XY pair of waves for X and Y values and it does not work with complex waves.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If *x1* or *x2* are not within the X range of *XWaveName*, faverageXY limits them to the nearest X range limit of *XWaveName*.

faverageXY returns the area divided by (*x2* -*x1*).

If any values in the X range are NaN, faverageXY returns NaN.

Reversing the order of *x1* and *x2* does not change the sign of the returned value.

The values in *XWaveName* may be increasing or decreasing. faverageXY assumes that the values in *XWaveName* are monotonic. If they are not monotonic, Igor does not complain, but the result is not meaningful. If any X values are NaN, the result is NaN.

The faverageXY function is not multidimensional aware. See Chapter II-6, **Multidimensional Waves** for details on multidimensional waves, particularly **Analysis on Multidimensional Waves** on page II-95.

See Also

Integrate, **area**, **areaXY**, **faverage** and **Poly2D Example 3**

FBinRead

FBinRead [***flags***] ***refNum***, ***objectName***

The FBinRead operation reads binary data from the file specified by *refNum* into the named object.

For simple applications of loading binary data into numeric waves you may find the GBLoadWave operation simpler to implement.

Parameters

refNum is a file reference number from the Open operation used to open the file.

objectName is the name of a wave, numeric variable, string variable, or structure.

Flags

/B[=*b*] Specifies file byte ordering.

b=0:	Native (same as no /B).
b=1:	Reversed (same as /B).
b=2:	Big-endian (Motorola).
b=3:	Little-endian (Intel).

/F=*f* Controls the number of bytes read and how the bytes are interpreted.

f=0:	Native binary format of the object (default).
f=1:	Signed byte; one byte.
f=2:	Signed 16-bit word; two bytes.
f=3:	Signed 32-bit word; four bytes.
f=4:	32-bit IEEE floating point; four bytes.
f=5:	64-bit IEEE floating point; eight bytes.
f=6:	64-bit integer; eight bytes. Requires Igor Pro 7.00 or later.

/U Integer formats (/F=1, 2, or 3) are unsigned. If /U is omitted, integers are signed.

Details

If *objectName* is the name of a string variable then /F does not apply. The number of bytes read is the number of bytes in the string *before* the FBinRead operation is called. You can use the **PadString** function to set the size of a string.

The binary format that FBinRead uses for a numeric variable depends on the /F flag. If you omit /F, the native data type of the variable, which is 8-byte double-precision floating point, is used. So, when reading into a real numeric variable, depending on /F, FBinRead reads 1, 2, 4, or 8 bytes from the file, converts those bytes to double-precision floating point if necessary, and stores the resulting value in the variable. When reading into a complex numeric variable, this process is repeated twice, once for the real part and once for the imaginary part.

Reading real waves works like reading real variables except that a real wave has multiple elements each of which is 1, 2, 4, or 8 bytes depending on the wave's data type. For each element of a real wave, FBinRead reads the number of bytes implied by /F or by the wave's native data type, converts those bytes to the wave's data type if necessary, and stores the resulting value in the corresponding wave element. When reading into a complex wave, this process is repeated twice, once for the real part of each element and once for the imaginary part.

Reading structures is different. The /F flag has no effect. FBinReads reads the number of bytes required to fill the structure which depends on the sizes of the individual fields and the fact that Igor uses 2-byte structure alignment. After the bytes are read from the file into the structure, FBinRead byte-swaps the individual fields if you include the /B flag.

The FBinRead operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

See Also

FBinWrite, **Open**, **FGetPos**, **FSetPos**, **FStatus**, **GBLoadWave**

FBinWrite

`FBinWrite` `[`***`flags`***`]` ***`refNum, objectName`***

The FBinWrite operation writes the named object in binary to a file.

Parameters

refNum is a file reference number from the **Open** operation used to open the file.

objectName is the name of a wave, numeric variable, string variable, or structure.

Flags

/B[=*b*] Specifies file byte ordering.

b=0:	Native (same as no /B).
b=1:	Reversed (same as /B).
b=2:	Big-endian (Motorola).
b=3:	Little-endian (Intel).

/F=*f* Controls the number of bytes written and how the bytes are formatted.

f=0:	Native binary format of the object (default).
f=1:	Signed byte; one byte.
f=2:	Signed 16-bit word; two bytes.
f=3:	Signed 32-bit word; four bytes.
f=4:	32-bit IEEE floating point; four bytes.
f=5:	64-bit IEEE floating point; eight bytes.
f=6:	64-bit integer; eight bytes. Requires Igor Pro 7.00 or later.

/P Adds an IgorBinPacket to the data. This is used for PPC or Apple event result packets (*refNum* = 0) and is not normally of use when writing to a file.

/U Integer formats (/F=1, 2, or 3) are unsigned. If /U is omitted, integers are signed.

Details

A zero value of *refNum* is used in conjunction with Program-to-Program Communication (PPC) or Apple events (*Macintosh*) or **ActiveX Automation** (*Windows*). The data that would normally be written to a file is appended to the PPC or Apple event or ActiveX Automation result packet.

If the object is a string variable then /F doesn’t apply. The number of bytes written is the number of bytes in the string.

The binary format that FBinWrite uses for numeric variables or waves depends on the /F flag. If no /F flag is present, FBinWrite uses the native binary format of the named object.

Byte ordering refers to the order in which a multibyte datum is written to a file. For example, a 16-bit word (sometimes called a “short”) consists of a high-order byte and a low-order byte. Under big-endian byte ordering, which is commonly used on Macintosh, the high-order byte is written to the file first. Under little-endian byte ordering, which is commonly used on Windows, the low-order byte is written to the file first.

FBinWrite will write an entire structure to a disk file. The individual fields of the structure will be byte-swapped if the /B flag is designated.

The FBinWrite operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

See Also

FBinRead, **Open**, **FGetPos**, **FSetPos**, **FStatus**, **GBLoadWave**

FetchURL

FetchURL(*urlStr*)

The FetchURL function returns a string containing the server's response to a request to get the contents of the URL specified by *urlStr*. If *urlStr* contains a URL that uses the file:// scheme, the contents of the local file is returned.

Parameters

urlStr is a string containing the URL to retrieve. You can include a username, password, and server port number as part of the URL.

FetchURL expects that *urlStr* has been percent-encoded if it contains reserved characters. See **Percent Encoding** on page IV-268 for additional information on when percent-encoding is necessary and how to do it.

See **URLs** on page IV-267 for details about how to correctly specify the URL.

FetchURL supports only the http://, https://, ftp://, and file:// schemes. See **Supported Network Schemes** on page IV-268 for details.

There are two special values of *urlStr* that can be used to get information about the network library that Igor uses. The keyword=value pairs returned when *urlStr* is "curl_version_info" may be useful to programmers in that the features and protocols available in the library are specified.

```
FetchURL("curl_version")
FetchURL("curl_version_info")
```

Details

If FetchURL encounters an error, it returns a NULL string. You should check for errors before using the returned string. In a user-defined function, use the **GetRTError** function.

```
String urlStr = "http://www.badserver"
String response = FetchURL(urlStr)
Variable error = GetRTError(1)       // Check for error before using response
if (error != 0)
   // FetchURL produced an error
   // so don't try to use the response.
endif
```

Limitations

It is possible for FetchURL to return a valid server response even though the URL you requested does not exist on the server or requires a username and password that you did not provide. In this situation, the response returned by the server will usually be a web page stating that the page was not found or another error message. You can check for this kind of error in your own code by examining the response.

FetchURL does not support advanced features such as network proxies, file or data uploads, setting the timeout period, or saving the server's response directly to a file. When using the http:// scheme, only the GET method is supported. This means that you cannot use FetchURL to submit form data to a web server that requires using the http POST method. Use the **URLRequest** operation if you need any of these features.

Igor Pro is not capable of displaying the contents of a URL in a rendered form like a web browser.

Examples

```
// Retrieve the contents of the WaveMetrics home page.
String response
```

```
response = FetchURL("http://www.wavemetrics.com")

// Get a binary image file from a web server and then
// save the image to a file on the desktop.
String url = "http://www.wavemetrics.net/images/tbg.gif"
String imageBytes = FetchURL(url)
Variable error = GetRTError(1)
if (error != 0)
   Print "Error downloading image."
else
   Variable refNum
   String localPath = SpecialDirPath("Desktop", 0, 0, 0) + "tbg.gif"
   Open/T=".gif" refNum as localPath
   FBinWrite refNum, imageBytes
   Close refNum
endif
```

See Also

FTPDownload, **URLEncode**, **URLRequest**

Network Communication on page IV-267, **Network Connections From Multiple Threads** on page IV-271.

FFT

`FFT` `[`***`flags`***`]` ***`srcWave`***

The FFT operation computes the Discrete Fourier Transform of *srcWave* using a multidimensional prime factor decomposition algorithm. By default, *srcWave* is overwritten by the FFT.

Output Wave Name

For compatibility with earlier versions of Igor, if you use FFT with no flags or with just the /Z flag, the operation overwrites *srcWave*.

If you use any flag other than /Z, FFT uses default output wave names: W_FFT for a 1D FFT and M_FFT for a multidimensional FFT.

We recommend that you use the /DEST flag to make the output wave explicit and to prevent overwriting *srcWave*.

Flags

/COLS — Computes the 1D FFT of 2D *srcWave* one column at a time, storing the results in the destination wave.

$$I[t_1][n]=\sum_{k=0}^{N-1} f[t_1][k]\exp\left(i2\pi kn / N\right).$$

You must specify a destination wave using the /DEST flag. No other flags are allowed with this flag. The number of rows must be even. If *srcWave* is a real (NxM) wave, the output matrix will be (1+N/2,M) in analogy with 1D FFT. To avoid changes in the number of points you can convert *srcWave* to complex data type. This flag applies only to 2D source waves. See also the /ROWS flag.

/DEST=*destWave* — Specifies the output wave created by the FFT operation.

It is an error to attempt specify the same wave as both *srcWave* and *destWave*.

The default output wave name is W_FFT for a 1D FFT and M_FFT for a multidimensional FFT.

When used in a function, the FFT operation by default creates a complex wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-72 for details.

/FREE Creates *destWave* as a free wave.

/FREE is allowed only in functions and only if *destWave*, as specified by /DEST, is a simple name or wave reference structure field.

See **Free Waves** on page IV-91 for more discussion.

The /FREE flag was added in Igor Pro 7.00.

/HCC Hypercomplex transform (cosine). Computes the integral

$$I_c(\omega_1,\omega_2)=\int\int_{-\infty}^{\infty} f(t_1,t_2)\cos(t_1\omega_1)\exp(it_2\omega_2)dt_1dt_2$$

using the 2D FFT (see **Details**).

/HCS Hypercomplex transform (sine). Computes the integral

$$I_s(\omega_1,\omega_2)=\int\int_{-\infty}^{\infty} f(t_1,t_2)\sin(t_1\omega_1)\exp(it_2\omega_2)dt_1dt_2$$

using the 2D FFT (see **Details**).

/MAG Saves just the magnitude of the FFT in the output wave. See comments under /OUT.

/MAGS Saves the squared magnitude of the FFT in the output wave. See comments under /OUT.

/OUT=*mode* Sets the output wave format.

mode=1:	Complex output (default)
mode=2:	Real output
mode=3:	Magnitude
mode=4:	Magnitude square
mode=5:	Phase
mode=6:	Scaled magnitude
mode=7:	Scaled magnitude squared

You can also identify modes 2-4 using the convenience flags /REAL, /MAG, and /MAGS. The convenience flags are mutually exclusive and are overridden by the /OUT flag.

The scaled quantities apply to transforms of real valued inputs where the output is normally folded in the first dimension (because of symmetry). The scaling applies a factor of 2 to the squared magnitude of all components except the DC. The scaled transforms should be used whenever Parseval's relation is expected to hold.

/PAD={*dim1* [, *dim2*, *dim3*, *dim4*]}

Converts *srcWave* into a padded wave of dimensions *dim1*, *dim2*…. The padded wave contains the original data at the start of the dimension and adds zero entries to each dimension up to the specified dimension size. The *dim1*… values must be greater than or equal to the corresponding dimension size of *srcWave*. If you need to pad just the lowest dimension(s) you can omit the remaining dimensions; for example, /PAD=*dim1* will set *dim2* and above to match the dimensions in *srcWave*.

/REAL Saves just the real part of the transform in the output wave. See comments under /OUT.

/ROWS Calculates the FFT of only the first dimension of a 2D *srcWave*. It thus computes the 1D FFT of one row at a time, storing the results in the destination wave.

$$N[n][t_2]=\sum_{k=0}^{M-1} f[k][t_2]\exp(i2\pi kn/M)$$

You must specify a destination wave using the /DEST flag. No other flags are allowed with this flag. The number of columns must be even. If *srcWave* is a real (NxM) wave, the output matrix will be (N,1+M/2) in analogy with 1D FFT. To avoid changes in the number of points you can convert *srcWave* to complex data type. See also /COLS flag.

/RP=[*startPoint*, *endPoint*]

/RX=(*startX*, *endX*) Defines a segment of a 1D *srcWave* that will be transformed. By default the operation transforms the whole wave. It is sometimes useful to take advantage of this feature in order to transform just the defined interval, which includes both end points. You can define the interval using wave point indexing with the /RP flag or using the X-values with the /RX flag. The interval must include at least four data points and the total number of points must be an even number.

/WINF=*windowKind*

Premultiplies a 1D *srcWave* with the selected window function.

If you include the /PAD flag, the window function is applied to the pre-padded data.

See **Window Functions** below for details.

/Z Disables rotation of the FFT of a complex wave. Igor normally rotates the FFT result (which is also complex) by N/2 so that x=0 is at the center point (N/2). When /Z is specified, Igor does not perform this rotation and leaves x=0 at the first point (0).

Details

The data type of *srcWave* is arbitrary. The first dimension of *srcWave* must be an even number and the minimum length of *srcWave* is four points. When *srcWave* is a double precision wave, the FFT is computed in double precision. All other data types are transformed using single precision calculations. The result of the FFT operation is always a floating point number (single or double precision).

Depending on your choice of outputs, you may not be able to invert the transform in order to obtain the original *srcWave*.

srcWave or any of its intervals must have at least four data points and must not contain NaNs or INFs.

The FFT algorithm is based on prime number decomposition, which decomposes the number of points in each dimension of the wave into a product of prime numbers. The FFT is optimized for primes < 5. In time consuming applications it is frequently worthwhile to pad the data so that the total number of points factors into small prime numbers.

The hypercomplex transforms are computed by writing the sine and cosine as a sum of two exponentials. Let the 2D Fourier transform of the input signal be

$$F\left[n_1\right]\left[n_2\right]=\sum_{k_1=0}^{N_1-1}\sum_{k_2=0}^{N_2-1} f\left[k_1\right]\left[k_2\right]\exp\left(i2\pi k_1\frac{n_1}{N_1}\right)\exp\left(i2\pi k_2\frac{n_2}{N_2}\right)$$

then the two hypercomplex transforms are given by

$$I_c\left[n_1\right]\left[n_2\right]=\frac{1}{2}\left(F[n_1][n2]+F[-n_1][n2]\right)$$

and

$$I_s\left[n_1\right]\left[n_2\right]=\frac{1}{2i}\left(F[n_1][n2]-F[-n_1][n2]\right)$$

Window Functions

The /F=*windowKind* flag premultiplies a 1D *srcWave* with the selected window function.

In the following window definitions, *w*(*n*) is the value of the window function that multiplies the signal, *N* is the number of points in the signal wave (or range if /R is specified), and *n* is the wave point index. With /R, *n*=0 for the first datum in the range.

Choices for *windowKind* are in bold.

Bartlet:

A synonym for Bartlett.

Bartlett:

$$w(n)=\begin{cases}\dfrac{2n}{N} & n=0,1,...\dfrac{N}{2}\\ 2-\dfrac{2n}{N} & n=\dfrac{N}{2},...N-1\end{cases}$$

Blackman367, **Blackman361**, **Blackman492**, **Blackman474:**

$$w(n)=a_0-a_1\cos\left(\frac{2\pi}{N}n\right)+a_2\cos\left(\frac{2\pi}{N}2n\right)-a_3\cos\left(\frac{2\pi}{N}3n\right).$$

$$n=0,1,2...N-1.$$

windowKind	a_0	a_1	a_2	a_3
Blackman367	0.42659071	0.49656062	0.07684867	0
Blackman361	0.44959	0.49364	0.05677	0
Blackman492	0.35875	0.48829	0.14128	0.01168
Blackman474	0.40217	0.49703	0.09392	0.00183

Cos1, **Cos2**, **Cos3**, **Cos4:**

$$w(n)=\cos\left(\frac{n}{N}\pi\right)^{\alpha},$$

$$n=-\frac{N}{2},...,-1,0,1,...,\frac{N}{2}.$$

windowKind	α
Cos1:	α = 1
Cos2:	α = 2
Cos3:	α = 3
Cos4:	α = 4

Hamming:

$$w(n)=\begin{cases} 0.54+0.46\cos\left(\dfrac{2\pi n}{N}\right) & n=-\dfrac{N}{2},\ldots,-1,0,1,\ldots,\dfrac{N}{2} \\ 0.54-0.46\cos\left(\dfrac{2\pi n}{N}\right) & n=0,1,2,\ldots,N-1 \end{cases}$$

Hanning:

$$w(n)=\begin{cases} \dfrac{1}{2}\left[1+\cos\left(\dfrac{2\pi n}{N}\right)\right] & n=-\dfrac{N}{2},\ldots,-1,0,1,\ldots,\dfrac{N}{2} \\ \dfrac{1}{2}\left[1-\cos\left(\dfrac{2\pi n}{N}\right)\right] & n=0,1,2,\ldots,N-1 \end{cases}$$

$$w(n) = \begin{cases} \frac{1}{2}\left[1+\cos\left(\frac{2n}{N}\pi\right)\right] & n = -\frac{N}{2}, \ldots, -1, 0, 1, \ldots, \frac{N}{2} \\ \frac{1}{2}\left[1-\cos\left(\frac{2n}{N}\pi\right)\right] & n = 0, 1, 2, \ldots, N-1 \end{cases}$$

KaiserBessel20, KaiserBessel25, KaiserBessel30:

$$w(n)=\frac{I_0\left(\pi\alpha\sqrt{1-\left(\dfrac{2n}{N}\right)^2}\right)}{I_0(\pi\alpha)} \qquad 0\le|n|\le\frac{N}{2}.$$

where I_0 is the zero-order modified Bessel function of the first kind.

windowKind	α
KaiserBessel20:	α = 2.
KaiserBessel25:	α = 2.5.
KaiserBessel30:	α = 3.

Parzen:

$$w(n)=1-\left|\frac{2n}{N}\right|^2 \qquad 0\le|n|\le\frac{N}{2}.$$

Poisson2, Poisson3, Poisson4:

$$w(n)=\exp\left(-\alpha\frac{2|n|}{N}\right) \qquad 0\le|n|\le\frac{N}{2}.$$

windowKind	α
Poisson2:	α = 2.
Poisson3:	α = 3.
Poisson4:	α = 4.

Riemann:

$$w(n)=\frac{\sin\left(\frac{2\pi n}{N}\right)}{\left(\frac{2\pi n}{N}\right)} \qquad 0\le |n| \le \frac{N}{2}.$$

Flat-Top:

The flat-top windows are defined as a sum of cosine terms:

$$w(n)=\sum_{k=0}^{m} c_k \cos(kz), \qquad z=\frac{2\pi j}{N}, \qquad j=0,1,...N-1.$$

Here are the supported flat-top window keywords for use as *windowKind* with the /WINF flag. These keywords require Igor Pro 8.00 or later:

windowKind	**Cosine Terms**
SFT3F	c0=0.26526, c1=-0.5, c2=0.23474.
SFT3M	c0=0.28235, c1=-0.52105, c2=0.19659.
FTNI	c0=0.2810639, c1=-0.5208972, c2=0.1980399.
SFT4F	c0=0.21706, c1=-0.42103, c2=0.28294, c3=-0.07897.
SFT5F	c0=0.1881, c1=-0.36923, c2=0.28702, c3=-0.13077, c4=0.02488.
SFT4M	c0=0.241906, c1=-0.460841, c2=0.255381, c3=-0.041872.
FTHP	c0=1.0, c1=-1.912510941, c2=1.079173272, c3=-0.1832630879.
HFT70	c0=1.0, c1=-1.90796, c2=1.07349, c3=-0.18199.
FTSRS	c0=1.0, c1=-1.93, c2=1.29, c3=-0.388, c4=0.028.
SFT5M	c0=0.209671, c1=-0.407331, c2=0.281225, c3=-0.092669, c4=0.0091036.
HFT90D	c0=1.0, c1=-1.942604, c2=1.340318, c3=-0.440811, c4=0.043097.
HFT95	c0=1.0, c1=-1.9383379, c2=1.3045202, c3=-0.4028270, c4=0.0350665.
HFT116D	c0=1.0, c1=-1.9575375, c2=1.4780705, c3=-0.6367431, c4=0.1228389, c5=-0.0066288.
HFT144D	c0=1.0, c1=-1.96760033, c2=1.57983607, c3=-0.81123644, c4=0.22583558, c5=-0.02773848, c6=0.00090360.
HFT169D	c0=1.0, c1=-1.97441842, c2=1.65409888, c3=-0.95788186, c4=0.33673420, c5=-0.06364621, c6=0.00521942, c7=-0.00010599.
HFT196D	c0=1.0, c1=-1.979280420, c2=1.710288951, c3=-1.081629853, c4=0.448734314, c5=-0.112376628, c6=0.015122992, c7=-0.000871252, c8=0.000011896.
HFT223D	c0=1.0, c1=-1.98298997309, c2=1.75556083063, c3=-1.19037717712, c4=0.56155440797, c5=-0.17296769663, c6=0.03233247087, c7=-0.00324954578, c8=0.00013801040, c9=-0.0000013275.
HFT248D	c0=1.0, c1=-1.985844164102, c2=1.791176438506, c3=-1.282075284005, c4=0.667777530266, c5=-0.240160796576, c6=0.056656381764, c7=-0.008134974479, c8=0.000624544650, c9=-0.000019808998, c10=0.000000132974.

See Also

See **Fourier Transforms** on page III-270 for discussion. The inverse operation is **IFFT**.

Spectral Windowing on page III-275. For 2D windowing see **ImageWindow**. Also the **Hanning** window operation.

IFFT, **DWT**, **CWT**, **STFT**, **HilbertTransform**, **WignerTransform**, **DSPPeriodogram**, **LombPeriodogram**, **Unwrap**, **MatrixOp**

References

For more information about the use of window functions see:

Harris, F.J., “On the use of windows for harmonic analysis with the discrete Fourier Transform“, *Proc, IEEE*, *66*, 51-83, 1978.

Heinzel, G., Rüdiger, A., & Schilling, R. (2002). “Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new at-top windows“, http://hdl.handle.net/11858/00-001M-0000-0013-557A-5.

FGetPos

FGetPos refNum

The FGetPos operation returns the file position for a file.

FGetPos is a faster alternative to FStatus if the only thing you are interested in is the file position.

The FGetPos operation was added in Igor Pro 7.00.

Parameters

refNum is a file reference number obtained from the **Open** operation.

Details

FGetPos supports very big files theoretically up to about 4.5E15 bytes in length.

FGetPos sets the following variables:

`V_flag`	Nonzero (true) if *refNum* is valid.
`V_filePos`	Current file position for the file in bytes from the start.

See Also

Open, **FSetPos**, **FStatus**

FIFO2Wave

FIFO2Wave [/R/S] ***FIFOName, channelName, waveName***

The FIFO2Wave operation copies FIFO data from the specified channel of the named FIFO into the named wave. FIFOs are used for data acquisition.

Flags

/R=[*startPoint*,*endPoint*]	Dumps the specified FIFO points into the wave.
/S=*s*	Controls the wave’s X scaling and number type: *s*=0: Same as no /S. *s*=1: Sets the wave’s X scaling x_0 value to the number of the first sample in the FIFO. *s*=2: Changes the wave’s number type to match the FIFO channel’s type. *s*=3: Combination of *s*=1 and *s*=2.

Details

The FIFO must be in the valid state for FIFO2Wave to work. When you create a FIFO, using NewFIFO, it is initially invalid. It becomes valid when you issue the start command via the CtrlFIFO operation. It remains valid until you change a FIFO parameter using CtrlFIFO.

If you specify a range of FIFO data points, using /R=[*startPoint*,*endPoint*] then FIFO2Wave dumps the specified FIFO points into the wave after clipping *startPoint* and *endPoint* to valid point numbers.

The valid point numbers depend on whether the FIFO is running and on whether or not it is attached to a file. If the FIFO is running then *startPoint* and *endPoint* are truncated to number of points in the FIFO. If the FIFO is buffering a file then the range can include the full extent of the file.

If you specify no range then FIFO2Wave transfers the most recently acquired FIFO data to the wave. The number of points transferred is the smaller of the number of points in the FIFO and number of points in the wave.

FIFO2Wave may or may not change the wave's X scaling and number type, depending on the current X scaling and on the /S flag.

Think of the wave's X scaling as being controlled by two values, x_0 and dx, where the X value of point p is x_0 + p*dx. FIFO2Wave always sets the wave's dx value equal to the FIFO's deltaT value (as set by the CtrlFIFO operation). If you use no /S flag, FIFO2Wave does not set the wave's x_0 value nor does it set the wave's number type.

If you are using FIFO2Wave to update a wave in a graph as quickly as possible, the /S=0 flag gives the highest update rate. The other /S values trigger more recalculation and slow down the updating.

If the wave's number type (possibly changed to match the FIFO channel) is a floating point type, FIFO2Wave scales the FIFO data before transferring it to the wave as follows:

```
scaled_value = (FIFO_value - offset) * gain
```

If the FIFO channel's gain is one and its offset is zero, the scaling would have no effect so FIFO2Wave skips it.

If the specified FIFO channel is an image strip channel (one defined using the optional vectPnts parameter to NewFIFOChan), then the resultant wave will be a matrix with the number of rows set by vectPnts and the number of columns set by the number of points described above for one-dimensional waves. To create an image plot that looks the same as the corresponding channel in a Chart, you will need to transpose the wave using **MatrixTranspose**.

See Also

The **NewFIFO** and **CtrlFIFO** operations, and **FIFOs and Charts** on page IV-313 for more information on FIFOs and data acquisition. For an explanation of waves and wave scaling, see **Changing Dimension and Data Scaling** on page II-68.

FIFOStatus

`FIFOStatus [/Q]` ***`FIFOName`***

The FIFOStatus operation returns miscellaneous information about a FIFO and its channels. FIFOs are used for data acquisition.

Flags

/Q	Doesn't print in the history area.

Details

FIFOStatus sets the variable V_flag to nonzero if a FIFO of the given name exists. If the named FIFO does exist then FIFOStatus stores information about the FIFO in the following variables:

`V_FIFORunning`	Nonzero if FIFO is running.
`V_FIFOChunks`	Number of chunks of data placed in FIFO so far.
`V_FIFOnchans`	Number of channels in the FIFO.
`S_Info`	Keyword-packed information string.

The keyword-packed information string consists of a sequence of sections with the following form: *keyword*:*value*;

You can pick a value out of a keyword-packed string using the **NumberByKey** and **StringByKey** functions. Here are the keywords for S_Info:

In addition, FIFOStatus writes fields to S_Info for each channel in the FIFO. The keyword for the field is a combination of a name and a number that identify the field and the channel to which it refers. For example, if channel 4 is named "Pressure" then the following would appear in the S_Info string: NAME4:Pressure.

Keyword	Type	Meaning
DATE	Number	The date/time when start was issued via CtrlFIFO.
DELTAT	Number	The FIFO’s deltaT value as set by CtrlFIFO.
DISKTOT	Number	Current number of chunks written to the FIFO’s file.
FILENUM	Number	The output file refNum or review file refNum as set by CtrlFIFO. This will be zero if the FIFO is connected to no file.
NOTE	String	The FIFO’s note string as set by CtrlFIFO.
VALID	Number	Zero if FIFO is not valid.
DATATYPE	Number	Channel's data type as if set by NewFIFOCHAN/Y=(*numType*) where *numType* is a value as returned by the **WaveType** function.

In the following table, the channel’s number is represented by “#”.

Keyword	Type	Meaning
FSMINUS#	Number	Channel’s minus full scale value as set by NewFIFOChan.
FSPLUS#	Number	Channel’s plus full scale value as set by NewFIFOChan.
GAIN#	Number	Channel’s gain value as set by NewFIFOChan.
NAME#	String	Name of channel.
OFFSET#	Number	Channel’s offset value as set by NewFIFOChan.
UNITS#	String	Channel’s units as set by NewFIFOChan.

See Also

The **NewFIFO**, **CtrlFIFO**, and **NewFIFOChan** operations, **FIFOs and Charts** on page IV-313 for more information on FIFOs and data acquisition.

The **NumberByKey** and **StringByKey** functions for parsing keyword-value strings.

FilterFIR

FilterFIR [***flags***] ***waveName*** [, ***waveName***]…

The FilterFIR operation convolves each *waveName* with automatically-designed filter coefficients or with *coefsWaveName* using time-domain methods.

The automatically-designed filter coefficients are simple lowpass and highpass window-based filters or a maximally-flat notch filter. Multiple filter designs are combined into a composite filter. The filter can be optionally placed into the first *waveName* or just used to filter the data in *waveName*.

FilterFIR filters data faster than **Convolve** when there are many fewer filter coefficient values than data points in *waveName*.

Note: FilterFIR replaces the obsolete **SmoothCustom** operation.

Parameters

waveName is a destination wave that is overwritten by the convolution of itself and the filter.

waveName may be multidimensional, but only one dimension selected by /DIM is filtered (for two-dimensional filtering, see **MatrixFilter**).

If *waveName* is complex, the real and imaginary parts are filtered independently.

Flags

/COEF [=*coefsWaveName*]

Replaces the first output *waveName* by the filter coefficients instead of the filtered results or, when *coefsWaveName* is specified, replaces the output wave(s) by the result of convolving *waveName* with coefficients in *coefsWaveName*.

coefsWaveName must not be one of the destination *waveName*s. It must be single- or double-precision numeric and one-dimensional.

To avoid shifting the output with respect to the input, *coefsWaveName* must have an odd length with the “center” coefficient in the middle of the wave.

The coefficients are usually symmetrical about the middle point, but FilterFIR does not enforce this.

/DIM=*d* Specifies the wave dimension to filter.

d=-1: Treats entire wave as 1D (default).

d=0: Operates along rows.

d=1: Operates along columns.

d=2: Operates along layers.

d=3: Operates along chunks.

Use /DIM=0 to apply the filter to each individual column (each one a channel, say left and right) in a multidimensional *waveName* where each row comprises all of the sound samples at a particular time.

/E=*endEffect* Determines how the ends of the wave (*w*) are handled when fabricating missing neighbor values. *endEffect* has values:

0: Bounce method (default). Uses w[*i*] in place of the missing w[-*i*] and w[*n*-*i*] in place of the missing w[*n*+*i*].

1: Wrap method. Uses w[*n*-*i*] in place of the missing w[-*i*] and vice versa.

2: Fill with 0. Same as /ENDV={0}.

3: Fill method. Uses w[0] in place of the missing w[-*i*] and w[*n*] in place of the missing w[*n*+*i*].

/ENDV={*sv* [, *ev*]} When fabricating missing neighbor values for each filtered wave, missing values before the start of data are replaced with *sv*.

Missing values after the end of data are replaced with *ev* if specified, or with *sv* if *ev* is omitted. /ENDV implies /E=2.

/ENDV was added in Igor Pro 9.00.

/HI={*f1*, *f2*, *n*} Creates a high-pass filter based on the windowing method, using the Hanning window unless another window is specified by /WINF.

f1 and *f2* are filter design frequencies measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).

f1 is the end of the reject band, and *f2* is the start of the pass band:

$0 < f1 < f2 < 0.5$

n is the number of FIR filter coefficients to generate. A larger number gives better stop-band rejection. A good number to start with is 101.

Use both /HI and /LO to create a bandpass filter.

/LO={*f1*, *f2*, *n*} Creates a low-pass filter. *f1* is the end of the pass band, *f2* is the start of the reject band, and *n* is the number of FIR filter coefficients. See /HI for more details.

/NMF={*fc*, *fw* [, *eps*, *nMult*]}

Creates a maximally-flat notch filter centered at *fc* with a -3dB width of *fw*. *fc* and *fw* are filter design frequencies measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).

The longest filter length allowed is 400,001 points, which requires fw >= 0.000789 (0.0789% of the sampling frequency).

Prior to Igor Pro 8.03 the longest filter length was 4,001 points, with fw >= 0.00789 (0.789% of the sampling frequency).

Coefficients at the ends that are smaller than the optional *eps* parameter are removed, making the filter shorter (and faster), though less accurate. The default is 2^{-40}. Use 0 to retain all coefficients, no matter how small, even zero coefficients. Retaining all coefficients will substantially increase the execution time as *fw* is made smaller.

nMult specifies how much longer the filter may be to obtain the most accurate notch frequency. The default is 2 (potentially twice as many coefficients). Set *nMult* <= 1 to generate the shortest possible filter.

The maximally flat notch filter design is based on Zahradník and Vlcek, and uses arbitrary precision math (see **APMath**) to compute the coefficients.

/WINF=*windowKind*

Applies the named “window” to the filter coefficients. Windows alter the frequency response of the filter in obvious and subtle ways, enhancing the stop-band rejection or steepening the transition region between passed and rejected frequencies. They matter less when many filter coefficients are used.

If /WINF is not specified, the Hanning window is used. For no coefficient filtering, use /WINF=None.

Choices for *windowKind* are:

Bartlett, Blackman367, Blackman361, Blackman492, Blackman474, Cos1, Cos2, Cos3, Cos4, Hamming, Hanning, KaiserBessel20, KaiserBessel25, KaiserBessel30, Parzen, Poisson2, Poisson3, Poisson4, and Riemann.

See **FFT** for window equations and details.

Details

If *coefsWaveName* is specified, then /HI, /LO, and /NMF are ignored.

If more than one of /HI, /LO, and /NMF are specified, the filters are combined using linear convolution. The length of the combined filter is slightly less than the sum of the individual filter lengths.

A band pass or band reject filter results when both /LO and /HI are specified. A band pass filter results from /LO frequencies greater than the /HI frequencies (the pass bands of the low pass and high pass filters overlap). Beginning with Igor Pro 9.00, a band reject filter results when /LO frequencies are less than the /HI frequencies (the stop bands of the filters overlap).

The filtering convolution is performed in the time-domain. That is, the FFT is not employed to filter the data. For this reason the coefficients length should be small in comparison to the destination waves.

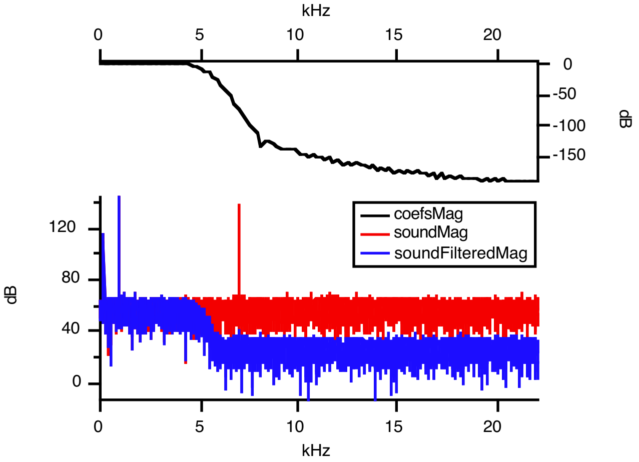
FilterFIR assumes that the middle point of *coefsWaveName* corresponds to the delay = 0 point. The “middle” point number = trunc(numpnts(*coefsWaveName* -1)/2). *coefsWaveName* usually contains the two-sided impulse response of a filter, and usually contains an odd number of points. This is the kind of coefficients data generated by /HI, /LO, and /NMF.

FilterFIR ignores the X scaling of all waves, except when /COEF creates a coefficients wave, which preserves the X scale deltax and alters the leftx value so that the zero-phase (center) coefficient is located at x=0.

Examples

```
// Make test sound from three sine waves
Variable/G fs= 44100                          // Sampling frequency
Variable/G seconds= 0.5                       // Duration
Variable/G n= 2*round(seconds*fs/2)
Make/O/W/N=(n) sound                          // 16-bit integer sound wave
SetScale/p x, 0, 1/fs, "s", sound
```

```
Variable/G f1= 200, f2= 1000, f3= 7000
Variable/G a1=100, a2=3000,a3=1500
sound= a1*sin(2*pi*f1*x)
sound += a2*sin(2*pi*f2*x)
sound += a3*sin(2*pi*f3*x)+gnoise(10)          // Add a noise floor

// Compute the sound's spectrum in dB
FFT/MAG/WINF=Hanning/DEST=soundMag sound
soundMag= 20*log(soundMag)
SetScale d, 0, 0, "dB", soundMag

// Apply a 5 kHz low-pass filter to the sound wave
Duplicate/O sound, soundFiltered
FilterFIR/E=3/LO={4000/fs, 6000/fs, 101} soundFiltered

// Compute the filtered sound's spectrum in dB
FFT/MAG/WINF=Hanning/DEST=soundFilteredMag soundFiltered
soundFilteredMag= 20*log(soundFilteredMag)
SetScale d, 0, 0, "dB", soundFilteredMag

// Compute the filter's frequency response in dB
Make/O/D/N=0 coefs                             // Double precision is recommended
SetScale/p x, 0, 1/fs, "s", coefs
FilterFIR/COEF/LO={4000/fs, 6000/fs, 101} coefs
FFT/MAG/WINF=Hanning/PAD={(2*numpnts(coefs))}/DEST=coefsMag coefs
coefsMag= 20*log(coefsMag)
SetScale d, 0, 0, "dB", coefsMag

// Graph the frequency responses
Display/R/T coefsMag as "FIR Lowpass Example";DelayUpdate
AppendToGraph soundMag, soundFilteredMag;DelayUpdate
ModifyGraph axisEnab(left)={0,0.6}, axisEnab(right)={0.65,1}
ModifyGraph rgb(soundFilteredMag)=(0,0,65535), rgb(coefsMag)=(0,0,0)
Legend
```



```
// Graph the unfiltered and filtered sound time responses
Display/L=leftSound sound as "FIR Filtered Sound";DelayUpdate
AppendToGraph/L=leftFiltered soundFiltered;DelayUpdate
ModifyGraph axisEnab(leftSound)={0,0.45}, axisEnab(leftFiltered)={0.55,1}
ModifyGraph rgb(soundFiltered)=(0,0,65535)

// Listen to the sounds
PlaySound sound                // This has a very high frequency tone
PlaySound soundFiltered        // This doesn't
```

References

Zahradník, P., and M. Vlcek, Fast Analytical Design Algorithms for FIR Notch Filters, *IEEE Trans. on Circuits and Systems*, *51*, 608 - 623, 2004.

<http://euler.fd.cvut.cz/publikace/files/vlcek/notch.pdf>

See Also

Smoothing on page III-292; the **Smooth**, **Convolve**, **MatrixConvolve**, and **MatrixFilter** operations.

FilterIIR

FilterIIR [***flags***] [***waveName,…***]

The FilterIIR operation applies to each *waveName* either the automatically-designed IIR filter coefficients or the IIR filter coefficients in *coefsWaveName*. Multiple filter designs are combined into a composite filter. The filter can be optionally placed into the first *waveName* or just used to filter the data in *waveName*.

The automatically-designed filter coefficients are bilinear transforms of the Butterworth analog prototype with an optional variable-width notch filter.

To design more advanced IIR filters, see **Designing the IIR Coefficients**.

Parameters

waveName may be multidimensional, but only the one dimension selected by /DIM is filtered (for two-dimensional filtering, see **MatrixFilter**).

waveName may be omitted for the purpose of checking the format of *coefsWaveName*. If the format is detectably incorrect an error code will be returned in V_flag. Use /Z to prevent command execution from stopping.

Flags

/CASC Specifies that *coefsWaveName* contains cascaded bi-quad filter coefficients. The cascade implementation is more stable and numerically accurate for high-order IIR filtering than Direct Form 1 filtering. See **Cascade Details**.

/COEF [=*coefsWaveName*]

Replaces the first output *waveName* by the filter coefficients instead of the filtered results or, when *coefsWaveName* is specified, replaces the output wave(s) by the result of filtering *waveName* with the IIR coefficients in *coefsWaveName*.

coefsWaveName must not be one of the destination *waveName*s. It must be single- or double-precision numeric and two-dimensional.

When used with /CASC, *coefsWaveName* must have 6 columns, containing real-valued coefficients for a product of ratios of second-order polynomials (cascaded bi-quad sections).

If /ZP is specified, it must be complex, otherwise it must be real.

See **Details** for the format of the coefficients in *coefsWaveName*.

/DIM=*d* Specifies the wave dimension to filter.

d=-1:	Treats entire wave as 1D (default).
d=0:	Operates along rows.
d=1:	Operates along columns.
d=2:	Operates along layers.
d=3:	Operates along chunks.

Use /DIM=0 to apply the filter to each individual column (each one a channel, say left and right) in a multidimensional *waveName* where each row comprises all of the sound samples at a particular time.

/ENDV=*ev* Values before the beginning of each filtered wave are replaced with *ev*. If you omit /ENDV they are replaced with zeros. To prevent filter startup artifacts set *ev* to the first value or a localized mean value at the start of the wave to be filtered.

/ENDV was added in Igor Pro 9.00.

/HI=*fHigh* Creates a high-pass Butterworth filter with the -3dB corner at *fHigh*. The order of the filter is controlled by the /ORD flag.

fHigh is a filter design frequency measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).

/LO=*fLow* Creates a low-pass Butterworth filter with the -3dB corner at *fLow*. The /ORD flag controls the order of the filter.

fLow is a filter design frequency measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).

Create bandpass and bandreject filters by specifying both /HI and /LO. For a bandpass filter, set *fLow* > *fHigh*, and for a band reject filter, set *fLow* < *fHigh*.

/N={*fNotch*, *notchQ*} Creates a notch filter with the center frequency at *fNotch* and a -3dB width of *fNotch*/*notchQ*.

fNotch is a filter design frequency measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).

notchQ is a number greater than 1, typically 10 to 100. Large values produce a filter that "rings" a lot.

/ORD=*order* Sets the order of the Butterworth filter(s) created by /HI and /LO. The default is 2 (second order), and the maximum is 100.

/Z=*z* Prevents procedure execution from aborting when an error occurs. Use /Z=1 to handle this case in your procedures using **GetRTError**(1) rather than having execution abort. /Z=0 is the same as no /Z at all.

/ZP Specifies that *coefsWaveName* contains complex z-domain zeros (in column 0) and poles (in column 1) or, if *coefsWaveName* is not specified, that the first output *waveName* is to be replaced by filter coefficients in the zero-pole format. See **Zeros and Poles Details**.

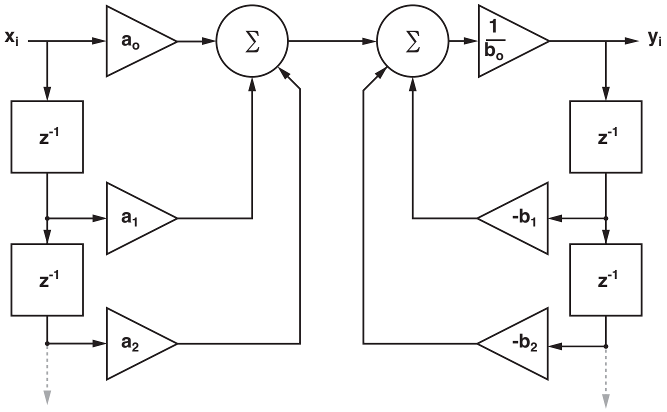
Details

FilterIIR sets V_flag to 0 on success or to an error code if an error occurred. Command execution stops if an error occurs unless the /Z flag is set. Omit /Z and call **GetRTError** and **GetRTErrMessage** under similar circumstances to see what the error code means.

Direct Form 1 Details

Unless /CASC or /ZP are specified, the coefficients in *coefsWaveName* describe a ratio of two polynomials of the Z transform:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2} + \dots}{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots}$$



Direct Form I Implementation

$$y_i = \frac{a_0 x_i + a_1 x_{i-1} + a_2 x_{i-2} + \dots - b_1 y_{i-1} - b_2 y_{i-2} + \dots}{b_0}$$

where *x* is the input wave *waveName* and *y* is the output wave (either *waveName* again or *destWaveName*).

FilterIIR computes the filtered result using the Direct Form I implementation of *H*(*z*).

The rational polynomial numerator (a_i) coefficients in are column 0 and denominator (b_i) coefficients in column 1 of *coefsWaveName*.

The coefficients in row 0 are the nondelayed coefficients a_0 (in column 0) and b_0 (in column 1).

The coefficients in row 1 are the z^{-1} coefficients, a_1 and b_1.

The coefficients in row n are the z^{-n} coefficients, a_n and b_n.

The number of coefficients for the numerator can differ from the number of coefficients for the denominator. In this case, specify 0 for unused coefficients.

Note: If all the coefficients of the denominator are 0 (b_i= 0 except b_0 = 1), then the filter is actually a causal FIR filter (Finite Impulse Response filter with delay of n-1). In this sense, FilterIIR implements a superset of the FilterFIR operation.

Alternate Direct Form 1 Notation

The designation of a_i, etc. as the numerator is at odds with many textbooks such as *Digital Signal Processing*, which uses b for the numerator coefficients of the rational function, a for the denominator coefficients with an implicit a_0 = 1, in addition to reversing the signs of the remaining denominator coefficients so that they can write $H(z)$ as:

$$H(z)=\frac{Y(z)}{X(z)}=\frac{\sum_{i=0}^{n} b_i z^{-i}}{1-\sum_{i=1}^{n} a_i z^{-i}}.$$

Coefficients derived using this notation need their denominator coefficients sign-reversed before putting them into rows 1 through n of column 1 (the second column), and the “missing” nondelayed denominator coefficient of 1.0 placed in row 0, column 1.

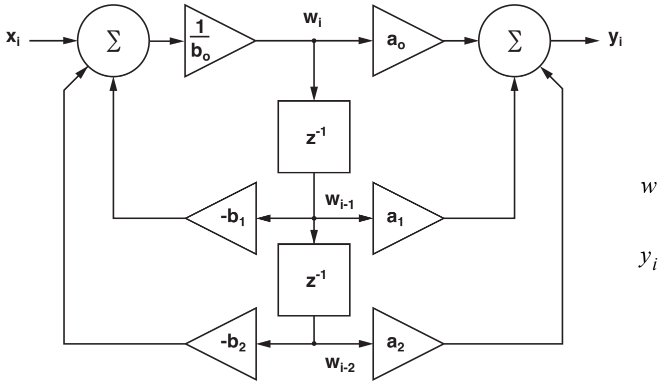
Cascade Details

When using /CASC, coefficients in *coefsWaveName* describe the product of one or more ratios of two quadratic polynomials of the Z transform:

$$H(z)=\frac{Y(z)}{X(z)}=\prod_{k=1}^{K}\frac{a_{0_k}+a_{1_k}z^{-1}+a_{2_k}z^{-2}}{b_{0_k}+b_{1_k}z^{-1}+b_{2_k}z^{-2}}.$$

Each product term implements a “cascaded bi-quad section”, and $H(z)$ can be realized by feeding the output of one section to the next one.

The cascade coefficients filter the data using a Direct Form II cascade implementation:



$$w_i = \frac{x_i - b_1 w_{i-1} - b_2 w_{i-2}}{b_0}$$

$$y_i = a_0 w_i + a_1 w_{i-1} + a_2 w_{i-2}$$

Cascaded Bi-Quad Direct Form II Implementation

The cascade implementation is more stable and numerically accurate for high-order IIR filtering than Direct Form I filtering. Cascade IIR filtering is recommended when the filter order exceeds 16 (a 16th-order Direct Form I filter has 17 numerator coefficients and 17 denominator coefficients).

coefsWaveName must be a six-column real-valued numeric wave. Each row describes one bi-quad section. The coefficients for the second term (or "section") of the product (*k*=2) are in the following row, etc.:

k	Row	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5
1	0	a_{0_1}	a_{1_1}	a_{2_1}	b_{0_1}	b_{1_1}	b_{2_1}
2	1	a_{0_2}	a_{1_2}	a_{2_2}	b_{0_2}	b_{1_2}	b_{2_2}
	…						

The number of coefficients for the numerator (*a*'s) is allowed to differ from the number of coefficients for the denominator (*b*'s). In this case, specify 0 for unused coefficients.

For example, a third order filter (three poles and three zeros) cascade implementation is a single-order section combined with a second order section. The values for a_{2_k}, b_{2_k} for that section (*k*) would be 0. Here the second section is specified as the first-order section:

k	Row	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5
1	0	a_{0_1}	a_{1_1}	a_{2_1}	b_{0_1}	b_{1_1}	b_{2_1}
2	1	a_{0_2}	a_{1_2}	0	b_{0_2}	b_{1_2}	0

Alternate Cascade Notation

In the DSP literature, the b_{0_k} gain values are typically one and the *H*(*z*) expression contains an overall gain value, usually *K*. Here each product term (or "section") has a user-settable gain value. Computing the correct gain values to control overflow in integer implementations is the responsibility of the user. For floating implementations, you might as well set all b_{0_k} values to one except, say, b_{0_1}, to control the overall gain.

Zeros and Poles Details

When using /ZP, coefficients in *coefsWaveName* contains complex zeros and poles in the (also complex) Z transform domain:

$$H(z)=\frac{Y(z)}{X(z)}=\frac{(z-z_0)(z-z_1)(z-z_2)\ldots}{(z-p_0)(z-p_1)(z-p_2)\ldots}$$

coefsWaveName must be a two-column complex wave with zero0, zero1,… zeroN in the first column of N+1 rows, and pole0, pole1,… poleN in the second column of those same rows:

k	Row	Col 0	Col 1
1	0	(zero0Real, zero0Imag)	(pole0Real, pole0Imag)
2	1	(zero1Real, zero1Imag)	(pole1Real, pole1Imag)
3	2	(zero2Real, zero2Imag)	(pole2Real, pole2Imag)
	…		

If a zero or pole has a nonzero imaginary component, the conjugate zero or pole must be included in *coefsWaveName*. For example, if a zero is placed at (0.7, 0.5), the conjugate is (0.7, -0.5), and that value must also appear in column 0. These two zeros form what is known as a "conjugate pair". The conjugate values must match within the greater of 1.0e-6 or 1.0e-6 * |zeroOrPole|.

Use (0,0) for unused poles or zeros, as a zero or pole at *z*= (0,0) has no effect on the filter frequency response.

The /ZP format for the coefficients is internally converted into the Direct Form 1 implementation, or into the Cascade Direct Form 2 implementation if /CASC is specified. There is no option for returning these implementation-dependent coefficients in a wave.

Designing the IIR Coefficients

Simple IIR filters can be used or created by specifying the /LO, /HI, /ORD, /N, /CASC, and /ZP flags. Use /COEF without *coefsWaveName* to put these simple IIR filter coefficients into the first *waveName*.

More advanced IIR filters (Bessel, Chebyshev) can be designed using the separate Igor Filter Design Laboratory (IFDL). IFDL is an Igor package that you use to design FIR (Finite Impulse Response) and IIR (Infinite Impulse Response) filters and to apply them to your data. The IIR design software creates IIR coefficients based on bilinear transforms of analog prototype filters such as Bessel, Butterworth, and Chebyshev.

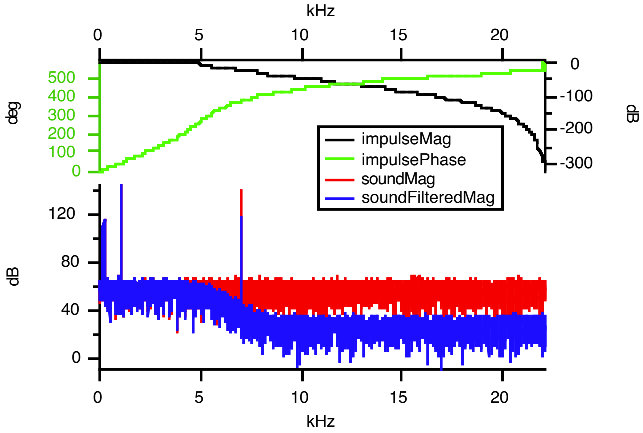
Even without IFDL, you can create custom IIR filters by manually placing poles and zeros in the Z plane using the Pole and Zero Filter Design procedures. Copy the following line to your Procedure window and click the Compile button at the bottom of the procedure window:

```
#include <Pole And Zero Filter Design>
```

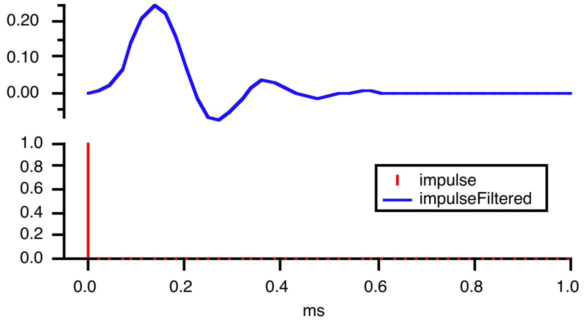
Then choose Pole and Zero Filter Design from the Analysis menu.

Examples

```
// Make test sound from three sine waves
Variable/G fs= 44100                        // Sampling frequency
Variable/G seconds= 0.5                     // Duration
Variable/G n= 2*round(seconds*fs/2)
Make/O/W/N=(n) sound                        // 16-bit integer sound wave
SetScale/p x, 0, 1/fs, "s", sound
Variable/G f1= 200, f2= 1000, f3= 7000
Variable/G a1=100, a2=3000,a3=1500
sound= a1*sin(2*pi*f1*x)
sound += a2*sin(2*pi*f2*x)
sound += a3*sin(2*pi*f3*x)+gnoise(10)           // Add a noise floor

// Compute the sound's spectrum in dB
FFT/MAG/WINF=Hanning/DEST=soundMag sound
soundMag= 20*log(soundMag)
SetScale d, 0, 0, "dB", soundMag

// Apply a 5 kHz, 6th order low-pass filter to the sound wave
Duplicate/O sound, soundFiltered
FilterIIR/LO=(5000/fs)/ORD=6 soundFiltered      // Second order by default

// Compute the filtered sound's spectrum in dB
FFT/MAG/WINF=Hanning/DEST=soundFilteredMag soundFiltered
soundFilteredMag= 20*log(soundFilteredMag)
SetScale d, 0, 0, "dB", soundFilteredMag

// Compute the filter's frequency and phase by filtering an impulse
Make/O/D/N=2048 impulse= p==0                   // Impulse at t==0
SetScale/P x, 0, 1/fs, "s", impulse
Duplicate/O impulse, impulseFiltered
FilterIIR/LO=(5000/fs)/ORD=6 impulseFiltered
FFT/MAG/DEST=impulseMag impulseFiltered
impulseMag= 20*log(impulseMag)
SetScale d, 0, 0, "dB", impulseMag
FFT/OUT=5/DEST=impulsePhase impulseFiltered
impulsePhase *= 180/pi                          // Convert to degrees
SetScale d, 0, 0, "deg", impulsePhase
Unwrap 360, impulsePhase                        // Continuous phase

// Graph the frequency responses
Display/R/T impulseMag as "IIR Lowpass Example"
AppendToGraph/L=phase/T impulsePhase
AppendToGraph soundMag, soundFilteredMag
ModifyGraph axisEnab(left)={0,0.6}
ModifyGraph axisEnab(right)={0.65,1}
ModifyGraph axisEnab(phase)={0.65,1}
ModifyGraph freePos=0, lblPos=60, rgb(soundFilteredMag)=(0,0,65535)
ModifyGraph rgb(impulseMag)=(0,0,0), rgb(impulsePhase)=(0,65535,0)
ModifyGraph axRGB(phase)=(3,52428,1), tlblRGB(phase)=(3,52428,1)
Legend
```



```
// Graph the unfiltered and filtered impulse time responses
Display/L=leftImpulse impulse as "IIR Filtered Impulse"
AppendToGraph/L=leftFiltered impulseFiltered
ModifyGraph axisEnab(leftImpulse)={0,0.45}, axisEnab(leftFiltered)={0.55,1}
ModifyGraph freePos=0, margin(left)=50
ModifyGraph mode(impulse)=1, rgb(impulseFiltered)=(0,0,65535)
SetAxis bottom -0.00005,0.001
Legend
```



```
// Listen to the sounds
PlaySound sound                  // This has a very high frequency tone
PlaySound soundFiltered          // This doesn't
```

References

Embree, P.M., and B. Kimble, *C Language Algorithms for Signal Processing*, 456 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Lynn, P.A., and W. Fuerst, *Introductory Digital Signal Processing with Computer Applications*, 479 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1998.

Oppenheim, A.V., and R.W. Schafer, *Digital Signal Processing*, 585 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1975.

Terrell, T.J., *Introduction to Digital Filters*, 2nd ed., 261 pp., John Wiley & Sons, New York, 1988.

See Also

Smoothing on page III-292; the **FFT** and **FilterFIR** operations.

FindContour

`FindContour [`*`flags`*`]` ***`matrixWave, level`***

The FindContour operation creates an XY pair of waves representing the locus of the solution to *matrixWave=level* .

The FindContour operation was added in Igor Pro 7.00.

Flags

/DSTX=*destX*	Saves the output X data in the specified destination wave. The destination wave is created or overwritten if it already exists.
/DSTY=*destY*	Saves the output Y data in the specified destination wave. The destination wave is created or overwritten if it already exists.

Details

FindContour uses a contour-following algorithm to generate a pair of waves describing the locus of the solution to *matrixWave*=*level*.

If you omit /DSTX the output X data is written to W_XContour in the current data folder.

If you omit /DSTY the output Y data is written to W_YContour in the current data folder.

The output waves are written as double-precision floating point. They use NaNs to separate different contiguous solution points.

Example

```
Make/N=(100,200) dataWave = 1e4*gauss(x,50,10,y,100,20)
FindContour dataWave,4     // Find solution to dataWave=4
NewImage dataWave
AppendToGraph/T W_YContour vs W_XContour
```

See Also

AppendMatrixContour, **ContourZ**

FindDimLabel

`FindDimLabel(`*`waveName`*`, `*`dimNumber`*`, `*`labelString`*`)`

Returns the index value corresponding to the label for the given dimension. Returns -1 if the label is for the entire dimension. Returns -2 if the label is not found.

Use *dimNumber* =0 for rows, 1 for columns, 2 for layers, or 3 for chunks.

See Also

GetDimLabel, **SetDimLabel**, **CopyDimLabels**

FindDuplicates

`FindDuplicates [`*`flags`*`] `*`srcWave`*

The FindDuplicates operation identifies duplicate values in a wave and optionally creates various output waves. *srcWave* can be either numeric or text.

When *srcWave* is numeric, the /DN, /INDX, /RN and /SN flags create output waves as described below. If you omit all of these flags then FindDuplicates does nothing.

When *srcWave* is text, the /DT, /INDX, /RT and /ST flags create output waves as described below. If you omit all of these flags then FindDuplicates does nothing.

The FindDuplicates operation was added in Igor Pro 7.00.

Flags

/FREE	Creates all output waves as free waves. The /FREE flag was added in Igor Pro 8.00. /FREE is permitted in user-defined functions only. If you use /FREE then all output wave parameters must be simple names, not paths or $ expressions. See **Free Waves** on page IV-91 for details on free waves.
/INDX=*indexWave*	Creates a numeric output wave containing the index of each encountered duplicate. The index is the point number in *srcWave* where a duplicate value was encountered. This flag applies to both numeric and text inputs.
/Z	Do not report any errors.

Flags for Numeric Source Wave

/DN=*dupsWave* Creates a numeric output wave that contains the duplicates.

/RN=*dupsRemovedWave*

Creates a numeric output wave that contains the source data with all duplicates removed.

/SN=*replacement* Creates a numeric output wave with all duplicates replaced with *replacement*. *replacement* can be any numeric value including NaN or INF.

The output wave is W_ReplacedDuplicates in the current data folder unless you specify a different output wave using the /SNDS flag.

/SNDS=*dupsReplacedWave*

Specifies the output wave generated by /SN. If you omit /SNDS then the output wave created by /SN is W_ReplacedDuplicates in the current data folder. /SNDS without /SN has no effect.

/TOL=*tolerance* Specifies the tolerance value for single-precision and double-precision numeric source waves.

Two values are considered duplicates if

```
abs(value1-value2) <= tolerance
```

By default *tolerance* is zero.

/UN=*uniqueNumbersWave*

Creates a numeric output wave that contains the unique numbers in *srcWave* sorted from small to large. When *srcWave* contains NaN entries they are sorted as the last point in *uniqueNumbersWave*.

/UN is incompatible with /RN which maintains the order of entries in *srcWave*.

The /UN flag was added in Igor Pro 9.00.

/UNC=*uniqueCounts*

Creates a numeric output wave that contains the count of each entry in the *uniqueNumbersWave* created by the /UN flag.

The /UNC flag was added in Igor Pro 9.00.

Flags for Text Source Wave

/CI Performs case-insensitive text comparisons on ASCII characters only. For example, "A" and "a" are considered duplicates.

The /CI flag was added in Igor Pro 9.00.

/DT=*dupsWave* Creates a text output wave that contains the duplicates.

/LOC Performs locale-aware text comparisons which take case into account for both ASCII and non-ASCII characters. For example, the non-ASCII characters "Å" and "å" are considered duplicates as well as the ASCII characters "A" and "a".

/LOC is ignored unless you also include /CI.

The /LOC flag was added in Igor Pro 9.00.

/RT=*dupsRemovedWave*

Creates a text output wave that contains the source data with all duplicates removed.

/ST=*replacementStr* Creates a text output wave with all duplicates replaced with *replacementStr*. *replacementStr* can be any text value including "".

The output wave is T_ReplacedDuplicates in the current data folder unless you specify a different output wave using the /STDS flag.

/STDS=*dupsReplacedWave*

Specifies the output wave generated by /ST. If you omit /STDS then the output wave created by /ST is T_ReplacedDuplicates in the current data folder. /STDS without /ST has no effect.

Details

FindDuplicates scans *srcWave* and identifies duplicate values. The first instance of any value is not considered a duplicate. Duplicates are either identical, as is the case with integer or text waves, or values that are within a specified tolerance in the case of single-precision or double-precision numeric waves.

Text comparison is case-sensitive unless you use /CI or /CI/LOC.

The operation creates wave references for the waves specified by the various flags above. See **Automatic Creation of WAVE References** on page IV-72 for details.

Example

```
Function DemoFindDuplicates(mode)
   int mode        // 0=case sensitive; 1=/CI; 2=/CI/LOC

   Make/O/T sourceText={"A","a", "Å","å", "B","b"}
   switch(mode)
      case 0:      // Case sensitive
         // Returns {"A","a","Å","å","B","b"}
         FindDuplicates/FREE/RT=output sourceText
         break
      case 1:      // Case insensitive for ASCII only
         // Returns {"A","Å","å","B"}
         FindDuplicates/FREE/RT=output/CI sourceText
         break
      case 2:      // Case insensitive, locale aware
         // Returns {"A","Å","B"}
         FindDuplicates/FREE/RT=output/CI/LOC sourceText
         break
   endswitch

   Print sourceText
   Print output
End
```

See Also

FindLevels, **FindValue**, **Sort**, **TextHistogram**

FindLevel

`FindLevel [`*`flags`*`] `*`waveName, level`*

The FindLevel operation searches the named wave to find the X value at which the specified Y *level* is crossed.

Flags

/B=*box*

Sets box size for sliding average. If /B=*box* is omitted or *box* equals 1, no averaging is done. If you specify an even box size then the next higher (odd) integer is used. If you use a box size greater than 1, FindLevel will be unable to find a level crossing that occurs in the first or last (*box*-1)/2 points of the wave since these points don't have enough neighbors for computing the derived average wave values.

/EDGE=*e*

Specifies searches for either increasing or decreasing level crossing.

e=1: Searches only for crossing where Y values are increasing as *level* is crossed from wave start towards wave end.

e=2: Searches only for crossing where the Y values are decreasing as *level* is crossed from wave start towards wave end.

e=0: Same as no /EDGE flag (searches for either increasing and decreasing level crossing).

/P	Computes the X crossing location in terms of point number. If /P is omitted, the level crossing location is computed in terms of X values.
/Q	Don’t print results in history and don’t report error if *level* is not found.
/R=(*startX*,*endX*)	Specifies an X range of the wave to search. You may exchange *startX* and *endX* to reverse the search direction.
/R=[*startP*,*endP*]	Specifies a point range of the wave to search. You may exchange *startP* and *endP* to reverse the search direction. If you specify the range as /R=[*startP*] then the end of the range is taken as the end of the wave. If /R is omitted, the entire wave is searched.
/T=*dx*	Search for two level crossings. *dx* must be less than *minWidthX*, so you must also specify /M if you use /T. (FindLevel limits *dx* so that second search start isn’t beyond where the first search for next edge will be.)
/T=*dx*	Performs a second search after finding the initial level crossing. The second search starts *dx* units beyond the initial level crossing and looks back in the direction of the initial crossing. If FindLevel finds a second level crossing, it sets V_LevelX to the average of the initial and second crossings. Otherwise, it sets V_LevelX to the initial crossing.

Details

FindLevel scans through the wave comparing *level* to values derived from the Y values of the wave. Each derived value is a sliding average of the Y values.

FindLevel searches for two derived wave values that straddle *level*. If it finds these values it computes the X value at which *level* is located by linearly interpolating between the straddling Y values.

FindLevel does not locate values exactly equal to *level*; it locates transitions through *level*. See **BinarySearch** for one method of locating exact values.

FindLevel reports its results by setting these variables:

`V_flag`	0: *level* was found. 1: *level* was not found.
`V_LevelX`	Interpolated X value at which *level* was found, or the corresponding point number if /P is specified.
`V_rising`	0: Y values at the crossing are decreasing from wave start towards wave end. 1: Y values at the crossing are increasing.

If you omit the /Q flag then FindLevel also reports its results by printing them in the history area.

If *level* is not found, and if you omit the /Q flag, FindLevel generates an error which puts up an error alert and halts execution of any command line or macro that is in progress.

V_LevelX is returned in terms of the X scaling of the named wave unless you use the /P flag, in which case it is in terms of point number.

FindLevel Handling of NaNs

In Igor Pro 8.00 and later, FindLevel handles NaN values differently than previous versions. Now if level falls between two non-NaN wave Y values with NaNs between them, those two Y values and their associated X scaling values are used to linearly interpolate the X location of the level crossing. Igor7 and earlier fail to find a crossing and set V_LevelX to NaN.

For example:

```
// Prints 2.5 in Igor8 or later, NaN in Igor7 or before
Make/O wave0 = {0, 1, NaN, NaN, 4, 5}
FindLevel/Q wave0, 2.5; Print V_levelX
```

You can revert to the pre-Igor 8 behavior by executing:

```
SetIgorOption UseIP6FindLevel = 1
```

FindLevel and Multidimensional Waves

The FindLevel operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

See Also

The **EdgeStats**, **FindLevels**, **FindValue**, and **PulseStats** operations and the **BinarySearch** and **BinarySearchInterp** functions.

FindLevels

`FindLevels [`***`flags`***`] `***`waveName, level`***

The FindLevels operation searches the named wave to find one or more X values at which the specified Y *level* is **crossed**.

To find where the wave is equal to a given value, use **FindValue** instead.

Flags

/B=*box* Sets box size for sliding average. See the **FindLevel** operation.

/D=*destWaveName* Specifies wave into which FindLevels is to store the level crossing values. If /D and /DEST are omitted, FindLevels creates a wave named W_FindLevels to store the level crossing values in.

/DEST=*destWaveName*

Same as /D. Both /D and /DEST create a real wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-72 for details.

/EDGE=*e* Specifies searches for either increasing or decreasing level crossing.

e=1: Searches only for crossings where the Y values are increasing as level is crossed from wave start towards wave end.

e=2: Searches only for crossings where the Y values are decreasing as level is crossed from wave start towards wave end.

e=0: Same as no /EDGE flag (searches for both increasing and decreasing level crossings).

/M=*minWidthX* Sets the minimum X distance between level crossings. This determines where FindLevels searches for the next crossing after it has found a level crossing. The search starts *minWidthX* X units beyond the crossing. The default value for *minWidthX* is 0.

/N=*maxLevels* Sets a maximum number of crossings that FindLevels is to find. The default value for *maxLevels* is the number of points in the specified range of *waveName*.

/P Compute crossings in terms of points. See the **FindLevel** operation.

/Q Doesn’t print to history and doesn’t abort if no levels are found.

/R=(*startX*,*endX*) Specifies X range. See the **FindLevel** operation.

/R=[*startP*,*endP*] Specifies point range. See the **FindLevel** operation.

/T=*dx* Search for two level crossings. *dx* must be less than *minWidthX*, so you must also specify /M if you use /T. (FindLevels limits *dx* so that second search start isn’t beyond where the first search for next edge will be.) See **FindLevel** for more about /T.

Details

The algorithm for finding a level crossing is the same one used by the **FindLevel** operation.

If FindLevels finds *maxLevels* crossings or can not find another level crossing, it stops searching.

FindLevels sets the following variables:

`V_flag`	0: *maxLevels* level crossings were found. 1: At least one but less than *maxLevels* level crossings were found. 2: No level crossings were found.
`V_LevelsFound`	Number of level crossings found.

Example

```
Macro FindLevelsExample()
   // Sample data
   Make/O/N=1024 peaks;SetScale/P x,0,0.001,"" peaks // 1ms sampling
   peaks=(exp(sawtooth(-(x-1)*300/pi)^3)-1)/(exp(1)-1)
   SetRandomSeed 0; peaks += gnoise(0.05)
   // locate rising and falling x crossings of y = 0.5
   Variable minDX=20*deltax(peaks) // min 20 samples between crossings
   Variable level = 0.5 // peaks y range nominally 0..1
   FindLevels/Q/D=risingEdges/EDGE=1/M=(minDX) peaks, level
   FindLevels/Q/D=fallingEdges/EDGE=2/M=(minDX) peaks, level

   // Indicate found edges
   Duplicate/O risingEdges, risingYs; risingYs = peaks(risingEdges[p])
   Duplicate/O fallingEdges, fallingYs; fallingYs = peaks(fallingEdges[p])
   // show level
   Make/O/N=2 showLevel = level;CopyScales/I peaks, showLevel

   Display /W=(35.25,41,856.5,249.5) peaks
   AppendToGraph risingYs vs risingEdges
   AppendToGraph fallingYs vs fallingEdges
   AppendToGraph showLevel
   ModifyGraph mode(risingYs)=8,mode(fallingYs)=8
   ModifyGraph marker(risingYs)=17,marker(fallingYs)=23
   ModifyGraph lStyle(showLevel)=1
   ModifyGraph
   rgb(risingYs)=(19675,39321,1),rgb(fallingYs)=(0,0,0),rgb(showLevel)=(1,16019,65535)
   Legend/C/N=text0/X=3.55/Y=1.90
End
```

See Also

The **FindLevel** operation for details about the level crossing detection algorithm and the /B, /P, /Q, /R, and /T flag values.

The FindLevels operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

FindListItem

`FindListItem(`*`itemStr, listStr`*` [, `*`listSepStr`*` [, `*`start`*` [, `*`matchCase`*` ]]])`

The FindListItem function returns the byte offset into *listStr* where *itemStr* begins. *listStr* should contain items separated by the *listSepStr*.

Use FindListItem to locate the start of an item in a string containing a "wave0;wave1;" style list such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

Use WhichListItem to determine the index of an item in the list.

If *itemStr* is not found, if *listStr* is `""`, or if *start* is not within the range of 0 to strlen(*listStr*)-1, then -1 is returned.

listSepStr, *startIndex*, and *matchCase* are optional; their defaults are ";", 0, and 1 respectively.

Details

ItemStr may have any length.

listStr is searched for the first instance of the item string bound by a *listSepStr* on the left and a *listSepStr* on the right. The returned number is the byte index where the first character of *itemStr* was found in *listSepStr*.

The search starts from the byte position in *listStr* specified by *start*. A value of 0 starts with the first character in *listStr*, which is the default if *start* is not specified.

listString is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *listSepStr* are always case-sensitive. The comparison of *itemStr* to the contents of *listStr* is usually case-sensitive. Setting the optional *matchCase* parameter to 0 makes the comparison case insensitive.

In Igor6, only the first byte of *listSepStr* was used. In Igor7 and later, all bytes are used.

If *startIndex* is specified, then *listSepStr* must also be specified. If *matchCase* is specified, *startIndex* and *listSepStr* must be specified.

Examples

```
Print FindListItem("w1", "w0;w1;w2,")                   // prints 3
Print FindListItem("v2", "v1,v2,v3,", ",")              // prints 3
Print FindListItem("v2", "v0,v2,v2,", ",", 4)           // prints 6
Print FindListItem("C", "a;c;C;")                       // prints 4
Print FindListItem("C", "a;c;C;", ";", 0, 0)            // prints 2
```

See Also

The **AddListItem**, **strsearch**, **StringFromList**, **RemoveListItem**, **RemoveFromList**, **ItemsInList**, **WhichListItem**, **WaveList**, **TraceNameList**, **StringList**, **VariableList**, and **FunctionList** functions.

FindAPeak

FindAPeak [/B=*baseWaveName*] *minamp*, *pol*, *box*, *peakWave* [(*startX*,*endX*)]

FindAPeak locates the maximum or minimum of a peak by analyzing smoothed first and second derivatives.

The FindAPeak operation is used primarily by the Igor Technical Note #20 and its variants. For most purposes, use the more flexible **FindPeak** operation instead of FindAPeak.

Parameters

minamp is minimum amplitude ("threshold") of a peak. Use it to reject small or spurious peaks.

pol is the expected peak polarity. Specify 1 to search for a positive-going peak or 2 to search for a negative-going peak.

box is the number of peak values to include in the sliding average when smoothing the derivatives. If you specify an even number, the next-higher odd number is used.

peakWave specifies the wave containing the peak.

[*startX*,*endX*] is an optional subrange to search in point numbers.

(*startX*,*endX*) is an optional subrange to search in X values.

If you omit the subrange, *startX* defaults to the first point in *peakWave* and *endX* defaults to the last point in *peakWave*.

The search always with *startX* and ends at *endX*, regardless of whether *startX* is less than or greater than *endX*. You can use this to control the direction of the search.

Flags

/B=*baseWave*	Specifies a base wave containing values to subtract from *peakWave* to compute the derived data which FindAPeak searches for peaks.

Details

FindAPeak creates a temporary smoothed version of *peakWave* and a temporary first derivative of the smoothed data. It scans through the first derivative for the first zero-crossing where the smoothed data exceeds the minimum amplitude as specified by *minamp*. The location of the zero-crossing is then more accurately determined by reverse linear interpolation. The smoothed second derivative is computed at that point to see if the peak is a positive-going or negative-going peak.

Output Variables

FindAPeak reports results through these output variables:

V_Flag	0 if a peak is found and to 1 if no peak is found.
V_peakX	The interpolated X value of the peak center.
V_peakP	The interpolated fractional point number of the peak center.

See Also

FindPeak, **EstimatePeakSizes**

FindPeak

`FindPeak [`*`flags`*`]` ***`waveName`***

The FindPeak operation searches for a minimum or maximum by analyzing the smoothed first and second derivatives of the named wave. Information about the peak position, amplitude, and width are returned in the output variables.

Flags

Some of the flags have the same meaning as for the FindLevel operation.

/B=*box*	Sets box size for sliding average.
/I	Modify the search criteria to accommodate impulses (peaks of one sample) by requiring only one value to exceed *minLevel*. The default criteria requires that two successive values exceed *minLevel* for a peak to be found (or two successive values be less than the /M level when searching for negative peaks). Impulses can also be found by omitting *minLevel*, in which case /I is superfluous.
/M=*minLevel*	Defines minimum level of a peak. /N changes this to maximum level (see **Details**).
/N	Searches for a negative peak (minimum) rather then a positive peak (maximum).
/P	Location output variables (see **Details**) are reported in terms of (floating point) point numbers. If /P is omitted, they are reported as X values.
/Q	Doesn’t print to history and doesn’t abort if no peak is found.
/R=(*startX*,*endX*)	Specifies X range and direction for search.
/R=[*startP*,*endP*]	Specifies point range and direction for search.

Details

FindPeak sets the following variables:

`V_flag`	Set only when using the /Q flag. 0: Peak was found. Any nonzero value means the peak was not found.
`V_LeadingEdgeLoc`	Interpolated location of the peak edge closest to *startX* or *startP*. If you use the /P flag, V_LeadingEdgeLoc is a point number rather than to an X value. If the edge was not found, this value is NaN.
`V_PeakLoc`	Interpolated X value at which the peak was found. If you use the /P flag, FindPeak sets V_PeakLoc to a point number rather than to an X value. Set to NaN if peak wasn’t found.
`V_PeakVal`	The *approximate* Y value of the found peak. If the peak was not found, this value is NaN (Not a Number).
`V_PeakWidth`	Interpolated peak width. If you use the /P flag, V_PeakWidth is expressed in point numbers rather than as an X value. V_PeakWidth is never negative. If either peak edge was not found, this value is NaN.
`V_TrailingEdgeLoc`	Interpolated location of the peak edge closest to *endX* or *endP*. If you use the /P flag, V_TrailingEdgeLoc is a point number rather than to an X value. If the edge was not found, this value is NaN.

FindPeak computes the sliding average of the input wave using the BoxSmooth algorithm with the *box* parameter. The peak center is found where the derivative of this smoothed result crosses zero. The peak edges are found where the second derivative of the smoothed result crosses zero. Linear interpolation of the derivatives is used to more precisely locate the center and edges. The peak value is simply the greater of the two unsmoothed values surrounding the peak center (if /N, then the lesser value).

FindPeak is not a high-accuracy measurement routine; it is intended as a simple peak-finder. Use the **PulseStats** operation for more precise statistics.

Without /M, a peak is found where the derivative crosses zero, regardless of the peak height.

If you use the /M=*minLevel* flag, FindPeak ignores peaks that are lower than *minLevel* (i.e., the Y value of a found peak will exceed *minLevel*) in the box-smoothed input wave. If /N is also specified (search for minimum), FindPeak ignores peaks whose amplitude is greater than *minLevel* (i.e., the Y value of a found peak will be *less* than *minLevel*).

Without /I, a peak must have two successive values that exceed *minLevel*. Use /I when you are searching for peaks that may have only one value exceeding *minLevel*.

The search for the peak begins at *startX* (or the first point of the wave if /R is not specified), and ends at *endX* (or the last point of the wave if no /R). Searching backwards is permitted, and exchanges the values of V_LeadingEdgeLoc and V_TrailingEdgeLoc.

A simple automatic peak-finder is implemented in the "Peak Autofind.ipf" procedure file used in the Multipeak Fitting package. Execute `DisplayHelpTopic "Multipeak Fitting"` for details.

The FindPeak operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

See Also

The **PulseStats** operation, the **FindLevel** operation for details about the /B, /P, /Q, and /R flag values.

FindPointsInPoly

`FindPointsInPoly` ***`xWaveName`*****`,`** ***`yWaveName`*****`,`** ***`xPolyWaveName`*****`,`** ***`yPolyWaveName`***

The FindPointsInPoly operation determines if points fall within a certain polygon. It can be used to write procedures that operate on a subset of data identified graphically in a graph.

Details

FindPointsInPoly determines which points in *yWaveName* vs *xWaveName* fall within the polygon defined by *yPolyWaveName* vs *xPolyWaveName*.

xWaveName must have the same number of points as *yWaveName* and *xPolyWaveName* must have the same number of points as *yPolyWaveName*.

FindPointsInPoly creates an output wave named W_inPoly with the same number of points as *xWaveName*. FindPointsInPoly indicates whether the point *yWaveName*[p] vs *xWaveName*[p] falls within the polygon by setting `W_inPoly[p]=1` if it is within the polygon, or `W_inPoly[p]=0` if it is not.

FindPointsInPoly uses integer arithmetic with a precision of about 1 part in 1000. This should be good enough for visually determined (hand-drawn) polygons but might not be sufficient for mathematically generated polygons.

The FindPointsInPoly operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

See Also

GraphWaveDraw, **PolygonOp**

FindRoots

`FindRoots` `[`***`flags`***`]` ***`funcspec`*****`,`** ***`pWave`*** `[,` ***`funcspec`*****`,`** ***`pwave`*** `[, …]]`

`FindRoots /P=`***`PolyCoefsWave`***

The FindRoots operation determines roots or zeros of a specified nonlinear function or system of functions. The function or system of functions must be defined in the form of Igor user procedures.

Using the second form of the command, FindRoots finds all the complex roots of a polynomial with real coefficients. The polynomial coefficients are specified by *PolyCoefsWave*.

Flags for roots of nonlinear functions

/B [= *doBracket*]
Specifies bracketing for roots of a single nonlinear function only.

doBracket=0: Skips an initial check of the root bracketing values and the possible search for bracketing values. This means that you must provide good bracketing values via the /L and /H flags. See /L and /H flags for details on bracketing of roots. /B alone is the same as /B=0.

doBracket=1: Uses default root bracketing.

/F=*trustRegion*
Sets the expansion factor of the trust region for the search algorithm when finding roots of systems of functions. Smaller numbers will result in a more stable search, although for some functions larger values will allow the search to zero in on a root more rapidly. Default is 1.0; useful values are usually between 0.1 and 100.

/I=*maxIters*
Sets the maximum number of iterations in searching for a root to *maxIters*. Default is 100.

/L=*lowBracket*
/H=*highBracket*
/L and /H are used only when finding roots of a single nonlinear function. *lowBracket* and *highBracket* are X values that bracket a zero crossing of the function. A root is found between the bracketing values.

If *lowBracket* and *highBracket* are on the same side of zero, it will try to find a minimum or maximum between *lowBracket* and *highBracket*. If it is found, and it is on the other side of zero, Igor will find two roots.

If *lowBracket* and *highBracket* are on the same side of zero, but no suitable extreme point is found between, it will search outward from these values looking for a zero crossing. If it is found, Igor determines one root.

If *lowBracket* and *highBracket* are equal, it adds 1.0 to *highBracket* before looking for a zero crossing.

The default values for *lowBracket* and *highBracket* are zero. Thus, not using either *lowBracket* or *highBracket* is the same as /L=0/H=1.

/Q
Suppresses printout of results in the history area. Ordinarily, the results of root searches are printed in the history.

/T=*tol*
Sets the acceptable accuracy to *tol*. That is, the reported root should be within ±*tol* of the real root.

/X=*xWave*
/X={*x1*, *x2*, …}
Sets the starting point for searching for a root of a system of functions. There must be as many X values as functions. The starting point can be specified with a wave having as many points as there are functions, or you can write out a list of X values in braces. If you are finding roots of a single function, use /L and /H instead.

If you specify a wave, this wave is also used to receive the result of the root search.

/Z=*yValue*
Finds other solutions, that is, places where f(x) = *yValue*. FindRoots usually finds zeroes — places where f(x) = 0.

Flag for roots of polynomials

/P=*PolyCoefsWave*
Specifies a wave containing real polynomial coefficients. With this flag, it finds polynomial roots and does not expect to find user function names on the command line.

The /P flag causes all other flags to be ignored.

Use of this flag is not permitted in a thread-safe function.

Parameters

func specifies the name of a user-defined function.

pwave gives the name of a parameter wave that will be passed to your function as the first parameter. It is not modified. It is intended for your private use to pass adjustable constants to your function.

These parameters occur in pairs. For a one-dimensional problem, use a single *func*, *pwave* pair. An N-dimensional problem requires N pairs unless you use the combined function form (see **Combined Format for Systems of Functions**).

Function Format for 1D Nonlinear Functions

Finding roots of a nonlinear function or system of functions requires that you realize the function in the form of an Igor user function of a certain form. In the FindRoots command you then specify the functions with one or more function names paired with parameter wave names. See **Finding Function Roots** on page III-338 for detailed examples.

The functions must have a particular form. If you are finding the roots of a single 1D function, it should look like this:

```
Function myFunc(w,x)
    Wave w
    Variable x

    return f(x)         // an expression …
End
```

Replace “`f(x)`” with an appropriate expression. The FindRoots command might then look like this:

```
FindRoots /L=0 /H=1 myFunc, cw       // cw is a parameter wave for myFunc
```

Function Format for Systems of Multivariate Functions

If you need to find the roots of a system of multidimensional functions, you can use either of two forms. In one form, you provide N functions with N independent variables. You must have a function for each independent variable. For instance, to find the roots of two 2D functions, the functions must have this form:

```
Function myFunc1(w, x1, x2)
    Wave w
    Variable x1, x2

    return f1(x1, x2)       // an expression …
End

Function myFunc2(w, x1, x2)
    Wave w
    Variable x1, x2

    return f2(x1, x2)       // an expression …
End
```

In this case, the FindRoots command might look like this (where `cw1` and `cw2` are parameter waves that must be made before executing FindRoots):

```
FindRoots /X={0,1} myFunc1, cw1, myFunc2, cw2
```

You can also use a wave to pass in the X values. Make sure you have the right number of points in the X wave — it must have N points for a system of N functions.

```
Function myFunc1(w, xW)
    Wave w, xW

    return f1(xW[0], xW[1])       // an expression …
End

Function myFunc2(w, xW)
    Wave w, xW

    return f2(xW[0], xW[1])       // an expression …
End
```

Combined Format for Systems of Functions

For large systems of equations it may get tedious to write a separate function for each equation, and the FindRoots command line will get very long. Instead, you can write it all in one function that returns N Y values through a Y wave. The X values are passed to the function through a wave with N elements. The parameter wave for such a function must have N columns, one column for each equation. The parameters for equation N are stored in column N-1. FindRoots will complain if any of these waves has other than N rows.

Here is an template for such a function:

```
Function myCombinedFunc(w, xW, yW)
    Wave w, xW, yW

    yW[0] = f1(w[0][...], xW[0], xW[1],..., xW[N-1])
    yW[1] = f2(w[1][...], xW[0], xW[1],..., xW[N-1])
```

```
   …
   yW[N-1] = fN(w[N-1][...], xW[0], xW[1],..., xW[N-1])
End
```

When you use this form, you only have one function and parameter wave specification in the FindRoots command:

```
Make/N=(nrows, nequations) paramWave
fill in paramWave with values
Make/N=(number of equations) guessWave
guessWave = {x0, x1, …, xN}
FindRoots /X=guessWave myCombinedFunc, paramWave
```

FindRoots has no idea how many actual equations you have in the function. If it doesn't match the number of rows in your waves, your results will not be what you expect!

Coefficients for Polynomials

To find the roots of a polynomial, you first create a wave with the correct number of points. For a polynomial of degree N, create a wave with N+1 points. For instance, to find roots of a cubic equation you need a four-point wave.

The first point (row zero) of the wave contains the constant coefficient, the second point contains the coefficient for X, the third for X^2, etc.

There is no hard limit on the maximum degree, but note that there are significant numerical problems associated with computations involving high-degree polynomials. Roundoff error most likely limits reasonably accurate results to polynomials with degree limited to 20 to 30.

Ultimately, if you are willing to accept very limited accuracy, the numerical problems will result in a failure to converge. In limited testing, we found no failures to converge with polynomials up to at least degree 100. At degree 150, we found occasional failures. At degree 200 the failures were frequent, and at degree 500 we found no successes.

Note that you really can't evaluate a polynomial with such high degree, and we have no idea if the computed roots for a degree-100 polynomial have any practical relationship to the actual roots.

While FindRoots is a thread-safe operation, finding polynomial roots is not. Using FindRoots/P=polyWave in a ThreadSafe function results in a compile error.

Results for Nonlinear Functions and Systems of Functions

The FindRoots operation reports success or failure via the V_flag variable. A nonzero value of V_flag indicates the reason for failure.

`V_flag`=0: Success, but check V_YatRoot, V_YatRoot2 or W_YatRoot to make sure that the convergence point is sufficiently small to indicate a true root. It is very unlikely for the Y values to be exactly zero. "Sufficiently small" depends on the scale of your problem.

`V_flag`=1: User abort.

`V_flag`=3: Exceeded maximum allowed iterations

`V_flag`=4: /T=*tol* was too small. Reported by the root finder for systems of nonlinear functions.

`V_flag`=5: The search algorithm wasn't making sufficient progress. It may mean that /T=*tol* was set to too low a value, or that the search algorithm has gotten trapped at a false root. Try restarting from a different starting point.

`V_flag`=6: Unable to bracket a root. Reported when finding roots of single nonlinear functions.

`V_flag`=7: Fewer roots than expected. Reported by the polynomial root finder. This may indicate that roots were successfully found, but some are doubled. Happens only rarely.

`V_flag`=8: Decreased degree. Reported by the polynomial root finder. This indicates that one or more of the highest-order coefficients was zero, and a lower degree polynomial was solved.

`V_flag`=9: Convergence failure or other numerical problem. Reported by the polynomial root finder. This indicates that a numerical problem was detected during the computation. The results are not valid.

The results of finding roots of a single 1D function are put into several variables:

`V_numRoots` The number of roots found. Either 1 or 2.

`V_Root` The root.

`V_YatRoot` The Y value of the function at the root. *Always* check this; some discontinuous functions may give an indication of success, but the Y value at the found root isn't even close to zero.

`V_Root2` Second root if FindRoots found two roots.

`V_YatRoot2` The Y value at the second root.

Results for roots of a system of nonlinear functions are reported in waves:

`W_Root` X values of the root of a system of nonlinear functions. If you used /X=*xWave*, the root is reported in your wave instead.

`W_YatRoot` The Y values of the functions at the root of a system of nonlinear functions.

Only one root is found during a single call to FindRoots.

Roots of a polynomial are reported in a wave:

`W_polyRoots` A complex wave containing the roots of a polynomial. The number of roots should be equal to the degree of the polynomial, unless a root is doubled.

See Also

Finding Function Roots on page III-338.

The FindRoots operation uses the Jenkins-Traub algorithm for finding roots of polynomials:

Jenkins, M.A., Algorithm 493, Zeros of a Real Polynomial, *ACM Transactions on Mathematical Software*, *1*, 178-189, 1975. Used by permission of ACM (1998).

FindSequence

`FindSequence [`*`flags`*`] `*`srcWave`*

The FindSequence operation finds the location of the specified sequence starting the search from the specified start point. The result of the search stored in V_value is the index of the entry in the wave where the first value is found or -1 if the sequence was not found.

Flags

/FNAN Specifies searching for a NaN value when srcWave is floating point.

This flag was added in Igor Pro 7.00.

/I=*wave* Specifies an integer sequence wave for integer search.

/M=*val* If there are repeating entries in the match sequence, *val* is a tolerance value that specifies the maximum difference between the number of repeats. So, for example, if the match sequence is aaabbccc and the *srcWave* contains a sequence aabbcc then the sequence will not be considered a match if *val*=0 but will be considered a match if *val*=1.

/R Searches in reverse from the point in *srcWave* specified by /S or, if you omit /S, from the end of *srcWave*. /R was added in Igor Pro 9.00.

/S=*start* Sets starting point of the search.

If you omit /S, the search starts from the start of *srcWave* or, if you include /R, from the end of *srcWave*.

/T=*tolerance* Defines the tolerance (value ± *tolerance* will be accepted) when comparing floating point numbers.

/U=*uValueWave* Specifies the match sequence wave in case of unsigned long range.

/V=*rValueWave*	Specifies the match sequence wave in the case of single/double precision numbers.
/Z	No error reporting.

Details

If the match sequence is specified via the /V flag, it is considered to be a floating point wave (i.e., single or double precision) in which case it is compared to data in the wave using a tolerance value. If the tolerance is not specified by the /T flag, the default value 1.0^{-7}.

If the match sequence is specified via the /I flag, the sequence is assumed to be an integer wave (this includes both signed and unsigned char, signed and unsigned short as well as long). In this case *srcWave* must also be of integer type and the operation searches for the sequence based on exact equality between the match sequence and entries in the wave as signed long integers.

If the match sequence is unsigned long wave use the /U flag to specify the value for an integer comparison.

You can also use this operation on waves of two or more dimensions. In this case you can calculate the rows, columns, etc. For example, in the case of a 2D wave:

```
col=floor(V_value/rowsInWave)
row=V_value-col*rowsInWave
```

See Also

FindValue, **StringToUnsignedByteWave**

FindValue

`FindValue [`*`flags`*`] `*`srcWave`*

`FindValue [`*`flags`*`] `*`txtWave`*

The FindValue operation finds the location of the specified value starting the search from the specified start point. It stores the results of the search in the variables V_value, V_row, V_col, V_layer, and V_chunk.

Flags

/FNAN	Specifies searching for a NaN value when *srcWave* is floating point. The /FNAN flag was added in Igor Pro 7.00.
/I=*ivalue*	Specifies an integer value for integer search.
/R	Reverses the order of the search. In the absence of /S and /RMD, the search starts from the end of the wave. /R is not compatible with /UOFV. The /R flag was added in Igor Pro 9.00.

/RMD=[*firstRow*,*lastRow*][*firstColumn*,*lastColumn*][*firstLayer*,*lastlayer*][*firstChunk*,*lastChunk*]

Designates a contiguous range of data in the source wave to which the operation is to be applied. This flag was added in Igor Pro 8.00.

You can include all higher dimensions by leaving off the corresponding brackets. For example:

```
/RMD=[firstRow,lastRow]
```

includes all available columns, layers and chunks.

You can use empty brackets to include all of a given dimension. For example:

```
/RMD=[][firstColumn,lastColumn]
```

means "all rows from column A to column B".

You can use a * to specify the end of any dimension. For example:

```
/RMD=[firstRow,*]
```

means "from firstRow through the last row".

/S=*start* Sets start of search in the wave. If /S is not specified, start is set to 0.

/T=*tolerance* Use this flag when comparing floating point numbers to define a non-negative tolerance such that the specified value ± *tolerance* will be accepted.

/TEXT=*templateString*

Specifies a template string that will be searched for in *txtWave*.

/TXOP=*txOptions* Specifies the search options using a combination of binary values.

1:	Case sensitive
2:	Whole word
4:	Whole wave element

/U=*uValue* Specifies the match value in case of unsigned long range.

/UOFV The /UOFV (unordered find value) flag, which was added in Igor Pro 8.00, runs the search using multiple threads with each thread searching a different section of the wave. The search terminates when any thread finds a matching value.

Use /UOFV when you need the fastest result and you do not care if it finds the first matching value in the wave or a subsequent matching value.

By default /UOFV is ignored if *srcWave* contains fewer than 2,000 points. You can modify this value using **MultiThreadingControl**. /UOFV is also ignored if you use the /RMD flag.

/V=*rValue* Specifies the match value in the case of single/double precision numbers. For most purposes you should also use /T to specify the tolerance.

/Z No error reporting.

Details

If the match value is specified via the /V flag, it is considered to be a floating point value in which case it is compared to data in the wave using a tolerance value. If the tolerance is not specified by the /T flag, the value 10^{-7} is used.

If the match value is specified via the /I flag, the value is assumed to be an integer. In this case *srcWave* must be of integer type and the operation searches for the value based on exact equality between the match value and entries in the wave as signed long integers.

If the match value is unsigned long use the /U flag to specify the value for an integer comparison.

The result of the search is stored in the output variables V_value, V_row, V_col, V_layer, and V_chunk. V_value is set to the index in *srcWave*, treating it as 1D regardless of its dimensionality, where the searched value was found or to -1 if it was not found. V_row, V_col, V_layer, and V_chunk are set to the row, column, layer, and chunk number where the searched value was found or to -1 if it was not found but the latter three are set to NaN if the corresponding dimension does not exist in *srcWave*.

When searching for text in a text wave the operation creates the variable V_value as above but it also creates the variable V_startPos to specify the position of *templateString* from the start of the particular wave element.

Example

```
Make jack = sin(x/8)       // Single-precision floating point
Display jack

// This prints -1 because 0.5 +/- 1.0E-7 does not occur in wave jack
FindValue /V=.5 jack; Print V_value

// This prints 21 because 0.5 +/- 0.01 does occur in wave jack
FindValue /V=.5 /T=.01 jack; Print V_value

// The value of jack(21), to 6 decimal digits of precision, is 0.493920
Print jack(21)
```

See Also

FindSequence, **FindLevel**, **FindLevels**, **FindDuplicates**

FitFunc

`FitFunc`

Marks a user function as a user-defined curve fit function. By default, only functions marked with this keyword are displayed in the Function menu in the Curve Fit dialog.

If you wish other functions to be displayed in the Function menu, you can select the checkbox labelled "Show old-style functions (missing FitFunc keyword)".

See Also

User-Defined Fitting Functions on page III-250.

floor

`floor(`*`num`*`)`

The floor function returns the closest integer less than or equal to *num*.

The result for INF and NAN is undefined.

See Also

The **round**, **ceil**, and **trunc** functions.

FMaxFlat

`FMaxFlat [/SYM[=`*`sym`*`] /Z[=z]]` ***`beta`*****`,`** ***`gamma`*****`,`** ***`coefsWave`***

The FMaxFlat operation calculates the coefficients of Kaiser's maximally flat filter.

FMaxFlat is primarily used for the Kaiser maximally flat filter feature of the Igor Filter Design Laboratory (IFDL) package.

Flags

/SYM[=*sym*]	Return symmetrical FIR filter coefficients without a leading length point (see ***Details*** below). The /SYM flag was added in Igor Pro 8.00.
/Z[=z]	Prevents procedure execution from aborting if FMaxFlat generates an error. The /Z flag was added in Igor Pro 8.00. V_Flag is set to a non-zero error code or zero if no error occurred. Use /Z or the equivalent, /Z=1, if you want to handle errors in your procedures rather than having execution abort. Unlike some other operations /Z does suppress invalid beta and gamma value parameter errors.

Parameters

beta is the transition frequency ("cutoff") expressed as a fraction of the sampling frequency, more than 0 and less than 0.5.

gamma is the transition width in fraction of sampling frequency, a number more than 0 and less than 0.5, and less than both beta*2 and 1-2*beta.

coefsWave is the 1D single- or double-precision floating point wave that receives the resulting coefficients. In Igor8 or later, FMaxFlat resizes *coefsWave* as necessary to fit the number of returned values. The upper bound on the number of coefficients can be computed as: `ceil(5/16/gamma/gamma)+1`

Details

The operation is based on the "mxflat" program as found in Elliot and Kaiser (see references below).

Use the /SYM flag to return a *coefsWave* with symmetrical coefficients suitable for use with **FilterFIR**, in which case the number of points in *coefsWave* identifies the number of filter coefficients.

If you omit /SYM or specify /SYM=0, only half of the coefficients are computed by FMaxFlat. The rest can be obtained by symmetry, but the first point of *coefsWave* contains the number of computed coefficients in the designed filter. This unusual format is compatible with pre-Igor8 IFDL procedures.

Example

```
// Make a maximally-flat low pass filter with cutoff at 1/4 sampling frequency
Make/O/D/N=0 coefs                              // coefs will be resized
```

```
FMaxFlat/SYM 0.25, 0.05, coefs                  // Make symmetrical FIR filter
Display coefs

// Analyze the filter's frequency response
FFT/OUT=3/PAD={256}/DEST=coefs_FFT coefs
Display coefs_FFT                               // Filter response for 1Hz sample rate

// Make sample data: a sweep from 0 to 20500 Hz
Make/O/N=1000 data= sin(p*p/1000*pi/2)          // 0 to fs/2
SetScale/P x, 0, 1/41000, "s" data              // 41000 Hz sample rate
Display data

// Analyse unfiltered data's frequency content
FFT/OUT=3/PAD={1000}/DEST=data_FFT  data        // Data frequency response
Display data_FFT

// Apply filter to copy of data
Duplicate/O data, filtered; DelayUpdate
FilterFIR/DIM=0/COEF=coefs filtered
Display filtered

// Analyse filtered data's frequency content
FFT/OUT=3/PAD={1000}/DEST=filtered_FFT  filtered  // Filtered data frequency response
Display filtered_FFT
TileWindows/O=1                                 // Tile Graphs
```

References

Elliot, Douglas F.,contributing editor, *Handbook of Digital Signal Processing Engineering Applications*, Academic Press, San Diego, CA, 1987.

Kaiser, J.F., *Design subroutine (MXFLAT) for symmetric FIR low pass digital filters with maximally flat pass and stop bands*.

IEEE Digital Signal Processing Committee, Editor, *Programs for Digital Signal Processing*, IEEE Press, New York, 1979.

See Also

Remez, **FilterFIR**

FontList

`FontList(`*`separatorStr`*` [, `*`options`*`])`

The FontList function returns a list of the installed fonts, separated by the characters in *separatorStr*.

Parameters

A maximum of 10 bytes from *separatorStr* are appended to each font name as the output string is generated. *separatorStr* is usually ";".

Use *options* to limit the returned font list according to font type. It is restricted to returning only scalable fonts (TrueType, PostScript, or OpenType), which you can do with *options* = 1.

To get a list of nonscalable fonts (bitmap or raster), use:

```
String bitmapFontList = RemoveFromList(FontList(";",1), FontList(";"))
```

(Most Mac OS X fonts are scalable, so bitmapFontList may be empty.)

Examples

```
Function SetFont(fontName)
    String fontName
    Prompt fontName,"font name:",popup,FontList(";")+"default;"
    DoPrompt "Pick a Font", fontName

    Print fontName

    Variable type= WinType("")                  // target window type
    String windowName= WinName(0,127)
    if((type==1) || (type==3) || (type==7))     // graph, panel, layout
        Print "Setting drawing font for "+windowName
        Execute "SetDrawEnv fname=\""+fontName+"\""
    else
        if( type == 5 )                         // notebook
            Print "Setting font for selection in "+windowName
            Notebook $windowName font=fontName
        endif
```

```
   endif
End
```

See Also

The **FontSizeStringWidth**, **FontSizeHeight**, and **WinType** functions, and the **Execute**, **SetDrawEnv**, and **Notebook** Operations.

FontSizeHeight

`FontSizeHeight(`*`fontNameStr`*`, `*`fontSize`*`, `*`fontstyle`*` [,`*`appearanceStr`*`])`

The FontSizeHeight function returns the line height in pixels of any string when rendered with the named font and the given font style and size.

Parameters

fontNameStr is the name of the font, such as `"Helvetica"`.

fontSize is the size (height) of the font in pixels.

fontStyle is text style (bold, italic, etc.). Use 0 for plain text.

Details

The returned height is the sum of the font’s ascent and descent heights. Variations in *fontStyle* and typeface design cause the actual font height to be different than *fontSize* would indicate. (Typically a font “height” refers to only the ascent height, so the total height will be slightly larger to accommodate letters that descend below the baseline, such as g, p, q, and y).

FontSize is in pixels. To obtain the height of a font specified in points, use the **ScreenResolution** function and the conversion factor of 72 points per inch (see Examples).

If the named font is not installed, FontSizeHeight returns NaN.

FontSizeHeight understands “default” to mean the current experiment’s default font.

fontStyle is a binary coded integer with each bit controlling one aspect of the text style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

To set bit 0 and bit 2 (bold, underline), use $2^0+2^2 = 1+4 = 5$ for *fontStyle*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The optional *appearanceStr* parameter has no effect on Windows.

On Macintosh, the *appearanceStr* parameter is used for determining the height of a string drawn by a control. Set *appearanceStr* to "native" if you are measuring the height of a string drawn by a "native GUI" control or to "os9" if not.

Set *appearanceStr* to "default" to use the appearance set by the user in the Miscellaneous Settings dialog. "os9" is the default value.

Usually you will want to set *appearanceStr* to the S_Value output of **DefaultGUIControls**/W=winName when determining the height of a string drawn by a control.

Examples

```
Variable pixels= 12 * ScreenResolution/72          // convert 12 points to pixels
Variable pixelHeight= FontSizeHeight("Helvetica",pixels,0)
Print "Height in points= ", pixelHeight * 72/ScreenResolution
```

```
Function FontIsInstalled(fontName)
   String fontName
   if( numtype(FontSizeHeight(fontName,10,0)) == 2 )
      return 0            // NaN returned, font not installed
   else
      return 1
   endif
End
```

See Also

The **FontList**, **FontSizeStringWidth**, **numtype**, **ScreenResolution**, and **DefaultGUIControls** functions.

FontSizeStringWidth

FontSizeStringWidth(*fontNameStr*, *fontSize*, *fontstyle*, *theStr* [,*appearanceStr*])

The FontSizeStringWidth function returns the width of *theStr* in pixels, when rendered with the named font and the given font style and size.

Parameters

fontNameStr is the name of the font, such as "`Helvetica`".

fontSize is the size (height) of the font in pixels.

fontStyle is text style (bold, italic, etc.). Use 0 for plain text.

theStr is the string whose width is being measured.

The optional *appearanceStr* parameter has no effect on Windows.

On Macintosh, the *appearanceStr* parameter is used for determining the width of a string drawn by a control. Set *appearanceStr* to "native" if you are measuring the width of a string drawn by a "native GUI" control or to "os9" if not.

Set *appearanceStr* to "default" to use the appearance set by the user in the Miscellaneous Settings dialog. "os9" is the default value.

Usually you will want to set *appearanceStr* to the S_Value output of **DefaultGUIControls**/W=winName when determining the width of a string drawn by a control.

Details

If the named font is not installed, FontSizeStringWidth returns NaN.

FontSizeStringWidth understands “default” to mean the current experiment’s default font.

FontSize is in pixels. To obtain the width of a font specified in points, use the **ScreenResolution** function and the conversion factor of 72 points per inch (see Examples).

fontStyle is a binary coded integer with each bit controlling one aspect of the text style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

To set bit 0 and bit 2 (bold, underline), use 2^0+2^2 = 1+4 = 5 for *fontStyle*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Examples

Example 1

```
Variable fsPix= 10 * ScreenResolution/72          // 10 point text in pixels
String text= "How long is this text?"
Variable WidthPix= FontSizeStringWidth("Helvetica",fsPix,0,text)
Print "width in inches= ", WidthPix / ScreenResolution
```

Example 2

```
Variable fsPix= 13 * ScreenResolution/72          // 13 point text in pixels
String text= "text for a control"
DefaultGUIControls/W=Panel0                       // Sets S_Value
Variable WidthPix= FontSizeStringWidth("Helvetica",fsPix,0,text,S_Value)
Print "width in points= ", WidthPix / ScreenResolution * 72
```

See Also

The **FontList**, **FontSizeHeight**, **ScreenResolution** and **DefaultGUIControls** functions.

for-endfor

```
for(<initialization>;<continuation test>;<update>)
   <loop body>
endfor
```

A for-endfor loop executes the loop body code until the continuation test evaluates as false (zero) or until a break statement is executed in the body code. When the loop starts, the initialization expressions are evaluated once. For each iteration, the continuation test is evaluated at the beginning and the update expressions are evaluated at the end.

```
for(<type> varName : <wave>)  // Range-based for loop added in Igor Pro 9.00
   <loop body>
endfor
```

A range-based for loop iterates over each element of a wave. The specified loop variable contains the value of the current wave element.

See Also

For Loop, **Range-Based For Loop**, **break**

for-var-in-wave

```
for(<type> varName : <wave>)  // Range-based for loop added in Igor Pro 9.00
   <loop body>
endfor
```

A range-based for loop iterates over each element of a wave. The specified loop variable contains the value of the current wave element.

See Also

Range-Based For Loop, **For Loop**, **break**

FPClustering

```
FPClustering [flags] srcWave
```

The FPClustering operation performs cluster analysis using the farthest-point clustering algorithm. The input for the operation *srcWave* defines M points in N-dimensional space. Outputs are the waves W_FPCenterIndex and W_FPClusterIndex.

Flags

/CAC	Computes all the clusters specified by /MAXC.
/CM	Computes the center of mass for each cluster. The results are stored in the wave M_clustersCM in the current data folder. Each row corresponds to a single cluster with columns providing the respective dimensional components.
/DSO	Returns the distance map of *srcWave* in M_DistanceMap. No other output is generated and all other flags are ignores. /DSO was added in Igor Pro 8.00. The distance map is the Cartesian distance between any two rows in *srcWave*. The results are stored in the upper triangle of the double-precision output wave M_DistanceMap. The lower triangle is set to zero (results can be obtained by symmetry). Each element of the distance map is given by:

$$M\_DistanceMap_{rc} = \sqrt{\sum_{i=0}^{nCols-1} \left(srcWave[r][i] - srcWave[c][i]\right)^2}$$

/INCD	Computes the inter-cluster distances. The result is stored in the current data folder in the wave M_InterClusterDistance, a 2D wave in which the [*i*][*j*] element contains the distance between cluster *i* and cluster *j*.
/MAXC=*nClusters*	Terminates the calculation when the number of clusters reaches the specified value. Note that this termination condition is sufficient but not necessary, i.e., the operation can terminate earlier if the farthest distance of an element from a hub is less than the average distance.
/MAXR=*maxRad*	Terminates the calculation when the maximum distance is less than or equal to *maxRad*.
/NOR	Normalizes the data on a column by column basis. The normalization makes each columns of the input span the range [0,1] so that even when *srcWave* contains columns that may be different by several orders of magnitude, the algorithm is not biased by a larger implied cartesian distance.
/Q	Don’t print information to the history area.
/SHUB=*sHub*	Specifies the row which is used as a starting hub number. By default the operation uses the first row in *srcWave*.
/Z	No error reporting.

Details

The input for FPClustering is a 2D wave *srcWave* which consists of M rows by N columns where each row represents a point in N-dimensional space. *srcWave* can contain only finite real numbers and must be of type SP or DP. The operation computes the clustering and produces the wave W_FPCenterIndex which contains the centers or “hubs” of the clusters. The hubs are specified by the zero-based row number in *srcWave* which contains the cluster center. In addition, the operation creates the wave W_FPClusterIndex where each entry maps the corresponding input point to a cluster index. By default, the operation continues to add clusters as long as the largest possible distance is greater than the average intercluster distance. You can also stop the processing when the operation has formed a specified number of clusters (see /MAXC).

The variable V_max contains the maximum distance between any element and its cluster hub.

It is possible that in some circumstances you can get slightly different clustering depending on your starting point. The default starting hub is row zero of *srcWave* but you can use the /SHUB flag to specify a different starting point.

FPClustering computes the Cartesian distance between points. As a result, if the scale of any dimension is significantly larger than other dimensions it might bias the clustering towards that dimension. To avoid this situation you can use the /NOR flag which normalizes each column to the range [0,1] and hence equalizes the weight of each dimension in the clustering process.

See Also

The **KMeans** operation.

References

Gonzalez, T., Clustering to minimize the maximum intercluster distance, *Theoretical Computer Science*, *38*, 293-306, 1985.

fprintf

fprintf *refNum*, *formatStr* [, ***parameter***]…

The fprintf operation prints formatted output to a text file.

Parameters

refNum is a file reference number from the **Open** operation used to open the file.

formatStr is the format string, as used by the **printf** operation.

parameter varies depending on *formatStr*.

Details

If *refNum* is 1, fprintf will print to the history area instead of to a file, as if you used printf instead of fprintf. This useful for debugging purposes.

A zero value of *refNum* is used in conjunction with Program-to-Program Communication (PPC), Apple events (*Macintosh*) or **ActiveX Automation** (*Windows*). Data that would normally be written to a file is appended to the PPC, Apple event or ActiveX Automation result packet.

The fprintf operation supports numeric (real only) and string fields from structures. All other structures field types cause a compile error.

Output to Standard Streams

If refNum is -1, Igor prints to the standard output stream (stdout).

If refNum is -2, Igor prints to the standard error stream (stderr).

This feature was added in Igor Pro 8.00. It is useful primarily when you launch Igor from a custom application. The application may be able to capture Igor's standard stream output.

On Macintosh, output printed to either of the standard streams is typically visible only if Igor is started from the Terminal. On Windows, stdout and stderr output is not visible even if Igor is started from the command line.

See Also

The **printf** operation for complete format and parameter descriptions. The **Open** operation and **Creating Formatted Text** on page IV-259.

FReadLine

FReadLine [/N /ENCG=*textEncoding* /T] *refNum*, *stringVarName*

The FReadLine operation reads bytes from a file into the named string variable. The read starts at the current file position and continues until a terminator character is read, the end of the file is reached, or the maximum number of bytes is read.

Parameters

refNum is a file reference number from the **Open** operation used to create the file.

Flags

/N=*n* Specifies the maximum number of bytes to read.

FReadLine can read lines text of length up to about 2 billion bytes.

/ENCG=*textEncoding*

/ENCG={*textEncoding*, *tecOptions* [,*mapErrorMode*]}

Specifies the text encoding of the plain text file being loaded.

This flag was added in Igor Pro 9.00.

See **Text Encoding Names and Codes** on page III-490 for a list of accepted values for *textEncoding*. The 32-bit Unicode text encodings UTF-32BE and UTF-32LE are currently not supported.

If you omit /ENCG, *textEncoding* defaults to binary (255) and does no text encoding conversion but does look for and skip a UTF-8 byte order mark.

For most purposes the default values for *tecOptions* (3) and *mapErrorMode* (1) are fine and you can use /ENCG=*textEncoding* instead of /ENCG={*textEncoding*, *tecOptions* [,*mapErrorMode*]}.

tecOptions is an optional bitwise parameter that controls text encoding conversion. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

tecOptions is defined as follows:

Bit 0: If cleared, the presence of null bytes causes FReadLine to consider the text invalid in all byte-oriented text encodings.

If set (default), null bytes are allowed in byte-oriented text encodings.

Bit 1: If cleared FReadLine does not validate text if the specified text encoding is UTF-8.

If set (default), FReadLine validates text even if the text encoding is UTF-8.

Bit 2: If cleared (default) FReadLine validates the text in the specified text encoding except that validation for UTF-8 is skipped if bit 1 is cleared.

If set, FReadLine assumes that the text is valid in the specified text encoding and does not validate it. Setting this bit makes FReadLine slightly faster but it is usually inconsequential.

UTF-8 because it is not valid in the input text encoding. *mapErrorMode* is one of these values:

1: FReadLine generates an error.

2: Return a substitute character for the unmappable character. The substitute character for Unicode is the Unicode replacement character, U+FFFD. For most non-Unicode text encodings it is either control-Z or a question mark.

3: Skip unmappable input character.

4: Return an escape sequence representing the unmappable code point.

If *mapErrorMode* is 2, 3 or 4, FReadLine does not return an error in the event of an unmappable character.

See the discussion of *mapErrorMode* and the examples for **ConvertTextEncoding** for further discussion.

See **FReadLine and Byte Order Marks** on page V-263 below for further discussion.

/T=*termcharStr* Specifies the terminator character.

`/T=(num2char(13))` specifies carriage return (CR, ASCII code 13).

`/T=(num2char(10))` specifies linefeed (LF, ASCII code 10).

`/T=";"` specifies the terminator as a semicolon.

`/T=""` specifies the terminator as null (ASCII code 0).

See **Details** for default behavior regarding the terminator.

Details

If /N is omitted, there is no maximum number of bytes to read. When reading lines of text from a normal text file, you will not need to use /N. It may be of use in specialized cases, such as reading text embedded in a binary file.

FReadLine can read lines text of length up to about 2 billion bytes.

If /T is omitted, FReadLine will terminate on any of the following: CR, LF, CRLF, LFCR. (Most Macintosh files use CR. Most Windows files use CRLF. Most UNIX files use LF. LFCR is an invalid terminator but some buggy programs generate files that use it.) FReadLine reads whichever of these appears in the file, terminates the read, and returns just a CR in the output string. This default behavior transparently handles files that use CR, LF, CRLF, or LFCR as the terminator and will be suitable for most cases.

If you use the /T flag, then FReadLine will terminate on the specified character only and will return the specified character in the output string.

Once you have read all of the bytes in the file, FReadLine will return zero bytes via *stringVarName*. The example below illustrates testing for this.

FReadLine and Text Encodings

For background information on text encodings, see **Text Encodings** on page III-459.

In Igor Pro 9.00 and later, you can specify the text encoding of the file you are reading using the /ENCG flag.

If you omit /ENCG, FReadLine treats the data as binary and does no text encoding conversion. However, it does skip a UTF-8 byte order mark at the start of the file if present. Treating the data as binary works if the data you are loading is pure ASCII or is valid UTF-8 text.

If you specify a text encoding using /ENCG, the FReadLine converts the text from the text encoding you specify into UTF-8 for internal storage. See **Text Encoding Names and Codes** on page III-490 for a list of accepted values for *textEncoding*. The 32-bit Unicode text encodings UTF-32BE and UTF-32LE are currently not supported.

In general it is not possible to accurately determine the text encoding of a plain text file programmatically so you need to know it *a priori*.

If the text is Unicode and includes a byte order mark then you can programmatically determine the text encoding by examining the byte order mark. You can do this using FBinRead to examine the first bytes of the file:

0xEF, 0xBB, 0xBF	UTF-8 byte order mark, *textEncoding*=1
0xFE, 0xFF	UTF-16BE byte order mark, *textEncoding*=100
0xFF, 0xFE	UTF-16LE byte order mark, *textEncoding*=101

If the file starts with a pattern other than these then you will need *a priori* knowledge or will need to make an assumption about the text encoding.

FReadLine and Byte Order Marks

For background information on byte order marks, see **Byte Order Marks** on page III-471.

If you want to check for the presence of a byte order mark, use **FBinRead** instead of FReadLine.

The default text encoding, used if you omit /ENCG, is binary (255). In this case, FReadLine does no text encoding conversion. However, it does look for a UTF-8 byte order mark and skips it if present.

If you specify UTF-8 or UTF-16 as the text encoding using the /ENCG flag, and if the file data begins with a byte order mark, then FReadLine skips it.

Example

```
Function PrintAllLinesInFile()
	Variable refNum
	Open/R refNum as ""			// Display dialog
	if (refNum == 0)
		return -1				// User canceled
	endif

	Variable lineNumber, len
	String buffer
	lineNumber = 0
	do
		FReadLine refNum, buffer
		len = strlen(buffer)
		if (len == 0)
			break				// No more lines to be read
		endif
		Printf "Line number %d: %s", lineNumber, buffer
		if (CmpStr(buffer[len-1],"\r") != 0)		// Last line has no CR ?
			Printf "\r"
		endif
		lineNumber += 1
	while (1)

	Close refNum
	return 0
End
```

See Also

The **Open** and **FBinRead** operations.

fresnelCos

`fresnelCos(`*`x`*`)`

The fresnelCos function returns the Fresnel cosine function *C*(*x*).

$$C(x) = \int_0^x \cos\left(\frac{\pi}{2}t^2\right)dt.$$

See Also

The **fresnelSin** and **fresnelCS** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

fresnelCS

`fresnelCS(`*`x`*`)`

The fresnelCS function returns both the Fresnel cosine in the real part of the result and the Fresnel sine in the imaginary part of the result.

See Also

The **fresnelSin** and **fresnelCos** functions.

fresnelSin

`fresnelSin(`*`x`*`)`

The fresnelSin function returns the Fresnel sine function *S*(*x*).

$$S(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right)dt.$$

See Also

The **fresnelCos** and **fresnelCS** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

FSetPos

`FSetPos `*`refNum`*`, `*`filePos`*

The FSetPos operation attempts to set the current file position to the given position.

Parameters

refNum is a file reference number obtained from the **Open** operation when the file was opened.

filePos is the desired position of the file in bytes from the start of the file.

Details

FSetPos generates an error if *filePos* is greater than the number of bytes in the file. You can ascertain this limit with the **FStatus** operation.

When a file that is open for writing is closed, any bytes past the end of the current file position are deleted by the operating system. Therefore, if you use FSetPos, make sure to set the current file position properly before closing the file.

FSetPos supports files of any length.

See Also

Open, **FGetPos**, **FStatus**

FStatus

`FStatus` *`refNum`*

The FStatus operation provides file status information for a file.

Parameters

refNum is a file reference number obtained from the **Open** operation.

Details

FStatus supports files of any length.

FStatus sets the following variables:

`V_flag`	Nonzero (true) if *refNum* is valid, in which case FStatus sets the other variables as well.
`V_filePos`	Current file position for the file in bytes from the start. In Igor7 or later, if you only want to know the current file position, use **FGetPos** instead of FStatus, which is slower.
`V_logEOF`	Total number of bytes in the file.
`S_fileName`	Name of the file.
`S_path`	Path from the volume to the folder containing the file. For example, "hd:Folder1:Folder2:". This is suitable for use as an input to the **NewPath** operation. Note that on the Windows operating system Igor uses a colon between folders instead of the Windows-standard backslash to avoid confusion with Igor's use of backslash to start an escape sequence (see **Escape Sequences in Strings** on page IV-14).
`S_info`	Keyword-packed information string.

The keyword-packed information string for S_info consists of a sequence of sections with the following form: *keyword*:*value*; You can pick a value out of a keyword-packed string using the **NumberByKey** and **StringByKey** functions.

Here are the keywords for S_info:

Keyword	Type	Meaning
PATH	string	Name of the symbolic path in which the file is located. This will be empty if there is no such symbolic path.
WRITEABLE	number	1 if file can be written to, 0 if not.

See Also

Open, **FGetPos**, **FSetPos**

FTPCreateDirectory

`FTPCreateDirectory` `[`***`flags`***`]` ***`urlStr`***

The FTPCreateDirectory operation creates a directory on an FTP server on the Internet.

For background information on Igor's FTP capabilities and other important details, see **File Transfer Protocol (FTP)** on page IV-272.

FTPCreateDirectory sets V_flag to zero if the operation succeeds or to a non-zero error code if it fails.

If the directory specified by *urlStr* already exists on the server, the server contents are not touched and V_flag is set to -1. This is not treated as an error.

Parameters

urlStr specifies the directory to create. It consists of a naming scheme (always "ftp://"), a computer name (e.g., "ftp.wavemetrics.com" or "38.170.234.2"), and a path (e.g., "/test/newDirectory"). For example:

```
   "ftp://ftp.wavemetrics.com/test/newDirectory"
```

urlStr must always end with a directory name, and must not end with a slash.

To indicate that *urlStr* contains an absolute path, insert an extra '/' character between the computer name and the path. For example:

```
ftp://ftp.wavemetrics.com//pub/test/newDirectory
```

If you do not specify that the path in *urlStr* is absolute, it is interpreted as relative to the FTP user's base directory. Since pub is the base directory for an anonymous user at wavemetrics.com, these URLs reference the same directory for an anonymous user:

```
ftp://ftp.wavemetrics.com//pub/test/newDirectory  // Absolute path
ftp://ftp.wavemetrics.com/test/newDirectory       // Relative to base directory
```

Special characters, such as punctuation, that are used in *urlStr* may be incorrectly interpreted by the operation. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. See **Percent Encoding** on page IV-268 for additional information.

Flags

/N=*portNumber* Specifies the server's TCP/IP port number to use (default is 21). In almost all cases, the default will be correct so you won't need to use the /N flag.

/U=*userNameStr* Specifies the user name to be used when logging in to the FTP server. If /U is omitted or if *userNameStr* is "", the login is done as an anonymous user. Use /U if you have an account on the FTP server.

/V=*diagnosticMode* Determines what kind of diagnostic messages FTPCreateDirectory will display in the history area. *diagnosticMode* is a bitwise parameter, with the bits defined as follows:

Bit 0: Show basic diagnostics. Currently this just displays the URL in the history.

Bit 1: Show errors. This displays additional information when errors occur.

Bit 2: Show status. This displays commands sent to the server and the server's response.

The default value for *diagnosticMode* is 3 (show basic and error diagnostics). If you are having difficulties, you can try using 7 to show the commands sent to the server and the server's response.

See **FTP Troubleshooting** on page IV-275 for other troubleshooting tips.

/W=*passwordStr* Specifies the password to be used when logging in to the FTP server. Use /W if you have an account on the FTP server.

If /W is omitted, the login is done using a default password that will work with most anonymous FTP servers.

See **Safe Handling of Passwords** on page IV-270 for information on handling sensitive passwords.

/Z Errors are not fatal. Will not abort procedure execution if an error occurs.

Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded, -1 if the specified directory already exists, or another nonzero value if an error occurred.

Examples

```
// Create a directory.
String url = "ftp://ftp.wavemetrics.com/pub/test/newDirectory"
FTPCreateDirectory url
```

See Also

File Transfer Protocol (FTP) on page IV-272.

FTPDelete, **FTPDownload**, **FTPUpload**, **URLEncode**

FTPDelete

`FTPDelete [`*`flags`*`]` ***`urlStr`***

The FTPDelete operation deletes a file or a directory from an FTP server on the Internet.

Warning: If you delete a directory on an FTP server, all contents of that directory and any subdirectories are also deleted.

For background information on Igor's FTP capabilities and other important details, see **File Transfer Protocol (FTP)** on page IV-272.

FTPDelete sets V_flag to zero if the operation succeeds and to nonzero if it fails. This, in conjunction with the /Z flag, can be used to allow procedures to continue to execute if an FTP error occurs.

Parameters

urlStr specifies the file or directory to delete. It consists of a naming scheme (always "ftp://"), a computer name (e.g., "ftp.wavemetrics.com" or "38.170.234.2"), and a path (e.g., "/test/TestFile1.txt"). For example: "ftp://ftp.wavemetrics.com/test/TestFile1.txt"

urlStr must always end with a file name if you are deleting a file or with a directory name if you are deleting a directory. In the case of a directory, *urlStr* must not end with a slash.

To indicate that *urlStr* contains an absolute path, insert an extra '/' character between the computer name and the path. For example:

```
ftp://ftp.wavemetrics.com//pub/test
```

If you do not specify that the path in *urlStr* is absolute, it is interpreted as relative to the FTP user's base directory. Since pub is the base directory for an anonymous user at wavemetrics.com, these URLs reference the same directory for an anonymous user:

```
ftp://ftp.wavemetrics.com//pub/test
ftp://ftp.wavemetrics.com/test
```

Special characters such as punctuation that are used in *urlStr* may be incorrectly interpreted by the operation. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. See **Percent Encoding** on page IV-268 for additional information

Flags

/D Deletes a complete directory and all its contents. Omit /D if you are deleting a file.

/N=*portNumber* Specifies the server's TCP/IP port number to use (default is 21). In almost all cases, the default will be correct so you won't need to use the /N flag.

/U=*userNameStr* Specifies the user name to be used when logging in to the FTP server. If /U is omitted or if userNameStr is "", the login is done as an anonymous user. Use /U if you have an account on the FTP server.

/V=*diagnosticMode* Determines what kind of diagnostic messages FTPDelete will display in the history area. *diagnosticMode* is a bitwise parameter, with the bits defined as follows:

Bit 0: Show basic diagnostics. Currently this just displays the URL in the history.

Bit 1: Show errors. This displays additional information when errors occur.

Bit 2: Show status. This displays commands sent to the server and the server's response.

The default value for *diagnosticMode* is 3 (show basic and error diagnostics). If you are having difficulties, you can try using 7 to show the commands sent to the server and the server's response.

See **FTP Troubleshooting** on page IV-275 for other troubleshooting tips.

/W=*passwordStr* Specifies the password to be used when logging in to the FTP server. Use /W if you have an account on the FTP server.

If /W is omitted, the login is done using a default password that will work with most anonymous FTP servers.

See **Safe Handling of Passwords** on page IV-270 for information on handling sensitive passwords.

/Z Errors are not fatal. Will not abort procedure execution if an error occurs.

Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded, or a nonzero value if an error occurred.

Examples

```
// Delete a file.
String url = "ftp://ftp.wavemetrics.com/test/TestFile1.txt"
FTPDelete url

// Delete a directory.
String url = "ftp://ftp.wavemetrics.com/test/TestDir1"
FTPDelete/D url
```

See Also

File Transfer Protocol (FTP) on page IV-272.

FTPCreateDirectory, **FTPDownload**, **FTPUpload**, **URLEncode**

FTPDownload

FTPDownload [***flags***] ***urlStr***, ***localPathStr***

The FTPDownload operation downloads a file or a directory from an FTP server on the Internet.

Warning: When you download a file or directory using the path and name of a file or directory that already exists on your local hard disk, all previous contents of the local file or directory are obliterated.

For background information on Igor’s FTP capabilities and other important details, see **File Transfer Protocol (FTP)** on page IV-272.

FTPDownload sets a variable named V_flag to zero if the operation succeeds and to nonzero if it fails. This, in conjunction with the /Z flag, can be used to allow procedures to continue to execute if a FTP error occurs.

If the operation succeeds, FTPDownload sets a string named S_Filename to the full file path of the downloaded file or, if the /D flag was used, the full path to the base directory that was downloaded. This is useful in conjunction with the /I flag.

If the operation fails, S_Filename is set to "".

Parameters

urlStr specifies the file or directory to download. It consists of a naming scheme (always "`ftp://`"), a computer name (e.g., "`ftp.wavemetrics.com`" or "`38.170.234.2`"), and a path (e.g., "`/Test/TestFile1.txt`"). For example: "`ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt`".

urlStr must always end with a file name if you are downloading a file or with a directory name if you are downloading a directory. In the case of a directory, *urlStr* must not end with a slash.

To indicate that *urlStr* contains an absolute path, insert an extra '/' character between the computer name and the path. For example:

```
ftp://ftp.wavemetrics.com//pub/test
```

If you do not specify that the path in *urlStr* is an absolute path, it is interpreted as a path relative to the FTP user's base directory. Since pub is the base directory for an anonymous user, this URL references the same directory:

```
ftp://ftp.wavemetrics.com/test
```

Special characters such as punctuation that are used in *urlStr* may be incorrectly interpreted by the operation. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. See **Percent Encoding** on page IV-268 for additional information.

localPathStr and *pathName* specify the name to use for the file or directory that will be created on your hard disk. If you use a full or partial path for *localPathStr*, see **Path Separators** on page III-451 for details on forming the path.

localPathStr must always end with a file name if you are downloading a file or with a directory name if you are downloading a directory. In the case of a directory, *localPathStr* must not end with a colon or backslash.

FTPDownload displays a dialog through which you can identify the local file or directory in the following cases:

1. You have used the /I (interactive) flag.
2. You did not completely specify the location of the local file or directory via *pathName* and *localPathStr*.
3. There is an error in *localPathStr*. This can be either a syntactical error or a reference to a nonexistent file or directory.
4. The specified local file or directory exists and you have not used the /O (overwrite) flag.

See **Examples** for examples of constructing a URL and local path.

Flags

/D Downloads a complete directory. Omit it if you are downloading a file.

/I Interactive mode which will prompt you to specify the name and location of the file or directory to be created on the local hard disk.

/M=*messageStr* Specifies the prompt message used by the dialog in which you specify the name and location of the file or directory to be created.

/N=*portNumber* Specifies the server's TCP/IP port number to use (default is 21). In almost all cases, this will be correct so you won't need to use the /N flag.

/O[=*mode*] Controls whether a local file or directory whose name is in conflict with the file or directory being downloaded is overwritten without prompting the user.

mode=0: Prompts the user to allow the overwrite. This is the default behavior if /O is omitted.

mode=1: Overwrites without prompting the user. If the /D flag is also used, all contents of the destination directory are deleted if it already exists. /O=1 is the same as /O.

mode=2: Merges files and subdirectories downloaded with the contents of the destination directory. Unlike /O=1, the contents of the destination directory are not deleted, however files and directories downloaded from the server will overwrite existing files and directories of the same name. When downloading a file this mode is accepted but has the same effect as /O=1.

/P=*pathName* Contributes to the specification of the file or directory to be created on your hard disk. *pathName* is the name of an existing symbolic path. See **Examples**.

/S=*showProgress* Determines if a progress dialog is displayed.

0: No progress dialog.
1: Show a progress dialog (default).

/T=*transferType* Controls the FTP transfer type.

0: Image (binary) transfer (default).
1: ASCII transfer.

See **FTP Transfer Types** on page IV-275 for more discussion.

/U=userNameStr Specifies the user name to be used when logging in to the FTP server. If this flag is omitted or if *userNameStr* is `""`, you will be logged in as an anonymous user. Use this flag if you have an account on the FTP server.

/V=diagnosticMode Determines what kind of diagnostic messages FTPDownload will display in the history area. *diagnosticMode* is a bitwise parameter, with the bits defined as follows:

Bit 0: Show basic diagnostics. Currently this just displays the URL in the history.

Bit 1: Show errors. This displays additional information when errors occur.

Bit 2: Show status. This displays commands sent to the server and the server's response.

The default value for *diagnosticMode* is 3 (show basic and error diagnostics). If you are having difficulties, you can try using 7 to show the commands sent to the server and the server's response.

See **FTP Troubleshooting** on page IV-275 for other troubleshooting tips.

/W=passwordStr Specifies the password to be used when logging in to the FTP server. Use this flag if you have an account on the FTP server.

If this flag is omitted, “nopassword” will be used for the login password. This will work with most anonymous FTP servers. Some anonymous FTP servers request that you use your email address as a password. You can do this by including the /W=“<your email address>” flag.

If /W is omitted, the login is done using a default password that will work with most anonymous FTP servers.

See **Safe Handling of Passwords** on page IV-270 for information on handling sensitive passwords.

/Z Errors are not fatal. Will not abort procedure execution if an error occurs.

Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded, -1 if the user canceled in an interactive dialog, or another nonzero value if an error occurred.

Examples

Download a file using a full local path:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String localPath = "hd:Test Folder:TestFile1.txt"
FTPDownload url, localPath
```

Download a file using a local symbolic path and file name:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String pathName = "Igor"      // Igor is the name of a symbolic path.
String fileName = "TestFile1.txt"
FTPDownload/P=$pathName url, fileName
```

Download a directory using a full local path:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestDir1"
String localPath = "hd:Test Folder:TestDir1"
FTPDownload/D url, localPath
```

See Also

File Transfer Protocol (FTP) on page IV-272.

FTPCreateDirectory, **FTPDelete**, **FTPUpload**, **URLEncode**, **FetchURL**.

FTPUpload

FTPUpload [*flags*] *urlStr*, *localPathStr*

The FTPUpload operation uploads a file or a directory to an FTP server on the Internet.

Warning: When you upload a file or directory to an FTP server, all previous contents of the server file or directory are obliterated.

For background information on Igor’s FTP capabilities and other important details, see **File Transfer Protocol (FTP)** on page IV-272.

FTPUpload sets a variable named V_flag to zero if the operation succeeds and to nonzero if it fails. This, in conjunction with the /Z flag, can be used to allow procedures to continue to execute if a FTP error occurs.

If the operation succeeds, FTPUpload sets a string named S_Filename to the full file path of the uploaded file or, if the /D flag was used, to the full path to the base directory that was uploaded. This is useful in conjunction with the /I flag.

If the operation fails, S_Filename is set to "".

Parameters

urlStr specifies the file or directory to create. It consists of a naming scheme (always "`ftp://`"), a computer name (e.g., "`ftp.wavemetrics.com`" or "`38.170.234.2`"), and a path (e.g., "`/Test/TestFile1.txt`"). For example: "`ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt`".

urlStr must always end with a file name if you are uploading a file or with a directory name if you are uploading a directory, in which case *urlStr* must not end with a slash.

To indicate that *urlStr* contains an absolute path, insert an extra '/' character between the computer name and the path. For example:

```
ftp://ftp.wavemetrics.com//pub/test
```

If you do not specify that the path in *urlStr* is an absolute path, it is interpreted as a path relative to the FTP user's base directory. Since pub is the base directory for an anonymous user, this URL references the same directory:

```
ftp://ftp.wavemetrics.com/test
```

Special characters such as punctuation that are used in *urlStr* may be incorrectly interpreted by the operation. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. See **Percent Encoding** on page IV-268 for additional information.

localPathStr and *pathName* specify the name and location on your hard disk of the local file to be uploaded. If you use a full or partial path for *localPathStr*, see **Path Separators** on page III-451 for details on forming the path.

localPathStr must always end with a file name if you are uploading a file or with a directory name if you are uploading a directory. In the case of a directory, *localPathStr* must not end with a colon or backslash.

FTPUpload displays a dialog that you can use to identify the file or directory to be uploaded in the following cases:

1. You used the /I (interactive) flag.
2. You did not completely specify the location of the file or folder to be uploaded via *pathName* and *localPathStr*.
3. There is an error in *localPathStr*. This can be either a syntactical error or a reference to a nonexistent directory.

See **Examples** for examples of constructing a URL and local path.

Flags

/D	Uploads a complete directory. Omit it if you are uploading a file.
/I	Interactive mode which displays a dialog for choosing the local file or directory to be uploaded.

/M=*messageStr* Specifies the prompt message used by the dialog in which you choose the local file or directory to be uploaded.

/N=*portNumber* Specifies the server’s TCP/IP port number to use (default is 21). In almost all cases, this will be correct so you won’t need to use the /N flag.

/O[=*mode*] Overwrite. FTPUpload *always* overwrites the specified server file or directory, whether /O is used or not.

If /O=2 is *not* used, all files and subdirectories in the destination directory on the server are first deleted and then the local files and directories are uploaded to the server.

If /O=2 *is* used, the existing contents the contents of the local source directory are merged into the remote directory instead of completely overwriting it.

/P=*pathName* Contributes to the specification of the file or directory to be uploaded. *pathName* is the name of an existing symbolic path. See **Examples**.

/S=*showProgress* Determines if a progress dialog is displayed.

0: No progress dialog.

1: Show a progress dialog (default).

/T=*transferType* Controls the FTP transfer type.

0: Image (binary) transfer (default).

1: ASCII transfer.

See **FTP Transfer Types** on page IV-275 for more discussion.

/U=*userNameStr* Specifies the user name to be used when logging in to the FTP server. If this flag is omitted or if *userNameStr* is `""`, you will be logged in as an anonymous user. Use this flag if you have an account on the FTP server.

/V=*diagnosticMode* Determines what kind of diagnostic messages FTPUpload will display in the history area. *diagnosticMode* is a bitwise parameter, with the bits defined as follows:

Bit 0: Show basic diagnostics. Currently this just displays the URL in the history.

Bit 1: Show errors. This displays additional information when errors occur.

Bit 2: Show status. This displays commands sent to the server and the server's response.

The default value for *diagnosticMode* is 3 (show basic and error diagnostics). If you are having difficulties, you can try using 7 to show the commands sent to the server and the server's response.

See **FTP Troubleshooting** on page IV-275 for other troubleshooting tips.

/W=*passwordStr* Specifies the password used when logging in to the FTP server. Use this flag if you have an account on the FTP server.

If this flag is omitted, “nopassword” will be used for the login password. This will work with most anonymous FTP servers. Some anonymous FTP servers request that you use your email address as a password. You can do this by including the /W=“<your email address>” flag.

If /W is omitted, the login is done using a default password that will work with most anonymous FTP servers.

See **Safe Handling of Passwords** on page IV-270 for information on handling sensitive passwords.

/Z Errors are not fatal. Will not abort procedure execution if an error occurs.

Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded, -1 if the user canceled in an interactive dialog, or another nonzero value if an error occurred.

Examples

Upload a file using a full local path:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String localPath = "hd:Test Folder:TestFile1.txt"
FTPUpload url, localPath
```

Upload a file using a local symbolic path and file name:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String pathName = "Igor"      // Igor is the name of a symbolic path.
String fileName = "TestFile1.txt"
FTPUpload/P=$pathName url, fileName
```

Upload a directory using a full local path:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestDir1"
String localPath = "hd:Test Folder:TestDir1"
FTPUpload/D url, localPath
```

See Also

File Transfer Protocol (FTP) on page IV-272.

FTPCreateDirectory, **FTPDelete**, **FTPDownload**, **URLEncode**.

FuncFit

FuncFit [***flags***] ***fitFuncName*, *cwaveName*, *waveName*** [***flag parameters***]

FuncFit [***flags***] **{*fitFuncSpec*}, *waveName*** [***flag parameters***]

The FuncFit operation performs a curve fit to a user defined function, or to a sum of fit functions using the second form (see **Fitting Sums of Fit Functions** on page V-275). Fitting can be done using any method that can be selected using the /ODR flag (see **CurveFit** for details).

FuncFit operation parameters are grouped in the following categories: flags, parameters (*fitFuncName*, *cwaveName*, *waveName* or {*fitFuncSpec* }, *waveName*), and flag parameters. The sections below correspond to these categories. Note that flags must precede the *fitFuncName* or `fitFuncSpec` and flag parameters must follow *waveName*.

In Igor Pro 9.00 and later, your fitting function can return complex values. The sections below indicate requirements for complex fitting functions. For details, see **Fitting with Complex-Valued Functions** on page III-248.

Flags

See **CurveFit** for all available flags.

Parameters

fitFuncName The user-defined function to fit to, which can be a function taking multiple independent variables (see also **FuncFitMD**). Multivariate fitting with FuncFit *requires* /X=*xwaveSpec*. To fit complex data waves, your fitting function must be declared complex (Function/C).

cwaveName Wave containing the fitting coefficients. If the fitting function is declared with a complex wave for the coefficient wave, then this wave must be complex.

waveName The wave containing the dependent variable data to be fit to the specified function. For functions of just one independent variable, the dependent variable data is often referred to as "Y data". You can fit to a subrange of the wave by supplying (*startX*,*endX*) or [*startP*,*endP*] after the wave name. See **Wave Subrange Details** below for more information on subranges of waves in curve fitting. If your fitting function returns complex values then this wave must be complex.

fitFuncSpec List of fit functions and coefficient waves, with some optional information. Using this format fits a model consisting of the sum of the listed fit functions. Intended for fitting multiple peaks, but probably useful for other applications as well. See **Fitting Sums of Fit Functions** on page V-275. Does not support complex fitting functions.

Flag Parameters

These flag parameters must follow *waveName*.

/*E=ewaveName* A wave containing the epsilon values for each parameter. Must be the same length as the coefficient wave. If your fitting function requires a complex coefficient wave then your epsilon wave must be complex. Not supported with complex fitting functions.

/STRC=*structureInstance*

Used only with **Structure Fit Functions** on page III-261. When using a structure fit function, you must specify an instance of the structure to FuncFit. This will be an instance that has been initialized by a user-defined function that you write in order to invoke FuncFit.

/X=*xwaveSpec* An optional wave containing X values for each of the input data values. If the fitting function has more than one independent variable, *xwaveSpec* is required and must be either a 2D wave with a column for each independent variable, or a list of waves, one for each independent variable. A list must be in braces: /X={*xwave0*, *xwave1*,…}. There must be exactly one column or wave for each independent variable in the fitting function. If your fitting function takes complex X values then your X waves must be complex.

/NWOK Allowed in user-defined functions only. When present, certain waves may be set to null wave references. Passing a null wave reference to FuncFit is normally treated as an error. By using /NWOK, you are telling FuncFit that a null wave reference is not an error but rather signifies that the corresponding flag should be ignored. This makes it easier to write function code that calls FuncFit with optional waves.

The waves affected are the X wave or waves (/X), weight wave (/W), epsilon wave (/E) and mask wave (/M). The destination wave (/D=wave) and residual wave (/R=wave) are also affected, but the situation is more complicated because of the dual use of /D and /R to mean "do autodestination" and "do autoresidual". See /AR and /AD.

If you don't need the choice, it is better not to include this flag, as it disables useful error messages when a mistake or run-time situation causes a wave to be missing unexpectedly.

Note: To work properly this flag must be the last one in the command.

Other parameters are used as for the **CurveFit** operation, with some exceptions for multivariate fits.

Details for Multivariate Fits

The dependent variable data wave, *waveName*, must be a 1D wave even for multivariate fits. For fits to data in a multidimensional wave, see **FuncFitMD**.

For multivariate fits, the auto-residual (/R with no wave specified) is calculated and appended to the top graph if the dependent variable data wave is graphed in the top graph as a simple 1D trace. Auto residuals are calculated but not displayed if the data are displayed as a contour plot.

The autodest wave (/D with no wave specified) for multivariate fits has the same number of points as the data wave, with a model value calculated at the X values contained in the wave or waves specified with /X=*xwaveSpec*.

Confidence bands are not supported for multivariate fits.

Wave Subrange Details

Almost any wave you specify to FuncFit can be a subrange of a wave. The syntax for wave subranges is the same as for the Display command (see **Subrange Display Syntax** on page II-321 for details). See **Wave Subrange Details** on page V-132 for a discussion of the use of subranges in curve fitting.

The backwards compatibility rules for CurveFit apply to FuncFit as well.

In addition to the waves discussed in the CurveFit documentation, it is possible to use subranges when specifying the coefficient wave and the epsilon wave. Since the coefficient wave and epsilon wave must have the same number of points, it might make sense to make them two columns from a single multicolumn wave. For instance, here is an example in which the first column is used as the coefficient wave, the second is used as the epsilon wave, and the third is used to save a copy of the initial guesses for future reference:

```
Make/D/N=(5, 3) myCoefs
myCoefs[][0] = {1,2,3,4,5}             // hypothetical initial guess
myCoefs[][1] = 1e-6                    // reasonable epsilon values
myCoefs[][2] = myCoefs[p][0]           // save copy of initial guess
FuncFit myFitFunc, myCoefs[][0] myData /E=myCoefs[][1] …
```

You might have a fit function that uses a subset of the coefficients that are used by another. It might be useful to use a single wave for both. Here is an example in which a function that takes four coefficients is used to fit a subset of the coefficients, and then that solution is used as the initial guess for a function that takes six coefficients:

```
Make/D/N=6 Coefs6={1,2,3,4,5,6}
FuncFit Fit4Coefs, Coefs6[0,3] fitfunc4Coefs …
FuncFit Fit6Coefs, Coefs6 ...
```

Naturally, the two fit functions must be worked out carefully to allow this.

Fitting Sums of Fit Functions

If Igor encounters a left brace at the beginning of the fit function name, it expects a list of fit functions to be summed during the fit. This is useful for, for instance, fitting several peaks in a data set to a sum of peak functions.

The fit function specification includes at least the name of the fitting function and an associated coefficient wave. A sum of fit functions requires multiple coefficient waves, one for each fit function. Any coefficient wave-related options must be specified in the fit function specification via keyword-value pairs.

The syntax of the sum-of-fit-functions specification is as follows:

`{{`*`func1`*`, `*`coef1`*`, `*`keyword`*`=`*`value`*`},{`*`func2`*`, `*`coef2`*`, `*`keyword`*`=`*`value`*`}, …}`

or

`{`*`string=fitSpecStr`*`}`

Within outer braces, each fit function specification is enclosed within inner braces. You can use one or more fit function specifications, with no intrinsic limit on the number of fit functions.

The second format is available to overcome limitations on the length of a command line in Igor. This format is just like the first, but everything inside the outer braces is contained in a string expression (which may be just a single string variable).

You can use any fit function that can be used for ordinary fitting, including the built-in functions that are available using the CurveFit operation. If you should write a user-defined fitting function with the same name as a built-in fit function, the user-defined function will be used (this is strongly discouraged).

Every function specification must include an appropriate coefficient wave, pre-loaded with initial guesses.

The comma between each function specification is optional.

The keyword-value pairs are optional, and are used to communicate further options on a function-by-function basis. Available keywords are:

HOLD=*holdstr*

Indicates that a fit coefficient should be held fixed during fitting. *holdstr* works just like the hold string specified via the /H flag for normal fitting, but applies only to the coefficient wave associated with the fit function it appears with.

If you include HOLD in a string expression (the `{string=fitSpecStr}` syntax), you must escape the quotation marks around the hold string.

If you use the command-line syntax `{{func1,coef1,HOLD=`*`holdStr`*`}, ...}`, *holdStr* may be a reference to a global variable acquired using SVAR, or it may be a quoted literal string.

If you use `{string=`*`fitSpecStr`*`}`, *fitSpecStr* is parsed at run-time outside the context of any running function. Consequently, you cannot use a general string expression. You can use either `HOLD="quotedLiteralString"` or `HOLD=root:globalString`.

CONST={*constants*} Sets the values of constants in the fitting function. So far, only two built-in functions take constants: exp_XOffset and dblexp_XOffset. They each take just one constant (the X offset), so you will have a “list” of one number inside the braces.

EPSW=*epsilonWave* Specifies a wave holding epsilon values. Use only with a user-defined fitting function to set the differencing interval used to calculate numerical estimates of derivatives of the fitting function.

STRC=*structureInstance* Specifies an instance of the structure to FuncFit when using a structure fit function. *structureInstance* is an instance that was initialized by a user-defined function that invokes FuncFit. This keyword (and structure fitting functions) can be used only when calling FuncFit from within a user-defined function. See **Structure Fit Functions** on page III-261 for more details.

For more details, and for examples of sums of fit functions in use, **Fitting Sums of Fit Functions** on page III-244.

See Also

The **CurveFit** operation for parameter details. See also **FuncFitMD** for user-defined multivariate fits to data in a multidimensional wave.

The best way to create a user-defined fitting function is using the Curve Fitting dialog. See **Using the Curve Fitting Dialog** on page III-181, especially the section **Fitting to a User-Defined Function** on page III-190.

For details on the form of a user-defined function, see **User-Defined Fitting Functions** on page III-250.

FuncFitMD

```
FuncFitMD [flags] fitFuncName, cwaveName, waveName [flag parameters]
```

The FuncFitMD operation performs a curve fit to the specified multivariate user defined *fitFuncSpec*. FuncFitMD handles gridded data sets in multidimensional waves. Most parameters and flags are the same as for the **CurveFit** and **FuncFit** operations; differences are noted below.

cwaveName is a 1D wave containing the fitting coefficients, and *functionName* is the user-defined fitting function, which has 2 to 4 independent variables.

FuncFitMD operation parameters are grouped in the following categories: flags, parameters (*fitFuncName*, *cwaveName*, *waveName*), and flag parameters. The sections below correspond to these categories. Note that flags must precede the *fitFuncName* and flag parameters must follow *waveName*.

Flags

/L=*dimSize*	Sets the dimension size of the wave created by the auto-trace feature, that is, /D without destination wave. The wave fit_*waveName* will be a multidimensional wave of the same dimensionality as *waveName* that has *dimSize* elements in each dimension. That is, if you are fitting to a matrix wave, fit_*waveName* will be a square matrix that has dimensions *dimSize* X*dimSize*. **Beware**: *dimSize* =100 requires 100 million points for a 4-dimensional wave!

Parameters

fitFuncName	User-defined function to fit to, which must be a function taking 2 to 4 independent variables.
cwaveName	1D wave containing the fitting coefficients.
waveName	The wave containing the dependent variable data to be fit to the specified function. For functions of just one independent variable, the dependent variable data is often referred to as "Y data". You can fit to a subrange of the wave by supplying (*startX*,*endX*) or [*startP*,*endP*] for each dimension after the wave name. See **Wave Subrange Details** below for more information on subranges of waves in curve fitting.

Flag Parameters

These flag parameters must follow *waveName*.

/E=*ewaveName*	A wave containing the epsilon values for each parameter. Must be the same length as the coefficient wave.
/T=*twaveName*	Like /X except for the T independent variable. This is a 1D wave having as many elements as *waveName* has chunks.
/X=*xwaveName*	The X independent variable values for the data to fit come from *xwaveName* instead of from the X scaling of *waveName*. This is a 1D wave having as many elements as *waveName* has rows.
/Y=*ywaveName*	Like /X except for the Y independent variable. This is a 1D wave having as many elements as *waveName* has columns.
/Z=*ywaveName*	Like /X except for the Z independent variable. This is a 1D wave having as many elements as *waveName* has layers.
/NWOK	Allowed in user-defined functions only. When present, certain waves may be set to null wave references. Passing a null wave reference to FuncFitMD is normally treated as an error. By using /NWOK, you are telling FuncFitMD that a null wave reference is not an error but rather signifies that the corresponding flag should be ignored. This makes it easier to write function code that calls FuncFitMD with optional waves. The waves affected are the X wave or waves (/X), the Y spacing wave (/Y), the Z spacking wave (/Z) the T spacing wave (/T), weight wave (/W), epsilon wave (/E) and mask wave (/M). The destination wave (/D=wave) and residual wave (/R=wave) are also affected, but the situation is more complicated because of the dual use of /D and /R to mean "do autodestination" and "do autoresidual". See /AR and /AD. If you don't need the choice, it is better not to include this flag, as it disables useful error messages when a mistake or run-time situation causes a wave to be missing unexpectedly. **Note**: To work properly this flag must be the last one in the command.

Details

Auto-residual (/R with no wave specified) and auto-trace (/D with no wave specified) for functions having two independent variables are plotted in a separate graph window if *waveName* is plotted as a contour or image in the top graph. An attempt is made to plot the model values and residuals in the same way as the input data.

By default the auto-trace and auto-residual waves are 50x50 or 25x25x25 or 15x15x15x15. Use /L=*dimSize* for other sizes. Make your own wave and use /D=*waveName* or /R=*waveName* if you want a wave that isn't square. In this case, the wave dimensions must be the same as the dependent data wave.

Confidence bands are not available for multivariate fits.

Wave Subrange Details

Almost any wave you specify to FuncFitMD can be a subrange of a wave. The syntax for wave subranges is the same as for the Display command; see **Subrange Display Syntax** on page II-321 for details. Note that the dependent variable data (*waveName*) must be a multidimensional wave; this requires an extension of the subrange syntax to allow a multidimensional subrange. See **Wave Subrange Details** on page V-274 for a discussion of the use of subranges in curve fitting.

The backwards compatibility rules for **CurveFit** apply to FuncFitMD as well.

A subrange could be used to pick a plane from a 3D wave for fitting using a fit function taking two independent variables:

```
Make/N=(100,100,3) DepData
FuncFitMD fitfunc2D, myCoefs, DepData[][][0] …
```

See Also

The **CurveFit** operation for parameter details.

The best way to create a user-defined fitting function is using the Curve Fitting dialog. See **Using the Curve Fitting Dialog** on page III-181, especially the section **Fitting to a User-Defined Function** on page III-190.

For details on the form of a user-defined function, see **User-Defined Fitting Functions** on page III-250.

FUNCREF

FUNCREF *protoFunc func* [**=** ***funcSpec***]

Within a user function, FUNCREF is a reference that creates a local reference to a function or a variable containing a function reference.

When passing a function as an input parameter to a user function, the syntax is:

FUNCREF *protoFunc func*

In this FUNCREF reference, *protoFunc* is a function that specifies the format of the function that can be passed by the FUNCREF, and *func* is a function reference used as an input parameter.

When you declare a function reference variable within a user function, the syntax is:

FUNCREF *protoFunc func = funcSpec*

Here, the local FUNCREF variable, *func*, is assigned a *funcSpec*, which can be a literal function name, a $ string expression that evaluates at runtime, or another FUNCREF variable.

See Also

Function References on page IV-107 for an example and further usage details.

FuncRefInfo

FuncRefInfo(*funcRef*)

The FuncRefInfo function returns information about a **FUNCREF**.

Parameters

funcRef is a function reference variable declared by a FUNCREF statement in a user-defined function.

Details

FuncRefInfo returns a semicolon-separated keyword/value string containing the following information:

See Also

Function References on page IV-107 and **FUNCREF** on page V-278.

Keyword	Information
NAME	The name of the reference function or "" if the FUNCREF variable has not been assigned to point to a function.
ISPROTO	0 if the FUNCREF variable has been assigned to point to a function. 1 if it has not been assigned and therefore still points to the prototype function.
ISXFUNC	0 if it points to a user-defined function. 1 if the FUNCREF points to an external function.

Function

`Function` `[`**`[/C /D /S /DF /WAVE`**`]` ***`functionName`*****`(`**`[`***`parameters`***`]`**`)`**

The Function keyword introduces a user-defined function in a procedure window.

The optional flags specify the return value type, if any, for the function.

Flags

/C	Returns a complex number.
/D	Returns a double-precision number. Obsolete, accepted for backward compatibility.
/S	Returns a string.
/DF	Returns a data folder reference. See **Data Folder Reference Function Results** on page IV-81.
/WAVE	Returns a wave reference. See **Wave Reference Function Results** on page IV-76.

Details

If you omit all flags, the result is a scalar double-precision number.

The /D flag is not needed because all numeric return values are double-precision.

See Also

Chapter IV-3, **User-Defined Functions** and **Function Syntax** on page IV-31 for further information.

FunctionInfo

`FunctionInfo(`***`functionNameStr`*****` [, `*****`procedureWinTitleStr`*****`])`**

The FunctionInfo function returns a keyword-value pair list of information about the user-defined or external function name in *functionNameStr*.

Parameters

functionNameStr a string expression containing the name or multipart name of a user-defined or external function. *functionNameStr* is usually just the name of a function, or "" to return information about the function that called FunctionInfo.

To return information about a static function, supply both the module name and the function name in MyModule#MyFunction format (see **Regular Modules** on page IV-236), or specify the function name and *procedureWinTitleStr* (see below).

To return information about a function in a different independent module, supply the independent module name in addition to any module name and function name (a double or triple name):

Name	What It Refers To
MyIndependentModule#MyFunction	Refers to a non-static function in an independent module.
MyIndependentModule#MyModule#MyFunction	Refers to a static function in a procedure file with `#pragma moduleName=MyModule` in an independent module.

(See **Independent Modules** on page IV-238 for details on independent modules.)

The optional *procedureWinTitleStr* can be the title of a procedure window (such as "Procedure" or "File Name Utilities.ipf") in which to search for the named user-defined function. The information about the named function in the specified procedure window is returned.

The *procedureWinTitleStr* parameter makes it possible to select one of several static functions with identical names among different procedure windows, even if they do not contain a `#pragma moduleName=myModule` statement.

The *procedureWinTitleStr* parameter can also be a title followed by an independent module name in brackets to return information about the named function in the procedure window of the given title that belongs to named independent module. You can use this syntax in an independent module when inquiring about a function not in that independent module. For example, use "Procedure [ProcGlobal]" to return information about functions in the main procedure window.

procedureWinTitleStr can also be just an independent module name in brackets to return information about the named nonstatic function in any procedure window that belongs to named independent module.

Omit the procedureWinTitleStr parameter or set it to "" when functionNameStr is "".

Details

The returned string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

User-Defined and External Functions

Keyword	Information Following Keyword
`NAME`	The name of the function. Same as contents of *functionNameStr* in most cases. Just the function name if you use the `module#function` format.
`TYPE`	Value is "UserDefined" or "XFunc".
`THREADSAFE`	Either "yes" or "no". See **ThreadSafe Functions** on page IV-106.

`RETURNTYPE`	The return type or types of the function. See **Return Type and Parameter Type Codes** on page V-281.
	For functions that return a single value, the return type is a simple number. For example, for a function that returns a single string:
	`RETURNTYPE:8192;`
	For functions that return multiple values, the return types are listed in the order of their declaration in brackets separated by commas. For example, for a function that returns a number and a string:
	`RETURNTYPE:[4100,12288];`

`N_PARAMS`	Number of parameters for this function.
`PARAM_`*n*`_TYPE`	Number encoding the type of each parameter. There will be N of these keywords, one for each parameter. The part shown as *n* will be a number from 0 to N-1.

See **Examples** for a method for decoding these keywords.

User-Defined Functions Only

Keyword	Information Following Keyword
PROCWIN	Title of procedure window containing the function definition.
MODULE	Module containing function definition (see **Regular Modules** on page IV-236).
INDEPENDENTMODULE	Independent module containing function definition (see **Independent Modules** on page IV-238).
SPECIAL	The value part has one of three values: no: Not a “special” function (not static or override).

User-Defined Functions Only

Keyword	Information Following Keyword	
	static:	Function is a static function. You must use the module#function format to get info about static functions.
	override:	Function is an override function. See **Function Overrides** on page IV-106.
SUBTYPE	The function subtype, for instance **FitFunc**. See **Procedure Subtypes** on page V-13 for others.	
PROCLINE	Line number within the procedure window of the function definition.	
VISIBLE	Either "yes" or "no". Set to "no" in the unlikely event that the function is defined in an invisible file.	
N_OPT_PARAMS	Number of optional parameters. Usually zero.	

External Functions Only

Keyword	Information Following Keyword
XOP	Name of the XOP module containing the function.

Return Type and Parameter Type Codes

Type	Code	Code in Hex
Complex	1	0x1
Single Precision	2	0x2
Variable	4	0x4
Double Precision	4	0x4
Byte	8	0x8
16-bit Integer	16	0x10
32-bit Integer	32	0x20
Unsigned	64	0x40
/WAVE	128	0x80
Data folder reference	256	0x100
Structure	512	0x200
Function reference	1024	0x400
Pass by reference parameter	4096	0x1000
String	8192	0x2000
Wave	16384	0x4000
/Z	32768	0x8000

Igor functions can return a numeric value, a string value, a wave reference, or a data folder reference.

A returned numeric value is always double precision and may be complex. The return type for a normal numeric function is 4, for a complex function (Function/C) is 5 (4 for number +1 for complex).

A string function (Function/S) is 8192 (string). A Function/WAVE has a return type of 16384.

The return and parameter codes may be combined to indicate combinations of attributes. For instance, the code for a variable is 4 and the code for complex is 1. Consequently, the code for a complex variable parameter is 5. The code for a complex variable parameter passed by reference is (4+1+4096) = 4101.

Variables are always double-precision, hence the code of 4.

Waves may have a variety of codes. Numeric waves will combine with one of the number type codes such as 2 or 16. This does not reflect the numeric type of any actual wave, but rather any flag you may have used in the Wave reference. Thus, if the beginning of your function looks like

```
Function myFunc(w)
   Wave w
```

the code for the parameter w will be 16386 (16384 + 2) indicating a single-precision wave. You can use a numeric type flag with the Wave reference:

```
Function myFunc(w)
   Wave/I w
```

In this case, the code will be 16416 (16384 + 32).

Such codes are not very useful, as it is very rare to use a numeric type flag because the numeric type will be resolved correctly at runtime regardless of the flag.

A text wave has no numeric type, so its code is exactly 16384 or 49152 if /Z is also specified. Thus, the numeric type part of the code for a numeric wave serves to distinguish a numeric wave from a text wave. And a Wave/WAVE (a wave that contains references to other waves) has a code of 16512 (16384 + 128), unless /Z is also specified, which adds 32768, resulting in a code of 49280.

When a function uses the multi-valued return syntax, the return value types indicate that they are passed by reference. For example, the type code for a string returned this way is 0x2000 + 0x1000 = 0x3000 (12288 decimal).

Examples

This function formats function information nicely and prints it in the history area in an organized fashion. You can copy it into the Procedure window to try it out. It uses the function `InterpretType()` below to print a human-readable version of the parameter and return types. To try `PrintFuncInfo()`, you will need to copy the code for `InterpretType()` as well.

```
Function PrintFuncInfo(functionName)
   String functionName

   String infostr = FunctionInfo(functionName)
   if (strlen(infostr) == 0)
      print "The function \""+functionName+"\" does not exist."
      return -1
   endif

   print "Name: ", StringByKey("NAME", infostr)

   String typeStr = StringByKey("TYPE", infostr)
   print "Function type: ", typeStr
   Variable IsUserDefined = CmpStr(typeStr, "UserDefined")==0

   // It's not really necessary to use an IF statement here;
   // it simply prevents lines with blank information being
   // printed for an XFUNC.

   if (IsUserDefined)
      print "Module: ", StringByKey("MODULE", infostr)
      print "Procedure window: ", StringByKey("PROCWIN", infostr)
      print "Subtype: ", StringByKey("SUBTYPE", infostr)
      print "Special? ", StringByKey("SPECIAL", infostr)

      // Note use of NumberByKey to get a numeric key value
      print "Line number: ", NumberByKey("PROCLINE", infostr)
   endif

   // See function InterpretType() below for example of
   // interpreting type information.

   Variable returnType = NumberByKey("RETURNTYPE", infostr)

   String returnTypeStr = InterpretType(returnType, 1)
```

```
    printf "Return type: %d (0x%X) %s\r", returnType, returnType, returnTypeStr

    Variable nparams = NumberByKey("N_PARAMS", infostr)
    print "Number of Parameters: ", nparams

    Variable nOptParams = 0
    if (IsUserDefined)
       nOptParams = NumberByKey("N_OPT_PARAMS", infostr)
       print "Optional Parameters: ", nOptParams
    endif

    Variable i
    for (i = 0; i < nparams; i += 1)
       // Note how the PARAM_n_TYPE keyword string is constructed here:
       String paramKeyStr = "PARAM_"+num2istr(i)+"_TYPE"
       Variable ptype = NumberByKey(paramKeyStr, infostr)

       String ptypeStr = InterpretType(ptype,0)
       String format = "Parameter %d; type as number: %g (0x%X); type as string: %s"
       String output
       sprintf output, format, i, ptype, ptype, pTypeStr
       print output
    endfor

    return 0
End
```

Function that creates a human-readable string with information about parameter and return types. Note that various attributes of the type info is tested using the bitwise AND operator (&) to test for individual bits. The constants are expressed as hexadecimal values (prefixed with “0x”) to make them more readable (at least to a programmer). Otherwise, 0x4000 would be 16384; at least, 0x4000 is clearly a single-bit constant.

```
Function/S InterpretType(type, isReturnType)
    Variable type
    Variable isReturnType      // 0: type is parameter type; 1: type is return type.

    String typeStr = ""

    // limit type to unsigned 16-bit values (remove sign extensions caused by 0x8000)
    type = type & 0xFFFF

    // isNumeric is flag to tell whether to print out "complex" and "real";
    // we don't want that information on strings, text waves or wave of wave references.
    Variable isNumeric = 1

    if (type & 0x4000)         // test for WAVE bit set
       typeStr += "Wave"

       if( !isReturnType )
          if (type & 0x80)     // test for WAVE/WAVE bit set
             typeStr += "/WAVE"
             // don't print "real" or "complex" for wave waves
             isNumeric = 0
          endif
          if (type & 0x8000)  // test for WAVE/Z bit set
             typeStr += "/Z"
          endif
       endif
       typeStr += " "

       if( (type == 0x4000) || (type == (0x4000 | 0x8000)) )    // WAVE/T or WAVE/Z/T
          if( !isReturnType )
             // For parameter types, if no numeric bits are set, it is a text wave.
             // A numeric wave has some other bits set causing the value
             // to be different from 0x4000 or 0xC000.
             typeStr += "text "
          endif
          // Function/WAVE doesn't (cannot) specify whether the returned wave
          // is text or numeric.
          // Don't print "real" or "complex" for text or unknown wave types.
          isNumeric = 0
       endif
    elseif (type & 0x2000)     // test for STRING bit set
       typeStr += "String "
```

```
      isNumeric = 0
   elseif (type & 4)          // test for VARIABLE bit
      typeStr += "Variable "
   elseif (type & 0x100)      // test for DFREF bit
      typeStr += "Data folder reference "
      isNumeric = 0
   elseif (type & 0x200)      // test for STRUCTURE bit
      typeStr += "Struct "
      isNumeric = 0
   elseif (type & 0x400)      // test for FUNCREF bit
      typeStr += "FuncRef "
      isNumeric = 0
   endif

   // print "real" or "complex" for numeric objects only
   if (isNumeric)
      if (type & 1)           // test for COMPLEX bit
         typeStr += "cmplx "
      else
         typeStr += "real "
      endif
   endif

   if( !isReturnType && (type & 0x1000) )  // test for PASS BY REFERENCE bit
      typeStr += "reference "
   endif

   return typeStr
End
```

See Also

The **StringByKey** and **NumberByKey** functions.

StringByKey, **NumberByKey**, and **FunctionList** functions.

Regular Modules on page IV-236 and **Independent Modules** on page IV-238.

FunctionList

FunctionList(*matchStr*, *separatorStr*, *optionsStr*)

The FunctionList function returns a string containing a list of built-in or user-defined function names satisfying certain criteria. This is useful for making a string to list functions in a pop-up menu control. Note that if the procedures need to be compiled, then FunctionList will not list user-defined functions.

Parameters

Only functions having names that match *matchStr* string are listed. Use `"*"` to match all names. See **WaveList** for examples.

separatorStr is appended to each function name as the output string is generated. *separatorStr* is usually `";"` for list processing (See **Processing Lists of Waves** on page IV-198 for details on list processing).

Use *optionsStr* to further qualify the list of functions. *optionsStr* is a string containing keyword-value pairs separated by commas. Available options are:

KIND:*nk* Controls the kinds of functions returned.

nk=1: List built-in functions.

nk=2: List normal and override user-defined functions.

nk=4: List external functions (defined by an XOP).

nk=8: List only curve fitting functions; must be summed with 1, 2, 4, or 16. For example, use 10 to list user-defined fitting functions.

nk=16: Include static user-defined functions; requires WIN: option, must be summed with 1, 2, or 8. To list only static functions, subtract the non-static functions using RemoveFromList.

SUBTYPE:*typeName*

Lists functions that have the type *typeName*. That is, you could use ButtonControl as *typeName* to list only functions that are action procedures for buttons.

VALTYPE:*nv* Restricts list to functions whose return type is a certain kind.

nv=1:	Real-valued functions.
nv=2:	Complex-valued functions.
nv=4:	String functions.
nv=8:	WAVE functions
nv=16:	DFREF functions.

Use a sum of these values to include more than one type. The return type is not restricted if this option is omitted.

NPARAMS:*np* Restricts the list to functions having exactly *np* parameters. Omitting this option lists functions having any number of parameters.

NINDVARS:*ni* Restricts the list to fitting functions for exactly *ni* independent variables. NINDVARS is ignored if you have not elected to list curve fitting functions using the KIND option. Functions for any number of independent variables are listed if the NINDVARS option is omitted.

NRETURN:*nRet* Restricts list to user-defined functions that use the **Multiple Return Syntax** on page IV-36 to return *nRet* values.

NRETURN is ignored if KIND:8 is specified.

NRETURN was added in Igor Pro 9.00.

PARAM_n_TYPE:*pType*

Restricts list to functions whose nth input parameter type is a certain kind as described by *pType*.

PARAM_n_TYPE is ignored if KIND:8 is specified.

PARAM_n_TYPE was added in Igor Pro 9.00.

Use PARAM_0_TYPE for the first parameter, PARAM_1_TYPE for the second parameter, and so on. You can use any number of PARAM_n_TYPE keyword=value pairs in a given FunctionList call.

For each parameter, set *pType* to a parameter type code as described under **Return Type and Parameter Type Codes** on page V-286 below.

See also **Return Type and Parameter Type Code Examples** on page V-287 below.

RETURN_n_TYPE:*rType*

Restricts list to user-defined functions whose nth Multiple Return Syntax return type is a certain kind described by *rType*.

RETURN_n_TYPE is ignored if KIND:8 is specified.

RETURN_n_TYPE was added in Igor Pro 9.00.

Use RETURN_0_TYPE for the first return value, RETURN_1_TYPE for the second return value, and so on. You can use any number of RETURN_n_TYPE keyword=value pairs in a given FunctionList call.

For each return value, set *rType* to a return type code as described under **Return Type and Parameter Type Codes** on page V-286 below.

See also **Return Type and Parameter Type Code Examples** on page V-287 below.

WIN:*windowTitle* Lists functions that are defined in the procedure window with the given title. “Procedure” is the title of the built-in procedure window.

Note: Because the *optionsStr* keyword-value pairs are comma separated and procedure window names can have commas in them, the WIN:keyword must be the last one specified.

WIN:windowTitle [*independentModuleName*]

Lists functions that are defined in the named procedure window that belongs to the independent module *independentModuleName*. See **Independent Modules** on page IV-238 for details. Requires `SetIgorOption IndependentModuleDev=1`, otherwise no functions are listed.

Requires *independentModuleName*=ProcGlobal or SetIgorOption independentModuleDev=1, otherwise no functions are listed.

Note: The syntax is literal and strict: the window title must be followed by one space and a left bracket, followed directly by the independent module name and a closing right bracket.

WIN:[*independentModuleName*]

Lists functions that are defined in any procedure file that belongs to the named independent module.

Requires *independentModuleName*=ProcGlobal or SetIgorOption independentModuleDev=1, otherwise no functions are listed.

Note: The syntax is literal and strict: 'WIN:' must be followed by a left bracket, followed directly by the independent module name and a closing right bracket, like this:

```
FunctionList(...,"WIN:[myIndependentModuleName]")
```

Return Type and Parameter Type Codes

These codes are used with the PARAM_N_TYPE and RETURN_N_TYPE keywords:

Type	Code	Code in Hex
Complex	1	0x1
Single Precision	2	0x2
Variable	4	0x4
Double Precision	4	0x4
Byte	8	0x8
16-bit Integer	16	0x10
32-bit Integer	32	0x20
Unsigned	64	0x40
/WAVE	128	0x80
Data folder reference	256	0x100
Structure	512	0x200
Function reference	1024	0x400
Pass by reference parameter	4096	0x1000
String	8192	0x2000
Wave	16384	0x4000
/Z	32768	0x8000

Special Return Type and Parameter Type Codes

These special codes are used with the PARAM_N_TYPE and RETURN_N_TYPE keywords:

Any real-valued numeric wave	-2
Any complex-valued numeric wave	-3

Any text wave	-4
Any WAVE/WAVE	-5
Any WAVE/DF	-6

Return Type and Parameter Type Code Examples

This section shows common values used with the PARAM_N_TYPE and RETURN_N_TYPE keywords:

Any real-valued numeric wave	-2	
Any complex numeric wave	-3	
Any text wave	-4	
Wave/S	16386	0x4000 + 2
Wave or Wave/D	16388	0x4000 + 4
Wave/C or Wave/D/C	16389	0x4000 + 4 + 1
Wave/T	16384	0x4000 + 0

If /Z is used, add 32768 or 0x8000.

For pass-by-reference parameters, add 4096 or 0x1000.

To list user-defined functions in the main procedure window that have either a Wave or Wave/D first parameter, with or without /Z:

```
Print FunctionList("*",";","KIND:2,PARAM_0_TYPE:-2,WIN:Procedure") // WIN must be last
```

Examples

To list user-defined fitting functions for two independent variables:

```
Print FunctionList("*",";","KIND:10,NINDVARS:2")
```

To list button-control functions that start with the letter *b* (note that button-control functions are user-defined):

```
Print FunctionList("b*",";","KIND:2,SUBTYPE:ButtonControl")
```

See Also

Independent Modules on page IV-238, **Multiple Return Syntax** on page IV-36, **Procedure Subtypes** on page IV-204.

FunctionInfo, **FuncRefInfo**, **MacroList**, **OperationList**, **StringFromList**, **WinList**, **DisplayProcedure**

FunctionPath

`FunctionPath(`*`functionNameStr`*`)`

The FunctionPath function returns a path to the file containing the named function. This is useful in certain specialized cases, such as if a function needs access to a lookup table of a large number of values.

The most likely use for this is to find the path to the file containing the currently running function. This is done by passing `""` for *functionNameStr*, as illustrated in the example below.

The returned path uses Macintosh syntax regardless of the current platform. See **Path Separators** on page III-451 for details.

If the procedure file is a normal standalone procedure file, the returned path will be a full path to the file.

If the function resides in the built-in procedure window the returned path will be `":Procedure"`. If the function resides in a packed procedure file, the returned path will be `":<packed procedure window title>"`.

If FunctionPath is called when procedures are in an uncompiled state, it returns ":".

Parameters

If *functionNameStr* is `""`, FunctionPath returns the path to the currently executing function or `""` if no function is executing.

Otherwise FunctionPath returns the path to the named function or "" if no function by that name exists.

Examples

This example loads a lookup table into memory. The lookup table is stored as a wave in an Igor binary wave file.

```
Function LoadMyLookupTable()
   String path

   path = FunctionPath("")    // Path to file containing this function.
   if (CmpStr(path[0],":") == 0)
      // This is the built-in procedure window or a packed procedure
      // file, not a standalone file. Or procedures are not compiled.
      return -1
   endif

   // Create path to the lookup table file.
   path = ParseFilePath(1, path, ":", 1, 0) + "MyTable.ibw"

   DFREF dfSave = GetDataFolderDFR()

   // A previously-created place to store my private data.
   SetDataFolder root:Packages:MyData

   // Load the lookup table.
   LoadWave/O path

   SetDataFolder dfSave

   return 0
End
```

See Also

The **FunctionList** function.

GalleryGlobal

GalleryGlobal#*pictureName*

The GalleryGlobal keyword is used in an independent module to reference a picture in the global picture gallery which you can view by choosing Misc→Pictures.

See Also

See **Independent Modules and Pictures** on page IV-244.

gamma

gamma(*num*)

The gamma function returns the value of the gamma function of *num*. If *num* is complex, it returns a complex result. Note that the return value for *num* close to negative integers is NaN, not ±Inf.

See Also

The **gammln** function.

gammaEuler

gammaEuler

The gammaEuler function returns the Euler-Mascheroni constant 0.5772156649015328606065.

The gammaEuler function was added in Igor Pro 7.00.

gammaInc

gammaInc(*a*, *x* [, *upperTail*])

The gammaInc function returns the value of the incomplete gamma function, defined by the integral

$$\Gamma(a,x) = \int_x^\infty e^{-t} t^{a-1}\, dt.$$

If *upperTail* is zero, the limits of integration are 0 to x. If *upperTail* is absent, it defaults to 1, and the limits of integration are x to infinity, as shown. Note that `gammaInc(a, x) = gamma(a) - gammaInc(a, x, 0)`.

Defined for x > 0, a ≥ 0 (*upperTail* = zero or absent) or a > 0 (*upperTail* = 0).

See Also

The **gamma**, **gammp**, and **gammq** functions.

gammaNoise

gammaNoise(*a* [, *b*])

The gammaNoise function returns a pseudo-random value from the gamma distribution

$$f(x) = \frac{x^{a-1}\exp\left(-\frac{x}{b}\right)}{b^a\Gamma(a)}, \qquad x>0,\ a>0,\ b>0,$$

whose mean is *ab* and variance is ab^2. For backward compatibility you can omit the parameter *b* in which case its value is set to 1. When *a*→1 gammaNoise reduces to **expNoise**.

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

References

Marsaglia, G., and W. W. Tsang, *ACM*, 26, 363-372, 2000.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-390.

Chapter III-12, **Statistics** for a function and operation overview.

gammln

gammln(*num* [, *accuracy*])

The gammln function returns the natural log of the gamma function of *num*, where *num* > 0. If *num* is complex, it returns a complex result. Optionally, *accuracy* can be used to specify the desired fractional accuracy. If *num* is complex, it returns a complex result. In this case, *accuracy* is ignored.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of (f_{actual} - $f_{returned}$)/f_{actual} be less than 10^{-7}.

For backward compatibility, if you don't include *accuracy*, gammln uses older code that achieves an accuracy of about $2x10^{-10}$.

With *accuracy*, newer code is used that is both faster and more accurate. The output has fractional accuracy better than $1x10^{-15}$ except for values near zero, where the absolute accuracy (f_{actual} - $f_{returned}$) is better than $2x10^{-16}$.

The speed of calculation depends only weakly on accuracy. Higher accuracy is significantly slower than lower accuracy only for *num* between 6 and about 10.

See Also

The **gamma** function.

gammp

gammp(*a*, *x* [, *accuracy*])

The gammp function returns the regularized incomplete gamma function P(*a*,*x*), where *a* > 0, *x* ≥ 0. Optionally, *accuracy* can be used to specify the desired fractional accuracy. Same as `gammaInc(a, x, 0)/gamma(a)`.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of (f_{actual} - $f_{returned}$)/f_{actual} be less than 10^{-7}.

For backward compatibility, if you don’t include *accuracy*, gammp uses older code that is slower for an equivalent accuracy, and cannot achieve as high accuracy.

The ability of gammp to return a value having full fractional accuracy is limited by double-precision calculations. This means that it will mostly have fractional accuracy better than about 10^{-15}, but this is not guaranteed, especially for extreme values of *a* and *x*.

See Also

The **gammaInc** and **gammq** functions.

gammq

gammq(*a*, *x* [, *accuracy*])

The gammq function returns the regularized incomplete gamma function 1-P(*a*,*x*), where $a > 0$, $x \geq 0$. Optionally, *accuracy* can be used to specify the desired fractional accuracy. Same as `gammaInc(a, x)/gamma(a)`.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of $(f_{actual} - f_{returned})/f_{actual}$ be less than 10^{-7}.

For backward compatibility, if you don’t include *accuracy*, gammq uses older code that is slower for an equivalent accuracy, and cannot achieve as high accuracy.

The ability of gammq to return a value having full fractional accuracy is limited by double-precision calculations. This means that it will mostly have fractional accuracy better than about 10^{-15}, but this is not guaranteed, especially for extreme values of *a* and *x*.

See Also

The **gammaInc** and **gammp** functions.

Gauss

Gauss(x,*xc*,*wx* [,y,*yc*,*wy* [,z,*zc*,*wz* [,t,*tc*,*wt*]]])

The Gauss function returns a normalized Gaussian for the specified dimension.

$$Gauss(\mathbf{r},\mathbf{c},\mathbf{w}) = \prod_{i=1}^{n} \frac{1}{w_i\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{r_i - c_i}{w_i}\right)^2 \right],$$

where *n* is the number of dimensions.

Parameters

xc, *yc*, *zc*, and *tc* are the centers of the Gaussian in the X, Y, Z, and T directions, respectively.

wx, *wy*, *wz*, and *wt* are the widths of the Gaussian in the X, Y, Z, and T directions, respectively.

Note that w_i here is the standard deviation of the Gaussian. This is different from the width parameter in the gauss curve fitting function, which is sqrt(2) times the standard deviation.

Note also that the Gauss function lacks the cross-correlation parameter that is included in the Gauss2D curve fitting function.

Examples

```
Make/N=100 eee=gauss(x,50,10)
Print area(eee,-inf,inf)
  0.999999
```

```
Make/N=(100,100) ddd=gauss(x,50,10,y,50,15)
Print area(ddd,-inf,inf)
  0.999137
```

See Also

Gauss1D (duplicates the Gauss built-in curve fitting function)

Gauss2D (duplicates the Gauss2D built-in curve fitting function)

Gauss1D

Gauss1D(*w*, *x*)

The Gauss1D function returns the value of a Gaussian peak defined by the coefficients in the wave *w*. The equation is the same as the Gauss curve fit:

$$w[0]+w[1]\exp\left[-\left(\frac{x-w[2]}{w[3]}\right)^2\right].$$

Examples

Do a fit to a Gaussian peak in a portion of a wave, then extend the model trace to the rest of the X range:

```
Make/O/N=100 junkg                      // fake data wave
Setscale/I x -1,1,junkg
Display junkg
junkg = 1+2.5*exp(-((x-.5)/.3)^2)+gnoise(.1)
Duplicate/O junkg, junkgfit
junkgfit = NaN
AppendToGraph junkgfit
CurveFit gauss junkg[50,99] /D=junkgfit
// now extend the model trace
junkgfit = Gauss1D(w_coef, x)
```

See Also

The **CurveFit** operation.

Gauss2D

Gauss2D(*w*, *x*, *y*)

The Gauss2D function returns the value of a two-dimensional Gaussian peak defined by the coefficients in the wave *w*. The equation is the same as the Gauss2D curve fit:

$$w[0]+w[1]\exp\left\{\frac{-1}{2\left(1-w[6]^2\right)}\left[\left(\frac{x-w[2]}{w[3]}\right)^2+\left(\frac{y-w[4]}{w[5]}\right)^2-\left(\frac{2w[6](x-w[2])(y-w[4])}{w[3]w[5]}\right)\right]\right\}.$$

Examples

Do a fit to a Gaussian peak in a portion of a wave, then extend the model trace to the rest of the X range (watch out for the very long wave assignment to junkg2D):

```
Make/O/N=(100,100) junkg2D                      // fake data wave
Setscale/I x -1,1,junkg2D
Setscale/I y -1,1,junkg2D
Display; AppendImage junkg2D
//Caution! Next command wrapped to fit page:
junkg2D = -1 + 2.5*exp((-1/(2*(1-.4^2)))*(((x-.1)/.2)^2+((y+.2)/.35)^2+2*.4*
         ((x-.1)/.2)*((y+.2)/.35)))
junkg2D += gnoise(.01)
Duplicate/O junkg2D, junkg2Dfit
junkg2Dfit = NaN
AppendMatrixContour junkg2Dfit
CurveFit gauss2D junkg2D[20,80][10,70] /D=junkg2Dfit[20,80][10,70]
// now extend the model trace
junkg2Dfit = Gauss2D(w_coef, x, y)
```

See Also

The **CurveFit** operation.

GBLoadWave

GBLoadWave [*flags*] [*fileNameStr*]

The GBLoadWave operation loads data from a binary file into waves.

For more complex applications such as loading structured data into Igor structures see the **FBinRead** operation.

Parameters

If *fileNameStr* is omitted or is "", or if the /I flag is used, GBLoadWave presents an Open File dialog from which you can choose the file to load.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/A
Automatically assigns arbitrary wave names using "wave" as the base name. Skips names already in use.

/A=*baseName*
Same as /A but it automatically assigns wave names of the form *baseName*0, *baseName*1.

/D=*d*
New programming should use the /T flag instead of the /D, /L and /F flags.

d=0: Creates single-precision waves.
d=1: Creates double-precision waves.

/D by itself is equivalent to /D=1.

/F=f
New programming should use the /T flag instead of the /D, /L and /F flags.

f specifies the data format of the file:

f=1: Signed integer (8, 16, 32 bits allowed)
f=2: Creates double-precision waves
f=3: Floating point (default, 32, 64 bits allowed)

/FILT=*fileFilterStr*
Provides control over the file filter menu in the Open File dialog. This flag was added in Igor Pro 7.00.

The construction of the *fileFilterStr* parameter is the same as for the /F=*fileFilterStr* flag of the Open operation. See **Open File Dialog File Filters** on page IV-149 for details.

/I [={*macFilterStr*, *winFilterStr*}]
Specifies interactive mode which displays the Open File dialog.

In Igor7 and later, the *macFilterStr* and *winFilterStr* parameters are ignored. Use the /FILT flag instead.

/J=*j*
Specifies how input floating point data is interpreted.

j=0: IEEE floating point (default)
j=1: VAX floating point

/L=*length*
New programming should use the /T flag instead of the /D, /L and /F flags.

length specifies the data length of the data in the file in bits (default = 32). Allowable data lengths are 8, 16, 32, 64.

/N
Same as /A except that, instead of choosing names that are not in use, it overwrites existing waves.

/N=*baseName*
Same as /N except that it automatically assigns wave names of the form *baseName0*, *baseName1*.

/O=*o*
Controls overwriting of waves in case of a name conflict.

o=0: Use unique wave names.
o=1: Overwrite existing waves.

/O by itself is equivalent to /O=1.

/P=*pathName*
Specifies the folder to look in for *fileNameStr*. *pathName* is the name of an existing symbolic path.

/Q=*q* Controls messages written to the history area of the command window.

q=0:	Write messages.
q=1:	Suppress messages.

/Q by itself is equivalent to /Q=1.

/S=*s* *s* is the number of bytes at the start of the file to skip. It defaults to 0.

/T={*fType*,*wType*} Specifies the data type of the file (*fType*) and the data type of the wave or waves to be created (*wType*). The allowed codes for both *fType* and *wType* are:

2:	Single-precision floating point
4:	Double-precision floating point
8:	8-bit signed integer
16:	16-bit signed integer
32:	32-bit signed integer
128:	64-bit signed integer (Igor7 or later)
72:	8-bit unsigned integer (8+64)
80:	16-bit unsigned integer (16+64)
96:	32-bit unsigned integer (32+64)
192:	64-bit unsigned integer (128+64) (Igor7 or later)

/U=*u* Specifies the number of points of data per array in the file.

The default is 0 which means “auto”. In this case GBLoadWave calculate the number of data pointers per array based on the number of bytes in the file, the number of bytes to be skipped at the start of the file (/S flag), and the number of arrays in the file (/W flag).

/V=*v* Specifies interleaving of data in the file.

v=0:	Data in file is not interleaved (default)
v=1:	Data in file is interleaved

/V by itself is equivalent to /V=1.

/W=*w* Specifies the number of arrays in the file. The default is 1.

If you omit /W but specify the number of points per data array in the file via /U then GBLoadWave calculates the number of waves to be loaded based on the number of bytes in the file, the number of bytes to be skipped at the start of the file (/S flag), and the specified number of points per data array in the file (/U flag). Therefore, if you specify /U and want to load just one wave you must also specify /W=1.

/Y={*offset*, *mult*} Data loaded into waves is scaled using offset and mult:

```
output data = (input data + offset) * multiplier
```

This is useful to convert integer data into scaled, real numbers.

Details

The /N flag instructs Igor to automatically name new waves "wave" (or *baseName* if /N=*baseName* is used) plus a nimber. The nimber starts from zero and increments by one for each wave loaded from the file. If the resulting name conflicts with an existing wave, the existing wave is overwritten.

The /A flag is like /N except that Igor skips names already in use.

The /T flag allows you to specify a data type for both the input (data in the file) and the output (data in the waves). You should use the /T flag instead of the /D, /L and /F flags. These flags are obsolete but are still supported.

GBLoadWave Open File Dialog

If you include the /I flag, or if the /P=*pathName* and *fileNameStr* parameters do not fully specify the file to be loaded, GBLoadWave displays the Open File dialog.

The /FILT=*fileFilterStr* flag provides control over the file filter menu in the Open File dialog. This flag was added in Igor Pro 7.00. The construction of the *fileFilterStr* parameter is the same as for the /F=*fileFilterStr* flag of the Open operation. See **Open File Dialog File Filters** on page IV-149 for details.

In Igor7 and later, the *macFilterStr* and *winFilterStr* parameters of the /I flag are ignored. Use the /FILT flag instead.

Output Variables

GBLoadWave sets the following output variables:

`V_flag`	Number of waves loaded or -1 if an error occurs during the file load.
`S_fileName`	Name of the file being loaded.
`S_path`	File system path to the folder containing the file.
`S_waveNames`	Semicolon-separated list of the names of loaded waves.

S_path uses Macintosh path syntax (e.g., "`hd:FolderA:FolderB:`"), even on Windows. It includes a trailing colon.

When GBLoadWave presents an Open File dialog and the user cancels, V_flag is set to 0 and S_fileName is set to `""`.

Example

```
// Load 128 point single precision version 2 Igor binary wave file
GBLoadWave/S=126/U=128 "fileName"

// Load 8 256 point arrays of 16 bit signed integers into single-precision waves
// after skipping 128 byte header
GBLoadWave/S=128/T={16,2}/W=8/U=256 "fileName"

// Load n 100 point arrays of double-precision floating point numbers
// into double-precision Igor waves with names like temp0, temp1, etc,
// overwriting existing waves. n is determined by the number of bytes
// in the file.
GBLoadWave/O/N=temp/T={4,4}/U=100 "fileName"

// Load a file containing a 1024 byte header followed by a 512 row
// by 384 column array of unsigned bytes into an unsigned byte matrix
// wave and display it as an image
GBLoadWave/S=1024/T={8+64,8+64}/N=temp "fileName"
Rename temp0, image
Redimension/N=(512,384) image
if (<file uses row-major order>)
        MatrixTranspose image
    endif
Display; AppendImage image
```

"Row-major order" relates to how a 2D array is stored in memory. In row-major order, all data for a given row is stored contiguously in memory. In column-major order, all data for a given column is stored contiguously in memory. Igor uses column-major order but row-major is more common.

See Also

Loading General Binary Files on page II-166.

FBinRead operation for more complex applications such as loading structured data into Igor structures.

gcd

`gcd(`*`A`*`, `*`B`*`)`

The gcd function calculates the greatest common divisor of *A* and *B*, which are both assumed to be integers.

Examples

Compute least common multiple (LCM) of two integers:

```
Function LCM(a,b)
    Variable a, b

    return((a*b)/gcd(a,b))
End
```

See Also

PrimeFactors, **RatioFromNumber**

GeometricMean

`GeometricMean(a,b)`

The GeometricMean function returns the arithmetic–geometric mean of two positive real numbers *a* and *b*. The mean is computed by creating two sequences $\{a_i\}$ and $\{b_i\}$ initialized to the input values: a0=a and b0=b with

$$a_{n+1} = \frac{1}{2}\left(a_n + b_n\right),$$

$$b_{n+1} = \sqrt{a_n b_n}.$$

The two sequences converge in a few iterations to a single value which is the arithmetic-geometric mean.

See Also

EllipticK, **EllipticE**

GetAxis

`GetAxis` `[`**`/W=`*`winName`* `/Q`**`]` ***`axisName`***

The GetAxis operation determines the axis range and sets the variables V_min and V_max to the minimum and maximum values of the named axis.

Parameters

axisName is usually `"left"`, `"right"`, `"top"` or `"bottom"`, though it may also be the name of a free axis such as `"VertCrossing"`.

Flags

/Q	Prevents values of V_flag, V_min, and V_max from being printed in the history area. The results are still stored in the variables.
/W=*winName*	Retrieves axis info from the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

GetAxis sets V_min according to the bottom of vertical axes or left of horizontal axes and V_max according to the top of vertical axes or right of horizontal axes. It also sets the variable V_flag to 0 if the specified axis is actually used in the graph, or to 1 if it is not.

Axis ranges and other graph properties are computed when the graph is redrawn. Since automatic screen updates are suppressed while a user-defined function is running, if the graph was recently created or modified, you must call DoUpdate to redraw the graph so you get accurate axis information.

See Also

The **AxisInfo** function.

GetBrowserLine

`GetBrowserLine(`*`fullPathStr`* `[,` *`mode`*`])`

The GetBrowserLine function returns the zero-based line number of the data folder referenced by *fullPathStr*.

Documentation for the GetBrowserLine function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "GetBrowserLine"
```

GetBrowserSelection

GetBrowserSelection(*index* [, *mode*])

The GetBrowserSelection function returns a string containing the full path, quoted if necessary, to a selected Data Browser item.

Documentation for the GetBrowserSelection function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "GetBrowserSelection"
```

GetCamera

GetCamera [***flags***] **[*keywords*]**

The GetCamera operation provides information about a camera window.

Documentation for the GetCamera operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "GetCamera"
```

GetDataFolder

GetDataFolder(*mode* [, ***dfr***]**)**

The GetDataFolder function returns a string containing the name of or full path to the current data folder or, if *dfr* is present, the specified data folder.

GetDataFolderDFR is preferred.

Parameters

If *mode*=0, it returns just the name of the data folder.

If *mode*=1, GetDataFolder returns a string containing the full path to the data folder.

dfr, if present, specifies the data folder of interest.

Details

GetDataFolder can be used to save and restore the current data folder in a procedure. However GetDataFolderDFR is preferred for that purpose.

Examples

```
String savedDataFolder = GetDataFolder(1)          // Save
SetDataFolder root:
Variable/G gGlobalRootVar
SetDataFolder savedDataFolder                      // and restore
```

See Also

Chapter II-8, **Data Folders**.

The **SetDataFolder** operation and **GetDataFolderDFR** function.

GetDataFolderDFR

GetDataFolderDFR()

The GetDataFolderDFR function returns the data folder reference for the current data folder.

Details

GetDataFolderDFR can be used to save and restore the current data folder in a procedure. It is like GetDataFolder but returns a data folder reference rather than a string.

Example

```
DFREF saveDFR = GetDataFolderDFR()                 // Save
SetDataFolder root:
Variable/G gGlobalRootVar
SetDataFolder saveDFR                              // and restore
```

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-78.

The **SetDataFolder** operation.

GetDefaultFont

`GetDefaultFont(`*`winName`*`)`

The GetDefaultFont function returns a string containing the name of the default font for the named window or subwindow.

Parameters

If *winName* is null (that is, `""`) returns the default font for the experiment.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

Only graph windows and the experiment as a whole have default fonts. If *winName* is the name of a window other than a graph (e.g., a layout), or if *winName* is not the name of any window, GetDefaultFont returns the experiment default font.

In user-defined functions, font names are usually evaluated at compile time. To use the output of GetDefaultFont in a user-defined function, you will usually need to build a command as a string expression and execute it with the **Execute** operation.

Examples

```
String fontName = GetDefaultFont("Graph0")
String command= "SetDrawEnv fname=\"" + fontName + "\", save"
Execute command
```

See Also

The **GetDefaultFontSize**, **GetDefaultFontStyle**, **FontSizeHeight**, and **FontSizeStringWidth** functions.

GetDefaultFontSize

`GetDefaultFontSize(`*`graphNameStr`*`, `*`axisNameStr`*`)`

The GetDefaultFontSize function returns the default font size of the graph or of the graph's axis (in points) in the specified window or subwindow.

Details

If *graphNameStr* is `""` the top graph is examined.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

If *axisNameStr* is `""`, the font size of the default font for the graph is returned.

If named axis exists, the default font size for the named axis in the graph is returned.

If named axis does not exist, NaN is returned.

See Also

The **GetDefaultFont**, **GetDefaultFontStyle**, **FontSizeHeight**, and **FontSizeStringWidth** functions.

GetDefaultFontStyle

`GetDefaultFontStyle(`*`graphNameStr`*`, `*`axisNameStr`*`)`

The GetDefaultFontStyle function returns the default font style of the graph or of the graph's axis in the specified window or subwindow.

Details

If *graphNameStr* is `""` the top graph is examined.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

If *axisNameStr* is `""`, the font style of the default font for the graph is returned.

If named axis exists, the default font style for the named axis in the graph is returned.

If named axis does not exist, NaN is returned.

The function result is a bitwise value with each bit identifying one aspect of the font style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

See Also

The **GetDefaultFont**, **GetDefaultFontSize**, **FontSizeHeight**, and **FontSizeStringWidth** functions.

GetDimLabel

`GetDimLabel(`*`waveName`*`, `*`dimNumber`*`, `*`dimIndex`*`)`

The GetDimLabel function returns a string containing the label for the given dimension or dimension element.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers and 3 for chunks.

If *dimIndex* is -1, it returns the label for the entire dimension. If *dimIndex* is ≥ 0, it returns the dimension label for that element of the dimension.

See Also

SetDimLabel, **FindDimLabel**, **CopyDimLabels**

Dimension Labels on page II-93 for further usage details and examples.

GetEnvironmentVariable

`GetEnvironmentVariable(`*`varName`*`)`

The GetEnvironmentVariable function returns a string containing the current value of the specified environment variable for the currently running Igor process. If the variable does not exist, an empty string ("") is returned.

The GetEnvironmentVariable function was added in Igor Pro 7.00.

Parameters

varName	The name of an environment variable which may or may not exist. It must not be an empty string and may not contain an equals sign (=). As a special case, if a single equals sign ("=") is passed for *varName*, a carriage return (\r) separated list of all current key=value environment variable pairs is returned.

Details

The environment of Igor's process is composed of a set of key=value pairs that are known as environment variables.

Any child process created by calling ExecuteScriptText inherits the environment variables of Igor's process.

On Windows, environment variable names are case-insensitive. On other platforms, they are case-sensitive.

GetEnvironmentVariable returns an empty string if *varName* does not exist or if it does exist but its value is empty. If you need to know whether or not the environment variable itself actually exists, you can use the following function:

```
Function EnvironmentVariableExists(varName)
    String varName
    String varList = GetEnvironmentVariable("=")
    // Replace \r with \n because GrepString treats \n only as line separator, not \r.
    varList = ReplaceString("\r", varList, "\n")
    String regExp = "(?m)^" + varName + "="
    return GrepString(varList, regExp)
End
```

Examples

```
String currentUser = GetEnvironmentVariable("USER")
String varList = GetEnvironmentVariable("=")
```

See Also

SetEnvironmentVariable, **UnsetEnvironmentVariable**

GetErrMessage

GetErrMessage(*errorCode* [, *substitutionOption*])

GetErrMessage returns a string containing an explanation of the error associated with *errorCode*. It is most useful for programmers providing custom error handling in user-defined functions.

The **GetRTErrMessage** provides a simpler way to get a description of an error in a user-defined function.

For an overview of error handling, see **Flow Control for Aborts** on page IV-48.

Details

errorCode is an Igor error code. Usually you obtain the error code during execution of a user-defined function via the **GetRTError** function or from the V_Flag variable created by many Igor operations.

If multiple errors occur in a user-defined function, this may result in GetErrMessage returning an incomplete error message. See the MultipleErrors example below. You can achieve more reliable error reporting by calling **GetRTErrMessage** from a **try-catch-endtry** block.

Substitution

For a few error codes, the corresponding error message is designed to be combined with "substituted" information available only immediately after the error occurs. An example is the "parameter out of range" error which produces an error message such as "expected number between x and y ". The optional *substitutionOption* parameter gives you control over substitution.

To get the correct error message, you must call GetErrMessage immediately after calling the function or operation that generated the error and you must pass the appropriate value for *substitutionOption*.

Igor maintains two contexts which store the substitution information: one for user-defined functions and one for all other contexts (command line execution, macros, and the Execute operation).

Set *substitutionOption* to one of these values:

substitutionOption	GetErrMessage Action
0	Substitution values are filled in with "_Not Available_". This is the default when *substitutionOption* is not specified.
1	Substitution values are blank.
2	Substitution is performed based on the assumption that the error was received while executing a macro or a command using Igor's command line. This includes a command executed via the Execute operation even from a user-defined function because such commands are executed as if entered in the command line.
3	Substitution is performed based on the assumption that the error was received while executing a user-defined function.

For most purposes you should pass 3 for *substitutionOption* when the error was generated in a user-defined function other than through the Execute operation and pass 2 otherwise.

Examples

```
// Macro, Execute or command line example
Execute/Q/Z "Duplicate/O nonexistentWave, dup"
Print GetErrMessage(V_Flag,2)
// Prints "expected wave name"
// Function example
Function Test()
   Make/O/N=(2,2) data= 0
   FilterIIR/COEF=data/LO=999/Z data    // Purposely wrong /LO value
   Print GetErrMessage(V_Flag,3)        // Substitution assuming user-defined function
End
// Executing Test() prints: "expected /LO frequency between 0 and 0.5"

// Multiple error example
// Because of the first error, an assignment to a null wave reference,
// the substitution information for the FilterIIR operation is not available.
Function MultipleErrors()
   Make/O/N=(2,2) data= 0
   WAVE ww = $""

   // Generates error because ww is not valid
   ww = 0

   // Generates another error because of purposely wrong /LO value
   FilterIIR/COEF=data/LO=999/Z data

   Print GetErrMessage(V_Flag,3)        // Substitution assuming user-defined function
End
// Executing MultipleErrors() prints:
// "expected  between  and "         // Wave error masks /LO error reporting
```

See Also

Flow Control for Aborts on page IV-48, **GetRTErrMessage**, **GetRTError**

GetFileFolderInfo

GetFileFolderInfo [*flags*][*fileOrFolderNameStr*]

The GetFileFolderInfo operation returns information about a file or folder.

Parameters

fileOrFolderNameStr specifies the file (or folder) for which information is returned. It is optional if /P=*pathName* and /D are specified, in which case information about the directory associated with *pathName* is returned.

If you use a full or partial path for *fileOrFolderNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Folder paths should not end with single Path Separators. See the **MoveFolder Details** section.

If Igor can not determine the location of the file from *fileOrFolderNameStr* and *pathName*, it displays a dialog allowing you to specify the file to be examined. Use /D to select a folder.

Flags

/D	Uses the Select Folder dialog rather than Open File dialog when *pathName* and *fileOrFolderNameStr* do not specify an existing file or folder.
/Omit	
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/Q	No information printed to the history area.
/UTC[=*u*]	If you include /UTC or /UTC=1, GetFileFolderInfo returns creation and modification dates in UTC (coordinated universal time). If you omit /UTC or specify /UTC=0, GetFileFolderInfo returns creation and modification dates in local time. The default, used if you omit /UTC, is local time. The /UTC flag was added in Igor Pro 9.00.

/Z[=z] Prevents procedure execution from aborting if GetFileFolderInfo tries to get information about a file or folder that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

/Z=0: Same as no /Z.

/Z=1: Used for getting information for a file or folder only if it exists. /Z alone has the same effect as /Z=1.

/Z=2: Used for getting information for a file or folder if it exists and displaying a dialog if it does not exist.

Output Variables

GetFileFolderInfo returns information in the following output variables:

`V_flag` 0: File or folder was found.
-1: User cancelled the Open File dialog.
Other: An error occurred, such as the specified file or folder does not exist.

`S_path` Full file system path to the specified file or folder using Macintosh path syntax.

`V_isFile` 1: *fileOrFolderNameStr* is a file.

`V_isFolder` 1: *fileOrFolderNameStr* is a folder.

`V_isInvisible` 1: File is invisible (*Macintosh*) or Hidden (*Windows*).

`V_isReadOnly` Set if the file is locked (*Macintosh*) or is read-only (*Windows*).

On Macintosh, V_isReadOnly is either 0 (unlocked) or 1 (locked). To set this manually, display the Finder Info window for the file and then check or uncheck the “Locked” checkbox.

On Windows, V_isReadOnly is either 0 (unlocked) or 1 (locked). To set this manually, display the Properties window for the file and then check or uncheck the “Read-only” checkbox.

On both Macintosh and Windows, V_isReadOnly tells you only about the property set in the Finder or Windows desktop. It does not tell you if you have write permission for the file or for the folder containing the file. If your goal is to determine if you can write to the file, the only way to do that is to try to write to it and catch any resulting error.

`V_creationDate` Number of seconds since midnight on January 1, 1904 when the file or folder was first created in local time or UTC depending on the /UTC flag. Use **Secs2Date** to format the date as text.

`V_modificationDate` Number of seconds since midnight on January 1, 1904 when the file or folder was last modified in local time or UTC depending on the /UTC flag. Use **Secs2Date** to format the date as text.

`V_isAliasShortcut`

1: File is an alias (*Macintosh*) or a shortcut (*Windows*) and S_aliasPath is also set.

If *fileOrFolderNameStr* refers to a file (not a folder), GetFileFolderInfo returns additional information in the following variables:

`S_aliasPath` If the specified file is an alias or shortcut, S_aliasPath is the full path to the target of the specified file. Otherwise it is "".

S_aliasPath uses Macintosh path syntax. When the source is a folder, it ends with a “:” character.

`V_isStationery` 1: The stationery bit is set (*Macintosh*) or (*Windows*) the file type is one of the stationery file types (.pxt, .uxt, .ift).

`S_fileType` Four-character file type code, such as 'TEXT' or 'IGsU' (packed experiment). On Windows, these codes are fabricated by translating from the equivalent file name extensions, such as .txt and .pxp.

`S_creator` Four-character creator code, such as 'IGR0' (Igor Pro creator code).

On Windows, S_creator is set to 'IGR0' if the file name extensions is one of those registered to Igor Pro, such as .pxp or .bwav (but not .txt). For other registered extensions, S_creator is set to the full file path of the registered application. Otherwise it is set to `""`.

`V_logEOF` Number of bytes in the file data fork. For other forks, use **Open**/F and **FStatus**.

`V_version` Version number of the file. On Macintosh, this is the value in the vers(1) resource. On Windows, a file version such as 3.10.2.1 is returned as 4.021: use S_fileVersion to avoid the problem of the second digit overflowing into the first digit.

“0”: File version can’t be determined, or the file can’t be examined because it is already open.

`S_fileVersion` The file version as a string.

On Macintosh, this is just a string representation of V_Version. On Windows, a file version such as 3.10.2.1 is returned as “3.10.2.1”.

“0”: (*Macintosh*) file version can’t be determined.

“0.0.0.0”: (*Windows*) file version can’t be determined.

Details

You can change some of the file information by using **SetFileFolderInfo**.

On Windows shortcuts have ".lnk" file name extensions that are hidden on the desktop. Prior to Igor Pro 9, *fileOrFolderNameStr* was required to include the ".lnk" extension. For consistency with operations such as NewPath and OpenNotebook, in Igor Pro 9.00 and later, it is optional. When *fileOrFolderNameStr* refers to a shortcut, the S_path output variable includes the ".lnk" extension.

Examples

Print the modification date of a file:

```
GetFileFolderInfo/Z "Macintosh HD:folder:afile.txt"
if( V_Flag == 0 && V_isFile )                  // file exists
   Print Secs2Date(V_modificationDate,0), Secs2Time(V_modificationDate,0)
endif
```

Determine if a folder exists (easier than creating a path with NewPath and then using PathInfo):

```
GetFileFolderInfo/Z "Macintosh HD:folder:subfolder"
if( V_Flag && V_isFolder )
   Print "Folder Exists!"
endif
```

Find the source for a shortcut or alias:

```
GetFileFolderInfo/Z "Macintosh HD:fileThatIsAlias"
if( V_Flag && V_isAliasShortcut )
   Print S_aliasPath
endif
```

See Also

The **SetFileFolderInfo**, **PathInfo**, and **FStatus** operations. The **IndexedFile**, **Secs2Date**, and **ParseFilePath** functions.

GetFormula

`GetFormula(`*`objName`*`)`

The GetFormula function returns a string containing the named object's dependency formula. The named object must be a wave, numeric variable or string variable.

Details

Normally an object will have an empty dependency formula and GetFormula will return an empty string (`""`). If you assign a expression to an object using the := operator or the SetFormula operation, the text on the right side of the := or the parameter to SetFormula is the object's dependency formula and this is what GetFormula will return.

Examples

```
Variable/G dependsOnIt
Make/O wave0 := dependsOnIt*2     //wave0 changes when dependsOnItdoes
Print GetFormula(wave0)
```

Prints the following in the history area:

```
  dependsOnIt*2
```

See Also

See **Dependency Formulas** on page IV-230, and the **SetFormula** operation.

GetGizmo

`GetGizmo [`*`flags`*`] `*`keyword`*` [=`*`value`*`]`

The GetGizmo operation provides information about a Gizmo display window.

Documentation for the GetGizmo operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "GetGizmo"
```

GetIndependentModuleName

`GetIndependentModuleName()`

The GetIndependentModuleName function returns the name of the currently running Independent Module. If no independent module is running, it returns "ProcGlobal".

See Also

Independent Modules on page IV-238.

IndependentModuleList.

GetIndexedObjName

`GetIndexedObjName(`*`sourceFolderStr`*`, `*`objectType`*`, `*`index`*`)`

The GetIndexedObjName function returns a string containing the name of the indexth object of the specified type in the data folder specified by the string expression.

GetIndexedObjNameDFR is preferred.

Parameters

sourceFolderStr can be either `":"` or `""` to specify the current data folder. You can also use a full or partial data folder path. *index* starts from zero. If no such object exists a zero length string (`""`) is returned. *objectType* is one of the following values:

objectType	**What You Get**
1	Waves
2	Numeric variables
3	String variables
4	Data folders

Examples

```
Function PrintAllWaveNames()
   String objName
   Variable index = 0
   do
      objName = GetIndexedObjName(":", 1, index)
      if (strlen(objName) == 0)
         break
      endif
      Print objName
      index += 1
   while(1)
End
```

See Also

The **CountObjects** function, and Chapter II-8, **Data Folders**.

GetIndexedObjNameDFR

GetIndexedObjNameDFR(*dfr*, *objectType*, *index*)

The GetIndexedObjNameDFR function returns a string containing the name of the indexth object of the specified type in the data folder referenced by *dfr*.

GetIndexedObjNameDFR is the same as GetIndexedObjName except the first parameter, *dfr*, is a data folder reference instead of a string containing a path.

Parameters

index starts from zero. If no such object exists a zero length string ("") is returned.

objectType is one of the following values:

objectType	**What You Get**
1	Waves
2	Numeric variables
3	String variables
4	Data folders

Examples

```
Function PrintAllWaveNames()
   String objName
   Variable index = 0
   DFREF dfr = GetDataFolderDFR()    // Reference to current data folder
   do
      objName = GetIndexedObjNameDFR(dfr, 1, index)
      if (strlen(objName) == 0)
         break
      endif
      Print objName
      index += 1
   while(1)
End
```

See Also

Data Folders on page II-107, **Data Folder References** on page IV-78, **Built-in DFREF Functions** on page IV-81, **CountObjectsDFR**

GetKeyState

GetKeyState(*flags*)

The GetKeyState function returns a bitwise numeric value that indicates the state of certain keyboard keys.

To detect keyboard events directed toward a window that you have created, such as a panel window, use the window hook function instead of GetKeyState. See **Window Hook Functions** on page IV-293 for details.

GetKeyState is normally called from a procedure that is invoked directly through a user-defined button or user-defined menu item. The procedure tests the state of one or more modifier keys and adjusts its behavior accordingly.

Another use for GetKeyState is to determine if Escape is pressed. This can be used to detect that the user wants to stop a procedure.

GetKeyState tests the keyboard at the time it is called. It does not tell you if keys were pressed between calls to the function. Consequently, when a procedure uses the escape key to break a loop, the user must press Escape until the running procedure gets around to calling the function.

Parameters

flags is a bitwise parameter interpreted as follows:

Bit 0: If set, GetKeyState reports keys keys even if Igor is not the active application. If cleared, GetKeyState reports keys only if Igor is the active application.

All other bits are reserved and must be zero.

Details

When set, the return value is interpreted bitwise as follows:

Bit 0: Command (*Macintosh*) or Ctrl (*Windows*) pressed.

Bit 1: Option (*Macintosh*) or Alt (*Windows*) pressed.

Bit 2: Shift pressed.

Bit 3: Caps Lock pressed.

Bit 4: Control pressed (*Macintosh only*).

Bit 5: Escape pressed.

Bit 6: Left arrow key pressed. Supported in Igor Pro 7.00 or later.

Bit 7: Right arrow key pressed. Supported in Igor Pro 7.00 or later.

Bit 8: Up arrow key pressed. Supported in Igor Pro 7.00 or later.

Bit 9: Down arrow key pressed. Supported in Igor Pro 7.00 or later.

To test if a particular key is pressed, do a bitwise AND of the return value with the mask value 1, 2, 4, 8, 16, 32, 64, 128, 256 and 512 for bits 0 through 9 respectively. To test if a particular key and only that key is pressed compare the return value with the mask value.

On Macintosh, it is currently possible to define a keyboard shortcut for a user-defined menu item and then, in the procedure invoked by the keyboard shortcut, to use GetKeyState to test for modifier keys. This is not possible on Windows because the keyboard shortcut will not be activated if a modifier key not specified in the keyboard shortcut is pressed. It is also possible that this ability on Macintosh will be compromised by future operating system changes. On both operating systems, however, you can test for a modifier key when the user chooses a user-defined menu item or clicks a user-defined button using the mouse.

Examples

```
Function ShiftKeyExample()
   Variable keys = GetKeyState(0)

   if (keys == 0)
      Print "No modifier keys are pressed."
   endif

   if (keys & 4)
      if (keys == 4)
         Print "The Shift key and only the Shift key is pressed."
      else
         Print "The Shift key is pressed."
      endif
   endif
End
```

```
Function EscapeKeyExample()
   Variable keys

   do
      keys = GetKeyState(0)
      if ((keys & 32) != 0)         // User is pressing escape?
         break
      endif
   while(1)
End
```

See Also

Keyboard Shortcuts on page IV-136. **Setting Bit Parameters** on page IV-12 for details about bit settings.

GetLastUserMenuInfo

GetLastUserMenuInfo

The GetLastUserMenuInfo operation sets variables in the local scope to indicate the value of the last selected user-defined menu item.

Details

GetLastUserMenuInfo creates and sets these special variables:

V_flag The kind of menu that was selected:

V_flag	Menu Kind
0	Normal text menu item, including **Optional Menu Items** (see page IV-130) and **Multiple Menu Items** (see page IV-131).
3	"*FONT*"
6	"*LINESTYLEPOP*"
7	"*PATTERNPOP*"
8	"*MARKERPOP*"
9	"*CHARACTER*"
10	"*COLORPOP*"
13	"*COLORTABLEPOP*"

See **Specialized Menu Item Definitions** on page IV-132 for details about these special user-defined menus.

V_value Which menu item was selected. The value also depends on the kind of menu the item was selected from:

V_flag	V_value meaning
0	Text menu item number (the first menu item is number 1).
3	Font menu item number (use S_Value, instead).
6	Line style number (0 is solid line)
7	Pattern number (1 is the first selection, a SW-NE light diagonal).
8	Marker number (1 is the first selection, the X marker).
9	Character as an integer, = char2num(S_Value). Use S_Value instead.
10	Color menu item (use V_Red, V_Green, V_Blue, and V_Alpha instead).
13	Color table list menu item (use S_Value instead).

S_value The menu item text, depending on the kind of menu it was selected from:

V_flag	S_value meaning
0	Text menu item text.
3	Font name or “default”.
6	Name of the line style menu or submenu.
7	Name of the pattern menu or submenu.
8	Name of the marker menu or submenu.
9	Character as string.
10	Name of the color menu or submenu.
13	Color table name.

In the case of **Specialized Menu Item Definitions** (see page IV-132), S_value will be the title of the menu or submenu, etc.

V_Red, V_Green, V_Blue, V_Alpha

If a user-defined color menu ("*COLORPOP*" menu item) was chosen then these values hold the red, green, and blue values of the selected color. The values range from 0 to 65535 - see **RGBA Values**.

These outputs are set to 0 if the last user-defined menu selection was not a color menu selection.

S_graphName, S_traceName, V_mouseX, V_mouseY

These variables are set only when the user chooses a user-defined menu item from the TracePopup, AllTracesPopup, or GraphPopup contextual menu.

S_graphName and S_traceName are initially "" until a user-defined menu selection is made from one of these contextual menus, and are not reset for each user-defined menu selection.

S_graphName is the full host-child specification for the graph. If the graph is embedded into a host window, S_graphName might be something like “Panel0#G0”. See **Subwindow Syntax** on page III-92.

S_traceName is name of the trace that was selected by the trace contextual menu, or **""** if the AllTracesPopup or GraphPopup menu was chosen. See **Trace Names** on page II-282.

V_mouseX and V_mouseY, added in Igor Pro 8.00, are the mouse location of the click that invoked the contextual menu. The location is in pixels; use **AxisValFromPixel** to determine the X and Y axis values that correspond to the pixel location.

S_tableName, S_firstColumnPath, S_columnName, V_mouseX, V_mouseY

Added in Igor Pro 9.00.

These variables are set only when the user chooses a user-defined menu item from the TablePopup contextual menu.

S_tableName, S_firstColumnPath and S_columnName are initially "" until a user-defined menu selection is made from a TablePopup contextual menus, and are not reset for each user-defined menu selection.

S_tableName is the full host-child specification for the table. If the table is embedded into a host window, S_tableName might be something like "Panel0#T0". See **Subwindow Syntax** on page III-92.

S_firstColumnPath is full path to the wave selected by the table contextual menu, or "" if multiple columns from different waves were chosen. The full path is identical to **GetWavesDataFolder**(wave,2).

S_columnName is name of the selected column as used in the **ModifyTable** command, or "" if multiple columns from different waves were selected.

You can obtain additional information about the selected cells in the table using the **GetSelection** operation.

V_mouseX and V_mouseY are the mouse location of the click that invoked the contextual menu. The location is in pixels relative to the top-left corner of the table.

Examples

A Multiple Menu Items menu definition:

```
Menu "Wave List", dynamic
    "Menu Item 1", <some command>
    "Menu Item 2", <some command>
    WaveList("*",";",""), DoSomethingWithWave()
End
```

The last item can create multiple menu items - one for each wave name returned by WaveList. If the user selects one of these items, the DoSomethingWithWave user function can call GetLastUserMenuInfo to determine which wave was selected:

```
Function DoSomethingWithWave()
    GetLastUserMenuInfo
    WAVE/Z selectedWave = $S_value
    // Use selectedWave for something
End
```

A trivial user-defined color menu definition:

```
Menu "Color"
    "*COLORPOP*", DoSomethingWithColor()
End

Function DoSomethingWithColor()
    GetLastUserMenuInfo
    ... do something with V_Red, V_Green, V_Blue, V_Alpha ...
End
```

See **Specialized Menu Item Definitions** on page IV-132 for another color menu example.

A Trace contextual menu Items menu definition:

```
Menu "TracePopup", dynamic        // menu when a trace is right-clicked
    "-"                           // separator divides this from built-in menu items
    ExportTraceName(), ExportSelectedTrace()
    "Draw XY Here", DrawXYHere()
End

Function/S ExportTraceName()
    GetLastUserMenuInfo           // Sets S_graphName, S_traceName, V_mouseX, V_mouseY
    if (strlen(S_traceName) > 0)
        String item = "Export "+S_traceName
        return item
    endif
    return ""        // No item is added to the menu
End
```

```
Function ExportSelectedTrace()
   GetLastUserMenuInfo
   // Do something with S_graphName, S_traceName
End

Function DrawXYHere()
   GetLastUserMenuInfo

   // Figure out mouse position within graph (requires Igor 8)
   String info = traceinfo(S_graphName, S_traceName, 0)
   String xaxis = stringbykey("XAXIS",info)
   String yaxis = stringbykey("YAXIS",info)
   Variable x_pos = AxisValFromPixel(S_graphName, xaxis, V_mouseX)
   Variable y_pos = AxisValFromPixel(S_graphName, yaxis, V_mouseY)

   // Draw
   GetAxis/W=$S_graphName/Q $yaxis
   Variable miny=V_min, maxy=V_max
   GetAxis/W=$S_graphName/Q $xaxis
   Variable minx=V_min, maxx=V_max
   SetDrawLayer/W=$S_graphName/K userFront
   SetDrawEnv/W=$S_graphName push
   SetDrawEnv/W=$S_graphName xcoord= $xaxis,ycoord=$yaxis,save
   DrawLine/W=$S_graphName minx, y_pos, maxx, y_pos      // horizontal line
   DrawLine/W=$S_graphName x_pos, miny, x_pos, maxy      // vertical line
   SetDrawEnv/W=$S_graphName pop
End
```

See Also

Chapter IV-5, **User-Defined Menus** and especially the sections **Optional Menu Items** on page IV-130, **Multiple Menu Items** on page IV-131, and **Specialized Menu Item Definitions** on page IV-132.

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

GetMarquee

GetMarquee [**/K/W=*winName*/Z**] [***axisName*** [, ***axisName***]]

The GetMarquee operation provides a way for you to use the marquee as an input mechanism in graphs and page layout windows. It puts information about the marquee into variables.

Parameters

If you specify *axisName* (allowed only for graphs) the coordinates are in axis units. If you specify an axis that does not exist, Igor generates an error.

If you specify only one axis then Igor sets only the variables appropriate to that axis. For example, if you execute “`GetMarquee left`” then Igor sets the V_bottom and V_top variables but does *not* set V_left and V_right.

Flags

/K	Kills the marquee. Usually you will want to kill the marquee when you call GetMarquee, so you should use the /K flag. This is modeled after what happens when you create a marquee in a graph and then choose Expand from the Marquee menu. There may be some situations in which you want the marquee to persist. Igor also automatically kills the marquee anytime the window containing the marquee is deactivated, including when a dialog is summoned.
/W=*winName*	Specifies the named window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.
/Z	No runtime error generated if the target window isn’t a graph or layout, but V_flag will be zero. /Z does not prevent other kinds of problems from generating a runtime error.

Details

GetMarquee is intended to be used in procedures invoked through user menu items added to the graph Marquee menu and the layout Marquee menu.

GetMarquee sets the following variables and strings:

`V_flag` 0: There was no marquee when GetMarquee was invoked.
1: There was a marquee when GetMarquee was invoked.

`V_left` Marquee left coordinate.

`V_right` Marquee right coordinate.

`V_top` Marquee top coordinate.

`V_bottom` Marquee bottom coordinate.

`S_marqueeWin` Name of window containing the marquee, or `""` if no marquee. If subwindow, subwindow syntax will be used.

When called from the command line, GetMarquee sets global variables and strings in the current data folder. When called from a procedure, it sets local variables and strings.

In addition, creating, adjusting, or removing a marquee may set additional marquee global variables (see the **Marquee Globals** section, below).

The target window must be a layout or a graph. Use /Z to avoid generating a runtime-error (V_flag will be 0 if the target window was not a layout or graph).

If the target is a layout then Igor sets the variables in units of points relative to the top/left corner of the paper.

If the target is a graph then Igor sets V_left and V_right based on the specified horizontal axis. If no horizontal axis was specified, V_left and V_right are set relative to the left edge of the base window in points.

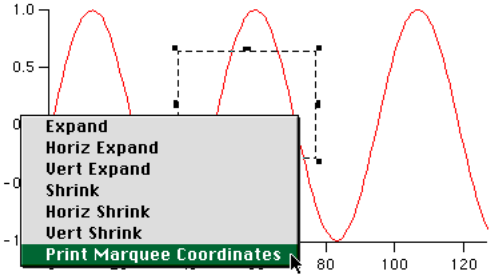
If the target is a graph then Igor sets V_bottom and V_top based on the specified vertical axis. If no vertical axis was specified, V_top and V_bottom are set relative to the top edge of the base window in points.

If there is no marquee when you invoke GetMarquee then Igor sets V_left, V_top, V_right, V_bottom based on the last time the marquee was active.

GetMarquee Example

```
Menu "GraphMarquee"
   "Print Marquee Coordinates", PrintMarqueeCoords()
End

Function PrintMarqueeCoords()
   GetMarquee left, bottom
   if (V_flag == 0)
      Print "There is no marquee"
   else
      printf "marquee left in bottom axis terms: %g\r", V_left
      printf "marquee right in bottom axis terms: %g\r", V_right
      printf "marquee top in left axis terms: %g\r", V_top
      printf "marquee bottom in left axis terms: %g\r", V_bottom
   endif
End
```

You can run this procedure by putting it into the procedure window, making a marquee in a graph, clicking in the marquee and choosing Print Marquee Coordinates:



The procedure calls GetMarquee to set the local marquee variables and then prints their values in the history area:

```
PrintMarqueeCoords()
  marquee left in bottom axis terms: 32.1149
  marquee right in bottom axis terms: 64.7165
  marquee top in left axis terms: 0.724075
  marquee bottom in left axis terms: -0.131061
```

Marquee Globals

You can cause Igor to update global marquee variables whenever the user adjusts the marquee (without the need for you to invoke GetMarquee) by creating a global variable named V_marquee in the root data folder:

```
Variable/G root:V_marquee = 1     //Creates V_marquee and sets bit 0 only
```

When the user adjusts the marquee Igor checks to see if root:V_marquee exists and which bits are set, and updates (and creates if necessary) these globals:

`Variable/G root:V_left`	Marquee left coordinate.
`Variable/G root:V_right`	Marquee right coordinate.
`Variable/G root:V_top`	Marquee top coordinate.
`Variable/G root:V_bottom`	Marquee bottom coordinate.
`String/G root:S_marqueeWin`	Name of window that contains marquee, or `""` if no marquee. Set only if root:V_Marquee has bit 15 (0x8000) set.

Unlike the local variables, for graphs these global variables are never in points. Root:V_left and V_right will be axis coordinates based on the first bottom axis created for the graph (if none, then for the first top axis). The axis creation order is the same as is returned by **AxisList**. Similarly, root:V_top and root:V_bottom will be axis coordinates based on the first left axis or the first right axis.

Igor examines the global root:V_marquee for bitwise flags to decide which globals to update, and when:

root:V_marquee Bit Meaning	Bit Number	Bit Value
Update global variables for graph marquees	0	1
Update global variables for layout marquees	2	4
Update S_marqueeWin when updating global variables	15	0x8000

Marquee Globals Example

By creating the global variable root:V_marquee this way:

```
Variable/G root:V_marquee = 1 + 4 + 0x8000
```

whenever the user creates, adjusts, or removes a marquee in any graph or layout Igor will create and update the global root:V_left, etc. coordinate variables and set the global string root:S_marqueeWin to the name of the window which has the marquee in it. When the marquee is removed, root:S_marqueeWin will be set to `""`.

This mechanism does neat things by making a **ValDisplay** or **SetVariable** control depend on any of the globals. See the Marquee Demo experiment in the Examples:Feature Demos folder for an example.

You can also cause a function to run whenever the user creates, adjusts, or removes a marquee by setting up a dependency formula using **SetFormula** to bind one of the marquee globals to one of the function's input arguments:

```
Variable/G root:dependencyTarget

SetFormula root:dependencyTarget, "MyMarqueeFunction(root:S_marqueeWin)"

Function MyMarqueeFunction(marqueeWindow)
   String marqueeWindow          // this will be root:S_marqueeWin

   if( strlen(marqueeWindow) )
      NVAR V_left= root:V_left, V_right= root:V_right
      NVAR V_top= root:V_top, V_bottom= root:V_bottom
      Printf marqueeWindow + " has a marquee at: "
      Printf "%d, %d, %d, %d\r", V_left, V_right, V_top, V_bottom
   else
```

```
        Print "The marquee has disappeared."
    endif

    return 0                        // return value doesn't really matter
End
```

See Also

The **SetMarquee** and **SetFormula** operations. **Setting Bit Parameters** on page IV-12 for information about bit settings.

GetMouse

`GetMouse [/W=`*`winName`*`]`

The GetMouse operation returns information about the position of the input mouse, and the state of the mouse buttons.

GetMouse is useful in situations such as background tasks where the mouse position and state aren't available as they are in control procedures and window hook functions.

Flags

/W=*winName* Returns the mouse position relative to the named window or subwindow. When identifying a subwindow with winName, see **Subwindow Syntax** on page III-92.

/W=kwTopWin Returns the mouse position relative to the currently frontmost non-floating window.

/W=kwCmdHist Returns the mouse position relative to the command window.

/W=Procedure Returns the mouse position relative to the main Procedure window.

Details

GetMouse returns the mouse position in local coordinates relative to the specified window unless /W is omitted in which case the returned coordinates are global.

On Windows, global coordinates are actually relative to the frame window. See **GetWindow** wsizeDC kwFrameInner.

Information is returned via the following string and numeric variables:

V_left Horizontal mouse position, in pixels.

V_top Vertical mouse position, in pixels.

V_flag Mouse button state. V_flag is a bitwise value with each bit reporting the mouse button states:

Bit 0: 1 if the primary mouse button (usually the left) is down, 0 if it is up.
Bit 1: 1 if the secondary mouse button (usually the right) is down, 0 if it is up.

On Macintosh, the secondary mouse button can be invoked by pressing the control key while clicking the primary (often the only) mouse button, but GetMouse does not report this with bit 1 set. Use **GetKeyState**'s bit 4 to test if the control key is pressed.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

S_name Name of the window or subwindow which the position is relative to, or "" if not a nameable window or if /W was omitted. Most useful with /W=kwTopWin. The result can be kwCmdHist or Procedure, or the name of a target window.

See Also

GetWindow, **GetKeyState**, **SetWindow**, **WMWinHookStruct**, **WMButtonAction**

Background Tasks on page IV-319, **Subwindow Syntax** on page III-92

GetRTError

`GetRTError(`*`flag`*`)`

The GetRTError function returns information about the error state of Igor's user-defined function runtime execution environment.

If *flag* is 0, GetRTError returns an error code if an error has occurred or 0 if no error has occurred.

If *flag* is 1, GetRTError returns an error code if an error has occurred or 0 if no error has occurred and it clears the error state of Igor's runtime execution environment. Use this if you want to detect and handle runtime errors yourself.

If *flag* is 2, GetRTError returns the state of Igor's internal abort flag but does not clear it.

For *flag*=0 and *flag*=1, you can call **GetErrMessage** to obtain the error message associated with the returned error code, if any.

In Igor Pro 7.00 or later, using GetRTError(1) on the same line as a command that causes an error overrides the debugger "debug on error" setting and prevents the debugger from activating for that error.

Example

```
// Detect and handle a runtime error rather than allowing it to cause
// Igor to abort execution or invoke the debugger. The error must be
// recorded, using GetErrMessage, and cleared, using GetRTError,
// on the same line as the error.
Function Demo()
	String msg
	Variable err
	Make/O/N=(-2) wave0; msg=GetErrMessage(GetRTError(0),3); err=GetRTError(1)
	if (err != 0)
		Print "Error in Demo: " + msg
		Print "Continuing execution"
	endif
	// Do more things here
End
```

See also

The **GetErrMessage** and **GetRTErrMessage** functions.

GetRTErrMessage

GetRTErrMessage()

In a user function, GetRTErrMessage returns a string containing the name of the operation that caused the error, a semicolon, and an error message explaining the cause of the error. This is the same information that appears in the alert dialog displayed. If no error has occurred, the string will be of zero length. GetRTErrMessage must be called before the error is cleared by calling GetRTError with a nonzero argument.

For an overview of error handling, see **Flow Control for Aborts** on page IV-48.

See also

Flow Control for Aborts on page IV-48, **GetRTError**, **GetErrMessage**

GetRTLocation

GetRTLocation(*sleepMS*)

GetRTLocation is used for profiling Igor procedures.

You will typically not call GetRTLocation directly but instead will use it through FunctionProfiling.ipf which you can access using this include statement:

```
#include <FunctionProfiling>
```

GetRTLocation is called from an Igor preemptive thread to monitor the main thread. It returns a code that identifies the current location in the procedure files corresponding to the procedure line that is executing in the main thread.

Parameters

sleepMs is the number of milliseconds to sleep the preemptive thread after fetching a value. *sleepMs* must be between 0.001 and 100.

Details

The result from GetRTLocation is passed to **GetRTLocInfo** to determine the location in the procedures. This samples the main thread only and the location becomes meaningless after any procedure editing.

See Also

GetRTLocInfo

GetRTLocInfo

GetRTLocInfo(*code*)

GetRTLocInfo is used for profiling Igor procedures.

You will typically not call GetRTLocInfo directly but instead will use it through FunctionProfiling.ipf which you can access using this include statement:

```
#include <FunctionProfiling>
```

GetRTLocation is called from an Igor preemptive thread to monitor the main thread. It returns a key/value string containing information about the procedure location associated with code or "" if the location could not be found.

Parameters

code is the result from a very recent call to **GetRTLocation**.

Details

The format of the result string is:

```
"PROCNAME:name;LINE:line;FUNCNAME:name;"
```

As of Igor Pro 7.03, if the code is in an independent module other than ProcGlobal then this appears at the beginning of the result string:

```
IMNAME:inName;
```

The line number is padded with zeros to facilitate sorting.

See Also

GetRTLocation

GetRTStackInfo

GetRTStackInfo(*selector*)

The GetRTStackInfo function returns information about "runtime stack" (the chain of macros and functions that are executing).

Details

If *selector* is 0, GetRTStackInfo returns a semicolon-separated list of the macros and procedures that are executing. This list is the same you would see in the debugger's stack list.

The currently executing macro or function is the last item in the list, the macro or function that started execution is the first item in the list.

If *selector* is 1, it returns the name of the currently executing function or macro.

If *selector* is 2, it returns the name of the calling function or macro.

If *selector* is 3, GetRTStackInfo returns a semicolon-separated list of routine names, procedure file names and line numbers. This is intended for advanced debugging by advanced programmers only.

For example, if RoutineA in procedure file ProcA.ipf calls RoutineB in procedure file ProcB.ipf, and RoutineB calls GetRTStackInfo(3), it will return:

```
RoutineA,ProcA.ipf,7;RoutineB,ProcB.ipf,12;
```

The numbers 7 and 12 would be the actual numbers of the lines that were executing in each routine. Line numbers are zero-based.

When called from a function started by **MultiThread** or **ThreadStart** the runtime stack information begins with the function that started threaded execution.

In future versions of Igor, *selector* may request other kinds of information.

Main Thread Example

```
Function Called()
   Print "Called by " + GetRTStackInfo(2) + "()"
   Print "Routines in calling chain: " + GetRTStackInfo(0)
End

Function Calling()
   Called()
End

Macro StartItUp()
   Calling()
End

// Executing StartItUp() prints:
  Called by Calling()
  Routines in calling chain: StartItUp;Calling;Called;
```

MultiThread Example

```
Macro BeginMultiThread(code)
   Variable code=3
   BeginMultiThreadFunc(code)
End

Function BeginMultiThreadFunc(Variable code)
   Make/O/N=4/T/FREE textWave
   MultiThread textWave = tsworker(code)
   Print textWave[0]
End

ThreadSafe Function/S tsworker(Variable code)
   String str= tssubr(code)
   return str
End

ThreadSafe Function/S tssubr(Variable code)
   String str= GetRTStackInfo(code)
   return str
End

// Executing BeginMultiThread(3) prints details for only the two threaded routines:
   tsworker,TSExample,16;tssubr,TSExample,21;
```

See Also

The Stack and Variables Lists, **ThreadSafe Functions and Multitasking**, **GetRTError**

GetScrapText

`GetScrapText()`

The GetScrapText function returns a string containing any plain text on the Clipboard (aka “scrap”). This is the text that would be pasted into a text document if you used Paste in the Edit menu.

See Also

The **PutScrapText** and **LoadPICT** operations.

GetSelection

`GetSelection` ***`winType`*****`,`** ***`winName`*****`,`** ***`bitflags`***

The GetSelection operation returns information about the current selection in the specified window.

Parameters

winType is one of the following keywords:

`graph, panel, table, layout, notebook, procedure`

winName is the name of a window of the specified type.

When identifying a subwindow with winName, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

If *winType* is procedure then *winName* is actually a procedure window title inside a $"" wrapper, such as:

```
GetSelection procedure $"DemoLoader.ipf", 3
```

bitflags is a bitwise parameter that is used in different ways for different window types, as described in **Details**. You should use 0 for undefined bits. **Setting Bit Parameters** on page IV-12 for further details about bit settings.

Details

For all window types, GetSelection sets V_flag:

`V_flag`	0: No selection when GetSelection was invoked.
	1: There was a selection when GetSelection was invoked.

Here is a description of what GetSelection does for each window type:

winType	***bitFlags***	**Action**
`graph`		Does nothing.
`panel`		Does nothing.
`table`	1	Sets V_startRow, V_startCol, V_endRow, and V_endCol based on the selected cells in the table. The top/left cell, not including the Point column, is (0, 0).
	2	Sets S_selection to a semicolon-separated list of column names.
	4	Sets S_dataFolder to a semicolon-separated list of data folders, one for each column.
`layout`	2	Sets S_selection to a semicolon separated list of selected objects in the layout layer (not any drawing layers). S_selection will be `""` if no objects are selected.
`notebook`	1	Sets V_startParagraph, V_startPos, V_endParagraph, and V_endPos based on the selected text in the notebook.
	2	Sets S_selection to the selected text.
	4	Requires Igor Pro 8.05 or later. Sets V_startParagraph and V_endParagraph to the left margin and right margin respectively of the current ruler in points relative to the ruler 0 position. Sets V_startPos, and V_endPos to the left edge and right edge respectively of the selection in points relative to the ruler 0 position.
`procedure`	1	Sets V_startParagraph, V_startPos, V_endParagraph, V_endPos based on the selected text in the procedure window.
	2	Sets S_selection to the selected text.

Examples

In a new experiment, make a table named “Table0” with some columns, and select some combination of rows and columns:

```
Make wave0 = p
Make wave1 = p + 1
Edit wave0, wave1
ModifyTable selection = (3,0,8,1,3,0)
```

Now execute these commands in a procedure or in the command line:

```
GetSelection table, Table0, 3
Print V_flag, V_startRow, V_startCol, V_endRow, V_endCol
Print S_selection
```

This will print the following in the history area:

```
  1  3  0  8  1
  wave0.d;wave1.d;
```

GetUserData

GetUserData(*winName*, *objID*, *userdataName*)

The GetUserData function returns a string containing the user data for a window, subwindow graph trace or control. The return string will be empty if no user data exists.

You set user data on a window or subwindow using the userData keyword of the SetWindow operation. You set it on a graph trace using the userData keyword of the ModifyGraph operation. You set it on a control using the userData keyword of the various control operations.

Parameters

winName may specify a window or subwindow name. Use "" for the top window.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

objID is a string specifying the name of a control or graph trace. Use "" for a window or subwindow.

userdataName is the name of the user data or "" for the default unnamed user data.

See Also

The **ControlInfo**, **GetWindow**, and **SetWindow** operations.

GetWavesDataFolder

GetWavesDataFolder(*waveName*, *kind*)

The GetWavesDataFolder function returns a string containing the name of a data folder containing the wave named by *waveName*. Variations on the theme are selected by *kind*.

The most common use for this is in a procedure, when you want to create a wave or a global variable in the data folder containing a wave passed as a parameter.

GetWavesDataFolderDFR is preferred.

Details

kind	**GetWavesDataFolder Returns**
0	Only the name of the data folder containing *waveName*.
1	Full path of data folder containing *waveName*, without wave name.
2	Full path of data folder containing *waveName*, including possibly quoted wave name.
3	Partial path from current data folder to the data folder containing *waveName*.
4	Partial path including possibly quoted wave name.

Kinds 2 and 4 are especially useful in creating command strings to be passed to **Execute**.

Examples

```
Function DuplicateWaveInDataFolder(w)
   Wave w
   DFREF dfSav = GetDataFolderDFR()
   SetDataFolder GetWavesDataFolder(w,1)
   Duplicate/O w, $(NameOfWave(w) + "_2")
   SetDataFolder dfSav
End
```

See Also

Chapter II-8, **Data Folders**.

GetWavesDataFolderDFR

GetWavesDataFolderDFR(*waveName*)

The GetWavesDataFolderDFR function returns a data folder reference for the data folder containing the specified wave.

GetWavesDataFolderDFR is the same as GetWavesDataFolder except that it returns a data folder reference instead of a string containing a path.

See Also

Data Folders on page II-107, **Data Folder References** on page IV-78, **Built-in DFREF Functions** on page IV-81.

GetWindow

`GetWindow [/Z] *winName*, *keyword*`

The GetWindow operation provides information about the named window or subwindow. Information is returned in variables, strings, and waves.

Parameters

winName can be the name of any target window (graph, table, page layout, notebook, control panel, Gizmo, camera, or XOP target window) or subwindow. It can also be the title of a procedure window or one of these four special keywords:

kwTopWin	Specifies the topmost target window.
kwCmdHist	Specifies the command history area.
kwFrameOuter	Specifies the "frame" or "application" window that Igor Pro has only under Windows. This is the window that contains Igor's menus and status bar.
kwFrameInner	Specifies the inside of the same "frame" window under Windows. This is the window that all other Igor windows are inside.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Only one of the following keywords may follow *winName*. The keyword chosen determines the information stored in the output variables:

active — Sets V_Value to 1 if the window is active or to 0 otherwise. Active usually means the window is the frontmost window.

activeSW — Stores the window "path" of currently active subwindow in S_Value. See **Subwindow Syntax** on page III-92 for details on the window hierarchy.

axsize — Reads graph axis area dimensions (where the traces are, including axis standoff) into V_left, V_right, V_top, and V_bottom in local coordinates. Dimensions are in points.

axsizeDC — Same as axSize but dimensions are in device coordinates (pixels).

backRGB — Sets V_Red, V_Green, V_Blue, and V_Alpha as **RGBA Values** to the background color of the window. The background color is set with **ModifyGraph (colors)** wbRGB, **ModifyLayout** bgRGB, and **Notebook** backRGB. Also returns the background color of procedure windows and the command/history windows. Other windows set these values to 65535 (opaque white).

Added in Igor Pro 7.00.

exterior — Sets V_value to 1 if the window is an exterior panel window or to 0 otherwise. Useful for window hook functions that must work for both regular windows and exterior panel windows, since exterior panels use their own hook function.

bgRGB — Another name for backRGB.

Added in Igor Pro 7.00.

cbRGB — Sets V_Red, V_Green, V_Blue, and V_Alpha as **RGBA Values** to the control panel area background color of the window in graphs and panel windows, as set by **ModifyGraph (colors)** cbRGB and **ModifyPanel** cbRGB. Other windows set these values to 65535 (opaque white).

Added in Igor Pro 7.00.

doScroll

If the window is a graph or panel window with scroll bars added by SetWindow doScroll, the variable V_value is set to 1.

If the window is a graph or panel window without scroll bars added by SetWindow doScroll, the variable V_value is set to 0.

If the window is other than a graph or panel window, an error is generated.

Added in Igor Pro 9.00.

drawLayer

If the specified window is a graph, layout, or panel window then the window's current drawing layer is returned in S_value. S_value is set to "" for other windows. See **Drawing Layers** on page III-68.

Added in Igor Pro 7.00.

expand

Graph windows set V_Value to the expand value set by **ModifyGraph (general)** expand=value, a value normally 0 or 1, where 1.5 means 150%.

Notebook, procedure and command windows set V_Value to the magnification, normally 100. See the **Notebook** magnification=m documentation for details.

Layout windows set V_Value to the **ModifyLayout** mag=m value, usually 0.5 (50%).

In Igor Pro 9.01 and later, panel windows set V_Value to the expand value set by **ModifyPanel** expand=value, a value normally 0 or 1, where 1.5 means 150%. This is the same value returned by **PanelResolution**(winName)/PanelResolution("").

Table windows set V_Value to 0, panels and other unsupported windows to NaN.

Added in Igor Pro 7.00.

file

Works for notebook and procedure windows only.

Returns via S_value a semicolon-separated list containing:

- the file name

- the Mac path to the folder containing the file with a colon at the end

- the name of a symbolic path pointing to that folder, if any

If the window was never saved to a standalone file then "" is returned in S_value. If the specified window is not a notebook or procedure window then "" is returned in S_value.

gbRGB

Sets V_Red, V_Green, V_Blue, and V_Alpha as **RGBA Values** to the plot area background color of the window in graph windows, as set by **ModifyGraph (colors)** gbRGB. Other windows set these values to 65535 (opaque white).

Use wbRGB to get the color of window background (the area outside of the axes).

Added in Igor Pro 7.00.

gsize

Reads graph outer dimensions into V_left, V_right, V_top, and V_bottom in local coordinates. This includes axes but not the tool palette, control bar, or info panel. Dimensions are in points.

gsizeDC

Same as gsize but dimensions are in device coordinates (pixels).

hide

Sets V_Value bit 0 if the window or subwindow is hidden.

Sets bit 1 if the host window is minimized.

Sets bit 2 if the subwindow is hidden only because an ancestor window or subwindow is hidden. Added in Igor Pro 7.00.

On Macintosh, if you execute `MoveWindow 0,0,0,0` to minimize a window to the dock, and then you immediately call `GetWindow hide`, bit 1 may not be correctly set because of the delay caused by the animation of the window sliding into the dock.

hook

Copies name of window hook function to S_value. See **Unnamed Window Hook Functions** on page IV-305.

hook(*hName*)

For the given named hook *hName*, copies name of window hook function to S_value. See **Named Window Hook Functions** on page IV-295.

logicalpapersize

Returns logical paper size of the page setup associated with the named window into V_left, V_right, V_top, and V_bottom. Dimensions are in points.

If the Page Setup dialog uses 100% scaling, these are also the physical dimensions of the page. V_left and V_top are 0 and correspond to the left top corner of the physical page.

On the Macintosh, using a Scale of 50% multiplies all of these dimensions by 2.

logicalprintablesize

Returns logical printable size of the page setup associated with the named window into V_left, V_right, V_top, and V_bottom. Dimensions are in points.

If the Page Setup dialog uses 100% scaling, these are also the physical dimensions of the page minus the margins. V_left and V_top are the number of points from the left top corner of the physical page to the left top corner of the printable area of page.

On the Macintosh, using a page setup scale of 50% multiplies all of these dimensions by 2.

magnification

Sets V_Value exactly the same way that expand does.

Added in Igor Pro 7.00.

maximize

Sets V_Value to 1 if the window is maximized, 0 otherwise. On Macintosh, V_Value is always 0.

needUpdate

Sets V_Value to 1 if window or subwindow is marked as needing an update.

note

Copies window note to S_value.

psize

Reads graph plot area dimensions (where the traces are) into V_left, V_right, V_top, and V_bottom in local coordinates. Dimensions are in points.

psizeDC

Same as psize but dimensions are in device coordinates (pixels).

sizeLimit

Returns the size limits imposed on a window via `SetWindow sizeLimit` in the V_minWidth, V_minHeight, V_maxWidth and V_maxHeight. The values are scaled for screen resolution to the same units as GetWindow wsize, which is points. Very large limits are returned as INF.

The sizeLimit keyword was added in Igor Pro 7.00.

Also returns a sizeLimit status value in V_Value. 0 means no SetWindow sizeLimit command will appear in the window's recreation macro, usually because no `SetWindow sizeLimit` command was applied to the window. 1 means a `SetWindow sizeLimit` command will appear in the window's recreation macro. -1 means it won't appear because of conflicts with graph absolute sizing modes.

title

Gets the title (set by as *titleStr* with NewPanel, Display, etc., or by the Window Control dialog) and puts it into S_value. S_value is set to "" if *winName* specifies a subwindow. See also the wtitle keyword, below.

userdata

Returns the primary (unnamed) user data for a window in S_value. Use **GetUserData** to obtain any named user data.

wavelist

Creates a 3 column text wave called W_WaveList containing a list of waves used in the graph in *winName*. Each wave occupies one row in W_WaveList. This list includes all waves that can be in a graph, including the data waves for contour plots and images.

wbRGB Another name for backRGB.

Added in Igor Pro 7.00.

wsize Reads window dimensions into V_left, V_right, V_top, and V_bottom in points from the top left of the screen. For subwindows, values are local coordinates in the host.

If the window is a graph or panel window with scroll mode turned on using SetWindow doScroll, then V_left and V_top are set to zero, and V_right and V_bottom are set to the width and height in points of the window content area.

wsizeDC Same as wsize but dimensions are in local device coordinates (pixels). The origin is the top left corner of the host window’s active rectangle.

If the window is a graph or panel window with scroll mode turned on using SetWindow doScroll, then V_left and V_top are set to zero, and V_right and V_bottom are set to the width and height in pixels of the window content area.

wsizeForControls Reads window width into V_right and the window height into V_bottom in panel units. "Panel units" are scaled so that a control having the dimensions of V_right and V_bottom exactly fills the window.

The wsizeForControls keyword was added in Igor Pro 9.00. See ***Examples*** below for an example.

wsizeOuter Reads window dimensions into V_left, V_right, V_top, and V_bottom in points from the top left of the screen. Dimensions are for the entire window including any frame and title bar. For subwindows, values are for the host window.

wsizeOuterDC Same as wsizeOuter but dimensions are in local device coordinates (pixels). The origin is the top left corner of the host window’s active rectangle, so V_top will be negative for a window with a title bar. V_left will be negative for windows with a frame; windows on Macintosh OS X have no frame, so V_left will be zero.

wsizeRM Generally the same as wsize, but these are the coordinates that would actually be used by a recreation macro except that the coordinates are in points even if the window is a panel. Also, if the window is minimized or maximized, the coordinates represent the window’s restored location.

On Windows, GetWindow kwFrameOuter wsizeRM returns the pixel coordinates of the MDI frame even when the frame is maximized. wsizeDC returns 2,2,2,2 in this case.

wtitle Gets the actual window title displayed in the window's title bar, regardless of whether it was set by the user (see the title keyword above) or is the default title created by Igor, and puts it into S_value.

S_value is set to "" if winName specifies a subwindow.

If winName is kwFrameOuter or kwFrameInner, on Macintosh S_Value is set to the name of the Igor application. On Windows it is set to the full title of the application as seen on the frame's window, which can be altered using DoWindow/T kwFrame.

Flags

/Z Suppresses error if, for instance, *winName* doesn't name an existing window. V_flag is set to zero if no error occurred or to a non-zero error code.

Details

winName can be the title of a procedure window. If the title contains spaces, use:

```
GetWindow $"Title With Spaces" wsize
```

However, if another window has a name which matches the given procedure window title, that window’s properties are returned instead of the procedure window.

The wsize parameter is appropriate for all windows.

The gsize, psize, gbRGB, and wavelist parameters are appropriate only for graph windows.

The logicalpapersize, logicalprintablesize and expand/magnification parameters are appropriate for all printable windows except for control panels and Gizmo plots.

Local Coordinates

"Local coordinates" are relative to the top left of the graph area, regardless of where that is on the screen or within the graph window. All dimensions are reported in units of points (1/72 inch) regardless of screen resolution. On the Macintosh, this is the same as screen pixels.

Frame Window Coordinates

kwCmdHist, kwFrameInner, and kwFrameOuter may be used with only the wsize keyword.

On Windows computers, kwFrameInner and kwFrameOuter return coordinates into V_left, V_right, V_top, and V_bottom. On the Macintosh, they always return 0, because Igor has no frame on the Macintosh.

kwFrameOuter coordinates are the location of the outer edges of Igor's application window, expressed in screen (pixel) coordinates suitable for use with `MoveWindow/F` to restore, minimize, or maximize the Igor application window.

If Igor is currently minimized, kwFrameOuter returns 0 for all values. If maximized, it returns 2 for all values. Otherwise, the screen (pixel) coordinates of the frame are returned in V_left, V_right, V_top, and V_bottom. This is consistent with `MoveWindow/F`.

kwFrameInner coordinates, however, are the location of the inner edges of the application window, expressed in Igor window coordinates (points) suitable for positioning graphs and other windows with **MoveWindow**.

If Igor is currently minimized, kwFrameInner returns the inner frame coordinates Igor would have if Igor were "restored" with `MoveWindow/F 1,1,1,1`.

V_left and V_top will always both be 0, and V_Bottom and V_Right will be the maximum visible (or potentially visible) window (not screen) coordinates in points.

The Wavelist Keyword

The format of W_WaveList, created with the wavelist keyword, is as follows:

Column 1	Column 2	Column 3
Wave name	partial path to the wave	special ID number

The wave name in column 1 is simply the name of the wave with no path. It may be the same as other waves in the list, if there are waves from different data folders.

The partial path in column 2 includes the wave name and can be used with the $ operator to get access to the wave.

The special ID number in column 3 has the format ##<number>##. A version of the recreation macro for the graph can be generated that uses these ID numbers instead of wave names (see the **WinRecreation** function). This makes it relatively easy to find every occurrence of a particular wave using a function like **strsearch**.

Examples

```
// These commands draw a red foreground rectangle framing
// the printable area of a page layout window.
GetWindow Layout0 logicalpapersize
DoWindow/F Layout0
SetDrawLayer/K userFront
SetDrawEnv linefgc=(65535,0,0), fillpat=0         // Transparent fill
DrawRect V_left+1, V_top+1, V_right-1, V_bottom-1

// These commands demonstrate the difference between title and wtitle.
Make/O data=x
Display/N=MyGraph data
GetWindow MyGraph title;Print S_Value             // Prints nothing (S_Value = "")
GetWindow MyGraph wtitle;Print S_Value            // Prints "MyGraph:data"
DoWindow/T MyGraph, "My Title for My Graph"
GetWindow MyGraph title;Print S_Value             // Prints "My Title for My Graph"
GetWindow MyGraph wtitle;Print S_Value            // Prints "My Title for My Graph"

// Create a panel expanded at 200% with a ListBox that fills the entire panel.
String list = CTabList()            // List of color tables
Make/O/T/N=(ItemsInList(list)) tw = StringFromList(p,list)
```

```
NewPanel/EXP=2/N=DemoPanel
GetWindow DemoPanel wsizeForControls
ListBox list0, win=DemoPanel, pos={0,0}, size={V_right, V_Bottom}, listWave=tw
```

See Also

The **SetWindow**, **GetUserData**, **MoveWindow** and **DoWindow** operations.

The **IgorInfo** function.

GetWindowBrowserSelection

GetWindowBrowserSelection(*mode*)

The GetWindowBrowserSelection function returns a semicolon seperated string that represents the selection in the Window Browser.

GetWindowBrowserSelection was added in Igor Pro 9.00.

Parameters

mode specifies which selection should be returned:

mode=0:	Returns the window names of the selected windows in the main window list.
	Other values for *mode* are reserved for possible future use.

If there is no selection, GetWindowBrowserSelection returns an empty string.

If the Window Browser is not open, GetWindowBrowserSelection returns an empty string and generates an error.

The order in which the selection is returned is undefined. If the order matters to you, you must sort the list yourself.

See Also

The Window Browser on page II-49, **SortList**

GizmoInfo

GizmoInfo(*nameStr*, *key*)

The GizmoInfo function is used to determine if a particular name is valid and unique as a Gizmo item names.

Documentation for the GizmoInfo function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "GizmoInfo"
```

GizmoPlot

GizmoPlot

GizmoPlot is a procedure subtype keyword that identifies a macro as being a Gizmo recreation macro. It is automatically used when Igor creates a window recreation macro for a Gizmo plot. See **Procedure Subtypes** on page IV-204 and **Saving and Recreating Graphs** on page II-350 for details.

GizmoScale

GizmoScale(*dataValue*, *dimNumber* [, *gizmoNameStr*])

The GizmoScale function returns a scaled *dataValue* for the specified dimension. The scaled values are used to position non-data drawing objects in the Gizmo window.

Documentation for the GizmoScale function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "GizmoScale"
```

gnoise

gnoise(*num* [, *RNG*])

The gnoise function returns a random value drawn from a Gaussian distribution such that the standard deviation of an infinite number of such values would be *num*.

gnoise returns a complex result if *num* is complex or if it is used in a complex expression. See **Use of gnoise With Complex Numbers** on page V-324.

The random number generator is initialized using a seed derived from the system clock when you start Igor. This almost guarantees that you will never get the same sequence twice. If you want repeatable “random” numbers, use **SetRandomSeed**.

The Gaussian distribution is achieved using a Box-Muller transformation of uniform random numbers.

The optional parameter RNG selects one of three pseudo-random number generators used to create the uniformly-distributed random numbers providing the input to the Box-Muller transformation.

If you omit the RNG parameter, gnoise uses RNG number 3, named "Xoshiro256**". This random number generator was added in Igor Pro 9.00 and is recommended for all new code. In earlier versions of Igor, the default was 1 (Linear Congruential Generator).

The available random number generators are:

RNG	**Description**
1	Linear Congruential Generator by L’Ecuyer with added Bayes-Durham shuffle. The algorithm is described in *Numerical Recipes* as the function `ran2()`. This option has nearly 23^2 distinct values and the sequence of random numbers has a period in excess of 10^{18}. This RNG is provided for backward compatibility only. New code should use RNG=3 (Xoshiro256**).
2	Mersenne Twister by Matsumoto and Nishimura. It is claimed to have better distribution properties and period of 2^{19937}-1. See Matsumoto, M., and T. Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Trans. on Modeling and Computer Simulation*, *8*, 3-30, 1998. This RNG is provided for backward compatibility only. New code should use RNG=3 (Xoshiro256**).
3	Xoshiro256** by David Blackman and Sebastiano Vigna. It has a period of 2^{256}-1, is 4 dimensionally equidistributed and passes all statistical tests at the time of writing. For technical details, see "Scrambled linear pseudorandom number generators", 2019 (http://vigna.di.unimi.it/ftp/papers/ScrambledLinear.pdf).

Use of gnoise With Complex Numbers

gnoise returns a complex result if *num* is complex or if it is used in a complex expression.

```
Function DemoGNoiseComplex()
   // gnoise(rv) in a complex expression returns a complex result
   // and is equivalent to cmplx(gnoise(rv),gnoise(rv))
   Variable rv = 1                  // A real variable
   SetRandomSeed 1
   Variable/C cv1 = gnoise(rv)      // Real parameter, complex result
   SetRandomSeed 1
   Variable/C cv2 = cmplx(gnoise(rv),gnoise(rv))     // Equivalent
   Print cv1, cv2

   // gnoise(cv) is equivalent to cmplx(gnoise(real(cv)),gnoise(imag(cv)))
   Variable/C cv = cmplx(1,1)       // A complex variable
   SetRandomSeed 1
   Variable/C cv3 = gnoise(cv)      // Complex parameter, complex result
   SetRandomSeed 1
   Variable/C cv4 = cmplx(gnoise(real(cv)),gnoise(imag(cv)))   // Equivalent
   Print cv3, cv4
End
```

See Also

SetRandomSeed, **enoise**, **Noise Functions**, **Statistics**

Graph

`Graph`

Graph is a procedure subtype keyword that identifies a macro as being a graph recreation macro. It is automatically used when Igor creates a window recreation macro for a graph. See **Procedure Subtypes** on page IV-204 and **Saving and Recreating Graphs** on page II-350 for details.

GraphMarquee

`GraphMarquee`

GraphMarquee is a procedure subtype keyword that puts the name of the procedure in the graph Marquee menu. See **Marquee Menu as Input Device** on page IV-163 for details.

GraphNormal

`GraphNormal` `[`***`/W=winName`***`]`

The GraphNormal operation returns the target or named graph to the normal mode, exiting any drawing mode that it may be in.

You would usually enter normal mode by choosing ShowTools from the Graph menu and clicking the graph tool (the top icon in the tool panel).

Flags

/W=*winName*	Reverts the named graph window. This must be the first flag specified when used in a Proc or Macro or on the command line.

See Also

The **GraphWaveDraw** and **GraphWaveEdit** operations.

GraphStyle

`GraphStyle`

GraphStyle is a procedure subtype keyword that puts the name of the procedure in the Style pop-up menu of the New Graph dialog and in the Graph Macros menu. See **Graph Style Macros** on page II-350 for details.

GraphWaveDraw

`GraphWaveDraw` `[`***`flags`***`] [`***`yWave`***`,` ***`xWave`***`]`

The GraphWaveDraw operation initiates drawing a curve composed of *yWave* vs *xWave* in the target or named graph. The user draws the curve using the mouse, and the values are stored in a pair of waves as XY data.

The user can manually initiate drawing by choosing ShowTools from the Graph menu and clicking in the appropriate tool.

Parameters

yWave and *xWave* can be simple names of waves in the current data folder or partial or full data folder paths to waves. If the waves already exist, GraphWaveDraw overwrites them. yWave and xWave can also be wave references pointing to existing waves in which case GraphWaveDraw overwrites them. Prior to Igor Pro 9.00, only simple names were accepted.

If *yWave* and *xWave* already exist, an error is generated unless you include the /O flag.

If you omit *yWave* and *xWave* then waves named W_YPoly*n* and W_XPoly*n* are created in the current data folder. *n* is an integer used to make the output wave names unique in their data folder, so Igor might create waves named W_XPoly0 and W_YPoly0, for example.

If there is no yWave vs xWave trace in the graph, GraphWaveDraw appends it.

Flags

/F[=*f*]	Initiates freehand drawing. In normal drawing, you click where you want a data point. In freehand drawing, you click once and then draw with the mouse button held down. If present, *f* specifies the smoothing factor. Max value is 8 (which is really slow), min value is 0 (default). The drawing tools use a value of 3 which is the recommended value.

/L/R/B/T Specifies which axes to use (Left, Right, Bottom, Top). Bottom and Left axes are used by default. Can specify free axes using /L=*axis name* type notation. See **AppendToGraph** for details. If necessary, the specified axes will be created. If an axis is created its range is set to -1 to 1.

/M Specifies that the curve being edited must be monotonic in the X dimension. The user is not allowed drag points so that they cross horizontally.

/O Overwrites *yWave* and *xWave* if they already exist.

/W=*winName* Draws in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

Once drawing starts no other user actions are allowed.

In normal mode, drawing stops when you double-click or when you click the first point (in which case the last point is set equal to the first point). When drawing finishes, the edit mode is entered.

In freehand mode, drawing stops when the mouse is released or when 10000 points have been drawn.

If you include /O and the waves are already on the graph then the first trace instance on the graph displaying them is used even if the trace uses a different pair of axes than specified by /L, /R, /B, and /T.

Output Variables

S_xWave Path to the X wave relative to the current data folder. S_xWave is create in Igor Pro 9.00 and later.

S_yWave Path to the X wave relative to the current data folder. S_yWave is create in Igor Pro 9.00 and later.

See Also

The **GraphNormal**, **GraphWaveEdit** and **DrawAction** operations.

GraphWaveEdit

`GraphWaveEdit [`*`flags`*`]` ***`traceName`***

The GraphWaveEdit operation initiates editing a wave trace in a graph. The wave trace must already be in the graph.

Normally, you would initiate editing by choosing ShowTools from the Graph menu and clicking in the appropriate tool rather than using GraphWaveEdit.

Parameters

traceName is a wave name, optionally followed by the # character and an instance number: "`myWave#1`" is the *second* instance of `myWave` appended to the graph ("`myWave`" is the first).

If *traceName* is omitted then you get to pick the wave trace to edit by clicking it.

Flags

/M Specifies that the edited trace must be monotonic in the X dimension. You cannot drag points so that they cross horizontally.

/ND Suppresses deletion of a data point when the user presses Option (*Macintosh*) or Alt (*Windows*) and clicks on the point.

/NI Suppresses insertion of a new data point when the user clicks between points.

/T=*t*	Sets the trace mode. *t*=0: Lines and small square markers (default). *t*=1: User settings unchanged.
/W=*winName*	Edits traces in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The GraphWaveEdit operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

See Also

The **GraphNormal**, **GraphWaveDraw** and **DrawAction** operations.

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

Grep

```
Grep [flags][srcFileStr ][srcTextWaveName][ as [destFileOrFolderStr]
    [destTextWaveName]]
```

The Grep operation copies lines matching a search expression from a file on disk, the Clipboard, or rows from a text wave to a new or existing file, an existing text wave, the History area, the Clipboard, or to S_value as a string list.

Source Parameters

The optional *srcFileStr* can be

- The full path to the file to copy lines from (in which case /P is not needed).
- The partial path relative to the folder associated with *pathName.*
- The name of a file in the folder associated with *pathName.*
- “Clipboard” to read lines of text from the Clipboard (in which case /P is ignored).

If Igor can not determine the location of the source file from *srcFileStr* and *pathName*, it displays a dialog allowing you to specify the source file.

The optional *srcTextWaveName* is the name or path to a text wave.

Only one of *srcFileStr* or *srcTextWaveName* may be specified. If neither is specified then an Open File dialog is presented allowing you to specify a source file.

Destination Parameters

The optional *destFileOrFolderStr* can be

- The name of (or path to) an existing folder when /D is specified.
- The name of (or path to) a possibly existing file.
- “Clipboard”, in which case the matching lines are copied to the Clipboard (and /P and /D are ignored). The text can be retrieved with the **GetScrapText** function.

If *destFileOrFolderStr* is a partial path, it is relative to the folder associated with *pathName*.

If /D is specified, the source file is created inside the folder using the source file name.

If Igor can not determine the location of the destination file from *pathName*, *srcFileStr*, and *destFileOrFolderStr*, it displays a Save File dialog allowing you to specify the destination file (and folder).

The optional *destTextWaveName* is the name or path to an existing text wave. It may be the same wave as *srcTextWaveName*.

Only one of *destFileOrFolderStr* or *destTextWaveName* may be specified.

If no destination file or text wave is specified then matching lines are printed in the history area, unless the /Q flag is specified, in which case the matching lines aren't printed or copied anywhere (though the output variables are still set).

Use /LIST to set S_value to a string list containing the matching lines or rows.

Use /INDX to create a wave W_index containing the zero-based row or line number where matches were found.

Parameter Details

If you use a full or partial path for either *srcFileStr* or *destFileOrFolderStr*, see **Path Separators** on page III-451 for details on forming the path.

Folder paths should not end with single path separators. See **MoveFolder**'s Details section.

Flags

/A Appends matching lines to the destination file, creating it if necessary, appends text to the Clipboard if the *destFileOrFolderStr* is "Clipboard", or appends rows to the destination text wave. Has no effect on output to the History area.

/D Interprets *destFileOrFolderStr* as the name of (or path to) an existing folder (or directory). Without /D, *destFileOrFolderStr* is the name of (or path to) a file. It is ignored if the destination is a text wave, the Clipboard, or the History area.

If *destFileOrFolderStr* is not a full path to a folder, it is relative to the folder associated with *pathName*.

/DCOL={*colNum*} Useful only when the destination is *destTextWaveName*.

Copies matching lines of text from the source file, Clipboard, or *srcTextWaveName* to column *colNum* of *destTextWaveName*, with any terminator characters removed.

The default when the source is a file or the Clipboard is `/DCOL={0}`, which copies matching lines into the first column of *destTextWaveName*.

The default when the source is *srcTextWaveName* is to copy each column of a matched row to the corresponding column in *destTextWaveName*.

/DCOL must be used with the /A flag, otherwise the destination wave will have only 1 column.

/DCOL={[*colNum*] [, *delimStr*], …}

Useful only when the source is *srcTextWaveName* and the destination is a file, the Clipboard, History area, or S_value.

Copies multiple columns in any order from matching rows of *srcTextWaveName* to the destination file, Clipboard, History area, or S_value.

Construct the line by appending the contents of the numbered column and the *delimStr* parameters in the order specified. The output line is terminated as described in **Line Termination**.

/E=*regExprStr* Specifies the Perl-compatible regular expression string. A line must match the regular expression to be copied to the output file. See **Regular Expressions**.

Multiple /E flags may be specified, in which case a line is copied only if it matches every regular expression.

/E={*regExprStr*, *reverse*}

Specifies the Perl-compatible regular expression string, *regExprStr*, for which the sense of the match can be changed by *reverse*.

reverse=1: Matching expressions are taken to **not** match, and vice versa. For example, use `/E={"CheckBox",1}` to list all lines that do not contain "CheckBox".

reverse=0: Same as /E=*regExptrStr*.

/ENCG=*textEncoding*

Specifies the text encoding of named text file. This flag applies if the source is a file and is ignored if the source is the clipboard or a text wave.

See **Text Encoding Names and Codes** on page III-490 for a list of accepted values for *textEncoding*.

If you omit /ENCG, Grep uses the default text encoding as specified by the Misc→Text Encoding→Default Text Encoding menu.

Passing 0 for textEncoding acts as if /ENCG were omitted. Passing 255 (binary) for textEncoding is treated as an error because the source text must be internally converted to UTF-8 and there is no valid conversion from binary to UTF-8.

/GCOL=*grepCol* Greps the specified column of *srcTextWaveName*, which is a two-dimensional text wave. The default search is on the first column (`grepCol=0`). Use `grepCol=-1` to match against any column of *srcTextWaveName*.

Does not apply if the source is a file or Clipboard.

/I Requires interactive searching even if *srcFileStr* and *destFileOrFolderStr* are specified and if the source file exists. Same as /I=3.

/I=*i* Specifies the degree of file search interactivity with the user.

i=0: Default; interactive only if *srcFileStr* is not specified or if the source file is missing. Same as if /I were not specified.

Note: This is different behavior than other commands such as **CopyFile**.

i=1: Interactive even if *srcFileStr* is specified and the source file exists.

i=2: Interactive even if *destFileOrFolderStr* is specified.

i=3: Interactive even if *srcFileStr* is specified, the source file exists, and *destFileOrFolderStr* is specified.

/INDX Creates in the current data folder an output wave W_Index containing the line numbers (or row numbers) where matching lines were found. If this is the only output you need, also use the /Q flag.

/LIST[=*listSepStr*] Creates an output string variable S_value containing a semicolon-separated list of the matching lines. If *listSepStr* is specified, then it is used to separate the list items. See **StringFromList** for details on string lists. If this is the only output you need, also use the /Q flag.

/M=*messageStr* Specifies the prompt message in any Open File dialog. If /S is not specified, then *messageStr* will be used for both the Open File and Save File dialogs.

/O Overwrites any existing destination file.

/P=*pathName* Specifies the folder containing the source file or the folder into which the file is copied. *pathName* is the name of an existing symbolic path.

Both *srcFileStr* and *destFileOrFolderStr* must be either simple file or folder names, or paths relative to the folder specified by *pathName*.

/Q Prevents printing results to an output file, text wave, History, or Clipboard. Use /Q to check for a match to *regExprStr* by testing the value of V_flag, V_value, S_value (/LIST), or W_Index (/INDX) without generating any other matching line output.

Note: When using /Q neither *destFileOrFolderStr* nor *destTextWaveName* may be specified.

/S=*saveMessageStr* Specifies the prompt message in any Save File dialog.

/T=*termCharStr*	Specifies the terminator character. `/T=(num2char(13))` Carriage return (CR, ASCII code 13). `/T=(num2char(10))` Linefeed (LF, ASCII code 10). `/T=";"` Semicolon. `/T=""` Null (ASCII code 0). See **Line Termination** for the default behavior of the terminator character.
/Z[=*z*]	Prevents aborting of procedure execution when attempting to open a nonexistent file for searching. Use /Z if you want to handle this case in your procedures rather than having execution abort. *z*=0: Same as no /Z at all. *z*=1: Open file only if it exists. `/Z` alone is the same as `/Z=1`. *z*=2: Open file if it exists and display a dialog if it does not exist.

Line Termination

Line termination applies mostly to source and destination files. (The Clipboard and history area delimit lines with CR, and text waves have no line terminators.)

If /T is omitted, Grep will break file lines on any of the following: CR, LF, CRLF, LFCR. (Most Macintosh files use CR. Most Windows files use CRLF. Most UNIX files use LF. LFCR is an invalid terminator but some buggy programs generate files that use it.)

Grep reads whichever of these terminator(s) appear in the source file and use them to write lines to any output file.

The terminator(s) are removed before the line is matched against the regular expression.

For lines that match *regExprStr*, terminator(s) in the input file are transferred to the output file unless the output is the Clipboard or history area, in which case the output terminator is always only CR (like **LoadWave**). This means you can transparently handle files that use CR, LF, CRLF, or LFCR as the terminator, and omitting /T will be suitable for most cases.

If you use the /T flag, then Grep will terminate line-from-file reads on the specified character only and will output the specified character into any output file.

"Lines" in One-dimensional Text Waves

Grep considers each row of *srcTextWaveName* or *destTextWaveName* to be a "line" of input or output.

When the destination is a file, the Clipboard, or the History area, Grep copies all of the text in a matching row of *srcTextWaveName* to the file and terminates the line. See **Line Termination** for the rules on line terminators.

When the destination is a *destTextWaveName*, Grep simply copies all the text in a matching row to a row in *destTextWaveName*, without adding or omitting any terminators.

"Lines" and Columns in Two-Dimensional Text Waves

Grep by default *matches* against only the first column (column 0) of each row of *srcTextWaveName*. You can use the `/GCOL=`*`grepCol`* flag to specify a different column to match against. Use `/GCOL=-1` to match against any column of *srcTextWaveName*.

When the source is a text wave and the destination is a file, the Clipboard, or the History area, Grep by default *copies* only the first column (column 0) to the destination.

Use the `/DCOL={`*`colNum1`*`, `*`delimStr1`*`, `*`colNum2`*`, `*`delimStr2`*`,...`*`colNumN`*`}` to print multiple columns (in any order) with delimiters after each column (the last column number need not be followed by a delimiter string). The output line is terminated with CR or *termcharStr* as described in **Line Termination**.

When both the source and destination are text waves and append (/A) is *not* specified, the destination text wave is redimensioned to have the same number of columns as the source text wave, and all columns of matching rows of *srcTextWaveName* are copied to *destTextWaveName*.

When both the source and destination are text waves and append /A is specified, then the number of columns in *destTextWaveName* is left unchanged, and each column of *srcTextWaveName* is copied to the corresponding column of *destTextWaveName*.

If the destination is a text wave and the source is a file or the Clipboard, each line (without the terminator) is copied to the first column of the destination text wave, or use `/DCOL={destColNum}` to put the text into a different column.

Output Variables

The Grep operation returns information in the following variables. When running in a user-defined function these are created as local variables. Otherwise they are created as global variables in the current data folder.

`V_flag`	0: Output successfully generated. -1: User cancelled either the Open File or Save File dialogs. Other: An error occurred, such as the specified file does not exist.
`V_value`	The number of input lines that matched the regular expression.
`V_startParagraph`	Zero-based line number into the file or Clipboard (or the row number of a source text wave) where the first regular expression was matched. Also see the **/INDX** flag.
`S_fileName`	Full path to the source file, the source text wave, or "Clipboard". If an error occurred or if the user cancelled, it is an empty string.
`S_path`	Full path to the destination file or destination text wave. "Clipboard": If *destFileOrFolderStr* was the Clipboard. "History": If the output was printed to the history area of the window. `""`: If an error occurred, if the user cancelled, or if /Q was specified.
`S_value`	Contains matching lines as a string list only if /LIST is specified.

Regular Expressions

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the "subject".

In the case of Grep, the "subject" is each line of the source file or Clipboard, or each row in the source text wave.

The regular expression syntax supported by Grep, **GrepList**, and **GrepString** is based on the "Perl-Compatible Regular Expression" (PCRE) library.

The syntax is similar to regular expressions supported by various UNIX and POSIX `egrep(1)` commands. See **Regular Expressions** on page IV-176 for more details.

As a trivial example, the pattern "Fred" as specified here:

```
Grep/P=myPath/E="Fred" "afile.txt" as "FredFile.txt"
```

matches lines that contain the string "Fred" anywhere on the line.

Character matching is *case-sensitive* by default, similar to strsearch. Prepend the Perl 5 modifier `(?i)` to match upper and lower-case versions of "Fred":

```
// Copy lines that contain "Fred", "fred", "FRED", "fREd", etc
Grep/P=myPath/E="(?i)fred" "afile.txt" as "AnyFredFile.txt"
```

To copy lines that do *not* match the regular expression, set the /E flag's reverse parameter:

```
// Copy lines that do NOT contain "Fred", "fred", "fREd", etc.
Grep/P=myPath/E={"(?i)fred",1} "afile.txt" as "NotFredFile.txt"
```

Note: Igor doesn't use the opening and closing regular expression delimiters that UNIX grep or Perl use: they would have used `"/Fred/"` and `"/(?i)fred/"`.

Regular expressions in Igor support the expected metacharacters and character classes that make the whole grep paradigm so useful. For example:

```
// Copy lines that START with a space or tab character
Grep/P=myPath/E="^[ \\t]" "afile.txt" as "LeadingTabsFile.txt"
```

For a complete description of regular expressions, see **Regular Expressions** on page IV-176, especially for a description of the many uses of the regular expression backslash character (see **Backslash in Regular Expressions** on page IV-179).

Note: Because Igor Pro also has special uses for backslash (see **Escape Sequences in Strings** on page IV-14), you must double the number of backslashes you would normally use for a Perl or grep pattern. Each pair of backslashes identifies a single backslash for the Grep command.

For example, to copy lines that contain "\z", the Perl pattern would be `\\z`, but the equivalent Grep expression would be `/E="\\\\z"`.

See **Backslash in Regular Expressions** on page IV-179 for a more complete description of backslash behavior in Igor Pro.

Examples

```
// Copy lines in afile.txt containing "Fred" (case sensitive)
// to an output file named "AnyFredFile.txt" in the same directory.
Grep/P=myPath/E="Fred" "afile.txt" as "AnyFredFile.txt"

// Copy lines in afile.txt containing "Fred" and "Wilma" (case-insensitive)
// to a text wave (which must exist andis overwritten):
Make/O/N=0/T outputTextWave
Grep/P=myPath/E="(?i)fred"/E="(?i)wilma" "afile.txt" as outputTextWave

// Print lines in afile.txt containing "Fred" and "Wilma" (case-insensitive)
// to the history area
Make/O/N=0/T outputTextWave
Grep/P=myPath/E="(?i)fred"/E="(?i)wilma" "afile.txt"

// Test whether afile.txt contains the word "boondoggle", and if so,
// on which line the first occurence was found, WITHOUT creating any output.
//
// Note: the \\b sequences limit matches to a word boundary before and after
// "boondoggle", so "boondoggles" and "aboondoggle" won't match.
//
Grep/P=myPath/Q/E="(?i)\\bBoondoggle\\b" "afile.txt"
if( V_value )                    // at least one instance was found
   Print "First instance of \"boondoggle\" was found on line", V_startParagraph
endif

// Create in S_value a string list of the lines as \r - separated list items:
Grep/P=myPath/LIST="\r"/Q/E="(?i)\\bBoondoggle\\b" "afile.txt"
if( V_Value )                    // some were found
   Print S_value
endif

// Create in W_index a list of the 0-based line numbers where "boondoggle"
// or "boondoggles", etc was found in afile.txt.
Grep/P=myPath/INDX/Q/E="(?i)boondoggle" "afile.txt"
if( V_flag == 0 )   // grep succeeded, perhaps none were found; let's see where
   WAVE W_Index     // needed if in a function
   Edit W_Index     // show line numbers in a table.
endif

// (Create a string list and text wave for the following examples.)
String list= CTabList()          // "Grays;Rainbow;YellowHot;..."
Variable items= ItemsInList(list)
Make/O/T/N=(items) textWave= StringFromList(p,list)

// Copy rows of textWave that contain "Red" (case sensitive)
// to the Clipboard as carriage-return separated lines.
Grep/E="Red" textWave as "Clipboard"

// Copy lines of the Clipboard that do NOT contain "Blue"
// (case in-sensitve) back to the Clipboard, overwriting what was there:
Grep/E={"(?i)blue",1} "Clipboard" as "Clipboard"
```

```
// Format matching text wave row to the history area
Grep/E=("Red")/DCOL={"prefix text --- ", 0, " --- suffix text"} textWave
// Printed output:
   prefix text --- BlueRedGreen --- suffix text
   prefix text --- RedWhiteBlue --- suffix text
   prefix text --- BlueRedGreen256 --- suffix text
   prefix text --- RedWhiteBlue256 --- suffix text
   prefix text --- Red --- suffix text
   prefix text --- RedWhiteGreen --- suffix text
   prefix text --- BlueBlackRed --- suffix text


// Re-copy rows of textWave that contain "Red" (case sensitive)
// to the Clipboard as carriage-return separated lines.
Grep/E="Red" textWave as "Clipboard"
// Create a 2-column text wave whose column 1 (the second column)
// contains the matching text from the Clipboard
Make/O/N=(0,2)/T outputTextWave
// Grep with /A to preserve 2 columns of outputTextWave
Grep/A/E="Red"/GCOL=1/DCOL={1} "Clipboard" as outputTextWave
Edit outputTextWave

// Examples with two-dimensional source text waves
Make/O/T/N=(10, 3) sourceTW= StringFromList(p+10*q,list)
Edit sourceTW

// Copy rows of textWave that contain "Red" in column 2 to outputTextWave.
Make/O/N=0/T outputTextWave
Grep/E="Red"/GCOL=2 sourceTW as outputTextWave
Edit outputTextWave

// Format matching text wave columns to the history area.
// Match lines that contain "Red" in any column of sourceTW:
Grep/E=("Red")/GCOL=-1/DCOL={0,", ",1,", ",2} sourceTW
// Printed output:
   YellowHot, BlueRedGreen256, Magenta
   BlueHot, RedWhiteBlue256, Yellow
   BlueRedGreen, PlanetEarth256, Copper
   RedWhiteBlue, Terrain256, Gold
   Terrain, Rainbow16, RedWhiteGreen
   Grays256, Red, BlueBlackRed
```

References

The regular expression syntax supported by Grep, **GrepString**, and **GrepList** is based on the *PCRE — Perl-Compatible Regular Expression Library* by Philip Hazel, University of Cambridge, Cambridge, England. The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5.

Visit <`http://pcre.org/`> for more information about the PCRE library.

A good book on regular expressions is: Friedl, Jeffrey E. F., *Mastering Regular Expressions*, 2nd ed., 492 pp., O'Reilly Media, 2002.

A helpful web site is: http://www.regular-expressions.info

See Also

Regular Expressions on page IV-176 and **Symbolic Paths** on page II-22.

Demo, **CopyFile**, **PutScrapText**, **LoadWave** operations. The **GrepString**, **GrepList**, **StringMatch**, and **CmpStr** functions.

GrepList

GrepList(*listStr*, *regExprStr* [,*reverse* [, *listSepStr*]])

The GrepList function returns each list item in *listStr* that matches the regular expression *regExprStr*.

ListStr should contain items separated by *listSepStr* which typically is ";".

regExprStr is a regular expression such as is used by the UNIX `grep(1)` command. It is much more powerful than the wildcard syntax used for **ListMatch**. See **Regular Expressions** on page IV-176 for *regExprStr* details.

reverse is optional. If missing, it is taken to be 0. If *reverse* is nonzero then the sense of the match is reversed. For example, if *regExprStr* is `"^abc"` and *reverse* is `1`, then all list items that do not start with "abc" are returned.

listSepStr is optional; the default is `";"`. In order to specify *listSepStr*, you must precede it with reverse.

Examples

To list ColorTables containing "Red", "red", or "RED" (etc.):

```
Print GrepList(CTabList(),"(?i)red")        // case-insensitive matching
```

To list window recreation commands starting with "\tCursor":

```
Print GrepList(WinRecreation("Graph0", 0), "^\tCursor", 0 , "\r")
```

See Also

Regular Expressions on page IV-176.

ListMatch, **StringFromList**, and **WhichListItem** functions and the **Grep** operation.

GrepString

`GrepString(`*`string`*`, `*`regExprStr`*`)`

The GrepString function tests *string* for a match to the regular expression *regExprStr*. Returns 1 to indicate a match, or 0 for no match.

Details

regExprStr is a regular expression such as is used by the UNIX grep(1) command. It is much more powerful than the wildcard syntax used for **StringMatch**. See **Regular Expressions** on page IV-176 for *regExprStr* details.

Character matching is case-sensitive by default, similar to **strsearch**. Prepend the Perl 5 modifier "(?i)" to match upper and lower-case text

Examples

Test for truth that the string contains at least one digit:

```
if( GrepString(str,"[0-9]+") )
```

Test for truth that the string contains at least one "abc", "Abc", "ABC", etc.:

```
if( GrepString(str,"(?i)abc") )             // case-insensitive test
```

See Also

Regular Expressions on page IV-176.

The **StringMatch**, **CmpStr**, **strsearch**, **ListMatch**, and **ReplaceString** functions and the **Demo** and **sscanf** operations.

GridStyle

`GridStyle`

GridStyle is a procedure subtype keyword that puts the name of the procedure in the Grid->Style Function submenu of the mover pop-up menu in the drawing tool palette. You can have Igor automatically create a grid style function for you by choosing Save Style Function from that submenu.

GroupBox

`GroupBox` `[`**`/Z`**`]` ***`ctrlName`*** `[`***`keyword = value`*** `[,` ***`keyword = value`*** `…]]`

The GroupBox operation creates a box to surround and group related controls.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the GroupBox control to be created or changed.

The following keyword=value parameters are supported:

align=*alignment*

Sets the alignment mode of the control. The alignment mode controls the interpretation of the *leftOrRight* parameter to the pos keyword. The align keyword was added in Igor Pro 8.00.

If *alignment*=0 (default), *leftOrRight* specifies the position of the left end of the control and the left end position remains fixed if the control size is changed.

If *alignment*=1, *leftOrRight* specifies the position of the right end of the control and the right end position remains fixed if the control size is changed.

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **DefaultGUIControls Default Fonts and Sizes** for how enclosed controls are affected by native groupbox appearance.

See **Button** for more appearance details.

disable=*d*

Sets user editability of the control.

d=0:	Normal.
d=1:	Hide.
d=2:	Draw in gray state.

fColor=(*r*,*g*,*b*[,*a*])

Sets color of the title text. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

font="*fontName*"

Sets font used for the box title, e.g., `font="Helvetica"`.

frame=*f*

Sets frame mode. If 1 (default), the frame has a 3D look. If 0, then a simple gray line is used. Generally, you should not use frame=0 with a title if you want to be in accordance with human interface guidelines.

fsize=*s*

Sets font size for box title.

fstyle=fs

Sets the font style of the title text. *fs* is a bitwise parameter with each bit controlling one aspect of the font style for the tick mark labels as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*}

Sets the help for the control.

helpStr is limited to 1970 bytes (255 in Igor Pro 8 and before).

You can insert a line break by putting “\r” in a quoted string.

labelBack=(*r*,*g*,*b*[,*a*]) or 0

Sets fill color for the interior. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

If you do not set labelBack then the interior is transparent.

If an opaque fill color is used, drawing objects can not be used because they will be covered up. Also, you will have to make sure the GroupBox is drawn before any interior controls.

The fidelity of the coloring is platform-dependent.

pos={*leftOrRight*,*top*} Sets the position in **Control Panel Units** of the top/left corner of the control if its alignment mode is 0 or the top/right corner of the control if its alignment mode is 1. See the align keyword above for details.

pos+={*dx*,*dy*} Offsets the position of the box in **Control Panel Units**.

size={*width*,*height*} Sets box size in **Control Panel Units**.

userdata(*UDName*)=*UDStr*

Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

title=*titleStr* Sets title to *titleStr*. Use `""` for no title.

win=*winName* Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

If no title is given and the width is less than 11 or height is specified as less than 6, then a vertical or horizontal separator line will be drawn rather than a box.

Note: Like TabControls, you need to click near the top of a GroupBox to select it.

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

The **GetUserData** function for retrieving named user data.

The **ControlInfo** operation for information about the control.

GuideInfo

`GuideInfo(`*`winNameStr, guideNameStr`*`)`

The GuideInfo function returns a string containing a semicolon-separated list of information about the named guide line in the named host window or subwindow.

Parameters

winNameStr can be `""` to refer to the top host window.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

guideNameStr is the name of the guide line for which you want information.

Details

The returned string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

The following keywords will be present only for user-defined guides:

See Also

The **GuideNameList**,s **StringByKey** and **NumberByKey** functions; the **DefineGuide** operation.

Keyword	Information Following Keyword
NAME	Name of the guide.
WIN	Name of the window or subwindow containing the guide.
TYPE	The value associated with this keyword is either *User* or *Builtin*. A *User* type denotes a guide created by the DefineGuide operation, equivalent to dragging a new guide from an existing one.
HORIZONTAL	Either 0 for a vertical guide, or 1 for a horizontal guide.
POSITION	The position of the guide in points. This is the actual position relative to the left or top edge of the window, not the relative position specified to DefineGuide.

Keyword	Information Following Keyword
GUIDE1	The guide is positioned relative to GUIDE1.
GUIDE2	In some cases, the guide is positioned at a fractional position between GUIDE1 and GUIDE2. If the guide does not use GUIDE2, the value will be `""`.
RELPOSITION	The position relative to GUIDE1 (and GUIDE2 if applicable). This is the same as the *val* parameter in DefineGuide. The returned value is in units of points if only GUIDE1 is used, or a fractional value if both GUIDE1 and GUIDE2 are used.

GuideNameList

GuideNameList(*winNameStr*, *optionsStr*)

The GuideNameList function returns a string containing a semicolon-separated list of guide names from the named host window or subwindow.

Parameters

winNameStr can be "" to refer to the top host window.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

optionsStr is used to further qualify the list of guides. It is a string containing keyword-value pairs separated by commas. Use "" to list all guides. Available options are:

TYPE:type	type = BuiltIn: List only built-in guides. type = User: List only user-defined guides, those created by the DefineGuide operation or by manually dragging a new guide from an existing one.
HORIZONTAL:h	h = 0: List only non-horizontal (that is, vertical) guides. h = 1: List only horizontal guides.

Example

```
String list = GuideNameList("Graph0", "TYPE:Builtin,HORIZONTAL:1")
```

See Also

The **DefineGuide** operation and the **GuideInfo** function.

Hanning

Hanning *waveName* [, *waveName*]…

Note: The **WindowFunction** operation has replaced the Hanning operation.

The Hanning operation multiplies the named waves by a Hanning window (which is a raised cosine function).

You can use Hanning in preparation for performing an FFT on a wave if the wave is not an integral number of cycles long.

The Hanning operation is not multidimensional aware. See Chapter II-6, **Multidimensional Waves**, particularly **Analysis on Multidimensional Waves** on page II-95 for details.

See Also

The **WindowFunction** operation implements the Hanning window as well as other forms such as Hamming, Parzen, and Bartlet (triangle).

ImageWindow, **DPSS**

Hash

`Hash(`*`inputStr`*`, `*`method`*`)`

The Hash function returns a cryptographic hash of the data in *inputStr*.

Parameters

inputStr is string of length up to 2^31 bytes. *inputStr* can contain binary or text data.

method is a number indicating the hash algorithm to use:

1	SHA-256 (SHA-2)
2	MD4
3	MD5
4	SHA-1
5	SHA-224 (SHA-2)
6	SHA-384 (SHA-2)
7	SHA-512 (SHA-2)

Prior to Igor Pro 7.00, only method 1 was supported.

See Also

WaveHash, **StringCRC**, **WaveCRC**

HCluster

`HCluster [ flags ] `*`sourceWave`*

The HCluster operation computes the information needed to create a cluster dendrogram using an agglomerative hierarchical clustering algorithm. "HCluster" stands for "hierarchical clustering". The HCluster operation was added in Igor Pro 9.00.

For background information, see **Hierarchical Clustering** on page III-162.

The input *sourceWave* represents either vectors in some data space or a square vector dissimilarity matrix (also called a "distance" matrix). You indicate which type of input you are providing using the /ITYP flag.

HCluster creates an output vector dissimilarity matrix wave or an output dendrogram wave or both, depending on the /OTYP flag. The output wave names default to M_HCluster_Dissimilarity and M_HCluster_Dendrogram but you can override the default using /DEST.

Flags

/ITYP=*it*	*it* is a keyword specifying the kind of data in *sourceWave*: *it*=`Vectors`: *sourceWave* rows represent data vectors (default). *it*=`DMatrix`: *sourceWave* contains a square vector dissimilarity matrix.

/OTYP=*ot*

ot is a keyword specifying what type of output to be produced:

ot=`DMatrix`: The output is a vector dissimilarity matrix. You can use /OTYP=DMatrix only if /ITYP=Vectors or if you omit /ITYP.

ot=`Dendrogram`: The output is a multi-column wave describing the nodes in a dendrogram illustrating the way original data is joined into clusters. This is the default if you omit /OTYP.

ot=`Both`: The output is both the vector dissimilarity matrix and a dendrogram.

See the /DEST flag for further discussion of the output wave or waves.

/LINK=*linkMethod*

linkMethod is a keyword specifying the method used to determine the dissimilarity between nodes in the dendrogram that represent more than one data vector. This is also referred to as the "linkage" method. Our definitions of node dissimilarities follows Python scipy.cluster.hierarchy.linkage.

The available keywordds for linkMethod are listed and described under **HCluster Linkage Calculation Methods** on page III-166.

If you omit /LINK, HCluster defaults to the average method.

/DISS=*dm*

dm is a keyword specifying the vector dissimilarity metric for calculating the dissimilarity between two data vectors. Our definitions of vector dissimilarity follows Python scipy.spatial.distance.pdist.

The available /DISS keywords are listed and described under **HCluster Vector Dissimilarity Calculation Methods** on page III-163.

If you omit /DISS, HCluster defaults to the Euclidean metric.

/P=*pow*

pow is the power for the Minkowski vector dissimilarity metric. The value of *pow* must be positive. The default is 2.0, equivalent to the Euclidean vector dissimilarity metric. Values that are too large can lead to floating-point overflow. Values less than 1.0 may give surprising results, as this can cause an inversion of the usual distance ordering. If the vector dissimilarity metric is not Minkowski this flag is ignored.

/VARW=*varWave*

Specifies the normalizing values Vj for use with the SEuclidean vector dissimilarity metric. Usually, the wave elements are variances of the vector elements over all the vectors. Thus, if you have a multi-column wave in which rows represent individual vectors, *varWave* should be filled with variances of the wave's columns. If your vectors have length of M, then varWave should be a 1D wave with M elements. This wave can be conveniently created using the MatrixOP operation, like this:

```
MatrixOp/O varWave = VarCols(rowVectorMatrix)^t
```

If the vector dissimilarity matrix is not SEuclidean, the /VARW flag is ignored.

/DEST=*outWaveName*

Specifies the output waves when you have specified /OTYP=`DMatrix` or /OTYP=`Dendrogram`.

If you specified /OTYP=`DMatrix`, *outWaveName* is the name of the output vector dissimilarity matrix wave to be created or overwritten, optionally preceded by a data folder path. If you omit /DEST, HCluster creates an output vector dissimilarity matrix named M_HCluster_Dissimilarity in the current data folder.

If you specified /OTYP=`Dendrogram`, *outWaveName* is the name of the output dendrogram wave to be created or overwritten, optionally preceded by a data folder path. If you omit /DEST, HCluster creates an output dendrogram named M_HCluster_Dendrogram in the current data folder.

/DEST={*dMatrixName*, *dendrodrogramName*}

Specifies the output waves when you have specified /OTYP=`Both`.

dMatrixName and *dendrodrogramName* are names of waves to be created or overwritten, optionally preceded by data folder paths.

If you specify /OTYP=Both and omit /DEST, HCluster creates an output vector dissimilarity matrix named M_HCluster_Dissimilarity and an output dendrogram wave named M_HCluster_Dendrogram, both in the current data folder.

/O If present, allows the destination waves specified by the /DEST flag to overwrite a pre-existing wave.

Parameter

If you specify /ITYP=`Vectors` or omit /ITYP, *sourceWave* is an N row x M column matrix containing N data vectors of length M in the rows. HCluster creates a vector dissimilarity matrix from this input using the distance calculation method specified by /LINK.

If you specify /ITYP=`DMatrix`, *sourceWave* is a square matrix of dissimilarities between data vectors. If you choose this format, you are responsible for computing the dissimilarities between vectors. If none of the vector dissimilarity metrics provided by the /DISS flag are suitable, or if you require more processing after computing dissimilarities, you can use this format.

Dendrogram Output Wave

The HCluster operation optionally produces a dendrogram output wave that can be used to create a dendrogram plot. See **Dendrogram Wave Format** on page III-167 for a description of the dendrogram output wave format.

Reference

The HCluster operation is based on code developed by Daniel Müllner. This reference gives details of the algorithm and the various distance and vector dissimilarity measures and node agglomeration methods:

Daniel Müllner, fastcluster: Fast Hierarchical, Agglomerative Clustering Routines for R and Python, Journal of Statistical Software, 53 (2013), no. 9, 1–18, http://www.jstatsoft.org/v53/i09/.

See Also

Hierarchical Clustering on page III-162

hcsr

`hcsr(`*`cursorName`*` [, `*`graphNameStr`*`])`

The hcsr function returns the horizontal coordinate of the named cursor (A through J) in the coordinate system of the top (or named) graph's X axis.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The X axis used is the one that controls the trace on which the cursor is placed.

Examples

```
Variable xAxisValueAtCursorA = hcsr(A)      // not hcsr("A")
String str="A"
Variable xA= hcsr($str,"Graph0")            // $str is a name, too
```

See Also

The **pcsr**, **qcsr**, **vcsr**, **xcsr**, and **zcsr** functions.

Programming With Cursors on page II-321.

HDF5AttributeInfo

HDF5AttributeInfo(*locationID*, *objectNameStr*, *objectType*, *attributeNameStr*, *options*, *di*)

The HDF5AttributeInfo function stores information about the attribute such as its rank, dimension sizes and type in the HDF5DataInfo structure pointed to by *di*.

Documentation for the HDF5AttributeInfo function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5AttributeInfo"
```

HDF5CloseFile

HDF5CloseFile [/A /Z] *fileID*

The HDF5CloseFile operation closes an HDF5 file previously opened by HDF5OpenFile or HDF5CreateFile.

Documentation for the HDF5CloseFile operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5CloseFile"
```

HDF5CloseGroup

HDF5CloseGroup [/Z] *groupID*

The HDF5CloseGroup operation closes an HDF5 group that you opened via HDFCreateGroup or HDF5OpenGroup.

Documentation for the HDF5CloseGroup operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5CloseGroup"
```

HDF5Control

HDF5Control [keyword=value [, keyword=value]]

The HDF5Control HDF5Control provides control of aspects of Igor’s use of the HDF5 file format.

Documentation for the HDF5CreateFile operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5Control"
```

HDF5CreateFile

HDF5CreateFile [/I /O /P=pathName /Z] *fileID* as *fileNameStr*

The HDF5CreateFile operation creates a new HDF5 file or overwrites an existing HDF5 file.

Documentation for the HDF5CreateFile operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5CreateFile"
```

HDF5CreateGroup

HDF5CreateGroup [/Z] *locationID*, *nameStr*, *groupID*

The HDF5CreateGroup operation creates an HDF5 group in the HDF5 file. It returns a group ID via *groupID*.

Documentation for the HDF5CreateGroup operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5CreateGroup"
```

HDF5CreateLink

HDF5CreateLink [/EXT={*pathName*,*filePath*} /HARD=*makeHardLink* /Q /Z]
targetLocationID*, *targetName*, *linkLocationID*, *linkName

The HDF5CreateLink operation creates a new hard, soft or external link in an HDF5 file.

Documentation for the HDF5CreateLink operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5CreateLink"
```

HDF5DataInfo

The HDF5DataInfo structure is used with the HDF5DataInfo and HDF5AttributeInfo functions.

Documentation for the HDF5DataInfo structure is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5DataInfo"
```

HDF5DatasetInfo

HDF5DatasetInfo(*locationID*, *datasetNameStr*, *options*, *di*)

The HDF5DatasetInfo function stores information about the dataset such as its rank, dimension sizes and type in the HDF5DataInfo structure referenced by *di*.

Documentation for the HDF5DatasetInfo function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5DatasetInfo"
```

HDF5DatatypeInfo

The HDF5DatatypeInfo structure is used with the HDF5TypeInfo function.

Documentation for the HDF5DatatypeInfo structure is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5DatatypeInfo"
```

HDF5DimensionScale

HDF5DimensionScale [***flags***] **[keyword=value [, keyword=value]]**

The HDF5DimensionScale operation supports the creation and querying of HDF5 dimension scales.

Documentation for the HDF5DimensionScale operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5DimensionScale"
```

HDF5Dump

HDF5Dump [***flags***] ***fileNameStr***

The HDF5Dump operation dumps information about the specified HDF5 file.

Documentation for the HDF5Dump operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5Dump"
```

HDF5DumpErrors

HDF5DumpErrors [/CLR=*clear* /Q]

The HDF5DumpErrors operation dumps information about HDF5 library errors encountered by Igor.

Documentation for the HDF5DumpErrors operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5DumpErrors"
```

HDF5FlushFile

HDF5FlushFile [/A /Z] *fileID*

The HDF5FlushFile operation flushes an HDF5 file previously opened by HDF5OpenFile or HDF5CreateFile.

Documentation for the HDF5FlushFile operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5FlushFile"
```

HDF5LibraryInfo

HDF5LibraryInfo(*options*)

The HDF5LibraryInfo function returns information about the HDF5 library used by the currently-running version of Igor.

Documentation for the HDF5LibraryInfo function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5LibraryInfo"
```

HDF5LinkInfo

HDF5LinkInfo(*locationID*, *pathStr*, *options*, *li*)

The HDF5LinkInfo function stores information about an HDF5 link in the HDF5LinkInfoStruct structure pointed to by *li*.

Documentation for the HDF5LinkInfo function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5LinkInfo"
```

HDF5LinkInfoStruct

The HDF5LinkInfoStruct structure is used with the HDF5LinkInfo function.

Documentation for the HDF5LinkInfoStruct structure is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5LinkInfoStruct"
```

HDF5ListAttributes

HDF5ListAttributes [/TYPE=type /Z] *locationID*, *nameStr*

The HDF5ListAttributes operation returns a semicolon-separated list of attributes associated with the object specified by *locationID* and *nameStr*.

Documentation for the HDF5ListAttributes operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5ListAttributes"
```

HDF5ListGroup

HDF5ListGroup [***flags***] ***locationID*, *nameStr***

The HDF5ListGroup operation returns a semicolon-separated list of the names of objects in the HDF5 file or group specified by *locationID* and *nameStr*.

Documentation for the HDF5ListGroup operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5ListGroup"
```

HDF5LoadData

HDF5LoadData [*flags*] ***locationID***, ***nameStr***

The HDF5LoadData operation loads a dataset or attribute from an HDF5 file.

Documentation for the HDF5LoadData operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5LoadData"
```

HDF5LoadGroup

HDF5LoadGroup [*flags*] ***dataFolderSpec***, ***locationID***, ***nameStr***

The HDF5LoadGroup operation loads an HDF5 group and its datasets into an Igor Pro data folder.

Documentation for the HDF5LoadGroup operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5LoadGroup"
```

HDF5LoadImage

HDF5LoadImage [*flags*] ***locationID***, ***nameStr***

The HDF5LoadImage operation loads an image dataset and in some cases a palette dataset from an HDF5 file using the format specified in the HDF5 Image and Palette Specification version 1.2.

Documentation for the HDF5LoadImage operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5LoadImage"
```

HDF5OpenFile

HDF5OpenFile [/I /P=pathName /R /Z] *fileID* as *fileNameStr*

The HDF5OpenFile operation opens an existing HDF5 file for reading or for reading and writing.

Documentation for the HDF5OpenFile operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5OpenFile"
```

HDF5OpenGroup

HDF5OpenGroup [/Z] *locationID*, ***nameStr***, ***groupID***

The HDF5OpenGroup operation opens an existing HDF5 group in the HDF5 file.

Documentation for the HDF5OpenGroup operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5OpenGroup"
```

HDF5SaveData

HDF5SaveData [*flags*] ***wave***, ***locationID* [, *nameStr*]**

The HDF5SaveData operation saves a single wave in an HDF5 file as a dataset or as an attribute if /A is present.

Documentation for the HDF5SaveData operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5SaveData"
```

HDF5SaveDataHookStruct

The HDF5SaveDataHookStruct structure is used with the HDF5SaveDataHook function.

Documentation for the HDF5DatatypeInfo structure is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5SaveDataHookStruct"
```

HDF5SaveGroup

HDF5SaveGroup [***flags***] ***dataFolderSpec***, ***locationID***, ***nameStr***

The HDF5SaveGroup operation saves the contents of an Igor data folder in an HDF5 file.

Documentation for the HDF5SaveGroup operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5SaveGroup"
```

HDF5SaveImage

HDF5SaveImage [***flags***] ***keyword*** [***=value***]

The HDF5SaveImage operation saves an image dataset and in some cases a palette dataset in an HDF5 file using the format specified in the HDF5 Image and Palette Specification version 1.2.

Documentation for the HDF5SaveImage operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5SaveImage"
```

HDF5TypeInfo

HDF5TypeInfo(*locationID*, *datasetOrGroupNameStr*, *attributeNameStr*, *memberName*, *options*, *dti*)

The HDF5TypeInfo function stores information about the datatype of a dataset or attribute in the HDF5DataTypeInfo structure referenced by *dti*.

Documentation for the HDF5TypeInfo function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5TypeInfo"
```

HDF5UnlinkObject

HDF5UnlinkObject [/Z] ***locationID***, ***nameStr***

The HDF5UnlinkObject operation unlinks the specified object (a group, dataset, datatype or link) from the HDF5 file.

Documentation for the HDF5UnlinkObject operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "HDF5UnlinkObject"
```

hermite

hermite(*n*, *x*)

The hermite function returns the Hermite polynomial of order *n*:

$$H_n(x) = (-1)^n \exp\left(x^2\right)\frac{d^n}{dx^n}\exp\left(-x^2\right).$$

The first few polynomials are:

$$1$$

$$2x$$

$$4x^2 - 2$$

$$8x^3 - 12x$$

See Also

The **hermiteGauss** function.

hermiteGauss

hermiteGauss(*n*, *x*)

The hermiteGauss function returns the normalized Hermite polynomial of order *n*:

$$H_n(x) = \frac{1}{\sqrt{\sqrt{\pi}\,2^n n!}}(-1)^n \exp\left(x^2\right)\frac{d^n}{dx^n}\exp\left(-x^2\right).$$

Here the normalization was chosen such that

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x) H_m(x)\,dx = \delta_{mn},$$

where δ_{nm} is the Kronecker symbol.

You can verify the Hermite-Gauss normalization using the following functions:

```
Function TestNormalization(order)
   Variable order

   Variable/G theOrder = order
   // The integrand vanishes in double-precision outside [-30,30]
   Print/D Integrate1D(hermiteIntegrand,-30,30,2)
End

Function HermiteIntegrand(inX)
   Variable inX

   NVAR n = root:theOrder
   return HermiteGauss(n,inx)^2*exp(-inx*inx)
End
```

See Also

The **hermite** function.

hide

#pragma hide = *value*

The hide pragma allows you to make a procedure file invisible.

See Also

The **The hide Pragma** on page IV-54 and **#pragma**.

HideIgorMenus

HideIgorMenus [***MenuNameStr*** [, ***MenuNameStr***]…

The HideIgorMenus operation hides the named built-in menus or, if none are explicitly named, hides all built-in menus in the menu bar.

The effect of HideIgorMenus is lost when a new experiment is opened. The state of HideIgorMenus is saved with the experiment.

User-defined menus are not hidden by HideIgorMenus unless attached to built-in menus and the menu definition uses the hideable keyword.

Parameters

MenuNameStr	The name of an Igor menu, like “File”, “Data”, or “Graph”.

Details

The optional menu names are in English and not abbreviated. This ensures that code developed for a localized version of Igor will run on all versions.

The built-in menus that can be shown or hidden (the Help menu can be hidden only on Windows) are those that appear in the menu bar:

File	Edit	Data	Analysis	Macros	Windows	Graph
Layout	Notebook	Panel	Procedure	Table	Misc	Help

Hiding a built-in menu to which a user-defined menu is attached results in a built-in menu with only the user-defined items. For example, if this menu definition attaches items to the built-in Graph menu:

```
Menu "Graph"
   "Do My Graph Thing", ThingFunction()
End
```

Calling `HideIgorMenus "Graph"` will still leave a Graph menu showing (when a Graph is the top-most target window) with only the user-defined menu(s) in it: in this example the one “Do My Graph Thing” item.

Hiding the Macros menu hides menus created from Macro definitions like:

```
Macro MyMacro()
   Print "Hello, world."
End
```

but does not hide normal user-defined “Macros” definitions like:

```
Menu "Macros"
   "Macro 1", MyMacro(1)
End
```

You can set user-defined menus to hide and show along with built-in menus by adding the optional hideable keyword to the menu definition:

```
Menu "Graph", hideable
   "Do My Graph Thing", ThingFunction()
End
```

Then `HideIgorMenus "Graph"` will hide those items, too. If all user-defined Graph menu definitions use the hideable keyword, then no Graph menu will appear in the menu bar.

Some WaveMetrics procedures use the `hideable` keyword so that only customer-defined menus remain when HideIgorMenus is executed.

See Also

ShowIgorMenus, **DoIgorMenu**, **SetIgorMenuMode**, Chapter IV-5, **User-Defined Menus**

HideInfo

`HideInfo` `[`***`/W=winName`***`]`

The HideInfo operation removes the info panel from a graph if it was previously shown by the **ShowInfo** operation.

Flags

/W=*winName*	Hides the info panel in the named window.

See Also

The **ShowInfo** operation.

Programming With Cursors on page II-321.

HideProcedures

`HideProcedures`

The HideProcedures operation hides all procedure windows without closing or killing them.

See Also

The **DisplayProcedure** and **DoWindow** operations.

HideTools

`HideTools` `[`***`/A/W=winName`***`]`

The HideTools operation hides the tool palette in the top graph or control panel if it was previously shown by the **ShowTools** operation.

Flags

/A Sizes the window automatically to make extra room for the tool palette. This preserves the proportion and size of the actual graph area.

/W=*winName* Hides the tool palette in the named window. This must be the first flag specified when used in a Proc or Macro or on the command line.

winName must be either the name of a top-level window or a path leading to an exterior panel window (see **Exterior Control Panels** on page III-443).

See Also

The **ShowTools** operation.

HilbertTransform

HilbertTransform [**/Z**][**/O**][**/DEST=*destWave***] ***srcWave***

The HilbertTransform operation computes the Hilbert transformation of *srcWave*, which is a real or complex (single or double precision) wave of 1-3 dimensions. The result of the HilbertTransform is stored in *destWave*, or in the wave W_Hilbert (1D) or M_Hilbert in the current data folder.

Flags

/DEST=*destWave* Creates a real wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-72 for details.

/O Overwrites *srcWave* with the transform.

/PAD={*dim1* [, *dim2*, *dim3*, *dim4*]}

Converts *srcWave* into a padded wave of dimensions *dim1*, *dim2*…. The padded wave contains the original data at the start of the dimension and adds zero entries to each dimension up to the specified dimension size. The *dim1*… values must be greater than or equal to the corresponding dimension size of *srcWave*. If you need to pad just the lowest dimension(s) you can omit the remaining dimensions; for example, /PAD=*dim1* will set *dim2* and above to match the dimensions in *srcWave*.

This flag was added in Igor Pro 7.00.

/Z No error reporting.

Details

The Hilbert transform of a function *f*(*x*) is defined by:

$$F(t) = \frac{1}{\pi t} \int_{-\infty}^{\infty} \frac{f(x)dx}{x-t}.$$

Theoretically, the integral is evaluated as a Cauchy principal value. Computationally one can write the Hilbert transform as the convolution:

$$F(t) = \frac{-1}{\pi t} * f(t),$$

which by the convolution theorem of Fourier transforms, may be evaluated as the product of the transform of *f*(*x*) with -*i**sgn(*x*) where:

$$\operatorname{sgn}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}.$$

Note that the Hilbert transform of a constant is zero. If you compute the Hilbert transform in more than one dimension and one of the dimensions does not vary (is a constant), the transform will be zero (or at least numerically close to zero).

There are various definitions for the extension of the Hilbert transform to more than one dimension. In two dimensions this operation computes the transform by multiplying the 2D Fourier transform of the input by the factor (-i)sgn(x)(-i)sgn(y) and then computing the inverse Fourier Transform. A similar procedure is used when the input is 3D.

Examples

Extract the instantaneous amplitude and frequency of a narrow-band signal:

```
Make/O/N=1000 w0,amp,phase
SetScale/I x 0,50,"", w0,amp,phase
w0 = exp(-x/10)*cos(2*pi*x)
HilbertTransform /DEST=w0h w0 // w0+i*w0h is the "analytic signal", i=cmplx(0,1)
amp = sqrt(w0^2 + w0h^2)   // extract the envelope
phase = atan2(-w0h,w0)     // extract the phase [SIGN CONVENTION?]
Unwrap 2*pi, phase         // eliminate the 2*pi phase jumps
Differentiate phase /D=freq   // would have less noise if fit to a line
                              // over interior points
freq /= 2*pi               // phase = 2*pi*freq*time
Display w0,amp   // original waveform and its envelope; note boundary effects
Display freq     // instantaneous frequency estimate, with boundary effects
```

See Also

The **FFT** operation.

References

Bracewell, R., *The Fourier Transform and Its Applications*, McGraw-Hill, 1965.

Compute the envelope of a signal:

```
Function calcEnvelope(Wave ddd)
HilbertTransform/dest=ht ddd
Matrixop/o sEnv=abs(cmplx(ddd,ht))
CopyScales ddd,sEnv
KillWaves/z ht
End
```

Histogram

Histogram [*flags*] *srcWaveName*, *destWaveName*

The Histogram operation generates a histogram of the data in *srcWaveName* and puts the result in *destWaveName* or in W_Histogram or in the wave specified by /DEST.

Parameters

srcWaveName specifies the wave containing the data to be histogrammed.

For historical reasons the meaning and use of *destWaveName* depend on the binning mode as specified by /B. See **Histogram Destination Wave** on page V-351 below for details.

Flags

/A Accumulates the histogram result with the existing values in the destination wave instead of replacing the existing values with the result. Assumes /B=2 unless the /B flag is present.

Note: The result will be incorrect if you also use /P.

/B=mode — Controls binning:

mode=1: Semi-automatic mode that sets the bin range based on the range of the Y values in *srcWaveName*. The number of bins is determined by the number of points in the destination wave.

mode=2: Uses the bin range and number of bins determined by the X scaling and number of points in the destination wave.

mode=3: Uses Sturges' method to determine optimal number of bins and redimensions the destination wave as necessary. By this method

```
numBins=1+log2(N)
```

where N is the number of data points in *srcWaveName*. The bins will be distributed so that they include the minimum and maximum values.

mode=4: Uses a method due to Scott, which determines the optimal bin width as

$$\texttt{binWidth=3.49*}\sigma\texttt{*N}^{-1/3}$$

where N is the number of data points in *srcWaveName* and σ is the standard deviation of the distribution. The bins will be distributed so that they include the minimum and maximum values.

method=5: Uses the Freedman-Diaconis method where

$$\texttt{binWidth=2*IQR*N}^{-1/3}$$

where IQR is the interquartile distance (see **StatsQuantiles**) and the bins are evenly distributed between the minimum and maximum values.

/B={*binStart*,*binWidth*,*numBins*}

Sets the histogram bins from these parameters rather than from *destWaveName*. Changes the X scaling and length of the destination wave.

/C — Sets the X scaling so that X values are in the centers of the bins, which is required when you do a curve fit to the histogram output. Ordinarily, wave scaling of the output wave is set with X values at the left bin edges.

/CUM — Requests a cumulative histogram in which each bin is the sum of bins to the left. The last bin will contain the total number of input data points, or, with /P, 1.0.

/CUM cannot be used with a weighted histogram (/W flag).

When used with /A, the destination wave must be the result of a histogram created with /CUM.

Note that if you use a binning mode (/B flag) that sets a bin range that does not include the entire range of the input data, then the output will not count all of input points and the last bin will not contain the total number of input points. Input points whose values fall below the left edge of the first bin or above the right edge of the last bin will not be counted.

/DEST=*destWave* — Saves the histogram output in a wave specified by *destWave*. The destination wave is created or overwritten if it already exists.

Creates a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-72 for details.

See **Histogram Destination Wave** on page V-351 below for further discussion.

The /DEST flag was added in Igor Pro 7.00.

/DP — Causes Histogram to create the destination wave as double-precision floating point instead of single-precision floating point. The /DP flag was added in Igor Pro 8.00.

Single-precision precisely represents integers up to 16,677,721 only. Double-precision precisely represents integers up to about 9 trillion.

/N

Creates a wave named W_SqrtN containing the square root of the number of counts in each bin. This is an appropriate wave to use as a weighting wave when doing a curve fit to the histogram results. /N cannot be used with a weighted histogram (/W flag).

/NLIN=*binsWave*

Computes a non-linear histogram using the bins specified in the wave *binsWave*. This option is not compatible with the flags /A, /B, /C, /CUM, /N, /P, /W.

The bins must be contiguous and non-overlapping so that *binsWave* contains monotonically increasing values with no NaNs and INFs. For example, if you want the 3 bins [1,10),[10,100),[100,1000), execute:

Make/O/N=4 *bins*={1,10,100,1000}

The upper end of each bin is open.

The /NLIN flag was added in Igor Pro 7.00.

/P

Normalizes the histogram as a probability distribution function, and shifts wave scaling so that data correspond to the bin centers.

When using the results with **Integrate**, you must use /METH=0 or /METH=2 to select rectangular integration methods.

/R=(*startX*,*endX*)

Specifies the range of X values of *srcWaveName* over which the histogram is to be computed.

/R=[*startP*,*endP*]

Specifies the range of points of *srcWaveName* over which the histogram is to be computed.

/RMD=[*firstRow*,*lastRow*][*firstColumn*,*lastColumn*][*firstLayer*,*lastlayer*][*firstChunk*,*lastChunk*]

Designates a contiguous range of data in the source wave to which the operation is to be applied. This flag was added in Igor Pro 7.00.

You can include all higher dimensions by leaving off the corresponding brackets. For example:

```
/RMD=[firstRow,lastRow]
```

includes all available columns, layers and chunks.

You can use empty brackets to include all of a given dimension. For example:

```
/RMD=[][firstColumn,lastColumn]
```

means "all rows from column A to column B".

You can use a * to specify the end of any dimension. For example:

```
/RMD=[firstRow,*]
```

means "from firstRow through the last row".

/W=*weightWave*

Creates a “weighted” histogram. In this case, instead of adding a single count to the appropriate bin, the corresponding value from *weightWave* is added to the bin. *weightWave* may be any number type, and it may be complex. If it is complex, then the destination wave will be complex.

/W cannot be used with a cumulative histogram (/CUM flag).

Histogram Destination Wave

For historical reasons there are multiple ways to specify the destination wave and the meaning and use of *destWaveName* depend on the binning mode as specified by /B. This section explains the details and then provides guidance and when to use which mode.

In binning modes 1 and 2 (/B=1 and /B=2, described above), the destination wave plays a role in determining the binning and *destWaveName* must be the name of an existing wave. If you omit /DEST then the output is written to *destWaveName*. If you provide /DEST then the output is written to the wave specified by /DEST.

In binning modes 3, 4 and 5 (/B=3, /B=4 and /B=5, described above), the destination wave plays no role in determining the binning. If you omit *destWaveName* and /DEST, Histogram stores its output in a wave named W_Histogram in the current data folder. If you omit /DEST and provide *destWaveName*, then *destWaveName* must name an existing wave to which the output is written. If you provide /DEST, you can omit *destWaveName*. If you provide both /DEST and *destWaveName* then *destWaveName* must name an existing wave but the operation ignores it.

Here is the recommended usage:

If you want to use specific binning that you have determined, use /B={*binStart*,*binWidth*,*numBins*}, use /DEST to specify the destination wave, and omit *destWaveName*.

If you want Igor to determine the binning, use /B=3, /B=4 or /B=5, use /DEST to specify the destination wave, and omit *destWaveName*.

For backward compatibility with Igor Pro 6, use /B=1, /B=2, /B=3, /B=4 or /B={*binStart*,*binWidth*,*numBins*}, create a destination wave, use it as *destWaveName* and omit /DEST.

Details

If you use /B={*binStart*, *binWidth*, *numBins*}, then the initial number of data points in the wave is immaterial since the Histogram operation changes the number of points.

Only one /B and only one /R flag is allowed.

If both /A and /B flags are missing, the bin range and number of bins is calculated as if /B=1 had been specified.

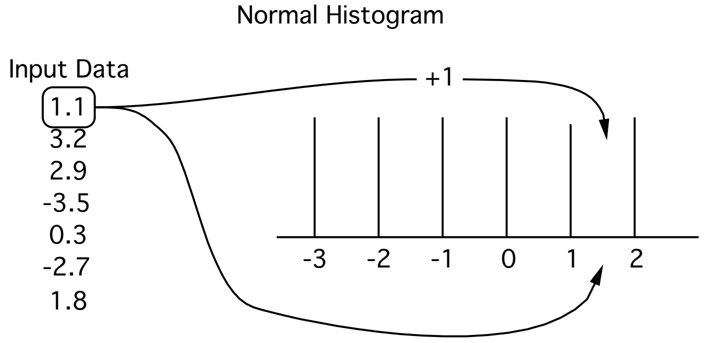
When accumulating multiple histograms in one output wave, typically you will want to use /B={*binStart*,*binWidth*,*numBins*} for the first histogram, and /A for successive histograms.

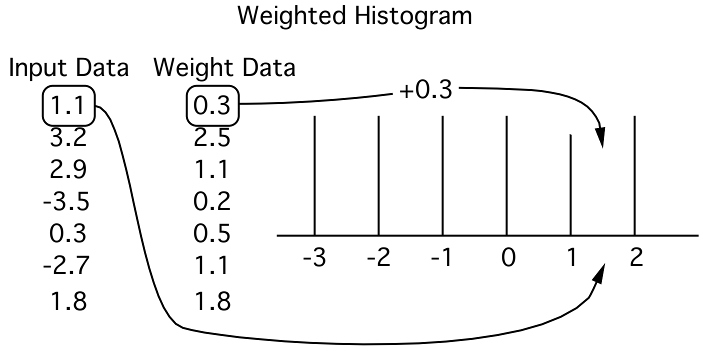
The Histogram operation works on single precision floating point destination waves. If necessary, Histogram redimensions the destination wave to be single precision floating point. However, Histogram/A requires that the destination wave already be single precision floating point.

For a weighted histogram, the destination wave will be double-precision.

If you specify the range as `/R=(start)`, then the end of the range is taken as the end of *srcWaveName*.

In an ordinary histogram, input data is examined one data point at a time. The operation determines which bin a data value falls into and a single count is added to that bin. A weighted histogram works similarly, except that it adds to the bin a value from another wave in which each row corresponds to the same row in the input wave.





The Histogram operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details. In fact, the Histogram operation can be usefully applied to multidimensional waves, such as those that represent images. The /R flag will not work as expected, however.

Examples

```
// Create histogram of two sets of data.
Make/N=1000 data1=gnoise(1), data2=gnoise(1)
Make/N=1 histResult

// Sets bins, does histogram.
Histogram/B={-5,1,10} data1, histResult
Display histResult; ModifyGraph mode=5

// Accumulates into existing bins.
Histogram/A data2, histResult
```

See Also

Histograms on page III-125, **ImageHistogram**, **JointHistogram**, **TextHistogram**

References

Sturges, H.A., The choice of a class-interval, *J. Amer. Statist. Assoc.*, *21*, 65-66, 1926.

Scott, D., On optimal and data-based histograms, *Biometrika*, *66*, 605-610, 1979.

hyperG0F1

hyperG0F1(*b*, *z*)

The hyperG0F1 function returns the confluent hypergeometric limit function

$$ {}_0F_1(b;z)=\sum_{i=0}^{\infty}\frac{z^i}{i!(b)_i}, $$

where $(b)_i$ is the Pochhammer symbol

$$ (b)_i = b(b+1)...(b+i-1). $$

The series evaluation may be computationally intensive. You can abort the computation by pressing the **User Abort Key Combinations**.

See Also

The **hyperG1F1**, **hyperG2F1**, and **hyperGPFQ** functions.

References

The PFQ algorithm was developed by Warren F. Perger, Atul Bhalla, and Mark Nardin.

hyperG1F1

hyperG1F1(*a*, *b*, *z*)

The hyperG1F1 function returns the confluent hypergeometric function

$$ {}_1F_1(a,b,z)=\sum_{n=0}^{\infty}\frac{(a)_n z^n}{(b)_n n!}, $$

where $(a)_n$ is the Pochhammer symbol

$$ (a)_n = a(a+1)\ldots(a+n-1). $$

The series evaluation may be computationally intensive. You can abort the computation by pressing the **User Abort Key Combinations**.

See Also

The **hyperG0F1**, **hyperG2F1**, and **hyperGPFQ** functions.

References

The PFQ algorithm was developed by Warren F. Perger, Atul Bhalla, and Mark Nardin.

hyperG2F1

`hyperG2F1(`*`a`*`, `*`b`*`, `*`c`*`, `*`z`*`)`

The hyperG2F1 function returns the confluent hypergeometric function

$$ {}_2F_1(a,b,c,z) = \sum_{n=0}^{\infty} \frac{(a)_n(b)_n z^n}{(c)_n n!} $$

$$ {}_2F_1(a,b,c;z) = \sum_{n=0}^{\infty} \frac{(a)_n(b)_n z^n}{(c)_n n!}, $$

where $(a)_n$ is the Pochhammer symbol

$$ (a)_n = a(a+1)\ldots(a+n-1). $$

Note: The series evaluation may be computationally intensive. You can abort the computation by pressing the **User Abort Key Combinations**.

See Also

The **hyperG0F1**, **hyperG1F1**, and **hyperGPFQ** functions.

References

The PFQ algorithm was developed by Warren F. Perger, Atul Bhalla, and Mark Nardin.

hyperGNoise

`hyperGNoise(`*`m`*`, `*`n`*`, `*`k`*`)`

The hyperGNoise function returns a pseudo-random value from the hypergeometric distribution whose probability distribution function is

$$ f(x;m,n,k) = \frac{\begin{pmatrix} n \\ x \end{pmatrix}\begin{pmatrix} m-n \\ k-x \end{pmatrix}}{\begin{pmatrix} m \\ k \end{pmatrix}} $$

where *m* is the total number of items, *n* is the number of marked items, and *k* is the number of items in a sample.

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

SetRandomSeed, **StatsHyperGCDF**, and **StatsHyperGPDF**.

Chapter III-12, **Statistics** for a function and operation overview.

Noise Functions on page III-390.

hyperGPFQ

`hyperGPFQ(`*`waveA`*`, `*`waveB`*`, `*`z`*`)`

The hyperGPFQ function returns the generalized hypergeometric function

$$ {}_pF_q\left(\left\{a_1,\ldots a_p\right\},\left\{b_1,\ldots b_q\right\};z\right)=\sum_{n=0}^{\infty}\frac{(a_1)_n(a_2)_n\ldots(a_p)_n z^n}{(b_1)_n(b_2)_n\ldots(b_q)_n n!}, $$

where $(a)_n$ is the Pochhammer symbol

$$ (a)_n = a(a+1)\ldots(a+n-1). $$

Note: The series evaluation may be computationally intensive. You can abort the computation by pressing the **User Abort Key Combinations**.

See Also

hyperG0F1, **hyperG1F1**, **hyperG2F1**

CosIntegral, **ExpIntegralE1**, **SinIntegral**

References

The PFQ algorithm was developed by Warren F. Perger, Atul Bhalla, and Mark Nardin.

i

```
i
```

The i function returns the loop index of the inner most iterate loop in a macro. Not to be used in a function. iterate loops are archaic and should not be used.

ICA

ICA [*flags*] *srcWave*

The ICA operation performs independent component analysis using the FastICA algorithm. Input data is in the form of a 2D wave where each column represents the equivalent of a single data acquisition channel. The results of the operation are stored in the waves M_ICAComponents, M_ICAUnMix and M_matrixW in the current data folder.

The ICA operation was added in Igor Pro 7.00.

Flags

/A=*alpha*

alpha is a constant used as a factor in the argument of the logCosh function. It is not used with the exp function.

alpha is in the range [1,2] and its default value is 1.

You will rarely need to change this value to affect the rate of convergence or the quality of the results.

/CF=num

Specifies the contrast function, also called the non-quadratic function, used by ICA.

num=0:	logCosh (default)
num=1:	exp

/COLS

Preconditions the input by subtracting the mean and then normalizing the input on a column-by-column basis. The algorithm appears to converge and produce better results when this flag is used.

As of Igor Pro 9.00, the input is preconditioned by default so /COLS is no longer required or recommended.

/DFLT

Use the deflation/vector method where iterations solve for a single vector of the "unmixing" matrix at a time. By default the operation uses the matrix method which solves for the complete unmixing matrix at one time.

/PCA
Save the output of the "PCA" stage which is also the form of the data before the fastICA iterations. The SVD U matrix is saved in the wave M_PCA and the eigevalues are saved in the wave W_PCAEV. Both are created in the current data folder.

/Q
Quiet mode; do not print anything in the history.

/TOL=*tolerance*
The tolerance value is used to determine when iterations converge.

In the deflation/vector method the tolerance measures the difference between the values of vectors in consecutive iterations.

In the matrix method the tolerance measures the average deviation of all components.

By default tolerance = 1e-5 for both methods.

/WINT=*w*
Provides an initial unmixing matrix W. If you do not provide this matrix the algorithm initializes using enoise.

The wave *w* must be 2D having the same number type as *srcWave* and having dimensions nCols x nCols, where nCols is the number of columns of *srcWave*. Providing an initial matrix is useful if you have obtained one from a previous set of iterations which may have converged using inadequate tolerance.

/Z
No error reporting.

Details

srcWave is a 2D wave of nRows by nCols. It must be a single or double precision real-valued wave containing no NaNs or INFs. Each column of *srcWave* corresponds to a single data acquisition channel that is assumed to consist of a linear superposition of independent components. This can be expressed as a matrix product

X=A (S^t)

where **S** is an nRows by nCols matrix of independent components, **^t** denotes the transpose, **A** is an nCols by nCols mixing matrix and **X** is the "mixed" input. The ICA operation attempts to find the independent components of **S** from the transformation

S=W X

so that the mutual information between the resulting columns of **S** is minimized. Since mutual information is not affected by a multiplication of components by scalar constants, the resulting independent components can be specified up to a scalar factor.

The operation uses the FastICA algorithm to compute the independent components.

The algorithm has two available methods for computation. The default is to attempt to evaluate the full **W** matrix at once. The second method (/DFLT flag) also known as "deflation" computes one row of **W** at a time. The deflation method might have advantages in cases where there are fewer independent components than there are columns in the input.

Example

```
// Create the source
Make/O/N=(1000,3) ddd
ddd[][0]=sin(2*pi*x/13)
ddd[][1]=sin(2*pi*x/17)
ddd[][2]=sin(2*pi*x/23)

// Create mixing matrix
Make/O/N=(3,3) AA
AA[0][0]= {0.291,0.6557,-0.5439}
AA[0][1]= {0.5572,0.3,-0.2}
AA[0][2]= {-0.1,-0.7,0.4}

// Do the mixing
MatrixOp/O xx=ddd x AA

// Try the ICA
ICA/DFLT/COLS xx
Display M_ICAComponents[][0]
Display M_ICAComponents[][1]
Display M_ICAComponents[][2]
```

References

A. Hyvarinen and E. Oja (2000) Independent Component Analysis: Algorithms and Applications, Neural Networks, (13)411-430.

See Also

PCA

if-elseif-endif

```
if ( <expression1> )
   <TRUE part 1>
elseif ( <expression2> )
   <TRUE part 2>
[…]
[else
   <FALSE part>]
endif
```

In an if-elseif-endif conditional statement, when an expression first evaluates as TRUE (nonzero), then only code corresponding to the TRUE part of that expression is executed, and then the conditional statement is exited. If all expressions evaluate as FALSE (zero) then *FALSE part* is executed when present. After executing code in any TRUE part or the FALSE part, execution will next continue with any code following the if-elseif-endif statement.

See Also

If-Elseif-Endif on page IV-40 for more usage details.

if-endif

```
if ( <expression> )
   <TRUE part>
[else
   <FALSE part>]
endif
```

An if-endif conditional statement evaluates *expression*. If *expression* is TRUE (nonzero) then the code in *TRUE part* is executed, or if FALSE (zero) then the optional *FALSE part* is executed.

See Also

If-Else-Endif on page IV-40 for more usage details.s

IFFT

```
IFFT [flags] srcWave
```

The IFFT operation calculates the Inverse Discrete Fourier Transform of *srcWave* using a multidimensional fast prime factor decomposition algorithm. This operation is the inverse of the **FFT** operation.

Output Wave Name

For compatibility with earlier versions of Igor, if you use IFFT without /ROWS or /COLS, the operation overwrites *srcWave*.

If you use the /ROWS flag, IFFT uses the default output wave name M_RowFFT and if you use the /COLS flag, IFFT uses the default output wave name M_ColFFT.

We recommend that you use the /DEST flag to make the output wave explicit and to prevent overwriting *srcWave*.

Parameters

srcWave is a complex wave. The IFFT of *srcWave* is a either a real or complex wave, according to the length and flags.

Flags

/C	Forces the result of the IFFT to be complex. Normally, the IFFT produces a real result unless certain special conditions are detected as described in **Details**.

/COLS Computes the 1D IFFT of 2D *srcWave* one column at a time, storing the results in the destination wave. You must specify a destination wave using the /DEST flag (no other flags are allowed). See the /ROWS flag and corresponding flags of the **FFT** operation.

/DEST=*destWave* Specifies the output wave created by the IFFT operation.

It is an error to specify the same wave as both *srcWave* and *destWave*.

In a function, IFFT by default creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-72 for details.

/FREE Creates *destWave* as a free wave.

/FREE is allowed only in functions and only if *destWave*, as specified by /DEST, is a simple name or wave reference structure field.

See **Free Waves** on page IV-91 for more discussion.

The /FREE flag was added in Igor Pro 7.00.

/R Forces real output when, due to a power of 2 number of points, IFFT would otherwise automatically produce a complex result.

/ROWS Calculates the IFFT of only the first dimension of 2D *srcWave*. It computes the 1D FFT one row at a time. You must specify a destination wave using the /DEST flag (no other flags are allowed). See the /COLS flag and corresponding flags of the **FFT** operation.

/Z Will not rotate *srcWave* when computing the IDFT of a complex wave whose length is an integral power of 2.

This length indicates that the Inverse DFT result will also be a complex wave. When the result is complex, *and* the x scaling of *srcWave* is such that the first point is *not* x=0, it normally rotates *srcWave* by -N/2 points before performing the IFFT. This inverts the process of performing an FFT on a complex wave. However when /Z is specified, it does not perform this rotation.

Details

The data type of *srcWave* must be complex and must not be an integer type. You should be aware that an IFFT on a number of points that is prime can be slow.

By default, IFFT assumes you are performing an inverse transform on data that was originally real and therefore it produces a real result. However, for historical and compatibility reasons, IFFT detects the special conditions of a one-dimensional wave containing an integral power of 2 data points and automatically creates a complex result.

When the result is complex, the number of points (N) in the resulting wave will be of the same length. Otherwise the resulting wave will be real and of length (N-1)*2.

In either the complex or real case the X units of the output wave are changed to “s”. The X scaling also is changed appropriately, cancelling out the adjustments made by the **FFT** operation. When the data is multidimensional, the same considerations apply to the additional dimensions. The scaling description and IDFT equation below pretend that the IFFT is not performed in-place. After computing the IFFT values, the X scaling of *waveOut* is changed as if Igor had executed these commands:

```
Variable points                  // time-domain points, NtimeDomain
if( waveIn was complex wave )
   points= numpnts(waveIn)
else                             // waveIn was real wave
   points= (numpnts(waveIn) - 1) * 2
endif
Variable deltaT= 1 / (points*deltaX(waveIn))      // 1/(NtimeDomaindx)
SetScale/P waveOut 0,deltaT,"s"
```

The IDFT equation is:

$$waveOut[n] = \frac{1}{N}\sum_{k=0}^{N-1} waveIn[k]\exp\left(\frac{2\pi ikn}{N}\right), \qquad where \quad i=\sqrt{-1}.$$

See Also

The **FFT**, **DSPPeriodogram**, and **MatrixOp** operations.

IgorInfo

`IgorInfo(`*`selector`*`)`

The IgorInfo function returns information about the Igor application and the environment in which it is running.

Details

selector is a number from 0 to 13.

Selector = 0

If *selector* is 0, IgorInfo returns a collection of assorted information. The result string contains five kinds of information. Each group is prefaced by a keyword and a colon, and terminated with a semicolon.

Keyword	Information Following Keyword For IgorInfo(0)
FREEMEM	The amount of free memory available to Igor.
PHYSMEM	The amount of total physical memory available to Igor. Added in Igor Pro 7.00.
USEDPHYSMEM	The amount of used physical memory used by Igor. Added in Igor Pro 7.00.
IGORKIND	The type of Igor application: "pro": Igor Pro 32-bit "pro demo": Igor Pro 32-bit in demo mode "pro64": Igor Pro 64-bit "pro64 demo": Igor Pro 64-bit in demo mode "pro64" and "pro64 demo" are returned by the 64-bit of Igor Pro 7.00 or later. The presence of "demo" indicates that Igor is running in demo mode, either because the user's fully-functional demo period has expired or because the user chose to run in demo mode using the License dialog.
IGORVERS	The version number of the Igor application. Also see IGORFILEVERSION returned by IgorInfo(3).
NSCREENS	Number of screens currently attached to the computer and used for the desktop.
SCREEN1	A description of the characteristics of screen 1. The format of the SCREEN1 description is: SCREEN1:DEPTH=*bitsPerPixel*,RECT=*left*,*top*,*right*,*bottom*; *left*, *top*, *right*, and *bottom* are all in pixels. If there are multiple screens, there will be additional SCREEN keywords, such as SCREEN2 and SCREEN3.

Selector = 1

IgorInfo(1) returns the name of the current Igor experiment.

Use IgorInfo(12) to get the file name including the extension.

Selector = *2*

IgorInfo(2) returns the name of the current platform: “Macintosh” or “Windows”.

Selector = *3*

IgorInfo(3) returns a collection of more detailed information about the operating system, localization information, and the actual file version of the Igor executable. The keywords are OS, OSVERSION, LOCALE, and IGORFILEVERSION.

Keyword	Information Following Keyword For IgorInfo(3)
IGORFILEVERSION	The actual version number of the Igor application file. On Macintosh, the version number is a floating point number with a possible suffix. Igor Pro 7.00, for example, returns “7.00”. Igor Pro 8.01 Beta 1 returns “8.01B01”. On Windows, the version format is a period-separated list of four numbers. Igor Pro 7.02 returns “7.0.2.X” where X is a subminor revision number. A revision to Igor Pro 7.02 would be indicated in the last digit, such as “7.0.2.12”.
LOCALE	Country for which this version of Igor Pro is localized. “US” for most versions, “Japan” for the Japanese versions.
OS	On Macintosh, the OS value is “Macintosh OS X”. On Windows, it is something like “Microsoft Windows 10 Home (21H1)”. The actual build number and format of the text will vary with the operating system and service pack.
OSVERSION	Operating system number. On Macintosh, this is something like “10.13.1”. On Windows, this is something like “10.0.19043.1466”.

Selector = *4*

IgorInfo(4) returns the name of the current processor architecture. Currently this is always “Intel”.

Selector = *5*

IgorInfo(5) returns, as a string, the serial number of the program if it is registered or "_none_" if it isn't registered. Use **str2num** to store the result in a numeric variable. str2num returns NaN if the program isn't registered.

Selector = *6*

IgorInfo(6) returns, as a string, the version of the Qt library under which Igor is running, for example “5.9.4”. This selector value was added in Igor Pro 7.00.

Selector = *7*

IgorInfo(7) returns, as a string, the name of the current user. This selector value was added in Igor Pro 7.00.

Selector = *8*

IgorInfo(8) returns, as a string, the group of the current user on Macintosh. On Windows, an empty string is always returned. This selector value was added in Igor Pro 8.00.

Selector = *9*

IgorInfo(9) returns "admin" if Igor's process is being run as an administrator (*Windows*) or if the user that started Igor's process is an administrator (*Macintosh*). Otherwise it returns "". This selector value was added in Igor Pro 8.00.

Selector = *10*

IgorInfo(10) returns a semicolon-separated list containing the names of activated XOPs. This selector value was added in Igor Pro 8.00.

Selector = *11*

IgorInfo(11) returns a string specifying the type of the current experiment. This will be one of the following:

"Packed"	Current experiment file is a .pxp file
"HDF5 packed"	Current experiment file is a .h5xp file
"Unpacked"	Current experiment file is a .uxp file
""	Current experiment was never saved to a file

See **Saving Experiments** on page II-16 for a discussion of the various experiment file formats.

***Selector* = 12**

IgorInfo(12) returns a string specifying the name of the current experiment file including the extension. If the experiment was never saved to a file, it returns "".

This selector value was added in Igor Pro 9.00.

Use IgorInfo(1) to get the file name without the extension.

***Selector* = 13**

In Igor Pro 9.00 and later, IgorInfo(13) returns a collection of information about the autosave settings. See **Autosave** on page II-36 for background information.

This selector value was added in Igor Pro 9.00.

***Selector* = 14**

IgorInfo(14) returns a collection of information about the HDF5 default compression settings. See **HDF5 Default Compression** on page II-214 for background information.

This selector value was added in Igor Pro 9.00.

The keywords are as follows:

Keyword	Information Following Keyword For IgorInfo(3)
ENABLED	0: Autosave is disabled 1: Autosave is enabled
MODE	1: Direct (saves to original files) 2: Indirect (saves to .autosave files)
INTERVAL	The interval in minutes between autosaves if autosave is enabled.
OPTIONS	A bitwise value with each bit indicating if a given feature is off (bit is 0) or on (bit is 1): Bit 0: Autosave entire experiment Bit 1: Autosave standalone procedure files Bit 2: Autosave standalone plain text notebooks Bit 3: Autosave standalone formatted text notebooks See **Setting Bit Parameters** on page IV-12 for details about bit settings.

See **Autosave** on page II-36 for details.

***Selector* = 15**

IgorInfo(15) returns a string specifying the default experiment file format.

This selector value was added in Igor Pro 9.00.

The returned value will be one of the following:

"Packed"	Default experiment file format is .pxp
"HDF5 packed"	Default experiment file format is .h5xp
"Unpacked"	Default experiment file format is .uxp

See **Saving Experiments** on page II-16 for a discussion of the various experiment file formats.

Selector = 16

IgorInfo(16) returns the total number of waves of all kinds (global, free, local) that currently exist. This selector value was added in Igor Pro 9.00.

IgorInfo(16) can help advanced Igor programmers detect wave leaks in their procedures. For details, see **Detecting Wave Leaks** on page IV-206.

Examples

```
Print NumberByKey("NSCREENS", IgorInfo(0))     // Number of active displays
Function RunningWindows()                      // Returns 0 if Macintosh, 1 if Windows
   String platform = UpperStr(IgorInfo(2))
   Variable pos = strsearch(platform,"WINDOWS",0)
   return pos >= 0
End
```

IgorVersion

`#pragma IgorVersion =` ***`versNum`***

When a procedure file contains the directive, `#pragma IgorVersion=`*versNum*, an error will be generated if *versNum* is greater than the current Igor Pro version number. It prevents procedures that use new features added in later versions from running under older versions of Igor in which these features are missing. However, this version check is limited because it does not work with versions of Igor older than 4.0.

See Also

The **The IgorVersion Pragma** on page IV-54 and #**pragma**.

IgorVersion

The IgorVersion function returns version number of the Igor application as a floating point number. Igor Pro 8.00 returns 8.00, as does Igor Pro 8.00A.

Details

You can use IgorVersion in conditionally compile code expressions, which can be used to omit calls to new Igor features or to provide backwards compatibility code.

```
#if (IgorVersion() >= 8.00)
   [Code that compiles only on Igor Pro 8.00 or later]
#else
   [Code that compiles only on earlier versions of Igor]
#endif
```

If at all possible, it is better to require your users to use a later version of Igor rather than writing conditional code. Attempting this kind of backward-compatibility multiplies your testing requirements and the chances for bugs.

See Also

IgorInfo, **Conditional Compilation** on page IV-108, **The IgorVersion Pragma** on page IV-54

ilim

`ilim`

The ilim function returns the ending loop count for the inner most iterate loop Not to be used in a function. iterate loops are archaic and should not be used.

imag

`imag(`*`z`*`)`

The imag function returns the imaginary component of the complex number *z* as a real (not complex) number.

See Also

The **cmplx**, **conj**, **p2rect**, **r2polar**, and **real** functions.

ImageAnalyzeParticles

`ImageAnalyzeParticles [`*`flags`*`]` ***`keyword imageMatrix`***

The ImageAnalyzeParticles operation performs one of two particle analysis operations on a 2D or 3D source wave *imageMatrix*. The source image wave must be binary, i.e., an unsigned char format where the particles are designated by 0 and the background by 255 (the operation will produce erroneous results if your data uses the opposite designation). Note that all nonzero values in the source image will be considered part of the background. Grayscale images must be thresholded before invoking this operation (you may need to use the /I flag with the **ImageThreshold** operation).

Note: ImageAnalyzeParticles does not take into account wave scaling. All image metrics are in pixels and all pixels are assumed to be square.

Parameters

keyword is one of the following names:

mark Creates a masking image for a single particle, which is specified by an internal (seed) pixel using the /L flag. The masking image is stored in the wave M_ParticleMarker, which is an unsigned char wave. All points in M_ParticleMarker are set to 64 (image operations on binary waves use the value 64 to designate the equivalent of NaN) except points in the particle which are set to the 0. This wave is designed to be used as an overlay on the original image (using the explicit=1 mode of ModifyImage). This keyword is superseded by the **ImageSeedFill** operation.

stats Measures the particles in the image. See **ImageAnalyzeParticles Stats** on page V-365 for details.

Flags

/A=*minArea* Specifies a minimum area as a threshold that must be exceeded for a particle to be counted (e.g., use *minArea*=0 to find single pixel particles). The minimum area is measured in pixels; its default value is *minArea*=5.

When the source wave is 3D, *minArea* specifies the minimum number of voxels that constitute a particle.

/A has no effect when used with the *mark* method.

/B Erases a 1 pixel wide frame inset from the boundary. This insures that no particles will have boundary pixels (see /EBPC below) and all boundary waves will describe close contours.

/CIRC={*minCircularity*,*maxCircularity*}

Use this flag to filter the output so that only particles in the range of the specified circularity are counted.

/D=*dataWave* Specify a wave from which the minimum, maximum, and total particle intensity are sampled when used with the stats keyword. dataWave must be of the same dimensions as the input binary image imageMatrix. It can be of any real numeric type. Results are returned in the waves W_IntMax, W_IntMin, and W_IntAvg.

/E Calculates an ellipse that best fits each particle. The equivalent ellipse is calculated by first finding the moments of the particle (i.e., average x-value, average y-value, average x^2, average y^2, and average x*y), and then requiring that the area of the ellipse be equal to that of the particle. The resulting ellipses are saved in the wave M_Moments. When *imageMatrix* is a 2D wave, the results returned in M_Moments are the columns: the X-center of the ellipse, the Y-center of the ellipse, the major axis, the minor axis, and the angle (radians) that the major axis makes with the X-direction. When *imageMatrix* is a 3D wave, the results in M_Moments include the sum of the X, Y, and Z components as well as all second order permutations of their products. They are arranged in the order: sumX, sumY, sumZ, sumXX, sumYY, sumZZ, sumXY, sumXZ, and sumYZ.

/EBPC Use this flag to exclude from counting any particle that has one or more pixels on any boundary of the image.

/F Fills 2D particles having internal holes and adjusts their area measure for the removal of holes. Internal boundaries around the holes are also eliminated. When the boundary of the particle consists of thin elements that cannot be traversed as a single closed path which passes each boundary pixel only once, the particle will not be filled. Note that filling particles may increase execution time considerably and on some images it may require large amount of memory. It is likely that a more efficient approach would be to preprocess the binary image and remove holes using morphology operations. This flag is not supported when *imageMatrix* is a 3D wave.

/FILL Use /FILL to fill holes inside particles. The reported values of area and perimeter are computed as if there are no holes. The filling algorithm could fail if, for example, there is a closed contour of zeros around the particles.

If you specify both /F and /FILL the operation used /FILL only.

Added in Igor Pro 7.00.

/L= (*row*,*col*) Specifies a 2D particle location in connection with the mark method. (*row*, *col*) is a seed value corresponding to any pixel inside the particle. If the seed belongs to the particle boundary, the particle will not be filled. This flag is not supported when *imageMatrix* is a 3D wave.

/M=*markerVal* Use this flag with the stats mode for 2D images. See **stats** keyword for a full description of the following waves:

markerVal=0:	No marker waves.
markerVal=1:	M_ParticlePerimeter.
markerVal=2:	M_ParticleArea.
markerVal=3:	M_Particle.

This flag does not apply to 3D waves.

/MAXA=*maxArea* Specifies an upper limit of the area of an acceptable particle when used with the stats keyword. The area is measured in pixels and the default value of *maxArea* is the number of pixels in the image. In 3D the maximum value applies to the number of voxels.

/NSW Creates the marker wave (see /M flag) but not the particle statistics waves when used with the stats keyword. This should reduce execution time in images containing many particles.

/P=*plane* Specifies the plane when operating on a single layer of a 3D wave.

/PADB Use this flag with the stats keyword to pad the image with a 1 pixel wide background. This has the effect that particles touching the image boundary are now interior particles with closed perimeter (that extend one pixel beyond the original image frame). In addition, entries in the wave W_ObjPerimeter will be longer for all boundary particles which will also affect other derived parameters such as circularity.

/PADB is different from /B in that it takes into account all pixels belonging to the particle that lie on the boundary of the image. The two flags are mutually exclusive.

/PADB was added in Igor Pro 7.00.

/PDLG Displays a progress dialog.

/PDLG is useful when you are processing very large 3D images. The progress dialog provides feedback and allows the user to abort the operation.

/PDLG was added in Igor Pro 9.00.

/Q Quiet flag, does not report the number of particles to the history area.

/R=*roiWave* Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u) that has the same number of rows and columns as *imageMatrix*. The ROI itself is defined by the entries or pixels in the *roiWave* with value of 0. Pixels outside the ROI may have any nonzero value. The ROI does not have to be contiguous. When *imageMatrix* is a 3D wave, *roiWave* can be either a 2D wave (matching the number of rows and columns in *imageMatrix*) or it can be a 3D wave that must have the same number of rows, columns and layers as *imageMatrix*. When using a 2D *roiWave* with a 3D *imageMatrix* the ROI is understood to be defined by *roiWave* for each layer in the 3D wave.

See **ImageGenerateROIMask** for more information on creating 2D ROI waves.

/U Saves the wave M_ParticleMarker as an 8-bit unsigned instead of the default 16-bit when used with the mark keyword.

/W Creates boundary waves W_BoundaryX, W_BoundaryY, and W_BoundaryIndex for a 2D *imageMatrix* wave. W_BoundaryX and W_BoundaryY contain the pixels along the particle boundaries. The boundary of each particle ends with a NaN entry in both waves. Each entry in W_BoundaryIndex is the index to the start of a new particle in W_BoundaryX and W_BoundaryY, so that you can quickly locate the boundary of each particle.

When there are holes in particles, the entries in W_BoundaryX and W_BoundaryY start with the external boundary followed by all the internal boundaries for that particle. There are no index entries for internal boundaries.

This flag is not supported when *imageMatrix* is a 3D wave.

Details

Particle analysis is accomplished by first converting the data from its original format into a binary representation where the particle is designated by zero and the background by any nonzero value. The algorithm searches for the first pixel or voxel that belongs to a particle and then grows the particle from that seed while keeping count of the area, perimeter and count of pixels or voxels in the particle. If you use additional flags, the algorithm must compute additional quantities for each pixel or voxel belonging to the particle.

If your goal is to mask only the particle, a more efficient approach is to use the **ImageSeedFill** operation, which similarly follows the particle but does not spend processing time on computing unrelated particle properties. ImageSeedFill also has the additional advantage of not requiring that the input wave be binary, which will save time on performing the initial threshold and, in fact, may produce much better results with the adaptive/fuzzy features that are not available in ImageAnalyzeParticles.

ImageAnalyzeParticles Stats

The ImageAnalyzeParticles stats keyword measures the particles in the image. Results of the measurements are reported for all particles whose area exceeds the *minArea* specified by the /A flag. The results of the measurements are:

V_NumParticles Number of particles that exceed the *minArea* limit.

W_ImageObjArea Area (in pixels) for each particle.

W_ImageObjPerimeter Perimeter (in pixels) of each particle. The perimeter calculation involves estimates for 45-degree pixel edges resulting in noninteger values.

W_circularity Ratio of the square of the perimeter to (4*π*objectArea). This value approaches 1 for a perfect circle.

W_rectangularity Ratio of the area of the particle to the area of the inscribing (nonrotated) rectangle. This ratio is π/4 for a perfectly circular object and unity for a nonrotated rectangle.

W_SpotX and W_SpotY Contain a single x, y point from each object. There is one entry per particle and the entries follow the same order as all other waves created by this operation. Each (x,y) point from these waves can used to define the position of a tag or annotation for a particle. Points can also be used as seed pixels for the associated *mark* method or for the ImageSeedFill operation.

W_xmin, W_xmax, W_ymin, W_ymax

Contain a single point for each particle defining an inscribing rectangular box with axes along the X and Y directions.

One of the following waves can be created depending on the /M specification. The waves are designed to be used as an overlay on the original image (using the explicit=1 mode of **ModifyImage**). **Note**: the additional time required to create these waves is negligible compared with the time it takes to generate the stats data.

M_ParticlePerimeter — Masking image of particle boundaries. It is an unsigned char wave that contains 0 values for the object boundaries and 64 for all other points.

M_ParticleArea — Masking image of the area occupied by the particles. It is an unsigned char wave containing 0 values for the object boundaries and 64 for all other points. It is also different from the input image in that particles smaller than the minimum size, specified by /A, are absent.

M_Particle — Image of both the area and the boundary of the particles. It is an unsigned char wave that contains the value 16 for object area, the value 18 for the object boundaries and the value 64 for all other points.

M_rawMoments — Contains five columns. The first column is the raw sum of the x values for each particle, and the second column contains the sum of the y values. To obtain the average or “center” of a particle divide these values by the corresponding area. The third column contains the sum of x^2, the fourth column the sum of y^2, and the fifth column the sum of x*y. The entries of this wave are used in calculating a fit to an ellipse (using the /E flag).

When *imageMatrix* is a 3D wave, the different results are packed into a single 2D wave M_3DParticleInfo, which consists of one row and 11 columns for each particle. Columns are arranged in the following order: minRow, maxRow, minCol, maxCol, minLayer, maxLayer, xSeed, ySeed, zSeed, volume, and area. Use `Edit M_3DParticleInfo.ld` to display the results in a table with dimension labels describing the different columns.

Examples

Convert a grayscale image (blobs) into a proper binary input:

```
ImageThreshold/M=4/Q/I blobs
```

Get the statistics on the thresholded image of blobs and create an image mask output wave for the perimeter of the particles:

```
ImageAnalyzeParticles/M=1 stats M_ImageThresh
```

Display an image of the blobs with a red overlay of the perimeter image:

```
NewImage/F blobs; AppendImage M_ParticlePerimeter
ModifyImage M_ParticlePerimeter explicit=1, eval={0,65000,0,0}
```

See Also

The **ImageThreshold**, **ImageGenerateROIMask**, **ImageSeedFill**, and **ModifyImage** operations. For more usage details see **Particle Analysis** on page III-375.

ImageBlend

ImageBlend [***/A=alpha /W=alphaWave***] ***srcWaveA***, ***srcWaveB*** [, ***destWave***]

The ImageBlend operation takes two RGB images (3D waves) in *srcWaveA* and *srcWaveB* and computes the alpha blending so that

destWave = *srcWaveA* * (1 - *alpha*) + *srcWaveB* * *alpha*

for each color component. If *destWave* is not specified or does not already exist, the result is saved in the current data folder in the wave M_alphaBlend.

The source and destination waves must be of the same data types and the same dimensions. The *alphaWave*, if used, must be a single precision (SP) float wave and it must have the same number of rows and columns as the source waves.

Flags

/A=*alpha*	Specifies a single alpha value for the whole image
/W=*alphaWave*	Single precision wave that specifies an alpha value for each pixel.

See Also

ImageComposite

ImageBoundaryToMask

ImageBoundaryToMask width=*w*, height=*h*, xwave=*xwavename*, ywave=*ywavename* [**, scalingWave=*scalingWaveName*,** [**seedX=*xVal*, seedY=*yVal***]]

The ImageBoundaryToMask operation scan-converts a pair of XY waves into an ROI mask wave.

Parameters

height = *h* Specifies the mask height in pixels.

scalingWave = *scalingWaveName*

2D or 3D wave that provides scaling for the mask. If specified, the scaling of the first two dimensions of scalingWave are copied to M_ROIMask, and both the X and Y waves are assumed to describe pixels in the scaled domain.

seedX = *xVal* Specifies seed pixel location. The operation fills the region defined by the seed and the boundary with the value 1. Background pixels are set to zero. Requires seedY.

seedY = *yVal* Specifies seed pixel location. The operation fills the region defined by the seed and the boundary with the value 1. Background pixels are set to zero. Requires seedX.

width = *w* Specifies the mask width in pixels.

xwave = *xwavename* Name of X wave for mask region.

ywave = *ywavename* Name of Y wave for mask region.

Details

ImageBoundaryToMask generates an unsigned char 2D wave named M_ROIMask, of dimensions specified by width and height. The wave consists of a background pixels that are set to 0 and pixels representing the mask that are set to 1.

The x and y waves can be of any type. However, if the waves describe disjoint regions there must be at least one NaN entry in each wave corresponding to the discontinuity, which requires that you use either single or double precision waves. The values stored in the waves must correspond to zero-based integer pixel values.

If the x and y waves include a vertex that lies outside the mask rectangle, the offending vertex is moved to the boundary before the associated line segment is scan converted.

If you want to obtain a true ROI mask in which closed regions are filled, you can specify the seedX and seedY keywords. The ROI mask is set with zero outside the boundary of the domain and 1 everywhere inside the domain.

Examples

```
Make/O/N=(100,200) src=gnoise(5)           // create a test image
SetScale/P x 500,1,"", src;DelayUpdate      // give it some funny scaling
SetScale/P y 600,1,"", src
Display; AppendImage src
Make/O/N=201 xxx,yyy                  // create boundary waves
xxx=550+25*sin(p*pi/100)              // representing a close ellipse
yyy=700+35*cos(p*pi/100)
AppendToGraph yyy vs xxx
```

Now create a mask from the ellipse and scale it so that it will be appropriate for src:

```
ImageBoundaryToMask ywave=yyy,xwave=xxx,width=100,height=200,scalingwave=src
```

To generate an ROI masked filled with 1 in a region defined by a seed value and the boundary curves:

```
ImageBoundaryToMask
   ywave=yyy,xwave=xxx,width=100,height=200,scalingwave=src,seedx=550,seedy=700
```

See Also

The **ImageAnalyzeParticles** and **ImageSeedFill** operations. For another example see **Converting Boundary to a Mask** on page III-378.

ImageComposite

`ImageComposite [/Z /FREE /DEST=`*`destWave`*`]` ***`srcImageA`*****`,`** ***`srcImageB`***

The ImageComposite operation creates a new image by combining *srcImageA* and *srcImageB* subject to one of 12 Porter-Duff compositing modes.

The ImageComposite operation was added in Igor Pro 8.00.

Flags

/AALP=*aWave* Specifies a 2D single-precision wave as the alpha associated with *srcImageA*. Alpha values are in the range 0 (transparent) to 1 (opaque).

/ACON=*a1* Specifies a single alpha value for the whole *srcImageA*. *a1* is in the range 0 (transparent) to 1 (opaque).

/BALP=*aWave* Specifies a 2D single-precision wave as the alpha associated with *srcImageB*. Alpha values are in the range 0 (transparent) to 1 (opaque).

/BCON=*a1* Specifies a single alpha value for the whole *srcImageB*. *a1* is in the range 0 (transparent) to 1 (opaque).

/DEST=*destWave* Specifies the wave to hold the composite image. If you omit /DEST the operation stores the image in the wave M_ImageComposite in the current data folder.

/FREE Creates output wave as free waves.

/FREE is permitted in user-defined functions only, not from the command line or in macros.

If you use /FREE then destWave must be simple name, not a path.

/NMOD=*mode* Selects one of the 12 Porter-Duff compositing modes. *mode* is a value from 1 to 12. The default is *mode*=4 corresponding to "A over B". See **Compositing Modes** on page V-368.

/OUT=*layers* Specifies the number of layers of the output image. Valid values are 3 (RGB) or 4 (RGBA). By default the operation creates an RGB image.

/PMA=*pmState* Set *pmState* to 1 if the RGB components in *srcImageA* are pre-multiplied. Use *pmState*=0 otherwise. By default *srcImageA* is assumed to be pre-multiplied. See **Pre-multiplication** on page V-368.

/PMB=*pmState* Set *pmState* to 1 if the RGB components in *srcImageB* are pre-multiplied. Use *pmStates*=0 otherwise. By default *srcImageB* is assumed to be pre-multiplied. See **Pre-multiplication** on page V-368.

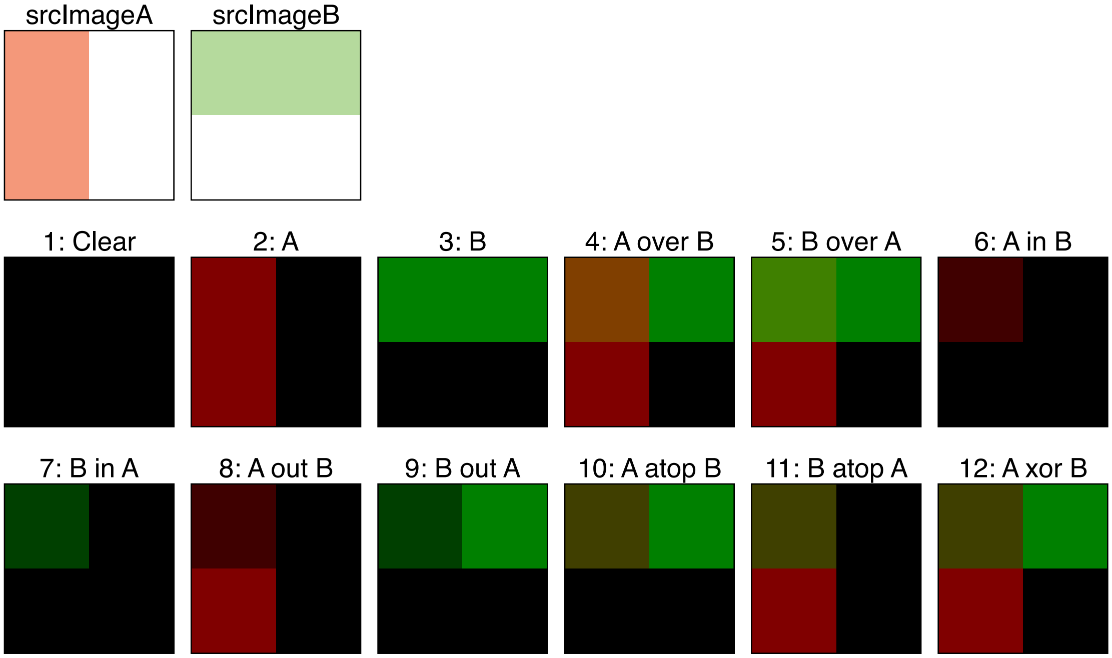
/Z No error reporting. The operation sets V_Flag to 0 if it succeeds or to an error code otherwise. You can use **GetErrMessage** to obtain a description of the error.

Pre-multiplication

An RGB value can be raw or pre-multiplied. "Pre-multiplied" means that the red, green, and blue values have been multiplied by normalized alpha values in the range 0 (transparent) to 1 (opaque). ImageComposite operation is faster when working with pre-multiplicated values. ImageComposite assumes that your RGB values are pre-multiplied unless you specify otherwise using /PMA=0 and /PMB=0.

Compositing Modes

Here are the 12 compositing modes supported by ImageComposite:



Details

ImageComposite computes an output RGB or RGBA image that result from compositing *srcImageA* and *srcImageB* using one of the Porter-Duff compositing modes shown in the table above. The waves *srcImageA* and *srcImageB* must have the same number of pixels and the same number type.

Supported number types are: unsigned char, unsigned short, unsigned int, single precision floating point and double precision floating point. When using integer waves expected alpha values are in the range [0,2^N-1] where N is the number of bits of the number type. Floating point waves should include alpha in the range [0,1].

There are three options to specify the alpha associated with each image:

1. The alpha can be expressed as the 4th layer in the wave.

2. The alpha can be specified by a single-precision wave that has the same number of pixels as the image using the /AALP and /BALP flags.

3. The alpha can be specified by a single number in the range [0,1] using the /ACON and /BCON flags.

Options 2 and 3 cannot override an alpha channel that is present in a source wave. To use these options you must delete the alpha channel in the source wave, if any.

Example

```
Function SetupImageCompositeDemo()          // Setup - Create two sample images
    Make/O/N=(128,128,4)/B/U imageA=0, imageB=0
    imageA[0,64][][0]=255
    imageA[0,64][][3]=128
    NewImage/S=0/N=imageAW imageA
    imageB[][0,64][1]=255
    imageB[][0,64][3]=128
    NewImage/S=0/N=imageBW imageB
    AutoPositionWindow/M=0/R=imageAW imageBW
End

Function CompositeAOverB()                  // Composite A over B
    Wave imageA, imageB
    ImageComposite/PMA=0/PMB=0/DEST=M_Comp1/NMOD=4 imageA,imageB
    NewImage/S=0/N=comp1 M_Comp1
    AutoPositionWindow/M=0/R=imageBW
End

Function CompositeAInB()                    // Composite A in B
    Wave imageA, imageB
    ImageComposite/PMA=0/PMB=0/DEST=M_Comp2/NMOD=6 imageA,imageB
    NewImage/S=0/N=comp2 M_Comp2
```

```
   AutoPositionWindow/M=0/R=comp1
End
Function RemoveAlphaChannel()             // Remove the alpha channel from imageA
   Wave imageA
   Duplicate/R=[][][0,2] imageA, imageA_rgb
   NewImage/S=0 imageA_rgb
End
```

References

T. Porter & T. Duff - Compositing Digital Images. Computer Graphics Volume 18, Number 3, July 1984 pp 253-259.

See Also

ImageBlend

ImageEdgeDetection

`ImageEdgeDetection [`*`flags`*`] `*`Method imageMatrix`*

The ImageEdgeDetection operation performs one of several standard image edge detection operations on the source wave *imageMatrix*.

Unless the /O flag is specified, the resulting image is saved in the wave M_ImageEdges.

The edge detection methods produce binary images on output; the background is set to 0 and the edges to 255. This is due, in most cases to a thresholding performed in the final stage.

Except for the case of marr and shen detectors, you can use the /M flag to specify a method for automatic thresholding; see the **ImageThreshold** /M flag.

Parameters

Method selects type of edge detection. *Method* is one of the following names:

canny — Canny edge detector uses smoothing before edge detection and thresholding. You can optionally specify the threshold using the /T flag and the smoothing factor using /S.

frei — Calculates the Frei-Chen edge operator (see Pratt p. 503) using only the row and column filters.

kirsch — Kirsch edge detector (see Pratt p. 509). Performs convolution with 8 masks calculating gradients.

marr — Marr-Hildreth edge detector. Performs two convolutions with Laplacian of Gaussian and then detects zero crossings. Use the /S flag to define the width of the convolution kernel.

prewitt — Calculates the Prewitt compass gradient filters. Returns the result for the largest filter response.

roberts — Calculates the square root of the magnitude squared of the convolution with the Robert's row and column edge detectors.

shen — Shen-Castan optimized edge detector. Supposed to be effective in the presence of noise. The flags that modify this operation are: /F for the threshold ratio (0.9 by default), /S for smoothness factor (0.9 by default), /W for window width (default is 10), /H for thinning factor which by default is 1.

sobel — Sobel edge detector using convolutions with row and column edge gradient masks (see Pratt p. 501).

Flags

/F=*fraction* — Determines the threshold value for the shen algorithm by starting from the histogram of the image and choosing a threshold such that *fraction* specifies the portion of the image pixels whose values are below the threshold. Valid values are in the interval (0 < *fraction* < 1).

/H=*thinning* — Thins edges when used with shen edge detector. By default the thinning value is 1. Higher values produce thinner edges.

/I — Inverts the output, i.e., sets the edges to 255 and the background to 0.

/M=*threshMethod* See the **ImageThreshold** automatic methods for obtaining a threshold value. Methods 1, 2, 4 and 5 are supported in this operation. If you use *threshMethod* = -1, threshold is not applied.

If you want to apply your own thresholding algorithm, use /M=6 to bypass the thresholding completely. The wave M_RawCanny contains the result regardless of any other flags you may have used.

/N Sets the background level to 64 (i.e., NaN)

/O Overwrites the source image with the output image.

/P=*layer* Applies the operation to the specified layer of a 3D wave.

/P is incompatible with /O.

/P was added in Igor Pro 7.00.

/R=*roiSpec* Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u). The ROI wave must have the same number of rows and columns as the image wave. The ROI itself is defined by entries/pixels whose values are 0. Pixels outside the ROI can be any nonzero value. The ROI does not have to be contiguous and can be any arbitrary shape. See **ImageGenerateROIMask** for more information on creating ROI waves.

In general, the *roiSpec* has the form {*roiWaveName*, *roiFlag*}, where *roiFlag* can take the following values:

roiFlag=0: Set pixels outside the ROI to 0.

roiFlag=1: Set pixels outside the ROI as in original image.

roiFlag=2: Set pixels outside the ROI to NaN (=64).

By default *roiFlag* is set to 1 and it is then possible to use the /R flag using the abbreviated form /R=*roiWave*.

/S= *smoothVal* Specifies the standard deviation or the width of the smoothing filter. By default the operation uses 1. Larger values require longer computation time. In the shen operation the default value is 0.9 and the valid range is (0 < *smoothVal*< 1).

/T=*thresh* Sets a manual threshold for any method above that uses a single threshold. This is faster than using /M.

/W=*width* Specifies window width when used in the shen operation. By default width is set to 10 and it is clipped to 49.

See Also

The **ImageGenerateROIMask** operation for creating ROIs and the **ImageThreshold** operation.

Edge Detectors on page III-365 for a number of examples.

References

Pratt, William K., *Digital Image Processing*, John Wiley, New York, 1991.

ImageFileInfo

`ImageFileInfo [/P=pathName] fileNameStr`

ImageFileInfo is no longer supported and always returns an error.

It is obsolete because it used QuickTime to obtain graphics file information and Apple is phasing out QuickTime.

ImageFilter

`ImageFilter [flags] Method dataMatrix`

The ImageFilter operation is identical to **MatrixFilter**, accepting the same parameters and flags, with the exception of the additional features described below.

Parameters

Method selects the filter type. *Method* is one of the following names:

avg3d	*n*x*n*x*n* average filter for 3D waves.
gauss3d	*n*x*n*x*n* gaussian filter for 3D waves.
hybridmedian	Implements ranking pixel values between two groups of pixels in a 5x5 neighborhood. The first group includes horizontal and vertical lines through the center, the second group includes diagonal lines through the center, and both groups include the center pixel itself. The resulting median value is the ranked median of both groups and the center pixel.
max3d	*n*x*n*x*n* maximum rank filter for 3D waves.
median3d	*n*x*n*x*n* median filter for 3D waves where *n* must be of the form 3^r (integer *r*), e.g., 3x3x3, 9x9x9 etc. The filter does not change the value of the voxel it is centered on if any of the filter voxels lies outside the domain of the data.
min3d	*n*x*n*x*n* minimum rank filter for 3D waves.
point3d	*n*x*n*x*n* point finding filter using normalized (n^3-1)*`center-outer` for 3D waves.

Flags

/N=*n*	Specifies the filter size. By default *n* =3. In most situations it will be useful to set *n* to an odd number in order to preserve the symmetry in the filters.
/O	Overwrites the source image with the output image. Used only with the hybridmedian filter, which does not automatically overwrite the source wave.

Details

You can operate on 3D waves using the 3D filters listed above. These filters are extensions of the 2D filters available under MatrixFilter. The avg3d, gauss3d, and point3d filters are implemented by a 3D convolution that uses an averaging compensation at the edges.

This operation does not support complex waves.

See Also

MatrixFilter for descriptions of the other available parameters and flags.

MatrixConvolve for information about convolving your own 3D kernels.

References

Russ, J., *Image Processing Handbook*, CRC Press, 1998.

ImageFocus

`ImageFocus [`*`flags`*`]` ***`stackWave`***

The ImageFocus operation creates in focus image(s) from a stack of images that contain in and out of focus regions. It computes the variance in a small neighborhood around each pixel and then takes the pixel value from the plane in which the highest variance is found.

Flags

/ED=*edepth*	Sets the effective depth in planes. For example, an effective depth of one means that it computes the best focus for each plane using a stack of three planes, which includes the current plane and any one adjacent plane above and below it. Does not affect the default method (/METH=0).
/METH=*method*	Specifies the calculation method. *method*=0: Computes a single plane output for the stack (default). *method*=1: Computes the best image for each plane using /ED.
/Q	Quiet mode; no output to history area.
/Z	No error reporting.

See Also

Chapter III-11, **Image Processing** contains links to and descriptions of other image operations.

ImageFromXYZ

`ImageFromXYZ [`*`flags`*`]` ***`xyzWave, dataMatrix, countMatrix`***

`ImageFromXYZ [`*`flags`*`] {`*`xWave,yWave,zWave`*`},` ***`dataMatrix, countMatrix`***

ImageFromXYZ converts XYZ data to matrix form. You might use it, for example, to convert a "sparse matrix" to an actual matrix for easier display and processing.

You provide the input data in the XYZ triplet *xyzWave* or in 1D waves *xwave*, *ywave*, and *zwave*.

dataMatrix and *countMatrix* receive output data but you must create them prior to calling ImageFromXYZ.

For each XY location in the input data, ImageFromXYZ adds the corresponding Z value to an element of *dataMatrix*. The element is determined based on the input XY location and the X and Y scaling of *dataMatrix*.

For each XY location in the input data, ImageFromXYZ increments the corresponding element of *countMatrix*. This permits you to obtain an average Z value if multiple input values fall into a given element of *dataMatrix*.

Parameters

xyzWave is a triplet wave containing the input XYZ data.

xWave, *yWave* and *zWave* are 1D input waves containing XYZ data.

You specify either *xyzWave* by itself or *xWave*, *yWave* and *zWave* in braces.

dataMatrix is a 2D wave to which the Z values are added. It must be either single-precision or double-precision floating point. The X and Y scaling of *dataMatrix* determines how input values are mapped to output matrix elements.

countMatrix is a 2D wave the elements of which store the number of Z values added to each corresponding element of *dataMatrix*. ImageFromXYZ sets it to 32-bit integer if it is not already so.

Flags

/AS	If /AS (autoscale) is specified, ImageFromXYZ clears both *dataMatrix* and *countMatrix* and sets the X and Y scaling of *dataMatrix* based on the range of X and Y input values.

Details

For each point in the XYZ input data, ImageFromXYZ adds the Z value to the appropriate element of *dataMatrix* and increments the corresponding element of *countMatrix*. Normally you will clear *dataMatrix* and *countMatrix* before calling it.

You can combine multiple XYZ datasets in one matrix by calling ImageFromXYZ multiple times with different input data and the same *dataMatrix* and *countMatrix*. In this case you would clear *dataMatrix* and *countMatrix* before the first call to ImageFromXYZ only.

What you do with the output is up to you but one technique is to divide *dataMatrix* by *countMatrix* to get the average and then use **MatrixFilter** NanZapMedian to eliminate any NaN values that result from zero divided by zero.

Example

```
Make /N=1000 /O wx=enoise(2), wy= enoise(2), wz= exp(-(wx^2+wy^2))
Make /O /N=(100,100) dataMat=0
SetScale x,-2,2,dataMat
SetScale y,-2,2,dataMat
Duplicate /O dataMat,countMat
ImageFromXYZ /AS {wx,wy,wz}, dataMat, countMat

// Execute these one at a time
NewImage dataMat
dataMat /= countMat                  // Replace cumulative z value with average
MatrixFilter NanZapMedian, dataMat   // Apply median filter, zapping NaNs
```

See Also

SetScale, **Image X and Y Coordinates** on page II-388.

ImageGenerateROIMask

ImageGenerateROIMask [**/W=*winName*/E=*e*/I=*i***] ***imageInstance***

The ImageGenerateROIMask operation creates a Region Of Interest (ROI) mask for use with other ImageXXX commands. It assumes the top (or /W specified) graph contains an image and that the user has drawn shapes using Igor's drawing tools in a specific manner.

ImageGenerateROIMask creates an unsigned byte mask matrix with the same x and y dimensions and scaling as the specified image. The mask is initially filled with zeros. Then the drawing layer, progFront, in the graph is scanned for suitable fillable draw objects. The area inside each shape is filled with ones unless the fill mode for the shape is set to erase in which case the area is filled with zeros.

Flags

/E=*e*	Changes value used for the exterior from the default zero values to *e*.
/I=*i*	Changes value used for the interior from the default one values to *i*.
/W=*winName*	Looks for the named graph window or subwindow containing appropriate image masks drawn by the user. If /W is omitted, ImageGenerateROIMask uses the top graph window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

To generate an ROI wave for use with most image processing operations you need to set the values of interior pixels to zero and exterior pixels to one using /E=1/I=0.

Suitable objects are those that can be filled (rectangles, ovals, etc.) and which are plotted in axis coordinate mode specified using the same axes by which the specified image instance is displayed. Objects plotted in plot relative mode are also used, However, this is not recommended because it will give correct results only if the image exactly fills the plot rectangle. If you use axis coordinate mode then you can zoom in or out as desired and the resulting mask will still be correct.

Note that the shapes can have their fill mode set to none. This still results in a fill of ones. This is to allow the drawn ROI to be visible on the graph without obscuring the image. However cutouts (fills with erase mode) will obscure the image.

Note also that nonfill drawing objects are ignored. You can use this fact to create callouts and other annotations.

In a future version of Igor, we may create a new drawing layer in graphs dedicated to ROIs.

The mask generated is named M_ROIMask and is generated in the current data folder.

Variable V_flag is set to 1 if the top graph contained draw objects in the correct layer and 0 if not. If 0 then the M_ROIMask wave was not generated.

Examples

```
Make/O/N=(200,400) jack=x*y; NewImage jack; ShowTools
SetDrawLayer ProgFront
SetDrawEnv linefgc=(65535,65535,0),fillpat=0,xcoord=top,ycoord=left,save
DrawRect 63.5,79.5,140.5,191.5
DrawRRect 61.5,206.5,141.5,280.5
SetDrawEnv fillpat= -1
DrawOval 80.5,169.5,126.5,226.5
ImageGenerateROIMask jack
NewImage M_ROIMask
AutoPositionWindow/E
```

See Also

For another example see **Generating ROI Masks** on page III-378.

ImageGLCM

ImageGLCM [*flags*] *srcWave*

The ImageGLCM operation calculates the gray-level co-occurrence matrix for an 8-bit grayscale image and optionally evaluates Haralick's texture parameters.

The ImageGLCM operation was added in Igor Pro 7.00.

Flags

/D=*distance* Sets the offset in pixels for which the co-occurrence matrix is calculated. The default value is 1.

/DEST=*destGLCM* Specifies the wave to hold the co-occurrence matrix. If you omit /DEST the operation stores the matrix in the wave M_GLCM in the current data folder.

/DETP=*destParamWave*

Specifies the wave to hold the computed texture parameters. If you omit /DETP the operation stores the texture parameters in the wave W_TextureParams in the current data folder.

If the destination wave already exists it is overwritten. Note that you must specify the /HTFP flag to compute the texture parameters.

/E=*structureBits* *structureBits* is a bitwise setting that lets you control the combination of co-occurrences that you want to compute.

Consider a wave displayed in a table and a pixel at position x

0	3	5
1	*x*	6
2	4	7

The *structureBits* corresponding to co-occurrence between x and any direction is simply 2^direction. By default the operation computes all combinations. This is equivalent to *structureBits*=255.

Note that the *structureBits* only define directions. The combination of the distance (/D) and the *structureBits* define the full co-occurrence calculation.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/FREE Creates output waves as free waves.

/FREE is permitted in user-defined functions only, not from the command line or in macros.

If you use /FREE then *destGLCM* and *destParamWave* must be simple names, not paths.

See **Free Waves** on page IV-91 for details on free waves.

/HTFP Computes Haralick's texture parameters. See the discussion in the Details section below for more information about the texture parameters.

/P=plane If the image consists of more than one plane you can use this flag to determine which plane in srcWave is analysed. By default it is plane zero.

/Z No error reporting.

Details

ImageGLCM computes the co-occurrence matrix for the image in *srcWave* and optionally evaluates Haralick's texture parameters. The operation supports 8-bit grayscale images and generates a 256x256 single-precision floating point co-occurrence matrix.

The elements of the matrix P[i][j] are defined as the normalized number of pixels that have a spatial relationship defined by the distance (/D) and the structure (/E) such that the first pixel has gray-level i and the second pixel has gray-level j. The matrix is normalized so that the sum of all its elements is 1.

If you specify the /HTFP flag the operation computes the 13 Haralick texture parameters and stores them sequentially in the destination wave (see /DETP). The wave is saved with dimension labels defining each element. The expressions for the texture parameters are:

$$f_1 = \sum_i \sum_j \left(p[i][j] \right)^2 ,$$

$$f_2 = \sum_{n=0}^{254} n^2 \left\{ \sum_{i=0}^{255} \sum_{\substack{j=0 \\ |i-j|=n}}^{255} p[i][j] \right\},$$

$$f_3 = \sum_i \sum_j \frac{(i-\mu_x)(j-\mu_y)p[i][j]}{\sigma_x \sigma_y},$$

$$f_4 = \sum_i \sum_j (i-\mu)^2 p[i][j],$$

$$f_5 = \sum_i \sum_j \frac{1}{1+(i-j)^2} p[i][j],$$

$$f_6 = \sum_i i p_{x+y}(i),$$

$$f_7 = \sum_i (i - f_6)^2 p_{x+y}(i),$$

$$f_8 = -\sum_i p_{x+y}(i) \log\left(p_{x+y}(i)\right),$$

$$f_{10} = Variance\left(p_{x-y}\right),$$

$$f_9 = -\sum_i \sum_j p[i][j] \log\left(p[i][j]\right),$$

$$f_{10} = Variance\left(p_{x-y}\right),$$

$$f_{11} = \sum_i p_{x-y}(i) \log\left(p_{x-y}(i)\right),$$

$$f_{12} = \frac{f_9 - HXY1}{\max\left(HX, HY\right)},$$

$$f_{13} = \sqrt{1 - \exp\left(-2\left(HXY2 - f9\right)\right)}.$$

Here

$$p_x(i) = \sum_j p[i][j], \quad p_y(j) = \sum_i p[i][j],$$

$$\mu_x = \sum_i i p_x(i), \quad \mu_y = \sum_i i p_y(i), \quad \mu = (\mu_x + \mu_y)/2.$$

$$\sigma_x = \sqrt{\sum_i \left(1 - \mu_x\right)^2 p_x(i)}, \qquad \sigma_y = \sqrt{\sum_i \left(1 - \mu_y\right)^2 p_y(i)},$$

$$p_{x+y}(k) = \sum_i \sum_{\substack{j \\ i+j=k}} p[i][j],$$

$$p_{x-y}(k) = \sum_i \sum_{\substack{j \\ |i-j|=k}} p[i][j],$$

$$HXY1 = -\sum_i \sum_j p[i][j] \log\left(p_x(i) p_y(j)\right),$$

$$HXY2 = -\sum_i \sum_j p_x(i) p_y(j) \log\left(p_x(i) p_y(j)\right),$$

$$HX = -\sum_i p_x(i) \log\left(p_x(i)\right), \quad HY = -\sum_i p_y(i) \log\left(p_y(i)\right).$$

There are at least two versions of f7 used in the literature and in software. We know of at least three versions of f14 so ImageGLCM does not compute it.

References

R.M. Haralick, K. Shanmugam and Itshak Dinstein, "Textural Features for Image Classification", IEEE Transactions on Systems, Man, and Cybernetics, 1973.

ImageHistModification

ImageHistModification [***flags***] ***imageMatrix***

The ImageHistModification operation performs a modification of the image histogram and saves the results in the wave M_ImageHistEq. If /W is not specified, the operation is a simple histogram equalization of *imageMatrix*. If /W is specified, the operation attempts to produce an image with a histogram close to *waveName*. If /A is specified, the operation performs an adaptive histogram equalization. *imageMatrix* is a wave of any noncomplex numeric type. Adaptive histogram equalization applies only to 2D waves and the other parts apply to both 2D and 3D waves.

Flags

/A — Performs an adaptive histogram equalization by subdividing the image into a minimum of 4 rectangular domains and using interpolation to account for the boundaries between adjacent domains. When the /C flag is specified with contrast factor greater than 1, this operation amounts to contrast-limited adaptive histogram equalization. By default the operation divides the image into 8 horizontal and 8 vertical regions. See /H and /V.

/B=*bins* — Specifies the number of *bins* used with the /A flag. If not specified, this value defaults to 256.

/C=*cFactor* — Specifies a contrast factor (or clipping value) above which pixels are equally distributed over the whole range. *cFactor* must be greater than 1, in the limit as *cFactor* approaches 1 the operation is a regular adaptive histogram equalization. **Note**: this flag is used only with the /A flag.

/H=*hRegions* — Specifies the number of horizontal subdivisions to be used with the /A feature. Note, the number of image pixels in the horizontal direction must be an integer multiple of *hRegions*.

/I — Extends the standard histogram equalization by using 2^{16} bins instead of 2^8 when calculating histogram equalization. This feature does not apply to the adaptive histogram equalization (/A flag).

/O — Overwrites the source image. If this flag is not specified, the resulting image is saved in the wave M_ImageHistEq.

/R=*roiSpec* Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u). The ROI wave must have the same number of rows and columns as *imageMatrix*. The ROI itself is defined by the entries whose values are 0. Regions outside the ROI can take any nonzero value. The ROI does not have to be contiguous and can take any arbitrary shape.

In general, the *roiSpec* has the form {*roiWaveName*, *roiFlag*}, where *roiFlag* can take the following values:

roiFlag=0:	Set pixels outside the ROI to 0.
roiFlag=1:	Set pixels outside the ROI as in original image (default).
roiFlag=2:	Set pixels outside the ROI to NaN (=64).

By default *roiFlag* is set to 1 and it is then possible to use the /R flag with the abbreviated form /R=*roiWave*. When *imageMatrix* is a 3D wave, *roiWave* can be either a 2D wave (matching the number of rows and columns in *imageMatrix*) or it can be a 3D wave which must have the same number of rows, columns, and layers as *imageMatrix*. When using a 2D *roiWave* with a 3D *imageMatrix*, the ROI is understood to be defined by *roiWave* for each layer in the 3D wave.

See **ImageGenerateROIMask** for more information on creating ROI waves.

/V=*vRegions* Specifies the number of vertical subdivisions to be used with the /A flag. The number of image pixels in the horizontal direction must be an integer multiple of *vRegions*. If the image dimensions are not divisible by the number of regions that you want, you can pad the image using ImageTransform **padImage**.

/W=*waveName* Specifies a 256-point wave that provides the desired histogram. The operation will attempt to produce an image having approximately the desired histogram values. This flag does not apply to the adaptive histogram equalization (/A flag)

See Also

The **ImageGenerateROIMask** and **ImageTransform** operations for creating ROIs. For examples see **Histograms** on page III-372 and **Adaptive Histogram Equalization** on page III-354.

ImageHistogram

`ImageHistogram [`*`flags`*`]` ***`imageMatrix`***

The ImageHistogram operation calculates the histogram of *imageMatrix*. The results are saved in the wave W_ImageHist. If *imageMatrix* is an RGB image stored as a 3D wave, the resulting histograms for each color plane are saved in W_ImageHistR, W_ImageHistG, W_ImageHistB.

imageMatrix must be a real-valued numeric wave.

Flags

/I Calculates a histogram with 65536 bins evenly distributed between the minimum and maximum data values. The operation first finds the extrema and then calculates the bins and the resulting histogram. Data can be a 2D wave of any type including float or double.

/P=*plane* Restricts the calculation of the histogram to a specific plane when *imageMatrix* is a non RGB 3D wave.

/R=*roiWave* Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u) that has the same number of rows and columns as *imageMatrix*. The ROI itself is defined by the entries o pixels in the *roiWave* with value of 0. Pixels outside the ROI may have any nonzero value. The ROI does not have to be contiguous. When *imageMatrix* is a 3D wave, *roiWave* can be either a 2D wave (matching the number of rows and columns in *imageMatrix*) or it can be a 3D wave that must have the same number of rows, columns and layers as *imageMatrix*. When using a 2D *roiWave* with a 3D *imageMatrix* the ROI is understood to be defined by *roiWave* for each layer in the 3D wave.

See **ImageGenerateROIMask** for more information on creating 2D ROI waves.

/S Computes the histogram for a whole 3D wave possibly subject to 2D or 3D ROI masking. The /S and /P flags are mutually exclusive.

Details

The ImageHistogram operation works on images, but it handles both 2D and 3D waves of any data type. Unless you use one of the special features of this operation (e.g., ROI or /P or /I) you could alternatively use the **Histogram** operation, which computes the histogram for the full wave and includes additional options for controlling the number of bins.

If the data type of *imageMatrix* is single byte, the histogram will have 256 bins from 0 to 255. Otherwise, the 256 bins will be distributed between the minimum and maximum values encountered in the data. Use the /I flag to increase the number of bins to 65536, which may be useful for unsigned short (/W/U) data.

See Also

ImageHistModification, **ImageGenerateROIMask**, **JointHistogram**, **Histograms** on page III-372

ImageInfo

`ImageInfo(`*`graphNameStr`*`, `*`imageWaveNameStr`*`, `*`instanceNumber`*`)`

The ImageInfo function returns a string containing a semicolon-separated list of information about the specified image in the named graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

imageWaveNameStr contains either the name of a wave displayed as an image in the named graph, or an image instance name (wave name with “#n” appended to distinguish the nth image of the wave in the graph). You might get an image instance name from the **ImageNameList** function.

If *imageWaveNameStr* contains a wave name, *instanceNumber* identifies which instance you want information about. *instanceNumber* is usually 0 because there is normally only one instance of a wave displayed as an image in a graph. Set *instanceNumber* to 1 for information about the second image of the wave, etc. If *imageWaveNameStr* is `""`, then information is returned on the *instanceNumber*th image in the graph.

If *imageWaveNameStr* contains an instance name, and *instanceNumber* is zero, the instance is taken from *imageWaveNameStr*. If *instanceNumber* is greater than zero, the wave name is extracted from *imageWaveNameStr*, and information is returned concerning the *instanceNumber*th instance of the wave.

Details

The string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon for ease of use with **StringByKey**. The keywords are as follows:

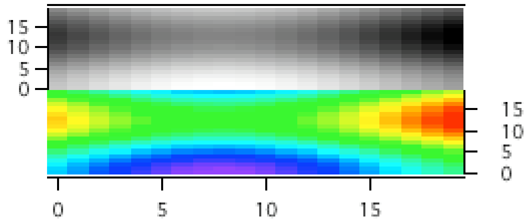
Keyword	Information Following Keyword
AXISFLAGS	Flags used to specify the axes. Usually blank because /L and /B (left and bottom axes) are the defaults.
COLORMODE	A number indicating how the image colors are derived: 1: Color table (see **Image Color Tables** on page II-392). 2: Scaled color index wave (see **Indexed Color Details** on page II-400). 3: Point-scaled color index (See **Example: Point-Scaled Color Index Wave** on page II-401). 4: Direct color (see **Direct Color Details** on page II-401). 5: Explicit Mode (see **ModifyImage** explicit keyword). 6: Color table wave (see **Color Table Waves** on page II-399).
RECREATION	Semicolon-separated list of *keyword=modifyParameters* commands for the ModifyImage command.
XAXIS	X axis name.
XWAVE	X wave name if any, else blank.
XWAVEDF	The full path to the data folder containing the X wave or blank if there is no X wave.
YAXIS	Y axis name.
YWAVE	Y wave name if any, else blank.
YWAVEDF	The full path to the data folder containing the Y wave or blank if there is no Y wave.
ZWAVE	Name of wave containing Z data used to calculate the image plot.
ZWAVEDF	The full path to the data folder containing the Z data wave.

The format of the RECREATION information is designed so that you can extract a keyword command from the keyword and colon up to the "`;`", prepend "`ModifyImage `", replace the "`x`" with the name of a image plot ("`data#1`" for instance) and then **Execute** the resultant string as a command.

Example 1

This example gets the image information for the second image plot of the wave "jack" (which has an instance number of 1) and applies its **ModifyImage** settings to the first image plot.

```
#include <Graph Utility Procs>, version>=6.1   // For WMGetRECREATIONFromInfo

// Make two image plots of the same data on different left and right axes
Make/O/N=(20,20) jack=sin(x/5)+cos(y/4)
Display;AppendImage jack                       // bottom and left axes
AppendImage/R jack                             // bottom and right axes

// Put image plot jack#0 above jack#1
ModifyGraph axisEnab(left)={0.5,1},axisEnab(right)={0,0.5}

// Set jack#1 to use the Rainbow color table instead of the default Grays
ModifyImage jack#1 ctab={*,*,Rainbow,0}
```



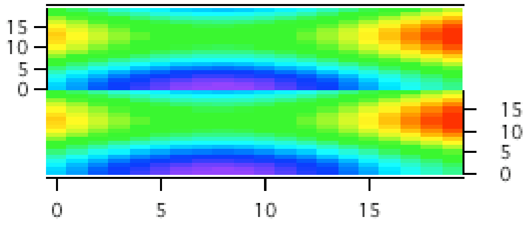
Now we peek at some of the image information for the second image plot of the wave "jack" (which has an instance number of 1) displayed in the top graph:

```
Print ImageInfo("","jack",1)[69,148]        // Just the interesting stuff
```

```
;ZWAVE:jack;ZWAVEDF:root:;COLORMODE:1;RECREATION:ctab= {*,*,Rainbow,0};plane= 0;
```

```
// Apply the color table, etc from jack#1 to jack:
String info= WMGetRECREATIONFromInfo(ImageInfo("","jack",1))
info= RemoveEnding(info)                        // Remove trailing semicolon
```

```
// Use comma instead of semicolon separators
String text = ReplaceString(";", info, ",")
Execute "ModifyImage jack " + text
```



Example 2

This example gets the full path to the wave containing the Z data from which the first image plot in the top graph was calculated.

```
String info= ImageInfo("","",0)      // 0 is index of first image plot
String pathToZ= StringByKey("ZWAVEDF",info)+StringByKey("ZWAVE",info)
Print pathToZ
   root:jack
```

See Also

The **ModifyImage**, **AppendImage**, **NewImage** and **Execute** operations.

Image Plots on page II-385.

Image Instance Names on page II-403.

ImageInterpolate

```
ImageInterpolate [flags] Method srcWave
```

The ImageInterpolate operation interpolates the source *srcWave* and stores the results in the wave M_InterpolatedImage in the current data folder unless you specify a different destination wave using the /DEST flag.

Parameters

Method selects type of interpolation. *Method* is one of the following names:

Affine2D — Performs an affine transformation on *srcWave* using parameters specified by the /APRM flag. The transformation applies to a general combination of rotation, scaling, and translation represented by a 3x3 matrix

$$M = \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & w \end{bmatrix}.$$

The upper 2x2 matrix is a composite of rotation and scaling, *tx* and *ty* are composite translations and *w* is usually 1. It computes the dimensions of the output wave and then uses the inverse transformation and bilinear interpolation to compute the value of each output pixel. When an output pixel does not map back into the source domain it is set to the user-specified background value. It supports 2D and 3D input waves. If *srcWave* is a 3D wave it applies on a layer by layer basis.

The output is stored in the wave M_Affine in the current data folder.

Bilinear — Performs a bilinear interpolation subject to the specified flag. You can use either the /F or /S flag, but not both.

Kriging — Uses Kriging to generate an interpolated matrix from a sparse data set. Kriging calculates interpolated values for a rectangular domain specified by the /S flag. The Kriging parameters are specified via the /K flag.

Kriging is computed globally for a single user-selected variogram model. If there are significant spatial variances within the domain occupied by the data, you should consider subdividing the domain along natural boundaries and use a single variogram model in each subdivision.

If there are N data points, the algorithm first computes the NxN matrix containing the distances between the data and then inverts an associated matrix of similar size to compute the result for the selected variogram model. Because inversion of an NxN matrix can be computationally expensive, you should consider restricting the calculation to regions that are similar to the range implied by the variogram. Such an approach can also be justified in the sense that the local interpolation should not be affected by a remote datum.

Note: Kriging does not support data containing NaNs or INFs. Wave scaling has no effect.

Pixelate — Creates a lower resolution (pixelated) version of *srcWave* by averaging the pixels inside domain specified by /PXSZ flag. The results are saved in the wave M_PixelatedImage in the current data folder.

The computed wave has the same numeric type as *srcWave* so the averaging may be inaccurate in the case of integer waves.

When *srcWave* is a 3D wave you have the option of averaging only over data in each layer using /PXSZ={nx,ny} or averaging over data in a rectangular cube using /PXSZ={nx,ny,nz}.

When not averaging over layers, the number of rows and columns of the new image are obtained by integer division of the original number by the respective size of the averaging rectangle and adding one more pixel for any remainder.

When averaging over layers the resulting rows and columns are obtained by truncated integer division ignoring remainders (if any).

See also **Multidimensional Decimation** on page II-98.

Resample Computes a new image based on the selected interpolation function and transformation parameters. Set the interpolation function with the /FUNC flag. Use the /TRNS flag to specify transformation parameters for grayscale images, or /TRNR, /TRNG, and /TRNB for the red, green, and blue components, respectively, of RGB images. M_InterpolatedImage contains the output image in the current data folder.

There are currently two transformation functions: the first magnifies an image and the second applies a radial polynomial sampling. The radial polynomial affects pixels based on their position relative to the image center. A linear polynomial reproduces the same image. Any nonlinear terms contribute to distortion (or correction thereof).

Spline Computes a 2D spline interpolation for 2D matrix data. The degree of the spline is specified by the /D flag.

Voronoi Generates an interpolated matrix from a sparse data set (*srcWave* must be a triplet wave) using Voronoi polygons. It calculates interpolated values for a rectangular domain as specified by the /S or /RESL flags.

It first computes the Delaunay triangulation of X, Y locations in the Z=0 plane (assuming that X, Y positions occupy a convex domain in the plane). It then uses the Voronoi dual to interpolate the Z values for X and Y pairs from the grid defined by the /S flag. The computed grid may exceed the bounds of the convex domain defined by the triangulation. Interpolated values for points outside the convex domain are set to NaN or the value specified by the /E flag.

Use the /I flag to iterate to finer triangulation by subdividing the original triangles into smaller domains. Each iteration increases computation time by approximately a factor of two, but improves the smoothness of the interpolation.

If you have multiple sets of data in which X,Y locations are unchanged, you can use the /STW flag to store one triangulation and then use the /PTW flag to apply the precomputed triangulation to a new interpolation. To use this option you should use the Voronoi keyword first with a triplet wave for *srcWave* and set xn = x0 and yn = y0. The operation creates the wave W_TriangulationData that you use in the next triangulation with a 1D wave as *srcWave*.

Voronoi interpolation is similar to what can be accomplished with the **ContourZ** function except that it does not require an existing contour plot, it computes the whole output matrix in one call, and it has the option of controlling the subdivision iterations.

See **Voronoi Interpolation Example** below.

XYWaves Performs bilinear interpolation on a matrix scaled using two X and Y 1D waves (specified by /W). The interpolation range is defined by /S. The data domain is defined between the centers of the first and last pixels (X in this example):

```
xmin=(xWave[0]+xWave[1])/2
xmax=(xWave[last]+xWave[last-1])/2
```

Values outside the domain of the data are set to NaN. The interpolation is contained in the M_InterpolatedImage wave, which is single precision floating point or double precision if *srcWave* is double precision.

Warp Performs image warping interpolation using a two step algorithm with three optional interpolation methods. The operation warps the image based on the relative positions of source and destination grids. The warped image has the same size as the source image. The source and destination grids are each specified by a pair of 2D X and Y waves where the rows and columns correspond to the relative location of the source grid. The smallest supported dimensions of grid waves are 2x2. All grid waves must be double-precision floating point and must have the same number of points corresponding to pixel positions within the image. Grid waves must not contain NaNs or INFs. Wave scaling is ignored.

Flags

/APRM={*r11*,*r12*,*tx*,*r21*,*r22*,*ty*,*w*,*background*}

Sets elements of the affine transformation matrix and the background value.

/ATOL

Allows ImageInterpolate to use a tolerance value if the result of the interpolation at any point is NaN. The algorithm returns the first non-NaN value it finds by adding or subtracting 1/10000th of the size of the X or Y interpolation step. At worst this algorithm is 5 times slower than the default algorithm if the interpolation is performed in a region which is completely outside the convex source domain.

/ATOL was added in Igor Pro 7.00.

/CMSH

Use this flag in Voronoi interpolation to create a triangle mesh surface representing the input data. After triangulating the X, Y locations (in a plane z=const), the mesh is generated from a sequence of XYZ vertices of all the resulting triangles. The output is stored in the wave M_ScatterMesh in the current data folder. It is in the form of a triplet wave where every consecutive 3 rows represent a disjoint triangle.

If the input contains a NaN or INF in any column, the corresponding row is excluded from the triangulation.

If you want to generate this mesh without generating the interpolated matrix you can omit the /S flag.

/CMSH was added in Igor Pro 7.00.

/CSAF=*factor*

In rare situations that usually involve spatial degeneracy, the Voronoi interpolation algorithm may need additional memory. You can use the /CSAF flag with an integer factor greater than 1 to increase the memory allocation.

/CSAF was added in Igor Pro 9.00.

/D=*splineDeg*

Specifies the spline degree with the Spline method. The default spline degree is 2. Supported values are 2, 3, 4, and 5.

/DEST=*destWave*

Specifies the wave to contain the output of the operation. If the specified wave already exists, it is overwritten.

Creates a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-72 for details.

/E=*outerValue*

Assigns *outerValue* to all points outside the convex domain of the Delaunay triangulation. By default *outerValue* = NaN.

/F={*fx*,*fy*}

Calculates a bilinear interpolation of all the source data. Here *fx* is the sampling factor for the X-direction and *fy* is the sampling factor in the Y-direction. The output number of points in a dimension is factor*(number of data intervals) +1. The number of data intervals is one less than the number of points in that dimension.

For example, if *srcWave* is a 2x2 matrix (you have a single data interval in each direction) and you use /F={2,2}, then the output wave is a 3x3 matrix (i.e., 2x2 intervals) which is a factor of 2 of the input. Sampling factors can be noninteger values.

/FDS

Performs spline interpolation to a "local" cubic spline that depends only on the 2D neighboring data. The interpolation goes exactly through the original data and provides continuity of the function and its first derivative. The second derivative is set to vanish on the boundary. The implemented algorithm was provided by Francis Dalaudier. /FDS was added in Igor Pro 7.00.

/FEQS

When performing Voronoi interpolation, forces the algorithm to use equal scaling for both axes. Normally the scaling for each axis is determined from its range. When the ranges of the two axes are different by an order of magnitude or more, it is helpful to force the larger range to be used for scaling both axes.

/FUNC=*funcName* Specifies the interpolation function. *funcName* can be:

nn Nearest neighbor interpolation uses the value of the nearest neighbor without interpolation. This is the fastest function.

bilinear Bilinear interpolation uses the immediately surrounding pixels and computes a linear interpolation in each dimension. This is the second fastest function.

cubic Cubic polynomial (photoshop-like) uses a 4x4 neighborhood value to compute the sampled pixel value.

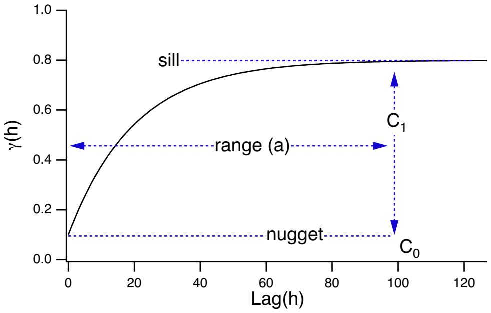
spline Spline smoothed sampled value uses a 4x4 neighborhood around the pixel.

sinc Slowest function using a 16x16 neighborhood.

/I=*iterations* Specifies the number of times the original triangulation is subdivided with the Voronoi interpolation method. By default the Voronoi interpolation computes the original triangulation without subdivision.

/K={*model*, *nugget*, *sill*, *range*}

Specifies the variogram parameters for kriging using standard notation, models are expressed in terms of the nugget value C_0, sill value C_0+C_1, and range a.



model Selects the variogram model. Values and models are:

1: Spherical. $\gamma(h) = C_0 + C_1 \cdot (3h/2a - 0.5 \cdot h^3/a^3)$

2: Exponential. $\gamma(h) = C_0 + C_1 \cdot (1 - \exp[-3 \cdot h/a])$

3: Gaussian. $\gamma(h) = C_0 + C_1 \cdot (1 - \exp[-3 \cdot (h/a)^2])$

nugget Specifies the lowest value in the variogram.

sill Specifies the maximum (plateau) value in the variogram range the characteristic length of the different variogram models.

Wave scaling has no effect on kriging calculations.

/PFTL=*tol* In Voronoi interpolation, controls the rectangular neighborhood about each datum position XY where the algorithm returns the original Z-value. The tolerance value 0 <= *tol* < 1 is tested separately in X and Y (i.e., it is not a geometric distance) when both X and Y are normalized to the range [0,1]. By default *tol*=1e-15.

Added in Igor Pro 7.00.

/PTW=*tWave* Uses a previous triangulation wave with Voronoi interpolation. *tWave* will be the wave saved by the /STW flag. You can't use a triangulation wave that was computed and saved on a different computer platform.

See **PTW Flag Example** on page V-388.

/PXSZ={*nx*,*ny*} Specifies the size in pixels of the averaging rectangle used by the Pixelate operation. nx is the number of rows and ny is the number of columns that are averaged to yield a single output pixel. If srcWave is a 3D wave the resulting wave has the same number of layers as srcWave with pixelation computed on a layer-by-layer basis.

/PXSZ={*nx*,*ny*,*nz*} Specifies the size in pixels of the averaging rectangle used by the Pixelate operation. nx is the number of rows and ny is the number of columns and nz is the number of layers that are averaged to yield a single output pixel.

This form of the /PXSZ flag was added in Igor Pro 7.00.

/RESL={*nx*, *ny*} Specifies resampling the full input image to an output image having *nx* rows by *ny* columns.

/S={*x0*,*dx*,*xn*,*y0*,*dy*,*yn*}

Calculates a bilinear interpolation of a subset of the source data. Here *x0* is the starting point in the X-direction, *dx* is the sampling increment, *xn* is the end point in the X-direction and the corresponding values for the Y-direction. If you set *x0* equal to *xn* the operation will compute the triangulation but not the interpolation.

/SPRT Skips the XY perturbation step. The perturbation step is designed to break degeneracies that originate when the XY data are sampled on a rectangular grid. If your data are not sampled on a rectangular grid you can skip the perturbation and get better accuracy in reproducing the Z-values at the sampled locations. See also **ModifyContour** with the keyword Perturbation.

Added in Igor Pro 7.00.

/STW Saves the triangulation information in the wave W_TriangulationData in the current data folder. W_TriangulationData can only be used on the computer platform where it was created.

/SV Saves the Voronoi interpolation in the 2D wave M_VoronoiEdges, which contains sequential edges of the Voronoi polygons. Edges are separated from each other by a row of NaNs. The outer most polygons share one or more edges with a large triangle containing the convex domain.

/TRNS={*transformFunc*,*p1*,*p2*,*p3*,*p4*}

Determines the mapping between a pixel in the destination image and the source pixel. *transformFunc* can be:

scaleShift Sets image scaling which could be anamorphic if the X and Y scaling are different.

radialPoly Corrects both color as well as barrel and pincushion distortion. In `radialPoly` the mapping from a destination pixel to a source pixel is a polynomial in the pixel's radius relative to the center of the image.

A source pixel, *sr,* satisfies the equation:

$$sr = ar + br^2 + cr^3 + dr^4,$$

where *r* is the radius of a destination pixel having an origin at the center of the destination image.

The corresponding parameters are:

transformFunc	*p1*	*p2*	*p3*	*p4*
scaleShift	xOffset	xScale	yOffset	yScale
radialPoly	a	b	c	d

/U=*uniformScale* Calculates a bilinear interpolation of all the source data as with the /F flag but with two exceptions: A single uniform scale factor applies in both dimensions, and the scale factor applies to the number of points — not the intervals of the data.

/W={*xWave*, *yWave*} Provides the scaling waves for XYWaves interpolation. Both waves must be monotonic and must have one more point than the corresponding dimension in *srcWave*. The waves contain values corresponding to the edges of data points in *srcWave*, so that the X value at the first data point is equal to `(xWave[0]+xWave[1])/2`.

Flags for Warp

/dgrx=*wave* Sets the wave containing the destination grid X data.

/dgry=*wave* Sets the wave containing the destination grid Y data.

/sgrx=*wave* Sets the wave containing the source grid X data.

/sgry=*wave* Sets the wave containing the source grid Y data.

/WM=*im* Sets the interpolation method for warping an image.

im=1: Fast selection of original data values.

im=2: Linear interpolation.

im=3: Smoothing interpolation (slow)

Details

When computing Bilinear or Spline interpolation *srcWave* can be a 2D or a 3D wave. When *srcWave* is a 3D wave the interpolation is computed on a layer by layer basis and the result is stored in a corresponding 3D wave. When the interpolation method is Kriging or Voronoi, *srcWave* is a 2D triplet wave (3-column wave) where each row specifies the X, Y, Z values of a datum. *srcWave* can be of any real data type. Results are stored in the wave M_InterpolatedImage. If *srcWave* is double precision so is M_InterpolatedImage; otherwise M_InterpolatedImage is a single precision wave.

Voronoi Interpolation Example

```
Function DemoVoronoiInterpolation()
    Make/O/N=(100,3) sampleTriplet
    sampleTriplet[][0]=enoise(5)
    sampleTriplet[][1]=enoise(5)
    sampleTriplet[][2]=sqrt(sampleTriplet[p][0]^2+sampleTriplet[p][1]^2)

    // Interpolate the data to a rectangular grid of 50x50 pixels
    ImageInterpolate/RESL={50,50}/DEST=firstImage voronoi sampleTriplet

    // Triangulate the XY locations and save the triangulation wave
    ImageInterpolate/STW voronoi sampleTriplet

    // Use the previous triangulation on the Z column of the sample
    MatrixOp/O zData=col(sampleTriplet,2)
    Wave W_TriangulationData
    ImageInterpolate/PTW=W_TriangulationData/RESL={50,50}/DEST=secondImage voronoi zData
End
```

PTW Flag Example

```
Function DemoPTW()
    // Create some random data
    Make/O/N=(100,3) eee = enoise(5)

    // Compute triangulation wave
    ImageInterpolate/RESL={1,1}/STW voronoi eee
    Wave W_TriangulationData

    // Copy the Z-column to a 1D wave
    MatrixOp/O e3 = col(eee,2)

    // Use the previous triangulation with 1D wave
    ImageInterpolate/PTW=W_TriangulationData /RESL={100,100} voronoi e3
    Wave M_InterpolatedImage
    Duplicate/O M_InterpolatedImage, oneD

    // Direct computation for comparison
    ImageInterpolate/RESL={100,100} voronoi eee
    // Display
    NewImage M_InterpolatedImage
    NewImage oneD
End
```

See Also

The **interp**, **Interp3DPath**, **ImageRegistration**, and **Loess** operations. The **ContourZ** function. For examples see **Interpolation and Sampling** on page III-359.

References

Unser, M., A. Aldroubi, and M. Eden, B-Spline Signal Processing: Part I-Theory, *IEEE Transactions on Signal Processing*, *41*, 821-832, 1993.

Douglas B. Smythe, “A Two-Pass Mesh Warping Algorithm for Object Transformation and Image Interpolation” ILM Technical Memo #1030, Computer Graphics Department, Lucasfilm Ltd. 1990.

ImageLineProfile

`ImageLineProfile [`*`flags`*`] xWave=`*`xwave`*`, yWave=`*`ywave`*`, srcWave=`*`srcWave`*` [, width=`*`value`*`, widthWave=`*`wWave`*`]`

The ImageLineProfile operation provides sampling of a source image along an arbitrary path specified by the two waves: *xWave* and *yWave*. The arbitrary path is made of line segments between every two consecutive vertices of *xWave* and *yWave*. In each segment the profile is calculated at a number of points (profile points) equivalent to the sampling density of the original image (unless the /V flag is used). Both *xWave* and *yWave* should have the same scaling as *srcWave*. If *srcWave* does not have the same scaling in both dimensions you should remove the scaling to compute an accurate profile.

At each profile point, the profile value is calculated by averaging samples along the normal to the profile line segment. The number of samples in the average is determined by the keyword width. The operation actually averages the interpolated values at N equidistant points on the normal to profile line segment, with `N=2(width+0.5)`. Samples outside the domain of the source image do not contribute to the profile value.

The profile values are stored in the wave W_ImageLineProfile. The actual locations of the profile points are stored in the waves W_LineProfileX and W_LineProfileY. The scaled distance measured along the path is stored in the wave W_LineProfileDisplacement.

When the averaging width is greater than zero, the operation can also calculate at each profile point the standard deviation of the values sampled for that point (see /S flag). The results are then stored in the wave W_LineProfileStdv. When using this operation on 3D RGB images, the profile values are stored in the 3 column waves M_ImageLineProfile and M_LineProfileStdv respectively.

Parameters

srcWave=*srcWave* Specifies the image for which the line profile is evaluated. The image may be a 2D wave of any type or a 3D wave or RGB data.

xWave=*xwave* Specifies the wave containing the x coordinate of the line segments along the path.

yWave=*ywave* Specifies the wave containing the y coordinate of the line segments along the path.

width=*value* Specifies the width (diameter) in pixels (need not be an integer value) in a direction perpendicular to the path over which the data is interpolated and averaged for each path point. If you do not specify width or use width=0, only the interpolated value at the path point is used.

widthWave=*wWave* Specifies the width of the profile (see definition above) on a segment by segment basis. *wWave* should be a 1D wave that has the same number of entries as xWave and yWave. If you provide a widthWave any value assigned with the width keyword is ignored. All values in the wave must be positive and finite.

Flags

/IRAD=*nRadIntervals*

/IRAD was added in Igor Pro 9.00.

Use /IRAD to estimate the integrated intensity for an annular domain defined by the /RAD flag and the width parameter. For example, to integrate the intensity in the annular domain centered around *Xc*=50, *Yc*=50 for the radial range [24,25]:

```
Make/O/N=(100,100) ddd=sqrt((x-50)^2+(y-50)^2)
ImageLineProfile/RAD={50,50,24.5,.5,.001}/IRAD=100 srcWave=ddd
Print V_integral
```

/P=*plane* Specifies which plane (layer) of a 3D wave is to be profiled. By default *plane* =-1 and the profiles are of either the single layer of a 2D wave or all three layers of a 3D RGB wave. Use *plane* =-2 if you want to profile all layers of a 3D wave.

/RAD={*Xc*, *Yc*, *RADc*, *radWidth* [, *deltaAngle*]}

/RAD was added in Igor Pro 9.00.

Use /RAD to compute a circular profile that is centered at (*Xc*,*Yc*) with a radius *RADc*. *Xc*, *Yc*, and *RADc* are expressed in terms of the scaled coordinates.

radWidth is in units of image pixels.

deltaAngle is the angle increment between samples in radians. If you omit it, the operation first computes the maximum radius (if width>0) and then computes the increment angle such that there are 5 (linearly interpolated) samples per path pixel. If your image data is relatively smooth you could reduce this sampling by specifying a large *deltaAngle*.

Here is an example using /RAD:

```
Make/O/N=(100,100) ddd=x*y    // Default scaling
ImagelineProfile/RAD={50,50,24.5,0} srcWave=ddd
Display W_ImageLineProfile
```

/S Calculates standard deviations for each profile point.

/SC Saves W_LineProfileX and W_LineProfileY using the X and Y scaling of *srcWave*.

/V Calculate profile points only at the vertices of xWave and yWave.

Examples

```
Make/N=(50, 50) sampleData
sampleData = sin((x-25) / 10) * cos((y-25) / 10)
NewImage sampleData
Make/n=2 xTrace={0,50} ,yTrace={20,20}
ImageLineProfile srcWave=sampleData, xWave=xTrace, yWave=yTrace
AppendtoGraph/T yTrace vs xTrace
Display W_ImageLineProfile
```

See Also

For additional examples see **ImageLineProfile Operation** on page III-372.

ImageLoad

ImageLoad [***flags***] [***fileNameStr***]

The ImageLoad operation loads an image file into an Igor wave. It can load PNG, JPEG, BMP, TIFF, and Sun Raster Files.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where pathName is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

If you want to force a dialog to select the file, omit the *fileNameStr* parameter or pass “” for it.

Flags

/AINF

Loads all of the image files in a disk folder into the current data folder. For example, if you have created an Igor symbolic path named ImagePath that points to a folder containing image files, you can execute:

```
ImageLoad/P=ImagePath/T=TIFF/AINF
```

When using /AINF, you must omit *fileNameStr* and you must include /T to specify the type of image file to be loaded.

This flag requires Igor Pro 7.03 or later.

/BIGT=*mode*

When mode is 1, ImageLoad uses the LibTIFF library to load TIFF files. This is the default if you omit /BIGT. The LibTIFF library supports the traditional TIFF file format and the Big TIFF file format, which supports file sizes greater than 4 GB and files containing compressed data.

When mode is 0, ImageLoad uses Igor’s internal TIFF code to load image data. This internal code does not support Big TIFF and is limited to file sizes less than 2 GB.

If you omit /BIGT, ImageLoad first attempts to load the file using LibTIFF. If an error occurs, it automatically attempts to load the file using Igor’s internal TIFF code.

The /SCNL, /STRP and /TILE flags require using LibTIFF. If you use any of these flags, /BIGT=1 is automatically in effect.

The /RAT and /RTIO flags require using Igor’s internal TIFF code. If you use these flags, /BIGT=0 is automatically in effect.

See ***Loading TIFF File***s below for more information about supported data types.

/C=*count*

Specifies the number of images to load from a TIFF stack containing multiple images. The images are stored in individual waves if /LR3D is omitted or in a single 3D wave if /LR3D is present.

By default, it loads only a single image (i.e., /C=1). Use /C=-1 to load all images. Images must be either 8 bits, 16 bits, or 32 bits/pixel for this option.

To load a subset of the images in a TIFF stack, use /S to specify the starting image.

If you specify a *count* that exceeds the number of images in the file, ImageLoad loads all images beginning with the first image or the image specified by /S.

/G

Displays the loaded image in a new image plot window.

/LR3D

Specifies that the images in a TIFF stack are to be loaded into a 3D wave rather than into multiple 2D waves. This option works with grayscale images only, not with full color (e.g., RGB).

To load a subset of the images into the 3D wave, also use /S and /C.

/LTMD

Reads data stored in TIFF tags belonging to the main Image File Directory. /LTMD works only when you use /BIGT=1 and is ignored otherwise. It was added in Igor Pro 8.00.

/LTMD creates a data folder named "Tag*n*" for each loaded image. The name of the data folder has the numeric suffix *n* starting from zero.

The "Tag*n*" data folder contains a text wave named T_Tags where each row contains the metadata associated with a single tag. The order of the rows in the wave T_Tags is indeterminate.

If you need to parse the metadata, you can search for the tag descriptor which always appears at the start of the line and is followed by a colon and one space (": ").

/N=*baseName* Stores the waves using *baseName* as the wave name. Only when *baseName* conflicts with an existing wave name will a numeric suffix be appended to the new wave names.

If you omit /N, ImageLoad uses the name of the file as the base name.

/O Overwrites an existing wave with the same name.

If you omit /O and there is an existing wave with the same name, a numeric suffix is appended to the image name to create a unique name.

/P=*pathName* Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/Q Quiet mode. Suppresses printing a description of the loaded data to the history area.

/RAT Read All Tags reads all of the tags in a TIFF file into one or more waves.

If you use /RAT, /BIGT=0 is automatically in effect. To load tags with /BIGT=1, use /LTMD instead of /RAT.

/RAT creates a data folder named "Tag*n*" with a numeric suffix, *n*, starting from zero for each loaded image. When reading multiple images from a stack TIFF file, /RAT creates a corresponding number of data folders.

Each data folder contains a text wave named T_Tags consisting of 5 columns. The first row contains the offset of the current Image File Directory (IFD) from the start of the file. The remaining rows describe the individual TIFF Tags as they appear in the IFD.

The first column contains the tag number, the second contains the tag description, the third contains the tag type, the fourth contains the tag length, and the fifth contains either the value of the tag or a statement identifying the name of the wave in which the data was stored. For example, a simple tag that contains a single value has the form:

Num	Desc	Type	Length	Value
256	IMAGEWIDTH	4	1	2560

A tag that contains more data, such as an array of values has the form:

Num	Desc	Type	Length	Value
273	STRIPOFFSETS	4	-120	tifTag273

Here the Length field is negative (-1*realLength) and the Value field contains the name of the wave tifTag273 which contains the array of strip offsets.

When the Value field consists of ASCII characters it is stored in the T_Tags wave itself. All other types are stored in a wave in the same Tag data folder.

Private tags are usually designated by negative tag numbers. If their data type is anything other than ASCII, they are saved in separate waves.

In Igor Pro 9.01 and later, /RAT sets the S_dataFolder output variable to the path to the data folder where the tag information is stored.

/RONI Stores the number of images in a TIFF stack file in the variable V_numImages. No images are loaded from the file. This file is not compatible with /BIGT=0. /RONI was added in Igor Pro 9.00.

/RTIO Reads tag information only from a TIFF file. /RTIO is similar to /RAT but it loads tag information only without loading any images.

If you use /RTIO, /BIGT=0 is automatically in effect. To load tags with /BIGT=1, use /LTMD instead of /RAT.

If you are loading a stack of images you can use the /C and /S flags to obtain tags from a specific range of images.

/S=*start* Specifies the first image to load from a TIFF stack containing multiple images.

start is zero-based and defaults to 0.

Use /C to specify the number of images to load.

/SCNL=*num* Reads the specified scanline from a TIFF file using LibTiff.

Added in Igor Pro 7.00.

/STRP=*num* Reads the specified strip from a TIFF file using LibTiff.

Added in Igor Pro 7.00.

/T=*type* Identifies what kind of image file to load. *type* is one of the following image file formats:

type	**Loads this Image Format**
any	Any graphic file type
bmp	Windows bitmap file
jpeg	JPEG file
png	PNG file
rpng	Raw PNG file (see **Details**)
sunraster	Sun Raster file
tiff	TIFF file (see also **Loading TIFF Files**).

If you omit /T or specifiy /T=any, Igor makes a guess based on the file name extension. ImageLoad reports an error if it is unable to determine the image file type.

/T=any allows the user to choose any file, regardless of its file name extension, if ImageLoad displays an Open File dialog.

When loading TIFF, we recommend that you use /T=tiff. See **Loading TIFF Files** below for details.

/TILE=*num* Reads the specified tile from a TIFF file using LibTiff.

Added in Igor Pro 7.00.

/Z No error reporting.

Details

The name of the wave created by ImageLoad is based on the file name or on *baseName* if you provide the /N=*baseName* flag. In either case, if and only if there is a name conflict, ImageLoad appends a number to create a unique wave name.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like “hd:Folder1:” or “C:\\Folder1\\”. See **Symbolic Paths** on page II-22 for details.

Output Variables

ImageLoad sets the following variables:

V_flag Set to 1 if the image was successfully loaded or to 0 otherwise.

S_fileName Set to the name of the file that was loaded.

S_path Set to the file system path to the folder containing the file. S_path uses Macintosh path syntax (e.g., "hd:FolderA:FolderB:"), even on Windows. It includes a trailing colon.

V_numImages Set to the number of images loaded. Applies to TIFF files only.

Also set by /RONI flag.

S_info When using /BIGT=1, S_info contains the text stored in the IMAGEDESCRIPTION (270) TIFF tag. See /RAT and /LTMD above for other tag data.

S_dataFolder Set by the /RAT flag to the path to the data folder where the tag information is stored. Added in Igor Pro 9.01.

S_waveNames Set to a semicolon-separated list of the names of loaded waves.

Loading PNG Files

If you use /T=rpng ("raw PNG") or if you omit /T and the file as a .png extension, ImageLoad interprets the PNG file as raw data.

We recommend that you use /T=rpng and use /T=png only if /T=rpng does not produce the desired results.

/T=rpng creates an 8-bit or 16-bit unsigned integer wave with 1 to 4 layers.

PNG images with physical units produce waves with X and Y units of meters.

If a PNG image has a color table, ImageLoad creates two waves: a main image wave with one layer and a color table wave of the same name but with an "_pal" suffix. If the name is too long it creates a wave named PNG_pal instead.

Loading TIFF Files

ImageLoad/BIGT=0 supports 1-bit, 8-bit, 16-bit, 24-bit, and 32-bit TIFF files as well as floating point TIFFs.

1-bit/pixel images are loaded into a unsigned byte waves

8-bit/pixel images are loaded into a unsigned byte waves

16-bit/pixel images are loaded into unsigned 16-bit waves

24-bit/pixel images and 32-bit/pixel images loaded into 3D RGB and RGBA waves respectively

ImageLoad/BIGT=1 supports the following data formats:

8-bit/sample signed or unsigned

12-bits/sample (packed into 16-bit unsigned)

16-bit/sample signed or unsigned

32-bit/sample IEEE single precision floating point, signed integer or unsigned integer

64-bit/sample IEEE double precision floating point, signed integer or unsigned integer

Loading a TIFF File With a Color Table

If your TIFF file includes a color table, ImageLoad/T=tiff/BIGT=0 loads the data into a 2D wave and loads the color table into a separate color table wave which can be used when creating an image plot.

If you want to load the TIFF file into a 3D RGB wave, use /T=tiff to load it into a 2D wave plus a color table and then use **ImageTransform** cmap2RGB to create the 3D RGB wave.

Loading TIFF Stacks

A TIFF stack is a TIFF file that contains multiple images. When loading a stack, you can:

- Load all images
- Load a range of images specified by /S (starting image) and /C (image count)

You can also load the images into:

- Separate 2D waves by omitting the /LR3D flag
- A single 3D wave by using the /LR3D flag

When you use /LR3D, ImageLoad stores each image from the TIFF file in a layer of the 3D output wave. This option works with grayscale images only, not with full color (e.g., RGB).

EXIF Metadata

Some applications embed metadata (information about the image) in EXIF format. In both JPEG and TIFF files, the metadata is stored using TIFF tags. To read the metadata, use the /RAT flag, even if you are loading a JPEG file.

Examples

```
// Load all images from a TIFF stack into separate 2D waves
ImageLoad /C=-1 /T=TIFF

// Load a single image from a TIFF stack into a 2D wave
ImageLoad/S=10/C=1/T=TIFF      // Load image 10 (zero based)

// Load all images from a TIFF stack into a single 3D wave
ImageLoad/LR3D/S=0/C=-1/T=TIFF

// Read all tags without loading any images
ImageLoad/C=-1/T=TIFF/RTIO

// Get the number of images in a TIFF stack
NewDataFolder/O/S tmp
ImageLoad/C=-1/T=TIFF/RTIO
Print V_numImages
KillDataFolder :
```

See Also

Loading Image Files on page II-157.

The **ImageSave** operation for saving waves as image files.

ImageMorphology

ImageMorphology [***flags***] ***Method imageMatrix***

The ImageMorphology operation performs one of several standard image morphology operations on the source *imageMatrix*. Unless the /O flag is specified, the resulting image is saved in the wave M_ImageMorph. The operation applies only to waves of type unsigned byte. All ImageMorphology methods except for watershed use a structure element. The structure element may be one of the built-in elements (see /E flag) or a user specified element.

Erosion, Dilation, Opening, and Closing are the only methods supported for a 3D *imageMatrix*.

Parameters

Method is one of the following names:

BinaryErosion	Erodes the source binary image using a built-in or user specified structure element (see /E and /S flags).
BinaryDilation	Dilates the source binary image using a built-in or user specified structure element (see /E and /S flags).
Closing	Performs the closing operation (dilation followed by erosion). The same structure element is used in both erosion and dilation. Note that this operation is an idempotent, which means that there is no point of executing it more than once.
Dilation	Performs a dilation of the source grayscale image using either a built-in structure element or a user specified structure element. The operation supports only 8-bit gray images.
Erosion	Erodes the source grayscale image using either a built-in structure element or a user specified structure element. The operation supports only 8-bit gray images.

Opening — Performs an opening operation (erosion followed by dilation). The same structure element is used in both erosion and dilation. Note that this operation is an idempotent which means that there is no point of executing it more than once.

TopHat — Calculates the difference between the eroded image and dilated image using the same structure element.

Watershed — Calculates the watershed regions for grayscale or binary image. Use the /N flag to mark all nonwatershed lines as NaNs. The /L flag switches from using 4 neighboring pixels (default) to 8 neighboring pixels.

Flags

/E=*id* — Uses a particular built in structure element. The following are the built-in structure element. The following are the built-in structure elements; make sure to use the appropriate id for the dimensionality of *imageMatrix*:

id	Element	Origin	Shape
1	2x2	(0,0)	square (default)
2	1x3	(1,1)	row (in 3x3 square)
3	3x1	(1,1)	column (in 3x3 square)
4	3x3	(1,1)	cross (in 3x3 square)
5	5x5	(2,2)	circle (in 5x5 square)
6	3x3	(1,1)	full 3x3 square
200	2x2x2	(1,1,1)	symmetric cube
202	2x2x2	(1,1,1)	2 voxel column in Y direction
203	2x2x2	(1,1,1)	2 voxel column in X direction
204	2x2x2	(1,1,1)	2 voxel column in Z direction
205	2x2x2	(1,1,1)	XY plane
206	2x2x2	(1,1,1)	YZ plane
207	2x2x2	(1,1,1)	XZ plane
300	3x3x3	(1,1,1)	symmetric cube
301	3x3x3	(1,1,1)	symmetric ball
302	3x3x3	(1,1,1)	3 voxel column in Y direction
303	3x3x3	(1,1,1)	3 voxel column in X direction
304	3x3x3	(1,1,1)	3 voxel column in Z direction
305	3x3x3	(1,1,1)	XY plane
306	3x3x3	(1,1,1)	YZ plane
307	3x3x3	(1,1,1)	XY plane
500	5x5x5	(2,2,2)	symmetric cube
501	5x5x5	(2,2,2)	symmetric ball
700	7x7x7	(3,3,3)	symmetric cube
701	7x7x7	(3,3,3)	symmetric ball

Note that this flag has no effect on watershed calculations.

/I= *iterations* — Repeats the operation the specified number of *iterations*.

/L — Uses 8-connected neighbors instead of 4.

/N Sets the background level to 64 (= NaN).

/O Overwrites the source wave with the output.

/R=*roiSpec* Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u). The ROI wave must have the same number of rows and columns as the image wave. The ROI itself is defined by the entries/pixels whose values are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous and can take any arbitrary shape. See **ImageGenerateROIMask** for more information on creating ROI waves.

In general, the *roiSpec* has the form {*roiWaveName*, *roiFlag*}, where *roiFlag* can take the following values:

roiFlag=0: Set pixels outside the ROI to 0.

roiFlag=1: Set pixels outside the ROI as in original image.

roiFlag=2: Set pixels outside the ROI to NaN (=64).

By default *roiFlag* is set to 1 and it is then possible to use the /R flag using the abbreviated form /R=*roiWave*.

/S= *seWave* Specifies your own structure element.

seWave must be of type unsigned byte with pixels that belong to the structure element set to 1 and background pixels set to 0.

There are no limitations on the size of the structure element and you can use the /X and /Y flags to specify the origin of your structure element.

/W= *whiteVal* Sets the white value in the binary image if it is different than 255. The black level is assumed to be zero.

/X= *xOrigin* Specifies the X-origin of a user-defined structure element starting at 0. If you do not use this flag Igor sets the origin to the center of the specified structure element.

/Y= *yOrigin* Specifies the Y-origin of a user defined structure element starting at 0. If you do not use this flag Igor sets the origin to the center of the specified structure element.

/Z= *zOrigin* Specifies the Z-origin of the element for 3D structure elements. If you do not use this flag Igor sets the origin to the center of the specified structure element.

Examples

If you would like to apply a morphological operation to a wave whose data type is not an unsigned byte and you wish to retain the wave's dynamic range, you can use the following approach:

```
Function ScaledErosion(inWave)
   Wave inWave

   WaveStats/Q inWave
   Variable nor=255/(V_max-V_min)
   MatrixOp/O tmp=nor*(inWave-V_min)
   Redimension/B/U tmp
   ImageMorphology/E=5 Erosion tmp
   Wave M_ImageMorph
   MatrixOp/O inWave=(M_ImageMorph/nor)+V_min
   KillWaves/Z tmp,M_ImageMorph
End
```

See Also

The **ImageGenerateROIMask** operation for creating ROIs. For details and usage examples see **Morphological Operations** on page III-368 and **Particle Analysis** on page III-375.

ImageNameList

ImageNameList(*graphNameStr*, *separatorStr*)

The ImageNameList function returns a string containing a list of image names in the graph window or subwindow identified by *graphNameStr*.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

separatorStr should contain a single ASCII character such as "," or ";" to separate the names.

An image name is defined as the name of the 2D wave that defines the image with an optional #ddd suffix that distinguishes between two or more images that have the same wave name. Since the image name has to be parsed, it is quoted if necessary.

Examples

The following command lines create a very unlikely image display. If you did this, you would want to put each image on different axes, and arrange the axes such that they don't overlap. That would greatly complicate the example.

```
Make/O/N=(20,20) jack,'jack # 2';
Display;AppendImage jack
AppendImage/T/R jack
AppendImage 'jack # 2'
AppendImage/T/R 'jack # 2'
Print ImageNameList("",";")
```

prints jack;jack#1;'jack # 2';'jack # 2'#1;

See Also

Another command related to images and waves: **ImageNameToWaveRef**.

For commands referencing other waves in a graph: **TraceNameList**, **WaveRefIndexed**, **XWaveRefFromTrace**, **TraceNameToWaveRef**, **CsrWaveRef**, **CsrXWaveRef**, **ContourNameList**, and **ContourNameToWaveRef**.

ImageNameToWaveRef

ImageNameToWaveRef(*graphNameStr*, *imageNameStr*)

The ImageNameToWaveRef function returns a wave reference to the 2D wave corresponding to the given image name in the graph window or subwindow named by *graphNameStr*.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

The image is identified by the string in *imageNameStr*, which could be a string determined using **ImageNameList**. Note that the same image name can refer to different waves in different graphs.

See Also

The **ImageNameList** function.

For a discussion of wave references, see **Wave Reference Functions** on page IV-197.

ImageRegistration

ImageRegistration [*flags*]**[testMask=*testMaskWave*] [refMask=*refMaskWave*]**
testWave=*imageWave1*, refWave=*imageWave2*

The ImageRegistration operation adjusts the test image wave, testWave, to match the reference image wave, refWave, possibly subject to auxiliary mask waves. The registration may involve offset, rotation, scaling or skewing.

Image data may be in two or three dimensions.

ImageRegistration is designed to find accurate registration for relatively small variation on the order of a few degrees of rotation and a few pixels offset between the reference and test images.

All the input waves are expected to be single precision float (SP) so you may have to redimension your images before using ImageRegistration.

ImageRegistration does not tolerate NaNs or INFs; use the masks if you need to exclude pixels from the registration process.

Parameters

refMask=*refMaskWave* Specifies an optional ROI wave used to mask refWave. Omit refMask to use all of the refWave.

The wave must have the same dimensions as refWave. refMask is a single precision floating point wave with nonzero entries marking the “ON” state. Note that the operation modifies this wave and that you should not use the same wave for both the reference and the test masks.

refWave=*imageWave2* Specifies the name of the reference image wave used to adjust testWave.

testMask=*testMaskWave* Specifies an optional ROI wave used to mask testWave. Omit testMask to use all of the testWave.

The wave must have the same dimensions as refWave. testMask is a single precision floating point wave with nonzero entries marking the “ON” state. Note that the operation modifies this wave and that you should not use the same wave for both the reference and the test masks.

testWave=*imageWave1* Specifies the name of the image wave that will be adjusted to match refWave.

testMaskWave and *refMaskWave* are optional ROI waves. The waves must have the same dimensions as testWave and refWave respectively. They must be single precision floating point waves with nonzero entries marking the “ON” state. If you need to include the whole region described by testWave or the whole region described by refWave you can omit the respective mask wave

Flags

/ASTP=*val* Sets the adaptation step for the Levenberg-Marquardt algorithm. Default value is 4.

/BVAL=*val* Enables clipping and sets the background values to which masked out voxels of the test data will be set.

/CONV=*val* Sets the convergence method.

val=0: Gravity, use if the difference between the images is only in translation. This option is frequently useful as a first step when the test and reference data are too far apart for accurate registration. The result of this registration is then passed to a subsequent ImageRegistration with /CONV=1.

val=1: Marquardt.

/CSNR=*val* Determines if the operation calculates the signal to noise ratio (SNR)

val=0: The SNR is not calculated.

val=1: The SNR is calculated (default).

Skipping the SNR calculation saves time and may be particularly useful when performing the registration on a stack of images.

/FLVL=*val* Specifies the finest level on which the optimization is to be performed.

If this is the same as /PRDL, then only the coarsest registration calculation is done.

If /FLVL=1 (default), then the full multiresolution pyramid is processed. You can use this flag to terminate the computation at a specified coarseness level greater than 1.

/GRYM Optimizes the gray level scaling factor.

It is sometimes dangerous to let the program adjust for gray levels because in some situations it might result in a null image.

/GRYR Renders output using the gray scaling parameter. This is more meaningful if the operation computes the optimal gray scaling (see /GRYM).

/GWDT={*sx*,*sy*,*sz*} Sets the three fields to the half-width of a Gaussian window that is used to smooth the data when computing the default masks. Defaults are {1,1,1}. See /REFM and /TSTM for more details.

/INTR=*val* Sets the interpolation method.

val=0: Nearest neighbor. Used when registering the center of gravity of the test and reference images.

val=1: Trilinear.

val=2: Tricubic (default).

/ISOS Optimizes the isometric scaling. This option is inappropriate if voxels are not cubic.

/ISR Computes the multiresolution pyramid with isotropic size reduction.

If the flag is not specified, the size reduction is in the XY plane only.

/MING=*val* Sets the minimum gain at which the computations will stop. Default value is zero, but you can use a slightly larger value to stop the iterations earlier.

/MSKC=*val* Sets mask combination value. During computation the masks for the test data and the mask for the reference data are also transformed. This flag determines how the two masks are to be combined. The registration criteria are computed for the combination of the two masks.

val=0: or.

val=1: nor.

val=2: and (default).

val=3: nand.

val=4: xor.

val=5: nxor.

/PRDL=*depth* Specifies the depth of the multiresolution pyramid. The finest level is *depth*=1. Each level of the pyramid decreases the resolution by a factor of 2. By default, the pyramid *depth*=4, which corresponds to a resolution reduction by a factor of $2^{(depth\text{-}1)}=8$.

The algorithm starts by computing the first registration on large scale features in the image (deepest level of the pyramid). It then makes small corrections to the registration at each consecutive pyramid level.

For best results, the coarsest representation the data should be between 30 and 60 pixels on a side. For example, for an image that is *H* by *V* pixels, you should choose the depth such that $H/2^{(depth\text{-}1)} \approx 30$.

/PSTK When performing registration of a stack of images, use this flag to apply the registration parameters of the previous layer as the initial guess for the registration of each layer after the first in the 3D stack.

/Q Quiet mode; no messages printed in the history area.

/REFM=*val* Sets the reference mask.

val=0: To leave blank and then every pixel is taken into account.

val=1: Value will be set if a valid reference mask is provided.

val=2: The test mask is computed (default).

When computing the reference mask it is assumed that brighter features are more important. This is done by using a low pass filter on the data (using the parameters in /GWDT) which is then converted into a binary mask. Note that you do not need to specify /REFM=1 if you are providing a reference mask wave. See also /TSTM.

/ROT={*rotX*,*rotY*,*rotZ*}

Determines if optimization will take into account rotation about the X, Y, or Z axes. A value of one allows optimization and zero prevents optimization from affecting the corresponding rotation parameter. Defaults are {0,0,1}, which are the appropriate values for rotating images.

/SKEW={*skewX*,*skewY*,*skewZ*}

Determines if optimization will take into account skewness about the X, Y, or Z axes. A value of one allows optimization and zero prevents optimization from affecting the corresponding skewness parameter. Defaults are {0,0,0}. Note that skewing and rotation or isometric scaling are mutually exclusive operations.

/STCK Use /STCK to perform the registration between a 2D reference image and each of the layers in a 3D image. The number of rows and columns of the refWave must match exactly the number of rows and columns in testWave. The transformation parameters are saved in the wave M_RegParams where each column contains the parameters for the corresponding layer in testWave.

/STRT=*val* Sets the first value of the adaptation parameter in the Levenberg-Marquardt algorithm.

The default value of this parameter is 1.

/TRNS={*transX*,*transY*,*transZ*}

Determines if optimization will take into account translation about the X, Y, or Z axes. A value of one allows optimization and zero prevents optimization from affecting the corresponding translation parameter. Defaults are {1,1,0}, which are appropriate for finding the X and Y translations of an image.

/TSTM=*val* Sets the test mask.

val=0:	To leave blank and then every pixel is taken into account.
val=1:	This value will be set if a valid reference mask is provided.
val=2:	The reference mask is computed. This is the default value.

The test image mask is computed in the same way as the reference image mask (see /REFM) using the same set of smoothing parameters. Note that you do not need to specify /TSTM=1 if you are providing a test mask wave.

/USER=*pWave* Provides a user transformation that will be applied to the input testWave in order to create the trasnsformed image. *pWave* must be a double precision wave which contains the same number of rows as W_RegParams.

Note: If you use a previously created W_RegParams make sure to change its name as it is overwritten by the operation.

If *pWave* has only one column and testWave contains multiple layers, then the same transformation applies to all layers. If *pWave* contains more than one column, then each layer of testWave is processed with corresponding column. If there are more layers than columns the first column is used in place of the missing columns.

/ZMAN Modifies the test and reference data by subtracting their mean values prior to optimization.

Details

ImageRegistration will register images that have sufficiently similar features. It will not work if key features are too different. For example, ImageRegistration can handle two images that are rotated relative to each other by a few degrees, but cannot register images if the relative rotation is as large as 45 degrees. The algorithm is capable of subpixel resolution but it does not handle large variations between the test image and the reference image. If the centers of the two images are too far from each other, you should first try ImageRegistration using /CON=0 to remove the translation offset before proceeding with a finer registration of details.

The algorithm is based on an iterative processing that proceeds from coarse to fine detail. Optimization uses a modified Levenberg-Marquardt algorithm and produces an affine transformation for the relative rotation and translation as well as for isometric scaling and contrast adjustment. The algorithm is most effective with square images where the center of rotation is not far from the center of the image.

When using gravity for convergence, skew parameters can't be evaluated (only translation is supported). Skew and isoscaling are mutually exclusive options. Mask waves are defined to have zero entries for pixels outside the region of interest and nonzero entries otherwise. If a mask is not provided, every pixel is used.

ImageRegistration creates the waves M_RegOut and M_RegMaskOut, which are both single precision waves. In addition, the operation creates the wave W_RegParams which stores 20 double precision registration parameters. M_RegOut contains the transformed (registered) test image and M_RegMaskOut contains the transformed mask (which is not affected by mask combination). ImageRegistration ignores wave scaling; images are compared and registered based on pixel values only.

The results printed in the history include:

dx, dy, dz	translation offsets measured in pixels.
a*ij*	Elements in the skewing transformation matrix.
phi	Rotation angle in degrees about the X-axis. Zero for 2D waves.
tht	Rotation angle in degrees about the Y-axis. Zero for 2D waves.
psi	Rotation angle in degrees about the Z-axis.
det	Absolute value of determinant of the skewing matrix (a*ij*).
err	Mean square error defined as $\frac{1}{N}\sum\left(x_i - y_i\right)^2,$ where x_i is the original pixel value, y_i the computed value, and N is the number of pixels.
snr	Signal to noise ratio in dB. It is given by: $10\log\left(\frac{\sum x_i^2}{\sum\left(x_i - y_i\right)^2}\right).$

These parameters are stored in the wave W_RegParams (or M_RegParams in the case of registering a stack). Angles are in radians. Dimension labels are used to describe the contents of each row of the output wave. Each column of the wave consists of the following rows (also indicated by dimension labels):

Point	Contents	Point	Contents	Point	Contents	Point	Content
0	dx	6	a21	12	gamma	17	origin_x
1	dy	7	a22	13	phi	18	origin_y
2	dz	8	a23	14	theta	19	origin_z
3	a11	9	a31	15	psi	21	MSE
4	a12	10	a32	16	lambda	21	SNR
5	a13	11	a33				

You can view the output waves with dimension labels by executing:

```
Edit W_RegParams.ld
```

See Also

The ImageInterpolate **Warp** operation.

References

The ImageRegistration operation is based on an algorithm described by:

Thévenaz, P., and M. Unser, A Pyramid Approach to Subpixel Registration Based on Intensity, *IEEE Transactions on Image Processing*, *7*, 27-41, 1998.

ImageRemoveBackground

ImageRemoveBackground /R=*roiWave* [***flags***] ***srcWave***

The ImageRemoveBackground operation removes a general background level, described by a polynomial of a specified order, from the image in *srcWave*. The result of the operation are stored in the wave M_RemovedBackground.

Flags

/F	Computes only the background surface fit. Will only store the resulting fit in M_RemovedBackground. This will not subtract the fit from the image.
/O	Overwrites the original wave.
/P=*polynomial order*	Specifies the order of the polynomial fit to the background surface. If omitted, the order is assumed to be 1.
/R=*roiWave*	Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/B/U), which has the same number of rows and columns as the image wave. Set the pixels that define the background region to 1. The remaining pixels can be any value other than 1. We recommend using 64 which Igor image processing operations often interpret as "blank" in unsigned byte image waves. The ROI does not have to be contiguous. See **ImageGenerateROIMask** for more information on creating ROI waves.
/W	Specifies that polynomial coefficients are to be saved in the wave W_BackgroundCoeff.

Details

The identification of the background is done via the ROI wave. Set the pixels that define the background region to 1. The remaining pixels can be any value other than 1. We recommend using 64 which Igor image processing operations often interpret as "blank" in unsigned byte image waves.

The operation first performs a polynomial fit to the points designated by the ROI wave using the specified polynomial order. A polynomial of order N corresponds to the function:

$$F_N(x,y) = \sum_{m=0}^{N} \sum_{n=0}^{m} c_{nm} x^{m-n} y^n .$$

Using the polynomial fit, a surface corresponding to the polynomial is subtracted from the source wave and the result is saved in M_RemovedBackground, unless the /O flag is used, in which case the original wave is overwritten.

Use the /W flag if you want polynomial coefficients to be saved in the W_BackgroundCoeff wave. Coefficients are stored in the same order as the terms in the sums above.

If you do not specify the polynomial order using the /P flag, the default order is 1, which means that the operation subtracts a plane (fitted to the ROI data) from the source image.

Note, if the image is stored as a wave of unsigned byte, short, or long, you might consider converting it into single precision (using `Redimension/S`) before removing the background. To see why this is important, consider an image containing a region of pixels equal to zero and subtracting a background plane corresponding to a nonconstant value. After subtraction, at least some of the pixels in the zero region should become negative, but because they are stored as unsigned quantities, they appear incorrectly as large values.

Examples

See **Background Removal** on page III-379.

See Also

The **ImageGenerateROIMask** operation for creating ROIs.

ImageRestore

```
ImageRestore [flags] srcWave=wSrc, psfWave=wPSF [, relaxationGamma=h,
   startingImage=wRecon ]
```

The ImageRestore operation performs the Richardson-Lucy iterative image restoration.

Flags

/DEST=*destWave*	Specifies the desired output wave. If /DEST is omitted, the output from the operation is stored in the wave M_Reconstructed in the current data folder.
/ITER=*iterations*	Specifies the number of iterations. The default number of iterations is 100.
/Z	Do not report errors.

Parameters

psfWave=*wPSF*	Specifies a known point spread function. *wPSF* must be a 2D (square NxN) wave of the same numeric type as *wSRC*. N must be an odd number greater than 1.
relaxationGamma=*h*	Specifies positive power gamma of in the relaxation mapping (see Details).
startingImage=*wRecon*	Use this keyword to specify a starting image that could be for example the output from a previous call to this operation. *wRecon* must have the same dimensions as *wSRC* and the same numeric type. You must make sure that *wRecon* is not the user-specified or the default destination wave of the operation.
srcWave=*wSrc*	Specifies the degraded image which must be a 2D single-precision or double-precision real wave.

Details

ImageRestore performs the Richardson-Lucy iteration solution to the deconvolution of an image. The input consists of the degraded image and point spread function as well as the desired number of iterations.

The operation allows you to apply additional iterations by setting the starting image to the restored output wave from a previous call to ImageRestore using the startingImage keyword. If startingImage is omitted, the starting image is created by ImageRestore with each pixel set to the value 1.

In the case of stellar images it may be useful to apply a relaxation step that involves scaling the correction evaluated at each iteration by

$$factor(v) = \sin\left(\frac{\pi}{2}\frac{v - v_{\min}}{v_{\max} - v_{\min}}\right)^{\gamma},$$

where v is pixel value, vmax and vmin are the maximum and minimum level pixels in the image and gamma is the user-specified relaxationGamma.

References

W.H. Richardson, "Bayesian-Based Iterative Method of Image Restoration". *JOSA 62, 1*: 55-59, 1972.

L.B. Lucy, "An iterative technique for the rectification of observed distributions", *Astronomical Journal 79, 6*: 745-754, 1974.

ImageRotate

ImageRotate [*flags*] *imageMatrix*

The ImageRotate operation rotates the image clockwise by *angle* (degrees) or counter-clockwise if /W is used.

The resulting image is saved in the wave M_RotatedImage unless the /O flag is specified. The size of the resulting image depends on the angle of rotation.

The portions of the image corresponding to points outside the domain of the original image are set to the default value 64 or the value specified by the /E flag.

You can apply ImageRotate to 2D and 3D waves of any data type.

Flags

/A=*angle*	Specifies the rotation angle measured in degrees in the counter-clockwise direction. For rotations by exactly 90 degrees use /C or /W instead.
/C	Specifies a counter-clockwise rotation by 90 degrees.
/E= *val*	Specifies the value for pixels that are outside the domain of the original image. By default pixels are set to 64. If you specify `/E=(NaN)` and your data is of type char, short, or long, the operation sets the external values to 64.
/F	Rotates image by 180 degrees.
/H	Flip the image horizontally.
/O	Overwrites the original image with the rotated image.
/Q	Quiet mode. Without this flag the operation writes warnings in the history area.
/RGBA=[*R*, *G*, *B* [, *A*])	Specifies the RGB or RGBA values of pixels that lie outside the domain occupied by the original image. This flag was added in Igor Pro 7.00.
/S	Uses source image wave scaling to preserve scaling and relative locations of objects in the image for rotation angles that are multiples of 90 degrees.
/V	Flip the image vertically.
/W	Specifies a clockwise rotation by 90 degrees.
/Z	Ignore errors.

See Also

The **MatrixTranspose** operation.

ImageSave

ImageSave [*flags*] *waveName* [[**as**]*fileNameStr*]

The ImageSave operation saves the named wave as an image file.

Previously this operation used QuickTime to implement the saving of some file formats. As of Igor Pro 7.00, it no longer uses QuickTime. Consequently, some file formats re no longer supported and some flags have changed.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where pathName is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

If you want to force a dialog to select the file, omit the *fileNameStr* parameter or specify `""` for *fileNameStr* or use the /I flag. The "as" keyword before *fileNameStr* is optional.

In Igor Pro 7.00 or later, if you specify "Clipboard" for *fileNameStr* and the file type is JPEG or PNG, Igor writes the image to the clipboard.

In Igor Pro 7.00 or later, if the file type is PNG or JPEG you can write the image to the picture gallery using the magic path name _PictGallery_. For example

```
ImageSave/T=PNG/P=_PictGallery_ waveName
```

Flags

/ALPH=*mode* Sets the value used for the ExtraSamples tag (338) in the TIFF file format. This flag was added in Igor Pro 7.00.

The tag tells a TIFF file reader how to use the alpha values. Supported modes are 1 for premultiplied and 2 for unassociated alpha. Premultiplied means that the RGB components have already been multiplied by alpha. Unassociated means that the RGB components are raw.

By default, alpha is assumed to be premultiplied (*mode*=1).

/BIGT Saves TIFF files in BigTIFF format, a newer TIFF standard that allows you to save large images (over 2GB) and provides support for several compression formats. Many applications that load TIFF files do not support BigTIFF.

The /BIGT flag was added in Igor Pro 9.00.

/C=*comp* Specifies the compression to use when saving BigTIFF:

comp	**BigTIFF Compression**
1	No compression (default)
2	CITTRLE
3	CCITTFAX3
4	CCITTFAX4
5	LZW
7	JPG
8	ADOBE_DEFLATE
32773	PACKBITS
32946	DEFLATE
32947	DCS
34712	JP2000s

The /C flag was added in Igor Pro 9.00.

/D=*depth* This flag is deprecated and should not be used in new code.

Specifies color depth in bits-per-pixel. Integer values of 1, 8, 16, 24, 32, and 40 are supported. A *depth* of 40 specifies 8-bit grayscale; a *depth* of 8 saves the file as 8-bits/pixel with a color lookup table. If /D is omitted, it acts like /D=8.

Saving with a color table may cause some loss of fidelity.

See also the discussion of saving TIFF files below.

/DS=*depth* Saves TIFF data in *depth* bits/sample.

Values for *depth* of 8, 16, 32 and 64 are supported. The default is 8.

The total number of bits/pixel is: (bits/sample) * (samples/pixel).

When using 32 or 64 bits/sample, *srcWave* is saved without normalization.

/F — This flag is deprecated and should not be used in new code.

Saves the wave as single precision float. The data is not normalized. Applies only to TIFF files.

/I — Interactive mode. Forces ImageSave to display a Save File dialog, even if the file and folder location are fully specified.

/IGOR — In Igor6 this flag told Igor to use internal code rather than QuickTime to write TIFF files. Igor no longer uses QuickTime and always uses internal code. This flag is still accepted but has no effect.

/O — Overwrites the specified file if it exists. If /O is omitted and the file exists, ImageSave displays a Save File dialog.

/P=*pathName* — Specifies the folder in which to save the image file. *pathName* is the name of an existing symbolic path.

/Q=*quality* — Specifies the quality of the compressed image. *quality* is a number between 0 and 1, with 1 being the highest quality. This is only applicable when saving in formats with lossy compression like JPEG.

/S — Saves as stack. Applies only to 3D or 4D source waves saved as TIFF files.

/T=*fileTypeStr* — Specifies the type of file to be saved.

fileTypeStr	**Saved Image Format**
"jpeg"	JPEG file
"png"	PNG file
"rpng"	raw PNG file (see **Saving as Raw PNG**)
"tiff"	TIFF file

If /T is omitted, the default type is "tiff".

/U — Prevents normalization. This works when saving TIFF only. See **Saving as TIFF** below.

/WT=*tagWave* — Saves TIFF files with file tags as specified by *tagWave*, which is a text wave consisting of a row and 5 columns for each tag (see description of the /RAT flag under **ImageLoad**). It ignores any information in the second column but will write the tags sequentially (only in the first Image File Directory (IFD) if there is more than one image). If the fourth column contains a negative number, there must be a wave, whose name is given in the fifth column of *tagWave*, in the same data folder as *tagWave*. You must make sure that: (1) tag numbers are legal and do not conflict with any existing tags; (2) the data type and data size are consistent with the amount of data saved in external waves.

/Z — No error reporting.

Details

Image files are characterized by the number of color components per pixel and the bit-depth of each components. Igor displays images that consist of 1 components (gray-scale/false color), 3 components (RGB) or 4 components (RGBA) per pixel. You can display waves of all real numeric types as images, but wave data are assumed to be either in the range of [0,255] for 8-bits/components or [0,65535] for all other numeric types. For more information see **Creating an Image Plot** on page II-386.

When you save a numeric wave as image file your options depend on the number of components (layers) of your wave and its number type.

Saving as PNG

You can save an Igor wave as a PNG file with 24 bits per pixel or 32 bits per pixel. The following table describes how the wave's data type corresponds to the PNG pixel format.

Source Wave	PNG Pixel Format
1-layer any type	RGB 24 bits per pixel gray normalized [0,255]
3-layer unsigned byte	RGB 24 bits per pixel no scaling
3-layer unsigned short	RGB 24 bits per pixel data divided by 256
3-layer all other types	RGB 24 bits per pixel data normalized [0,255]
4-layer unsigned byte	RGBA 32 bits per pixel no scaling
4-layer unsigned short	RGBA 32 bits per pixel data divided by 256
4-layer all other types	RGBA 32 bits per pixel data normalized [0,255]

Gray means that the three components of each pixel have the identical value. Data normalization consists of finding the global (over all layers) minimum and maximum values of *srcWave* followed by scaling to the full range, e.g.,

```
minValue = WaveMin(srcWave)
maxValue = WaveMax(srcWave)
outValue[i][j][k] = 255*(srcWave[i][j][k]-minValue)/(maxValue-minValue)
```

Saving as Raw PNG

The rpng image format requires a wave in 8- or 16-bit unsigned integer format with 1 to 4 layers. Use one layer for grayscale, 3 layers for rgb color, and the extra layer for an alpha channel. If X or Y scaling is not unity, they both must be valid and must be either inches per pixel or meters per pixel. If the units are not inches they are taken to be meters.

Saving as TIFF

ImageSave supports saving uncompressed TIFF images of 8, 16, 32 and 64 bits per color component with 1, 3 or 4 components per pixel.

Depending on the data type of your wave and the depth of the image file being created, ImageSave may save a "normalized" version of your data. The normalized version is scaled to fit in the range of the image file data type. For example, if you save a 16-bit Igor wave containing pixel values from -1000 to +1000 in an 8-bit grayscale TIFF file, ImageSave will map the wave values -1000 and +1000 to the file values 0 and 255 respectively. When saving an image file of 16-bits/component, Igor normalizes to 65535 as the full-scale value.

There is no normalization when you save in floating point (32 or 64 bits/component). Normalization is also not done when saving 8-bit wave data to an 8-bit image file. You can disable normalization with the /U flag.

Saving in floating point can lead to large image files (e.g., 64-bit/component RGBA has 256 bits/pixel) which are not supported by many applications.

Saving 3D or 4D Waves as a Stack of TIFF Images

If your Igor data is a 3D wave other than an RGB wave or a 4D wave, you can save it as a stack of grayscale images without a color map.

Use /S to indicate that you want to save a stack of images rather than a single image from the first layer of the wave.

Use /D=8 to save as 8 bits. Normalization is done except if the wave data is 8 bits.

Use /D=16 to save as 16 bits with normalization.

Use /F to save as single-precision floating point without normalization. Many programs can not read this format.

Use /U to prevent normalization.

Stacked images are normalized on a layer by layer basis. If you want to have uniform scaling and normalization you should convert your wave to the file data type before executing ImageSave.

See Also

The **ImageLoad** operation for loading image files into waves.

ImageSeedFill

ImageSeedFill [*flags*] [*keyword*]**, seedX=*xLoc*, seedY=*yLoc*, target=*setValue*, srcWave=*srcImage***

The ImageSeedFill operation takes a seed pixel and fills a contiguous region with the target value, storing the result in the wave M_SeedFill. The filled region is defined by all contiguous pixels that include the seed pixel and whose pixel values lie between the specified minimum and maximum values (inclusive). ImageSeedFill works on 2D and 3D waves.

Parameters

keyword is one of the following names:

adaptive=*factor* Invokes the adaptive algorithm where a pixel or voxel is accepted if its value is between the specified minimum and maximum or its value satisfies:

$$|val - avg| < factor * stdv.$$

Here *val* is the value of the pixel or voxel in question, *avg* is the average value of the pixels or voxels in the neighborhood and *stdv* is the standard deviation of these values. By choosing a small *factor* you can constrain the acceptable values to be very close to the neighborhood average. A large *factor* allows for more deviation assuming that the *stdv* is greater than zero.

This requirement means that a connected pixel has to be between the specified minimum and maximum value **and** satisfy the adaptive relationship. In most situations it is best to set wide limits on the minimum and maximum values and allow the adaptive parameter to control the local connectivity.

fillNumber=*num* Specifies the number, in the range 1 to 26, of voxels in each 3x3x3 cube that belong to the set. If fillNumber is exceeded, the operation fills the remaining members of the cube. If you do not specify this keyword, the operation does not fill the cube. Used only in the fuzzy algorithm.

fuzzyCenter=*fcVal* Specifies the center value for the fuzzy probability with the fuzzy algorithm (see **Details**). The default value is 0.25. Its standard range is 0 to 0.5, although interesting results might be obtained outside this range.

fuzzyProb=*fpVal* Specifies a probability threshold that must be met by a voxel to be accepted to the seeded set. The value must be in the range 0 to 1. The default value is 0.75.

fuzzyScale=*fsVal* Determines if a voxel is to be considered in a second stage using fuzzy probability. *fsVal* must be nonzero in order to invoke the fuzzy algorithm. The scale is used in comparing the value of the voxel to the value of the seed voxel. The scale should normally be in the range 0.5 to 2.0.

fuzzyWidth=*fwVal* Defines the width of the fuzzy probability distribution with the fuzzy algorithm (see **Details**). In most situations you should not need to specify this parameter. The default value is 1.

min=*minval* Specifies the minimum value that is accepted in the seeded set. Not needed when using fuzzy algorithm.

max=*maxval* Specifies the maximum value that is accepted to the seeded set. Not needed when using the fuzzy algorithm.

seedP=row Specifies the integer row location of the seed pixel or voxel. This avoids roundoff issues when srcWave has wave scaling. You must provide either seedP or seedX with all algorithms. It is sometimes convenient to use this with cursors e.g., `seedP=pcsr(a)`.

seedQ=col Specifies the integer column location of the seed pixel or voxel. This avoids roundoff difficulties when srcWave has wave scaling. You must provide either seedQ or seedY with all algorithms.

seedR=layer

Specifies the integer layer position of the seed voxel. When srcWave is a 3D wave you must use either seedR or seedZ.

seedX=*xLoc*

Specifies the pixel or voxel index. If srcWave has wave scaling, seedX must be expressed in terms of the scaled coordinate. This keyword or seedP is required with all algorithms.

seedY=*yLoc*

Specifies the pixel or voxel index. If srcWave has wave scaling, seedY must be expressed in terms of the scaled coordinate. This keyword or seedQ is required with all algorithms.

seedZ=*zLoc*

Specifies the voxel index. If srcWave has wave scaling, seedZ must be expressed in terms of the scaled coordinate. You must use this keyword or seedR whenever srcWave is 3D.

srcWave=*srcImage*

Specifies the source image wave.

target=*val*

Sets the value assigned to pixels or voxels that belonging to the seeded set.

Flags

/B=*bValue*

Specifies the value assigned to pixels or voxels that do not belong to the filled area. If you omit /B, these pixels or voxels are assigned the corresponding values of the wave specified by the srcWave keyword.

/C

Uses 8-connectivity where a pixel can be connected to any one of its neighbors and with which it shares as little as a single boundary point. The default setting is 4-connectivity where pixels can be connected if they are neighbors along a row or a column. This has no effect in 3D, where 26-connectivity is the only option.

/K=*killCount*

Terminates the fill operation after *killCount* elements have been accepted.

/O

Overwrites the source wave with the output (2D only).

/R=*roiWave*

Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u), that has the same number of rows and columns and layers as the image wave. The ROI itself is defined by the entries/pixels whose value are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous. See ImageGenerateROIMask for more information on creating ROI waves.

Details

In two dimensions, the operation takes a seed pixel, optional minimum and maximum pixel values and optional adaptive coefficient. It then fills a contiguous region (in a copy of the source image) with the target value. There are two algorithms for 2D seed fill. In direct seed fill (only min, max, seedX and seedY are specified) the filled region is defined by all contiguous pixels that include the seed pixel and whose pixel values lie between the specified minimum and maximum values (inclusive). In adaptive fill, there is an additional condition for the pixel or voxel to be selected, which requires that the pixel value must be within the standard deviation of the average in the 3x3 (2D) or 3x3x3 (3D) nearest neighbors. If you do not specify the minimum and maximum values then the operation selects only values identical to that of the seed pixel.

In 3D, there are three available algorithms. The direct seed fill algorithm uses the limits specified by the user to fill the seeded domain. In adaptive seed fill the algorithm requires the limits as well as the adaptive parameter. It fills the domain by accepting only voxels that lie within the adaptive factor times the standard deviation of the immediate voxel neighborhood. To invoke the third algorithm you must set fuzzyScale to a nonzero value. The fuzzy seed fill uses two steps to determine if a voxel should be in the filled domain. In the first step the value of the voxel is compared to the seed value using the fuzzy scale. If accepted, it passes to the second stage where a fuzzy probability is calculated based on the number of voxels in the 3x3x3 cell which passed the first step together with the user-specified probability center (fuzzyCenter) and width (fuzzyWidth). If the result is greater than fuzzyProb, the voxel is set to belong to the filled domain.

If the /O flag is not specified, the result is stored in the wave M_SeedFill.

If you specify a background value with the /B flag, the resulting image consists of the background value and the target value in the area corresponding to the seed fill. Although the wave is now bi-level, it retains the same number type as the source image.

ImageSeedFill returns a “bad seed pixel specification” if the seed pixel location derived from the various keywords above satisfies one or more of the following conditions:

- The computed integer pixel/voxel is outside the image.
- The value stored in the computed integer pixel/voxel location does not satisfy the min/max or fuzzy conditions. This is the most common condition when srcWave has wave scaling. To avoid this difficulty you should use the keywords seedP, seedQ, and seedR.

Examples

Using Cursor A position and value to supply parameter inputs for a 2D seedFill (**Warning**: command wrapped over two lines):

```
ImageSeedFill
   seedP=pcsr(a),seedQ=qcsr(a),min=zcsr(a),max=zcsr(a),target=0,srcwave=image0
```

Using the fuzzy algorithm for a 3D wave (**Warning**: command wrapped over two lines):

```
ImageSeedFill seedX=232,seedY=175,seedZ=42,target=1,fillNumber=10,fuzzyCenter=.25,
   fuzzyWidth=1,fuzzyScale=1,fuzzyProb=0.4,srcWave=ddd
```

See Also

For an additional example see **Seed Fill** on page III-377. To display the result of the operation for 3D waves it is useful to convert the 3D wave M_SeedFill into an array of quads. See ImageTransform **vol2surf**.

ImageSnake

ImageSnake [***flags***] ***srcWave***

The ImageSnake operation creates or modifies an active contour/snake in the grayscale image srcWave. The operation iterates to find the “lowest total energy” snake. The energy is defined by a range of optional flags, each corresponding to an individual term in the total energy expression. Iterations terminate by reaching the maximum number of iterations, when the snake does not move between iterations or when the user aborts the operation.

Flags

/ALPH=*alpha* Sets the coefficient of the energy term arising from the “tightness” of the snake.

/BETA=*beta* Sets the coefficient of the energy term corresponding to curvature of the snake. A high value for beta makes the snake more rounded.

/DELT=*delta* Sets the coefficient of the repulsion energy. A high value of *delta* keeps nonconsecutive snake points far from each other.

/EPS=*num* Sets the maximum number of vertices which are allowed to move in one iteration. If the number of vertices which move during an iteration is smaller than *num* then iterations terminate.

/EXEF=*eta* Sets the coefficient of the optional external energy component. By default this value is set to zero and there is no external energy contribution to the snakes energy. Note, this component is referred to as “external” because it is completely up to the user to specify both its coefficient and the value associated with each pixel. It should not be confused with what is called external snake energy in the literature, which usually applies to energy proportional to the gradient image (see /GAMM and /GRDI).

/EXEN=*wave* Specify a wave that contains energy values that will be added to the snakes energy calculation. The wave must have the same dimensions as *srcWave* and must be single precision float. Each pixel value corresponds to user defined energy which will be multiplied by the /EXEF coefficient and added to the sum which the snake minimizes. Note that when /EXEF is set to zero this component is ignored. An external energy wave may be useful, for example, if you want to attract the snake to the picture boundaries. In that case you can set:

```
Duplicate/O srcWave,extWave
Redimension/S extWave
Variable rows=DimSize(srcWave,0)-1
Variable cols=DimSize(srcWave,1)-1
extWave=(p==0 || q==0 || p==rows || q==cols) ? 0:1
```

/GAMM=*gamma* Sets the coefficient of the energy term corresponding to the gradient. A high value of gamma makes the snake follow lines of high image gradient.

/GRDI=*gWave* Specify the gradient image. This wave must have the same dimensions as *srcWave* and it must be single precision float. The wave corresponds to the quantity `abs(grad(gauss**srcWave))`, where grad is the gradient operator and ** denotes convolution of the source wave with a Gaussian kernel. It is best to run the operation the first time without specifying this wave. When the operation executes, it creates the wave M_GradImage which can then be used in subsequent executions of this operation. If you want to modify the wave to express some other form of energy that you want the operation to minimize, you should use the /EXEN and /EXEF flags.

/ITER=*iterations* Sets the maximum number of iterations. Convergence can be achieved if the value specified by /EPS is met. You can also terminate the process earlier by pressing the **User Abort Key Combinations**.

/N=*snakePts* Specify the number of vertices in the snake or the number of snake points. Note that if you are providing snake waves in /SX and /SY, you do not need to specify this flag. If you do not specify this flag the default value is 40.

/SIG=*sigma* Sets the size of the Gaussian kernel that is used to convolve the input image when creating the gradient image. Note that you do not need to use this flag if you provide a gradient image. *sigma* is 3 by default. You can use larger odd integers for larger Gaussian kernels which would correspond to a stronger blur.

/STRT={*centerX*, *centerY*, *radius*}

Sets the starting snake to be a circle with the given center and radius. If you use this flag you should also provide the number of snake points using the /N flag.

/STEP=*pixels* Sets the maximum radius of search. By default the radius of the search is 6 pixels and the search follows a clockwise pattern from radius of 1 pixel to maximum radius specified by this flag. Note: the search radius should be smaller than the dimension of a typical feature in the image. If the radius is larger the snake may encompass more than one object. Larger radius is also less efficient because many of the pixels in that range would result in a snake that crosses itself and hence get rejected in the process.

/SX=*xSnake* Provide an X-wave for the starting snake. You must also provide an appropriate Y-wave using /SY.

/SY=*ySnake* Provide a Y-Wave for the starting snake. Must work in combination with /SX.

/UPDM=*mode* Sets the update mode using any combination of the following:

Value	Update
0	Once when the operation terminates.
1	Once at the end of every iteration.
2	Once after every snake vertex moves.
4	Once for every search position.

/Q Quiet mode; don’t print information in the history.

/Z Don’t report any errors.

Details

A snake is a two-dimensional, usually closed, path drawn over an image. The snake is described by a pair of XY waves consisting of N vertices (sometimes called “snake elements” or “snaksels”). In this implementation it is assumed that the snake is closed so that the last point in the snake is connected to the first. Snakes are used in image segmentation, when you want to automatically select a contiguous portion of the plane based on some criteria. Unlike the classic contours, snakes do not have to follow a constant level. Their structure (or path) is found by associating the concept of energy with every snake configuration and attempting to find the configuration for which the associated energy is a minimum. The search for a minimum energy configuration is usually time consuming and it strongly depends on the format of the energy function and the initial conditions (as defined by the starting snake). The operation computes the energy as a sum of the following 5 terms:

1. The coefficient alpha times a sum of absolute deviations from the average snake segment length. This term tends to distribute the vertices of the snake at even intervals.
2. The coefficient beta times a sum of energies associated with the curvature of the snake at each vertex.
3. The coefficient gamma times a sum of energies computed from the negative magnitude of the gradient of a Gaussian kernel convolved with the image. This term is usually referred to in the literature as the external energy and usually drives the snake to follow the direction of high image gradients.
4. The coefficient delta times a repulsion energy. Repulsion is computed as an inverse square law by adding contributions from all vertices except the two that are immediately connected to each vertex. This energy term is designed to make sure that the snake does not fold itself into “valleys”.
5. The coefficient eta times the sum of values corresponding to the positions of all snake vertices in the wave you provide in /EXEN.

The energy calculation skips all terms for which the coefficient is zero. In addition there is a built-in scan which adds a very high penalty for configurations in which the snake crosses itself.

ImageSkeleton3D

ImageSkeleton3D [/DEST=*destWave* /METH=*method* /Z] *srcWave*

The ImageSkeleton3D operation computes the skeleton of a 3D binary object in srcWave by "thinning". Thinning is a layer-by-layer erosion until only the "skeleton" of an object remains. (See reference below.) It is used in neuroscience to trace neurons.

The ImageSkeleton3D operation was added in Igor Pro 7.00.

Parameters

srcWave is a 3D unsigned-byte wave where object voxels are set to 1 and the background is set to 0.

Flags

/DEST=*destWave* Specifies the wave to contain the output of the operation. If the specified wave already exists, it is overwritten.

When used in a user-defined function, ImageSkeleton3D creates wave reference for *destWave* if it is a simple name. See **Automatic Creation of WAVE References** on page IV-72 for details.

If you omit /DEST the output wave is M_Skeleton in the current data folder.

/METH=*m* Sets the method used to compute the skeleton.

m=1: Uses elements of an algorithm by Kalman Palagyi (default).

This is currently the only supported method.

/Z Do not report any errors.

Details

The output is stored in the wave M_Skeleton in the current data folder or in the wave specified by /DEST.

Skeleton voxels are set to the value 1 and background voxels are set to 0.

Example

```
// Create a cube with orthogonal square holes
Make/B/U/N=(30,30,30) ddd=0
ddd[2,27][2,27][2,27]=1
ddd[2,27][10,20][10,20]=0
ddd[10,20][2,27][10,20]=0
ddd[10,20][10,20][2,27]=0
ImageSkeleton3D ddd
```

See Also

Chapter III-11, **Image Processing**, **ImageMorphology**, **ImageSeedFill**

Reference

K. Palagyi, "A 3D fully parallel surface-thinning algorithm", *Theoretical Computer Science* **406** (2008) 119-135.

ImageStats

`ImageStats [ flags ] imageWave`

The ImageStats operation calculates wave statistics for specified regions of interest in a real matrix wave. The operation applies to image pixels whose corresponding pixels in the ROI wave are set to zero. It does not print any results in the history area.

Flags

/BEAM
Computes the average, minimum, and maximum pixel values in each layer of a 3D wave and 2D ROI. Output is to waves W_ISBeamAvg, W_ISBeamMax, and W_ISBeamMin in the current data folder. Use /RECT to improve efficiency for simple ROI domains. V_ variable results correspond to the last evaluated layer of the 3D wave. Do not use /G, /GS, or /P with this flag. Set /M=1 for maximum efficiency.

/BRXY={*xWave*, *yWave*}

Use this option with a 3D imageWave. It provides a more efficient method for computing average, minimum and maximum values when the set of points of interest is much smaller than the dimensions of an image.

Here *xWave* and *yWave* are 1D waves with the same number of points containing XY integer pixel locations specifying arbitrary pixels for which the statistics are calculated on a plane by plane basis as follows:

$$\mathrm{W\_ISBeamAvg[k]}=\frac{1}{\mathrm{n}}\sum_{i=1}^{n}\mathrm{Im}\,age[xWave[i]][yWave[i]].$$

Pixel locations are zero-based; non-integer entries may produce unpredictable results.

The calculated statistics for each plane are stored in the current data folder in the waves W_ISBeamAvg, W_ISBeamMax and W_ISBeamMin.

Note: This flag is not compatible with any other flag except /BEAM.

/C=*chunk*
When imageWave is a 4D wave, /C specifies the chunk for which statistics are calculated. By default *chunk* = 0.

Added in Igor Pro 7.00.

/G={*startP*, *endP*, *startQ*, *endQ*}

Specifies the corners of a rectangular ROI. When this flag is used an ROI wave is not required. This flag requires that *startP* ≤ *endP* and *startQ* ≤ *endQ*. When the parameters extend beyond the image area, the command will not execute and V_flag will be set to -1. You should therefore verify that V_flag=0 before using the results of this operation.

/GS={*sMinRow*,*sMaxRow*,*sMinCol*,*sMaxCol*}

Specifies a rectangular region of interest in terms of the scaled image coordinates. Each one of the 4 values will be translated to an integer pixel using truncation.

This flag, /G, and an ROI specification are mutually exclusive.

/M=*val*
Calculates the average and locates the minimum and the maximum in the ROI when /M=1. This will save you the computation time associated with the higher order statistical moments.

/P=*planeNumber*
Restricts the calculation to a particular layer of a 3D wave. By default, *planeNumber*= -1 and only the first layer of the wave is processed.

/R=*roiWave* Specifies a region of interest (ROI) in the image. The ROI is defined by a wave of type unsigned byte (/b/u), which has the same number of rows and columns as the image wave. The ROI itself is defined by the entries/pixels whose value are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous. See **ImageGenerateROIMask** for more information on creating ROI waves.

/RECT={*minRow, maxRow, minCol, maxCol*}

Limits the range of the ROI to a rectangular pixel range with /BEAM.

Details

The image statistics are returned via the following variables:

`V_adev`	Average deviation of pixel values.
`V_avg`	Average of pixel values.
`V_kurt`	Kurtosis of pixel values.
`V_min`	Minimum pixel value.
`V_minColLoc`	Specifies the location of the column in which the minimum pixel value was found or the first eligible column if no single column was found.
`V_minRowLoc`	Specifies the location of the row in which the minimum pixel value was found or the first eligible row if no single minimum was found.
`V_max`	Maximum pixel value.
`V_maxColLoc`	Specifies the location of the column in which the maximum pixel value was found or the first eligible column if no single column was found.
`V_maxRowLoc`	Specifies the location of the row in which the maximum pixel value was found or the first eligible row if no single maximum was found.
`V_npnts`	Number of points in the ROI excluding NaNs.
`V_rms`	Root mean squared of pixel values.
`V_sdev`	Standard deviation of pixel values.
`V_skew`	Skewness of pixel values.

Most of these statistical results are similarly defined as for the WaveStats operation. WaveStats will be more convenient to use when calculating statistics for an entire wave.

If *imageWave* is 4D it is often useful to use the reversible conversion

```
Redimension/N=(rows,cols,layers*chunks) ImageWave
```

which allows you to obtain the statistics for each layer and all chunks of the wave. To convert back to 4D, execute:

```
Redimension/N=(rows,cols,layers,chunks) ImageWave
```

See Also

The **ImageGenerateROIMask** and **WaveStats** operations. **ImageStats Operation** on page III-371.

ImageThreshold

`ImageThreshold` `[`***`flags`***`]` ***`imageMatrix`***

The ImageThreshold operation converts a grayscale *imageMatrix* into a binary image. The operation supports all data types. However, the source wave must be a 2D matrix. If *imageMatrix* contains NaNs, the pixels corresponding to NaN values are mapped into the value 64. The values for the On and Off pixels are 255 and 0 respectively. The resulting image is stored in the wave M_ImageThresh.

Flags

/C Calculates the correlation coefficient between the original image and the image generated by the threshold operation. The correlation value is printed to the history area (unless the /Q flag is specified), it is also stored in the variable V_correlation.

/I Inverts values written to the image, i.e., sets to zero all pixels above threshold.

/M= *method* Specifies the thresholding method. The calculated value will be printed to the history area (unless /Q is specified) and stored in the variable V_threshold.

method=0: Default. In thie case you must use the /T flag to specify a manually-selected threshold.

method=1: Automatically calculate a threshold value using an iterative method.

method=2: Image histogram is a simple bimodal distribution.

method=3: Adaptive thresholding. Evaluates threshold based on the last 8 pixels in each row, using alternating rows.

The output wave M_ImageThresh has the same numeric type as the input wave. In particular, when the input is signed byte, the on and off pixel values are 127 and 0 respectively.

Note that this method is not supported when used as part of the operation **ImageEdgeDetection**.

method=4: Fuzzy thresholding using entropy as the measure for “fuzziness”.

method=5: Fuzzy thresholding using a method that minimizes a “fuzziness” measure involving the mean gray level in the object and background.

method=6: Determines an ideal threshold by histograming the data and representing the image as a set of clusters that is iteratively reduced until there are two clusters left. The threshold value is then set to the highest level of the lower cluster. This method is based on a paper by A.Z. Arifin and A. Asano (see reference below) but modified for handling images with relatively flat histograms.

If the image histogram results in less than two clusters, it is impossible to determine a threshold using this method and the threshold value is set to NaN.

Added in Igor Pro 7.00.

method=7: Determines the ideal threshold value by maximizing the total variance between the “object” and the “background”. See http://en.wikipedia.org/wiki/Otsu%27s_method.

/N Sets the background level to 64 (i.e., NaN).

/O Overwrites the original image with the calculated threshold image.

If you do not specify the /O flag, the threshold image is written into the wave M_ImageThresh.

/P=*layer* When *imageMatrix* is a 3D wave /P selects a specific layer for which to compute the threshold. *layer* is the zero-based layer index.

If *layer* is -1, which is the default value, the threshold is computed for all layers of *imageMatrix*.

The /P flag is not compatible with /O.

The /P flag was added in Igor Pro 7.00.

/Q Suppresses printing calculated correlation coefficients (/C) and calculated thresholds (/M) to the history area.

/R=*roiSpec*	Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u). The ROI wave must have the same number of rows and columns as the image wave. The ROI itself is defined by the entries/pixels whose values are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous and can take any arbitrary shape. See **ImageGenerateROIMask** for more information on creating ROI waves. In general, the *roiSpec* has the form {*roiWaveName*, *roiFlag*}, where *roiFlag* can take the following values: *roiFlag*=0: Set pixels outside the ROI to 0. *roiFlag*=1: Set pixels outside the ROI as in original image. *roiFlag*=2: Set pixels outside the ROI to NaN (=64). By default *roiFlag* is set to 1 and it is then possible to use the /R flag using the abbreviated form /R=*roiWave*.
/T=*thresh*	Sets the threshold value.
/W= *Twave*	Sets the threshold intervals. Each interval is specified by a pair of values in the wave *Twave*. The first element in each pair is the low value and the second element is the high value. Pixel values that lie outside all the specified intervals are set to 0.

References

The automatic thresholding method (/M=1) is described in: T. W. Ridler and S. Calvard, *IEEE Transactions on Systems, Man and Cybernetics*, SMC-8, 630-632, 1978.

The thresholding method used with /M=6 is described in: A.Z. Arifin and A Asano, "Image segmentation by histogram thresholding using hierarchical cluster analysis", *Pattern Recognition Letters* 27 (2006) 1515-1521.

See Also

For usage examples see **Threshold Examples** on page III-356. The **ImageGenerateROIMask** and **ImageEdgeDetection** operations.

ImageTransform

`ImageTransform` `[`***`flags`***`]` ***`Method imageMatrix`***

The ImageTransform operation performs one of a number of transformations on *imageMatrix*. The result of using most keywords is a new wave stored in the current data folder. Most flags in this operation are exclusive to the keywords in which they are mentioned.

Parameters

Method selects type of transform. It is one of the following names:

averageImage	Computes an average image for a stack of images contained in 3D *imageMatrix*. The average image is stored in the wave M_AveImage and the standard deviation for each pixel is stored in the wave M_StdvImage in the current data folder. You can use this keyword together with the optional /R flag where a region of interest is defined by zero value points in a ROI wave. The operation sets to NaN the entries in M_AveImage and M_StdvImage that correspond to pixels outside the ROI. *imageMatrix* must have at least three layers.
averageRGBImage s	Computes an average image from a sequence of RGB images represented by a 4D wave. The average is computed separately for the R, G and B channels and the resulting RGB image is stored in the wave M_AverageRGBImage in the current data folder. The operation supports all real data types. Added in Igor Pro 7.00.

backProjection Reconstructs the source image from a projection slice and stores the result in the wave M_BackProjection. The projection slice should either be a wave produced by the projectionSlice keyword of this operation or a wave that would be similarly scaled. The input must be a single or double precision real 2D wave. Row scaling must range symmetrically about zero. For example, if the reconstructed image is expected to have 256 rows then the row scaling of the input should be from -128 to 127. Similarly, the column scaling of the input should range between zero and π. An equivalent implementation as a user function is provided in the demo experiment. You can use this implementation as a starting point if you want to develop filtered back projection.

See the **projectionSlice** keyword and the RadonTransformDemo experiment. For algorithm details see the chapter "Reconstruction of cross-sections and the projection-slice theorem of computerized tomography" in Born and Wolf, 1999.

ccsubdivision Performs a Catmull-Clark recursive generation of B-spline surfaces. There are two valid inputs: triangular meshes or quad meshes.

Quad meshes are assumed to be in the form of a 3D wave where the first plane contains the X-values, the second the Y-values and the third the Z-values.

Triangle meshes are much more complicated to convert into face-edge-vertex arrays so they are less desirable. They are stored in a three column (triplet) wave where the first column corresponds to the X coordinate, the second to the Y coordinate and the third to the Z coordinate. Each triangle is described by three rows in the wave and common vertices have to be repeated so that each sequential three rows in the triplet wave correspond to a single triangle. You can also associate a scalar value with each vertex and it will be suitably interpolated as new vertices are computed and old ones are shifted. In this case the input source wave contains one more column in the case of a triplet wave or one more plane in the case of a quad wave. The scalar value is added everywhere as an additional dimension to the spatial part of any point calculation.

You can specify the number of iterations using the /I flag. By default the operation executes a single iteration. The output is saved in a quad wave M_CCBSplines that consists of 4 columns. Each row corresponds to a Quad where the 3 planes contain the X, Y, and Z components. If you are using an optional scalar in the input, the scalar result is stored in the wave M_CCBScalar.

In some situations you may encounter spatial degeneracies when two or more parts of the surface meet at common vertices. This leads to one or more error messages in the history area of the command window. You can try to break a degeneracy yourself by adding a small perturbation to the input coordinates or you can use the /PRTF flag which adds a random perturbation on the order of 1e-10 times the extent of the data in each dimension. The perturbation does not affect the scalar.

cmap2rgb Converts an image and its associated colormap wave (specified using the /C flag) into an RGB image stored in a 3D wave M_RGBOut.

CMYK2RGB Converts a CMYK image, stored as 4 layer unsigned byte wave, into a 3 layer, standard RGB image wave. The output wave is M_CMYK2RGB that is stored in the current data folder.

compress Compresses the data in the *imageWave* using a nonlossy algorithm and stores it in the wave W_Compressed in the current data folder. The compressed wave includes all data associated with *imageWave* including its units and wavenote. Use the decompress keyword to recover the original wave. The operation supports all numeric data types.

NOTE: The compression format for waves greater than 2GB in size was changed in version 6.30B02. If you compressed a wave greater than 2 GB in IGOR64 6.30B01, you will need to decompress it using the same version. You can not decompress it in 6.30B02 or later.

convert2gray Converts an arbitrary 2D wave into an 8-bit normalized 2D wave. The default output wave name is M_Image2Gray.

decompress — Decompresses a compressed wave. It saves a copy of the decompressed wave under the name W_DeCompressed in the current data folder.

NOTE: The compression format for waves greater than 2GB in size was changed in version 6.30B02. If you compressed a wave greater than 2 GB in IGOR64 6.30B01, you will need to decompress it using the same version. You can not decompress it in 6.30B02 or later.

distance — Computes the distance transform, also called "distance map", for the input image. Added in Igor Pro 7.00.

The distance transform/map is an image where every pixel belonging to the "object" is replaced by the shortest distance of that pixel from the background/boundary.

imageMatrix must be a 2D wave of type unsigned byte (Make/B/U) where pixels corresponding to the object are set to zero and pixels corresponding to the background are set to 255. Such an image can be obtained, for example, using **ImageThreshold**.

The resulting distance map is stored in the wave M_DistanceMap in the current data folder. The operation supports three metrics (Manhattan, Euclidean, Euclidean + scaling) set via the /METR flag.

extractSurface — Extracts values corresponding to a plane that intersects a 3D volume wave (*imageMatrix*). You must specify the extraction parameters using the /X flag. The volume is defined by the wave scaling of the 3D wave. The result, obtained by trilinear interpolation, is stored in the wave M_ExtractedSurface and is of the type NT_FP64. Points in the plane that lie outside the volume are set to NaN.

fht — Performs a Fast Hartley Transform subject to the /T flag. The source wave must be a 2D real matrix with a power of 2 number of rows and columns. Default output is saved in the double-precision wave M_Hartley in the current data folder. If you use the /O flag the result overwrites *imageMatrix* without changing the numeric type. If *imageMatrix* is single-precision float the conversion is straightforward. All other numeric types are scaled. Single- and double-byte types are scaled to the full dynamic range. 32 bit integers are scaled to the range of the equivalent 16 bit types (i.e., unsigned int is scaled to unsigned short range etc.). It does not support wave scaling or NaN entries.

fillImage — Fills a 2D target image wave with data from a 1D image wave (specified using /D). Both waves must be the same data type, and the number of data points in the target wave must match the number of points in the data wave. There are four fill modes that are specified via the /M flag. The operation supports all noncomplex numeric data types.

findLakes — Originally intended to identify lakes in geographical data, this operation creates a mask for a 2D wave for all the contiguous points whose values are close to each other. You can determine the minimum number of pixels within a contiguous region using the /LARA=*minPixels* flag (default is 100). You can determine how close values must be in order to belong to a contiguous region using the /LTOL=*tolerance* flag (default is zero). You can also limit the search region using the /LRCT flag. Use the flag /LTAR=*target* to set the value of the masked regions. By default, the algorithm uses 4-connectivity when looking at adjacent pixels. You can set it to 8-connectivity using the /LCVT flag. The result of the operation is saved in the wave M_LakeFill. It has the same data type as the source wave and contains all the source values outside the masked pixels.

flipCols — Rearrange pixels by exchanging columns symmetrically about a center column or the center of the image (if the number of columns is even). The exchange is performed in place and can be reverted by repeating the operation. When working with 3D waves, use the /P flag to specify the plane that you want to operate on.

flipRows	Rearrange pixels in the image by exchanging rows symmetrically about the middle row or the middle of the image (if the image has an even number of rows). The exchange is performed in place and can be reverted by repeating the operation. When working with 3D waves, use the /P flag to specify the plane that you want to operate on.

flipPlanes	Rearrange data in a 3D wave by exchanging planes symmetrically about the middle plane. The operation is performed in place and can be reverted by repeating the operation.

fuzzyClassify	Segments grayscale and color images using fuzzy logic. Iteration stops when it reaches convergence defined by /TOL or the maximum number of iterations specified by /I. It is a good practice to specify the tolerance and the number of iterations. If the number of classes is small, the operation prints the class values in the history. Use /Q to eliminate printing and increase performance. Use /CLAS to specify the number of classes and optionally use /CLAM to modify the fuzzy probability values. Use /SEG to generate the segmentation image. The classes are stored in the wave W_FuzzyClasses in the current data folder and it will be overwritten if it already exists.

When *imageMatrix* is a grayscale image, each class is a single wave entry. When *imageMatrix* is a RGB image, classes are stored consecutively in the wave W_FuzzyClasses. If you request more classes than are present in *imageMatrix*, you will likely find a degeneracy where the space of a data class is spanned by more than one class. It is a good idea to compute the Euclidean distance between every possible pair of classes and eliminate degeneracies when the distance falls below some threshold.

Any real data type is allowed but values in the range [0,255] are optimal. You can segment 3D waves of more than 3 layers in which a class will be a vector of dimensionality equal to the number of layers in *imageMatrix*.

For examples see Examples/Imaging/fuzzyClassifyDemo.pxp.

getBeam	Extracts a beam from a 3D wave.

A “beam” is a 1D array in the Z-direction. If a row is a 1D array in the first dimension and a column is a 1D array in the second dimension then a beam is a 1D array in the third dimension.

The number of points in a beam is equal to the number of layers in *imageWave*. Specify the beam with the /BEAM={*row*,*column*} flag. It stores the result in the wave W_Beam in the current data folder. W_Beam has the same numeric type as *imageWave*. Use **setBeam** to set beam values. (See also, **MatrixOp** beam.)

getChunk	Extracts the chunk specified by chunk index /CHIX from *imageMatrix* and stores it in the wave M_Chunk in the current data folder. For example, if *imageMatrix* has the dimensions (10,20,3,10), the resulting M_Chunk has the dimensions (10,20,3). See also **setChunk** and **insertChunk**.

getCol	Extracts a 1D wave, named W_ExtractedCol, from any type of 2D or 3D wave. You specify the column using the /G flag. For a 3D source wave, it will use the first plane unless you specify a plane using the /P flag. *imageMatrix* can be real or complex. (See also putCol keyword, **MatrixOp** col.)

getPlane	Creates a new wave, named M_ImagePlane, that contains data in the plane specified by the /P flag. The new wave is of the same data type as the source wave. You can specify the type of plane using the /PTYP flag. (See also setPlane keyword, **MatrixOp**.)

getRow	Extracts a 1D wave, named W_ExtractedRow, from any type of 2D or 3D wave. You specify the row using the /G flag. For a 3D source wave, it will use the first plane unless you specify a plane using the /P flag. *imageMatrix* can be real or complex. (See also setRow keyword, **MatrixOp** row.)

Hough

Performs the Hough transform of the input wave. The result is saved to a 2D wave M_Hough in which the columns correspond to angle and the rows correspond to the radial domain.

By default the output consists of 180 columns. Use the /F flag to modify the angular resolution.

If the input image has N rows and M columns then the number of rows in the M_Hough is set to 1+sqrt(N^2+M^2). The output radius should be read relative to the center row.

It is assumed that the input wave has square pixels of unit size and that is binary (/B/U) where the background value is 0.

See also **Hough Transform** on page III-364.

hsl2rgb

Transforms a 3-plane HSL wave into a 3-plane RGB wave. If the source wave for this operation is of any type other than byte or unsigned byte, the HSL values are expected to be between 0 and 65535. For all source wave types the resulting RGB wave is of type unsigned short. The result of the operation is the wave M_HSL2RGB (of type unsigned word), where the RGB values are in the range 0 to 65535.

hslSegment

Creates a binary image of the same dimensions as the source image, in which all pixels that belong to all three of the specified Hue, Saturation, and Lightness ranges are set to 255 and the others to zero. You can specify the HSL ranges using the /H, /S, and /L flags. Each flag takes as an argument either a pair of values or a wave containing pairs of values. You must specify the /H flag but you can omit the /S and /L flags in which case the default values (corresponding to full range 0 to 1) are used. *imageMatrix* is assumed to be an RGB image.

imageToTexture

Transforms a 2D or 3D image wave into a 1D wave of contiguous pixel components. The transformation is useful for creating an OpenGL texture (for Gizmo) or for saving a color image in a format requiring either RGB or RGBA sequences.

The /O flag does not apply to imageToTexture.

Use the /TEXT flag to specify the type of transformation. *imageMatrix* must be an unsigned byte wave. A 1D unsigned byte wave named W_Texture is created in the current data folder.

W_Texture's wave note is set to a semicolon-separated list of keyword -value pairs that can be parsed using **StringByKey** and **NumberByKey**:

Keyword	Information Following Keyword
WIDTHPIXELS	**DimSize**(imageMatrix,0) or truncated to nearest power of 2 if /TEXT value is odd
HEIGHTPIXELS	**DimSize**(imageMatrix,1) or truncated to nearest power of 2
LAYERS	**DimSize**(imageMatrix,2)
TEXTUREMODE	val parameter from /TEXT flag
SOURCEWAVE	**GetWavesDataFolder**(imageMatrix, 2)

indexWave

Creates a 1D wave W_IndexedValues in the current data folder containing values from *imageMatrix* that are pointed to by the index wave (see /IWAV). Each row in the index wave corresponds to a single value of *imageMatrix*. If any row does not point to a valid index (within the dimensions of *imageMatrix*), the corresponding value is set to zero and the operation returns an error. Indices are zero based integers; the operation does not support interpolation and ignores wave scaling.

insertChunk

Inserts a chunk (a 3D wave specified by the /D flag) into *imageMatrix* at chunk index specified by the /CHIX flag. The dimensions of the inserted chunk must match the first three dimensions of *imageMatrix*. The wave must also have the same numeric type. The 4th dimension of *imageMatrix* is incremented by 1 to accommodate the new data. See also **getChunk** and **setChunk**.

insertImage

Inserts the image specified by the flag /INSI into *imageMatrix* starting at the position specified by the flags /INSX and /INSY. If the *imageMatrix* is a 3D wave then it inserts the image in the layer specified by the /P flag. The inserted image and *imageMatrix* must be the same data type. The inserted data is clipped to the boundaries of *imageMatrix*.

insertXplane

Inserts a 2D wave as a new plane perpendicular to the X-axis in a 3D wave. The /P flag specifies the insertion point and the /INSW flag specifies the inserted wave. The 2D wave must be the same numeric data type as the 3D wave and its dimensions must be cols x layers of the 3D wave. For example, if the 3D wave has the dimensions (10x20x30) the 2D wave must be 20x30. If you do not use the /O flag, it stores the result in the wave M_InsertedWave in the current data folder.

insertYplane

Inserts a 2D wave as a new plane perpendicular to the Y-axis in a 3D wave. The /P flag specifies the insertion point and the /INSW flag specifies the inserted wave. The 2D wave must be the same numeric data type as the 3D wave and its dimensions must be rows x layers of the 3D wave. For example, if the 3D wave has the dimensions (10x20x30) the 2D wave must be 10x30. If you do not use the /O flag, it stores the result in the wave M_InsertedWave in the current data folder.

insertZplane

Inserts a 2D wave as a new plane perpendicular to the Z-axis in a 3D wave. The /P flag specifies the insertion point and the /INSW flag specifies the inserted wave. The 2D wave must be the same numeric data type as the 3D wave and its dimensions must be rows x cols of the 3D wave. For example, if the 3D wave has the dimensions (10x20x30) the 2D wave must be 10x20. If you do not use the /O flag, it stores the result in the wave M_InsertedWave in the current data folder. This keyword is included for completeness. You can accomplish the same task using InsertPoints.

invert

Converts pixel values using the formula `newValue=255-oldValue`. Works on waves of any dimension, but only on waves of type unsigned byte. The result is stored in the wave M_Inverted unless specifying the /O flag.

matchPlanes

Finds pixels that match test conditions in all layers of a 3D wave. It creates a 2D unsigned byte output wave, M_matchPlanes, that is set to the values 0 and 255. A value of 255 indicates that the corresponding pixel has satisfied test conditions in all layers of the wave for which conditions were provided. Otherwise the pixel value is 0.

Test conditions are entered as a 2D wave using the /D flag. The condition wave must be double precision and it must contain the same number of columns as the number of layers in the 3D source wave. A condition for layer j of the source wave is specified by two rows in column j of the condition wave. The first row entry, say *A*, and the second row entry, say *B*, imply a condition on pixels in layer *j* such that $A \le x < B$. You can have more than one condition for a given layer by adding pairs of rows to the condition wave. For example, if you add in consecutive rows the values *C* and *D*, this implies the test:

$$(A \le x \le B) || (C \le x \le D).$$

If you do not have any conditions for some layer, set its corresponding condition column to NaN. Similarly, if you have two conditions for the first layer and one condition for the second layer, pad the bottom of column 1 in the condition wave with NaNs. See Examples for use of this keyword to perform hue/saturation segmentation.

offsetImage

Shifts an image in the XY plane by dx, dy pixels (specified by the /IOFF flag). Pixels outside the shifted image will be set to the specified background value. The operation works on 2D waves or on 3D waves with the optional /P flag. When shifting a 3D wave with no specified plane, it creates a 3D wave with all planes offset by the same amount. The wave M_OffsetImage contains the result in the current data folder.

The /O flag is not supported with offsetImage.

padImage

Resizes the source image. When enlarged, values from the last row and column fill in the new area. The /N flag specifies the new image size in terms of the rows and columns change. The /W flag specifies whether data should be wrapped when padding the image. Unless you use the /O flag, the result is stored in the wave M_PaddedImage in the current data folder.

projectionSlice

Computes a projection slice for a parallel fan of rays going through the image at various angles. For every ray in the fan the operation computes a line integral through the image (equivalent to the sum of the line profile along the ray). The operation computes the line integrals for multiple fans defined by the number and position of the rays as well as the angle that they make with the positive X-direction. Use the /PSL flag to specify the projection parameters. The projection slice itself is stored in a 2D wave M_ProjectionSlice where the rows correspond to the rays and the columns correspond to the selected range of angles. The operation does not support wave scaling. If the source wave is 3D the projection slice currently supports slices that are perpendicular to the z-axis and specified by their plane number.

See the **backProjection** keyword and the RadonTransformDemo experiment. For algorithm details see the chapter “Reconstruction of cross-sections and the projection-slice theorem of computerized tomography” in Born and Wolf, 1999.

putCol

Sets a column of *imageMatrix* to the values in the wave specified by the /D flag. Use the /G flag to specify column number and the /P flag to specify the plane. Note that if there is a mismatch in the number of entries between the specified waves, the operation uses the smaller number. See also getCol keyword.

putRow

Sets a row of *imageMatrix* to the values in the wave specified by the /D flag. Use the /G flag to specify column number and the /P flag to specify the plane. Note that if there is a mismatch in the number of entries between the specified waves, the operation uses the smaller number. See also getRow keyword.

removeXplane

Removes one or more planes perpendicular to the X-axis from a 3D wave. The /P flag specifies the starting position. By default, it removes a single plane but you can remove more planes with the /NP flag. If you do not use the /O flag, it saves the result in the wave M_ReducedWave in the current data folder.

removeYplane

Removes one or more planes perpendicular to the Y-axis from a 3D wave. The /P flag specifies the starting position. By default, it removes a single plane but you can remove more planes with the /NP flag. If you do not use the /O flag, it saves the result in the wave M_ReducedWave in the current data folder.

removeZplane

Removes one or more planes perpendicular to the Z-axis from a 3D wave. The /P flag specifies the starting position. By default, it removes a single plane but you can remove more planes with the /NP flag. If you do not use the /O flag, it saves the result in the wave M_ReducedWave in the current data folder.

rgb2cmap

Computes a default color map of 256 colors to represent the input RGB image. The colors are computed by clustering the input pixels in RGB space. The resulting color map is stored in the wave M_ColorIndex in the current data folder. The operation also saves the wave M_IndexImage which contains an index into the colormap that can be used to display the image using the commands:

```
NewImage M_IndexImage
ModifyImage M_IndexImage cindex= M_ColorIndex
```

To change the default number of colors use the /NCLR flag. When the number of colors are greater than 256, M_IndexImage will be a 16-bit unsigned integer or a 32 bit integer wave depending on the number. rgb2cmap supports input images in the form of 3D waves of type unsigned byte or single precision float. The floating point option may be used to input images in colorspaces that use signed numeric data.

rgb2gray

When the input *imageMatrix* is a 3D RGB wave, rgb2gray produces a 2D wave of type unsigned byte containing the grayscale representation of the input. By default, the operation stores the output in the wave M_RGB2Gray in the current data folder. The RGB values are converted into the luminance Y of the YIQ standard using:

```
Y = 0.299R + 0.587G + 0.114B
```

When the input imageMatrix is a 4D wave containing multiple (3 layer) RGB chunks, the conversion produces a 3D wave where each layer corresponds to the grayscale conversion of the corresponding chunk in the input wave. In this case the numeric type of the output is the same as that of the input but the conversion formula is the same. The /O flag is not supported when transforming a 4D RGB wave.

rgb2hsl

Converts an RGB image stored in a 3D wave into another 3D wave in which the three planes correspond to Hue, Saturation and Lightness in the HSL color model. Values of all components are normalized to the range 0 to 255 unless the /U flag is used or if the source wave is not 8-bit, in which case the range is 0 to 65535. The default output wave name is M_RGB2HSL.

rgb2i123

Performs a colorspace conversion of an RGB image into the following quantities:

$$\begin{aligned} I_1 &= \left|R-G\right|D \\ I_2 &= \left|R-B\right|D \quad \text{where } D = \frac{255}{\left|R-G\right|+\left|R-B\right|+\left|G-B\right|}. \\ I_3 &= \left|G-B\right|D, \end{aligned}$$

*I*1, *I*2, and *I*3 are stored in the wave M_I123 (using the same data type as the original RGB wave) in the current data folder. *I*1 is stored in the first layer, *I*2 in the second and *I*3 in the third. This color transformation is said to have useful applications in machine vision.

For more information see: Gevers and Smeulders (1999).

rgb2xyz

Converts a 3D RGB image wave into a 3D wave containing the XYZ color space equivalent. The conversion is based on the D65 white point and uses the following transformation:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}.$$

The XYZ values are stored in a wave named "M_RGB2XYZ" unless the /O flag is used, in which case the source image is overwritten and converted into single precision wave (NT_FP32).

roiTo1D

Copies all pixels in an ROI and saves them sequentially in a 1D wave. The ROI is specified by /R. The ROI wave must have the same dimensions as *imageMatrix*. If *imageMatrix* is a 3D wave, the ROI must have as many layers as *imageMatrix*. The wave W_roi_to_1d contains the output in the current data folder, has the same numeric type as *imageMatrix*, and contains the selected pixels in a column-major order.

rotateCols

Rotates rows in place. This operation is analogous to the **Rotate** operation except that it works on images and rotates an integer number of rows. The number of rows is specified by the /G flag.

When *imageMatrix* contains multiple layers you can use the /P flag to specify the layer of the wave that will undergo rotation. By default, if you do not specify the /P flag and if *imageMatrix* consists of three layers (RGB), then all three layers are rotated. Otherwise the operation rotates only the first layer of the wave.

rotateRows

Rotates columns in place. This operation is analogous to the **Rotate** operation except that it works on images and rotates an integer number of columns. The number of columns is specified by the /G flag.

When *imageMatrix* contains multiple layers you can use the /P flag to specify the layer of the wave that will undergo rotation. By default, if you do not specify the /P flag and if *imageMatrix* consists of three layers (RGB), then all three layers are rotated. Otherwise the operation rotates only the first layer of the wave.

scalePlanes

Scales each plane of the 3D wave, *imageMatrix*, by a constant taken from the corresponding entry in the 1D wave specified by the /D flag. The result is stored in the wave M_ScaledPlanes unless the /O flag is specified, in which case scaling is done in place.

When using /O, first redimension the wave to a different data type to make sure there are no artifacts due to type clipping.

If *imageMatrix* is double precision, M_ScaledPlanes is double precision. Otherwise M_ScaledPlanes is single precision.

This operation also supports the optional flag.

Note that when you display M_ScaledPlanes, which has three planes that originated from scaling byte data, you will have to multiply the wave by 255 to see the image because the RGB format for single and double precision data requires values in the range 0 to 65535.

selectColor

Creates a mask for the image in which pixel values depend on the proximity of the color of the image to a given central color. The central color, the tolerance and a grayscale indicator must be specified using the /E flag.

For example, `/E={174,187,75,10,1}` specifies an RGB of (174,187,75), a tolerance level of 10 and a requested grayscale output.

RGB values must be provided in a range appropriate for the source image. If the source wave type is not byte or unsigned byte, then the range of the RGB components should be 0 to 65535.

The color proximity is calculated in the nonuniform RGB space and the tolerance applies to the maximum component difference from the corresponding component of the central color.

The tolerance, just like the central color, should be appropriate to the type of the source wave.

The generated mask is stored in the wave M_SelectColor in the current data folder. If a wave by that name exists prior to the execution of this operation, it is overwritten. You can also use the /R flag with this operation to limit the color selection to pixels in the ROI wave whose value is zero.

setBeam

Sets the data of a particular beam in *imageMatrix*.

A “beam” is a 1D array in the Z-direction. If a row is a 1D array in the first dimension and a column is a 1D array in the second dimension then a beam is a 1D array in the third dimension.

Specify the beam with the /BEAM={*row*,*column*} flag and the 1D beam data wave with the /D flag. The beam data wave must have the same number of elements as the number of layers and same numeric type as *imageMatrix*. Use **getBeam** to extract the beam.

setChunk

Overwrites the data in the wave *imageMatrix* at chunk index specified by /CHIX with the data contained in a 3D wave specified by the /D flag. The assigned data must be contained in a wave that matches the first three dimensions of *imageMatrix* and must have the same number type. See also **getChunk** and **insertChunk**.

setPlane

Sets a plane (given by the /P flag) in the designated image with the data in a wave specified by the /D flag. It is designed as a complement of the **getPlane** keyword to provide an easier (faster) way to create multiplane images. Note that the operation supports setting a plane when the source data is smaller than the destination plane in which case the source data is placed in memory contiguously starting from the corner pixel of the destination plane. If the source data is larger than the destination plane it is clipped to the appropriate rows and columns. If you are setting all planes in the destination wave using algorithmically named source waves you could use the **stackImages** keyword instead. See also **getPlane** keyword.

shading

Calculates relative reflectance of a surface for a light source position defined by the /A flag.

The operation estimates the slope of the surface and then computes a relative reflectance defined as the dot product of the direction of the light and the normal to the surface at the point of interest. Reflectivity is scaled using the expression:

outPixel = *shadingA* * (*sunDirection* · *surfaceNormal*) + *shadingB*

By default *shadingA*=1, *shadingB*=0.

The result is stored in the wave M_ShadedImage, which has the same data type as the source wave.

If the source wave is any integer type, and the value of *shadingA*=1 the operation sets that value to 255.

The smallest supported wave size is 4x4 elements.

Values along the boundary (1 pixel wide) are arbitrary because there are no derivatives calculable for those pixels, so these pixels are filled with duplicates of the inner rows and columns.

shrinkBox

Shrinks 3D imageMatrix to include only the minimum three dimension rectangular range that contains all the voxels whose values are different from an outer value. The outer value is specified with the /F flag. This feature is useful in situations where ImageSeedFill has set the voxels around an object of interest to some outer value and it is desired to extract the smallest box that contains interesting data. The output is stored in the wave M_shrunkBox.

Added in Igor Pro 7.00.

shrinkRect

Shrinks *imageMatrix* to include only the minimum rectangle that contains all the pixels whose value is different from an outer value. The outer value is specified with the /F flag. This is useful in situations where **ImageSeedFill** has set the pixels around the object of interest to some outer value and it is desired to extract the smallest rectangle that contains interesting data. The output is stored in the wave M_Shrunk.

stackImages

Creates a 3D or 4D stack from individual image waves in the current data folder. The waves should be of the form *baseNameN*, where *N* is a numeric suffix specifying the sequence order. *imageMatrix* should be the name of the first wave that you want to add to the stack. You can use the /NP flag to specify the number of waves that you want to add to the stack.

The result is a 3D or 4D wave named M_Stack, which overwrites any existing wave of that name in the current data folder.

With /K, it kills all waves copied into the stack.

sumAllCols

Creates a wave W_sumCols in which every entry is the sum of the pixels on the corresponding image column. For a 3D wave, unless you specify a plane using the /P flag it will use the first plane by default.

sumAllRows

Creates a wave W_sumRows in which every entry is the sum of the pixels on the corresponding image row. For a 3D wave, unless you specify a plane using the /P flag it will use the first plane by default.

sumCol

Stores in the variable V_value the sum of the elements in the column specified by /G flag and optionally the /P flag.

sumPlane

Stores in the variable V_value the sum of the elements in the plane specified by the /P flag.

sumPlanes

Creates a 2D wave M_SumPlanes which contains the same number of rows and columns as the 3D source wave. Each entry in M_SumPlanes is the sum of the corresponding pixels in all the planes of the source wave. M_SumPlanes is a double precision wave if the source wave is double precision. Otherwise it is a single precision wave.

sumRow

Stores in the variable V_value the sum of the elements of a row specified by /G flag and optionally the /P flag.

swap

Swaps image data following a 2D FFT. The transform swaps diagonal quadrants of the image in one or more planes. This keyword does not support any flags. The swapping is done in place and it overwrites the source wave.

swap3D

Swaps data following a 3D FFT. The transform swaps diagonal quadrants of the data. This keyword does not support any flags. The swapping is done in place and the source wave is overwritten.

transpose4D

Converts a 4D wave into a new 4D wave where the data are reordered by dimensions specified by the /TM4D flag. The results are stored in the wave M_4DTranspose in the current data folder. There is no option to overwrite the input wave so /O has no effect with transpose4D.

Added in Igor Pro 7.00.

transposeVol

Transposes a 3D wave. The transposed wave is stored in M_VolumeTranspose. The /O flag does not apply. The operation supports the following 5 transpose modes which are specified using the /G flag:

mode	**Equivalent Command**
1	`M_VolumeTranspose=`*`imageMatrix`*`[p][r][q]`
2	`M_VolumeTranspose=`*`imageMatrix`*`[r][p][q]`
3	`M_VolumeTranspose=`*`imageMatrix`*`[r][q][p]`
4	`M_VolumeTranspose=`*`imageMatrix`*`[q][r][p]`
5	`M_VolumeTranspose=`*`imageMatrix`*`[q][p][r]`

vol2surf

Creates a quad-wave output (appropriate for display in Gizmo) that wraps around 3D “particles”. A particle is defined as a region of nonzero value voxels in a 3D wave. The algorithm effectively computes a box at the resolution of the input wave which completely encloses the data. The output wave M_Boxy is a 2D single precision wave of 12 columns where each row corresponds to one disjoint quad and the columns provide the sequential X, Y, and Z coordinates of the quad vertices.

voronoi

Computes the voronoi tesselation of a convex domain defined by the X, Y positions of the input wave. *imageMatrix* must be a triplet wave where the first column contains the X-values, the second column contains the Y-values and the third column is an arbitrary (zero is recommended) constant. The result of the operation is stored in the two column wave M_VoronoiEdges which contains sequential edges of the Voronoi polygons. Edges are separated from each other by a row of NaNs. The outer most polygons share one or more edges with a large triangle which contains the convex domain.

The operation creates two output waves:

M_Circles is a three column wave containing the center and radius of each natural neighborhood. The radius is correct only if equal scaling is applied.

M_DelaunayTriangles consists of 3 columns for the vertices of the triangles. Each triangle consists of 4 vertices. The 4th vertex is equal to the first. An extra row of NaNs used as a separator.

For an example, see Voronoi Tesselation Example below.

xProjection

Computes the projection in the X-direction and stores the result in the wave M_xProjection. See **zProjection** for more information.

xyz2rgb

Converts a 3D single precision XYZ color-space data into RGB based on the D65 white point. The transformation used is:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

If you do not specify the /O flag, the results are saved in a single precision 3D wave (NT_FP32) “M_XYZ2RGB”.

Note that not all XYZ values map into positive RGB triplets (consider colors that reside outside the RGB triangle in the XYZ diagram). This operation gives you the following choices: by default, the output wave is a single precision wave that will include possible negative RGB values. If you specify the /U flag, for unsigned short output wave, the operation will set to zero all negative components and scale the remaining ones in the range 0 to 65535.

yProjection

Computes the projection in the Y-direction and stores the result in the wave M_yProjection. See **zProjection** for more information.

zDot

Computes the dot product of a beam in *srcWave* with a 1D zVector wave (specified with the /D flag). This will convert stacked images of spectral scans into RGB or XYZ depending on the scaling in zVector. The *srcWave* and zVector must be the same data type (float or double). The wave M_StackDot contains the result in the current data folder.

zProjection

Computes the projection in the Z-direction and stores the result in the wave M_zProjection. The source wave must be a 3D wave of arbitrary data type. The value of the projection depends on the method specified via the /METH flag.

Flags

/A={*azimuth*, *elevation* [, *shadingA*, *shadingB*]}

Specifies parameters for shade. Position of the light source is given by *azimuth* (measured in degrees counter-clockwise) and *elevation* (measured in degrees above the horizon).

The parameters *shadingA* and *shadingB* are optional. By default their values are 1 and 0, respectively.

/Beam={*row*,*column*}

Designates a beam in a 3D wave; both *row* and *column* are zero based.

/BPJ={*width*,*height*}

Specifies the **backProjection** parameters: *width* and *height* are the width and height of the reconstructed image and should be equal to the size of the original wave.

/C=*CMapWave*

Specifies the colormap wave for **cmap2rgb** keyword. The *CMapWave* is expected to be a 2D wave consisting of three columns corresponding to the RGB entries.

/CHIX=*chunkIndex*

Identifies the chunk index for getting, inserting or setting a chunk of data in a 4D wave. *chunkIndex* ranges from 0 to the number of chunks in *imageMatrix*.

/CLAM=*fuzzy*

Sets the value used to compute the fuzzy probability in **fuzzyClassify**. It must satisfy *fuzzy* > 1 (default is 2).

/CLAS=*num*

Sets the number of requested classes in **fuzzyClassify**. If you don't know the number of expected classes and *num* is too high, fuzzyClassify will likely produce some degenerate classes.

/D=*waveName*

Specifies a data wave. Check the appropriate keyword documentation for more information about this wave.

/F=*value*

Increases the sampling in the angle domain when used with the Hough keyword. By default *value*=1 and the operation results in single degree increments in the interval 0 to 180, and if *value*=1.5 there will be 180*1.5 rows in the transform.

Specifies the outer pixel value surrounding the region of interest when used with **shrinkRect** keyword.

/FEQS

When performing Voronoi tesselation, forces the algorithm to use equal scaling for both axes. Normally the scaling for each axis is determined from its range. When the ranges of the two axes are different by an order of magnitude or more, it is helpful to force the larger range to be used for scaling both axes.

/G=*colNumber*

Specifies either the row or column number used in connection with **getRow** or **getCol** keywords. This flag also specifies the transpose *mode* with the **transposeVol** keyword.

/H={*minHue*, *maxHue*}

/H=*hueWave*

Specifies the range of hue values for selecting pixels. The hue values are specified in degrees in the range 0 to 360. Hue intervals that contain the zero point should be specified with the higher value first, e.g., `/H={330,10}`.

Use *hueWave* when you have more than one pair of hue values that bracket the pixels that you want to select.

See **HSL Segmentation** on page III-374 for an example.

/I=*iterations*

Sets the number of iterations in ccsubdivision and in **fuzzyClassify**.

/INSI=*imageWave*

Specifies the wave, *imageWave*, to be used with the insertImage keyword. *imageWave* is a 2D wave of the same numeric data type as *imageMatrix*.

/INSW=*wave*

Specifies the 2D wave to be inserted into a 3D wave using the keywords: **insertXplane**, **insertYplane**, or **insertZplane**.

/INSX=*xPos*

Specifies the pixel position at which the first row is inserted. Ignores wave scaling.

/INSY=*yPos*

Specifies the pixel position at which the first column is inserted. Ignores wave scaling.

/IOFF={*dx*,*dy*,*bgValue*} Specifies the amount of positive or negative integer offset with *dx* and *dy* and the new background value, *bgValue*, with the offsetImage keyword.

/IWAV=*wave* Specifies the wave which provides the indices when used with the keyword **indexWave**. The wave should have as many columns as the dimensions of *imageMatrix* (2, 3, or 4). For example, to specify indices for pixels in an image, the wave should have two columns. The first column corresponds to the row designation and the second to the column designation of the pixel. The wave can be of any number type (other than complex) and entries are assumed to be integer indices; there is no support for interpolation or for wave scaling.

/L={*minLight, maxLight*}

/L=*lightnessWave*

Specifies the range of lightness for selecting pixels. The lightness values are in the range 0-1. If you do not use the /L flag than the default full range is used.

Use *lightnessWave* when you have more than one pair of lightness values corresponding to the pixels that you want to select. For each pair, values should be arranged so that the smaller one is first and the larger is second. There is no restriction on the order of pairs in the wave except that they match the other waves used by the operation.

/LARA=*minPixels* Specifies the minimum number of pixels required for an area to be masked by the **findLakes** keyword. If you do not specify this flag, the default value used is 100.

/LCVT Use 8-connectivity instead of 4-connectivity.

/LRCT={*minX*,*minY*,*maxX*,*maxY*}

Sets the rectangular region of interest for the **findLakes** keyword. The operation will not affect the original data outside the specified rectangle. The X and Y values are the scaled values (i.e., using wave scaling).

/LTAR=*target* Set the target value for the masked region in the **findLakes** keyword.

/LTOL=*tol* Specifies the tolerance for the **findLakes** keyword. By default the tolerance is zero. The tolerance must be a positive number. The operation uses the tolerance by requiring neighboring pixels to have a value between that of the current pixel V and V+*tol*.

/M=*n* Specifies the method by which a 2D target image is filled with data from a 1D wave using the **fillImage** keyword.

n =0: Straight column fill, which you can also accomplish by a redimension operation.

n=1: Straight row fill.

n=2: Serpentine column fill. The points from the data wave are sequentially loaded onto the first column and continue from the last to the first point of the second column, and then sequentially through the third column, etc.

n=3: Serpentine row fill.

/METH=*method* Determines the values of the projected pixels for **xProjection**, **yProjection**, and **zProjection** keywords.

method=1: Pixel (i,j) in M_zProjection is assigned the maximum value that (i,j,*) takes among all layers of *imageMatrix* (default).

method=2: Pixel (i,j) in M_zProjection is assigned the average value that (i,j,*) takes among all layers of *imageMatrix*.

method=3: Pixel (i,j) in M_zProjection is assigned the minimum value that (i,j,*) takes among all layers of *imageMatrix*.

/METR=*method* Sets the metric used by the distance transform. Added in Igor Pro 7.00.

method=0:	Manhattan distance. Default.
method=1:	Euclidean distance where the distance is measured between centers of pixels (assumed square).
method=2:	Euclidean distance that also takes into account actual pixel size using the input's wave scaling.

method=0 executes faster than the other methods. *method*=2 can be 2x slower especially if different scaling is applied along the two axes.

/N={*rowsToAdd, colsToAdd*}

Creates an image that is larger or smaller by *rowsToAdd*, *colsToAdd*. The additional pixels are set by duplicating the values in the last row and the last column of the source image.

/NCLR=*M* Specifies the maximum number of colors to find with the rgb2cmap keyword. *M* must be a positive number; the default value is 256 colors.

The result of the operation is saved in the wave M_paddedImage.

/NP=*numPlanes* Specifies the number of planes to remove from a 3D wave when using the removeXplane, removeYplane, or removeZplane keywords. Specifies the number of waves to be added to the stack with the stackImages keyword.

/O Overwrites the input wave with the result except in the cases of **Hough** transform and **cmap2rgb**. Does not apply to the transposeVol parameter.

/P=*planeNum* Specifies the plane on which you want to operate with the **rgb2gray** or **getPlane** keywords. Also used for **getRow** or **getCol** if the source wave is 3D.

/PSL={*xStart*,*dx*,*Nx*,*aStart*,*da*,*Na*}

Specifies projection slice parameters. *xStart* is the first offset of the parallel rays measured from the center of the image. *dx* is the directed offset to the next ray in the fan and *Nx* is the number of rays in the fan. *aStart* is the first angle for which the projection is calculated. The angle is measured between the positive X-direction and the direction of the ray. *da* is the offset to the next angle at which the fan of rays is rotated and *Na* is the total number of angles for which the projection is computed.

/PTRF Use /PTRF with the ccsubdivision keyword to apply small perturbation to the input coordinates in order to break spatial degeneracies that may occur when multiple facets meet at some point in space. See ccsubdivision above for details. /PTRF was added in Igor Pro 9.00.

/PTYP=*num* Specifies the plane to use with the **getPlane** keyword.

num=0:	XY plane.
num=1:	XZ plane.
num=2:	YZ plane.

/Q Quiet flag. When used with the **Hough** transform, it suppresses report to the history of the angle corresponding to the maximum.

/R=*roiWave* Specifies a region of interest (ROI) defined by *roiWave*. For use with the keywords: **averageImage**, **scalePlanes** and **selectColor**.

/S={*minSat, maxSat*}

/S=*saturationWave* Specifies the range of saturation values for selecting pixels. The saturation values are in the range 0 to 1. If you do not use the /S flag, the default value is the full saturation range.

Use *saturationWave* when you have more than one pair of saturation values. If you use a saturation wave you must also use a lightness wave (see /L). *saturationWave* should consist of pairs of values where the first point is the lower saturation value and the second point is the higher saturation value. There is no restriction on the order of pairs within the wave.

/SEG Computes the segmentation image for **fuzzyClassify**. The image is stored in the 2D wave M_FuzzySegments. The value of each pixel is 255**classIndex/number of classes*. Here *classIndex* is the index of the class to which the pixel belongs with the highest probability.

/T=*flag* Use one or more of the following flags.

1: Swaps the data so that the DC is at the center of the image.

2: Calculates the power defined as: $P(f) = 0.5 \cdot (H(f)^2 + H(-f)^2)$.

/TEXT=*val* Specifies the type of texture to create with the **imageToTexture** keyword. *val* is a binary flag that can be a combination of the following values.

val	Texture
1	Truncates each dimension to the nearest power of 2, which is required for OpenGL textures.
2	Creates a 1D texture (all other textures are for 2D applications).
4	Creates a single channel texture suitable for alpha or luminance channels.
8	Creates a RGB texture from a 3 (or more) layer data.
16	Creates a RGBA texture. If *imageMatrix* does not have a 4th layer, alpha is set to 255.

/TOL=*tolerance* Sets the tolerance for iteration convergence with **fuzzyClassify**. Convergence is satisfied when the sum of the squared differences of all classes drops below *tolerance*, which must not be negative.

/TM4D=*mode* Used with transpose4D to specify the format of the output wave. Here *mode* is a 4 digit integer that describes the mapping of the transformation. The digit 1 is used to represent the first dimension, 2 for the second, 4 for the third and 8 for the 4th dimension such that the original wave corresponds to mode=1248. All other modes are obtained by permuting one or more of the four digits and mode must consist of 4 distinct digits.

Added in Igor Pro 7.00.

/U Creates an HSL wave of type unsigned short that contains values between 0 and 65535 when used with **rgb2hsl**.

/W Pads the image by wrapping the data. If you are adding more rows or more columns than are available in the source wave, the operation cycles through the source data as many times as necessary.

/X={*Nx*,*Ny*,*x1*,*y1*,*z1*,*x2*,*y2*,*z2*,*x3*,*y3*,*z3*}

Nx and *Ny* are the rows and columns of the wave M_ExtractedSurface. The remaining parameters specify three 3D points on the extracted plane. The three points must be chosen at the vertices of the plane and entered in clock-wise order without skipping a vertex.

/Z Ignores errors.

Examples

If you want to insert a 2D (M x N) wave, plane0, into plane number 0 of an (M x N x 3) wave, rgbWave:

```
ImageTransform /P=0/D=plane0 setPlane rgbWave
```

If your source wave is 100 rows by 100 columns and you want to create a montage of this image use:

```
ImageTransform /W/N={200,200} padImage srcWaveName
```

Hue and Saturation Segmentation Example

```
Function hueSatSegment(hslW,lowH,highH,lowS,highS)
   Wave hslW
   Variable lowH,highH,lowS,highS

   Make/D/O/N=(2,3) conditionW
   conditionW={{lowH,highH},{lowS,highS},{NaN,NaN}}
   ImageTransform/D=conditionW matchPlanes hslW
   KillWaves/Z conditionW
End
```

Voronoi Tesselation Example

```
Make/O/N=(33,3) ddd=gnoise(4)
ImageTransform voronoi ddd
Display ddd[][1] vs ddd[][0]
ModifyGraph mode=3,marker=19,msize=1,rgb=(0,0,65535)
AppendToGraph M_VoronoiEdges[][1] vs M_VoronoiEdges[][0]
SetAxis left -15,15
SetAxis bottom -5,10
```

See Also

Chapter III-11, **Image Processing**, for many examples. In particular see: **Color Transforms** on page III-352, **Handling Color** on page III-379, and **General Utilities: ImageTransform Operation** on page III-381. The **MatrixOp** operation.

References

Born, Max, and Emil Wolf, *Principles of Optics*, 7th ed., Cambridge University Press, 1999.

Details about the rgb2i123 transform:

Gevers, T., and A.W.M. Smeulders, Color Based Object Recognition, *Pattern Recognition*, *32*, 453-464, 1999.

ImageUnwrapPhase

ImageUnwrapPhase [***flags***][**qualityWave=*qWave*,**] **srcwave=*waveName***

The ImageUnwrapPhase operation unwraps the 2D phase in srcWave and stores the result in the wave M_UnwrappedPhase in the current data folder. srcWave must be a real valued wave of single or double precision. Phase is measured in cycles (units of 2π).

Parameters

qualityWave=*qWave*	Specifies a wave, *qWave,* containing numbers that rate the quality of the phase stored in the pixels. *qWave* is 2D wave of the same dimensions as srcWave that can be any real data type and values can have an arbitrary scale. If used with /M=1 the quality values determine the order of phase unwrapping subject to branch cuts, with higher quality unwrapped first. If used with /M=2 the unwrapping is guided by the quality values only. This wave must not contain any NaNs or INFs.
srcwave=*waveName*	Specifies a real-valued SP or DP wave that may contain NaNs or INFs but is otherwise assumed to contain the phase modulo 1.

Flags

/E	Eliminate dipoles. Only applies to Goldstein's method (/M=1). Dipoles are a pair of a positive and negative residues that are side by side. They are eliminated from the unwrapping process by replacing them with a branch cut. The variable V_numResidues contains the number of residues remaining after removal of the dipoles.

/L Saves the lookup table(LUT) used in the analysis with /M=1. This information may be useful in analyzing your results. The LUT is saved as a 2D unsigned byte wave M_PhaseLUT in the current data folder. Each entry consists of 8-bit fields:

bit=0:	Positive residue.
bit=1:	Negative residue.
bit=2:	Branch cut.
bit=3:	Image boundary exclusion.

Other bits are reserved and subject to change. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/M=*method* Determines the method for computing the unwrapped phase:

method =0: Modified Itoh's algorithm, which assumes that there are no residues in the phase. The phase is unwrapped in a contiguous way subject only to the ROI or singularities in the data (e.g., NaNs or INFs). You will get wrong results for the unwrapped phase if you use this method and your data contains residues.

method=1: Modified Goldstein's algorithm. Creates the variables V_numResidues and V_numRegions. Optional *qWave* can determine order of unwrapping around the branch cuts.

method=2: Uses a quality map to decide the unwrapping path priority. The quality map is a 2D wave that has the same dimensions as the source wave but could have an arbitrary data type. The phase is unwrapped starting from the largest value in the quality map.

/MAX=*len* Specifies the maximum length of a branch cut. Only applicable to Goldstein's method (/M=1). By default this is set to the largest of rows or columns.

/Q Suppresses messages to the history.

/R=*roiWave* Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/B/U) that has the same number of rows and columns as *waveName*. The ROI itself is defined by entries or pixels in the *roiWave* with value of 1. Pixels outside the ROI should be set to zero. The ROI does not have to be contiguous but it is best if you choose a convex ROI in order to make sure that any branch cuts computed by the algorithm lie completely inside the ROI domain.

/REST = *threshold* Sets the threshold value for evaluating a residue. The residue is evaluated by the equivalent of a closed path integral. If the path integral value exceeds the threshold value, the top-left corner of the quad is taken to be a positive residue. If the path integral is less than -*threshold*, it is a negative residue.

Details

Phase unwrapping in two dimensions is difficult because the result of the operation needs to be such that any path integral over a closed contour will vanish. In many practical situations, certain points in the plane have the property that a path integral around them is not zero. These nonzero points are residues. We use the definition that when a counterclockwise path integral leads to a positive value the residue is called a positive residue.

ImageUnwrapPhase uses the modified Itoh's method by default. Phase is unwrapped with an offset equal to the first element that is allowed by the ROI starting at (0,0) and scanning by rows. If there are no residues or if you unwrap the phase using Itoh's algorithm, then the phase is unwrapped only subject to the optional ROI using a seed-fill type algorithm that unwraps by growing a region outward from the seed pixel. Each time that the region growing is terminated by boundaries (external or due to the ROI), the algorithm returns to the row scanning to find a new starting point.

If there are residues and you choose Goldstein's method, the residues are first mapped into a lookup table (LUT) and branch-cuts are determined between residues and boundaries. It is also possible to remove some residues (dipoles) using the /E flag. Phase is then unwrapped in regions bounded by branch cuts using a seed-fill type algorithm that does not cross branch cuts. With a quality wave, the algorithm follows the same seed-fill approach except that it gives priority to pixels with high quality level. The phase on the branch cuts themselves is subsequently calculated.

The output wave M_UnwrappedPhase has the same wave scaling and dimension units as srcWave. The unwrapped phase is units of cycles; you will have to multiply it by 2π if you need the results in radians.

The operation creates two variables:

V_numResidues	Number of residues encountered(if using /M=1).
V_numRegions	Number of independent phase regions. In Goldstein's method the regions are bounded by branch cuts, but in Itoh's method they depend on the content of the ROI wave.

Examples

```
// Unwrap the phase of a complex wave wCmplx
MatrixOP/O phaseWave=atan2(imag(wCmplx),real(wCmplx))/(2*pi)
ImageUnwrapPhase/M=1 srcWave=phaseWave

// Find the locations of positive residues in the phase
ImageUnwrapPhase/M=1/L srcWave=phaseWave
MatrixOP/O ee=greater(bitAnd(M_PhaseLUT,2),0)

// Find the branch cuts
MatrixOP/O bc=greater(bitAnd(M_PhaseLUT,8),0)
```

See Also

The **Unwrap** operation and the **mod** function.

References

The following reference is an excellent text containing in-depth theory and detailed explanation of many two-dimensional phase unwrapping algorithms:

Ghiglia, Dennis C., and Mark D. Pritt, *Two Dimensional Phase Unwrapping — Theory, Algorithms and Software*, Wiley, 1998.

ImageWindow

ImageWindow [**/I/O/P=*param***] ***method srcWave***

The ImageWindow operation multiplies the named waves by the specified windowing method.

ImageWindow is useful in preparing an image for FFT analysis by reducing FFT artifacts produced at the image boundaries.

Parameters

srcWave	Two-dimensional wave of any numerical type. See **WindowFunction** for windowing one-dimensional data.
method	Selects the type of windowing filter. See **ImageWindow Methods** on page V-436.

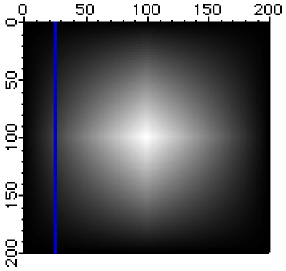
Flags

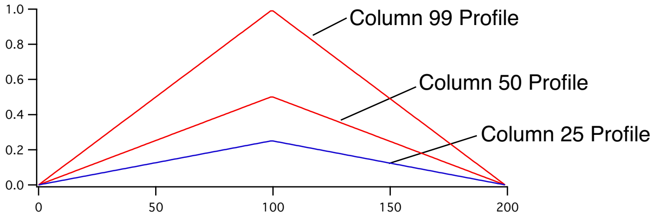
/I	Creates only the output wave containing the windowing filter values that are used to multiply each pixel in *srcWave*. It does not filter the source image.
/O	Overwrites the source image with the output image. If /O is not used then the operation creates the M_WindowedImage wave containing the filtered source image.
/P=*param*	Specifies the design parameter for the Kaiser window.

Details

The 1-dimensional window for each column is multiplied by the value of the corresponding row's window value. In other words, each point is multiplied by the both the row-oriented and column-oriented window value.

This means that all four edges of the image are decreased while the center remains at or near its original value. For example, applying the Bartlett window to an image whose values are all equal results in a trapezoidal pyramid of values:





The default output wave is created with the same data type as the source image. Therefore, if the source image is of type unsigned byte (/b/u) the result of using /I will be identically zero (except possibly for the middle-most pixel). If you keep in mind that you need to convert the source image to a wave type of single or double precision in order to perform the FFT, it is best if you convert your source image (e.g., `Redimension/S` *`srcImage`*) before using the ImageWindow operation.

The windowed output is in the M_WindowedImage wave unless the source is overwritten using the /O flag.

The necessary normalization value (equals to the average squared window factor) is stored in V_value.

ImageWindow Methods

This section describes the supported keywords for the *method* parameter. In all equations, *L* is the array width and *n* is the pixel number.

Hanning:

$$w(n) = \frac{1}{2}\left[1-\cos\left(\frac{2\pi n}{L-1}\right)\right] \quad 0 \le n \le L-1$$

Hamming:

$$w(n) = 0.54-0.46\cos\left(\frac{2\pi n}{L-1}\right) \quad 0 \le n \le L-1$$

Bartlet: Synonym for Bartlett.

Bartlett:

$$w(n) = \begin{cases} \dfrac{2n}{L-1} & 0 \le n \le \dfrac{L-1}{2} \\ 2-\dfrac{2n}{L-1} & \dfrac{L-1}{2} \le n \le L-1 \end{cases}$$

Blackman:

$$w(n) = 0.42-0.5\cos\left(\frac{2\pi n}{L-1}\right)+0.08\cos\left(\frac{4\pi n}{L-1}\right)$$

$$0 \le n \le L-1$$

Kaiser:

$$\frac{I_0\left(\omega_a\sqrt{\left(\frac{L-1}{2}\right)^2-\left(n-\frac{L-1}{2}\right)^2}\right)}{I_0\left(\omega_a\left(\frac{L-1}{2}\right)\right)} \qquad 0\le n\le L-1$$

where $I_0\{\ldots\}$ is the zeroth-order Bessel function of the first kind and ω_a is the design parameter specified by /P=*param*.

KaiserBessel20: $\alpha = 2.0$

KaiserBessel25: $\alpha = 2.5$

KaiserBessel30: $\alpha = 3.0$

$$w(n)=\frac{I_0\left(\pi\alpha\sqrt{1-\left(\frac{n}{L/2}\right)^2}\right)}{I_0(\pi\alpha)} \qquad 0\le|n|\le\frac{L}{2}$$

$$I_0(x)=\sum_{k=0}^{\infty}\frac{\left(x^2/4\right)^k}{(k!)^2}.$$

Examples

To see what one of the windowing filters looks like:

```
Make/N=(80,80) wShape                 // Make a matrix
ImageWindow/I/O Blackman wShape       // Replace with windowing filter
Display;AppendImage wShape                   // Display windowing filter
Make/N=2 xTrace={0,79},yTrace={39,39}        // Prepare for 1D section
AppendToGraph yTrace vs xTrace
ImageLineProfile srcWave=wShape, xWave=xTrace, yWave=yTrace
Display W_ImageLineProfile            // Display 1D section of filter
```

See Also

The **WindowFunction** operation for information about 1D applications.

Spectral Windowing on page III-275. Chapter III-11, **Image Processing** contains links to and descriptions of other image operations.

See **FFT** operation for other 1D windowing functions for use with FFTs; **DSPPeriodogram** uses the same window functions. See **Correlations** on page III-362.

DPSS

References

For further windowing information, see page 243 of:

Pratt, William K., *Digital Image Processing*, John Wiley, New York, 1991.

IndependentModule

`#pragma IndependentModule = `*`imName`*

The IndependentModule pragma designates groups of one or more procedure files that are compiled and linked separately. Once compiled and linked, the code remains in place and is usable even though other procedures may fail to compile. This allows functioning control panels and menus to continue to work regardless of user programming errors.

See Also

Independent Modules on page IV-238, **The IndependentModule Pragma** on page IV-55 and #**pragma**.

IndependentModuleList

IndependentModuleList(*listSepStr*)

The IndependentModuleList function returns a string containing a list of independent module names separated by listSepStr.

Use **StringFromList** to access individual names.

Parameters

listSepStr contains the character, usually ";", to be used to to separate the names in the returned list.

Details

In Igor6, only the first byte of *listSepStr* was used. In Igor7 and later, all bytes are used.

ProcGlobal is not in the returned list, and the order of returned names is not defined.

See Also

Independent Modules on page IV-238.

GetIndependentModuleName, **StringFromList**, **FunctionList**.

IndexedDir

IndexedDir(*pathName*, *index*, *flags* [, *separatorStr*])

The IndexedDir function returns a string containing the name of or the full path to the *index*th folder in the folder referenced by *pathName*.

Parameters

pathName is the name of an Igor symbolic path pointing to the parent directory.

index is the index number of the directory (within the parent directory) of interest starting from 0. If *index* is -1, IndexedDir will return the name of *all* of the folders in the parent, separated by semicolons or by *separatorStr* if specified.

flags is a bitwise parameter:

Bit 0:	Set if you want a full path. Cleared if you want just the directory name.

All other bits are reserved and should be cleared.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

separatorStr is an optional string argument used when index is -1. If you omit *separatorStr*, folder names are separated by ";". This parameter was added in Igor 9.01.

Details

You create the symbolic path identifying the parent directory using the **NewPath** operation or the New Path dialog (Misc menu).

Prior to Igor Pro 3.1, IndexedDir was an external function and took a string as the first parameter rather than a name. The *pathName* parameter can now be either a name or a string containing a name. Any of the following will work:

```
String str = "IGOR"
Print IndexedDir(IGOR, 0, 0)         // First parameter is a name.
Print IndexedDir($str, 0, 0)         // First parameter is a name.
Print IndexedDir("IGOR", 0, 0)       // First parameter is a string.
Print IndexedDir(str, 0, 0)          // First parameter is a string.
```

The acceptance of a string is for backward compatibility only. New code should be written using a name.

The returned path uses the native conventions of the OS under which Igor is running.

Examples

Example: Recursively Listing Directories and Files

Here is an example for heavy-duty Igor Pro programmers. It is an Igor Pro user-defined function that prints the paths of all of the files and folders in a given folder with or without recursion. You can rework this to do something with each file instead of just printing its path.

To try the function, copy and paste it into the Procedure window. Then execute the example shown in the comments.

```
// PrintFoldersAndFiles(pathName, extension, recurse, level)
// Shows how to recursively find all files in a folder and subfolders.
// pathName is the name of an Igor symbolic path that you created
// using NewPath or the Misc->New Path menu item.
// extension is a file name extension like ".txt" or "????" for all files.
// recurse is 1 to recurse or 0 to list just the top-level folder.
// level is the recursion level - pass 0 when calling PrintFoldersAndFiles.
// Example: PrintFoldersAndFiles("Igor", ".ihf", 1, 0)
Function PrintFoldersAndFiles(pathName, extension, recurse, level)
	String pathName		// Name of symbolic path in which to look for folders and files.
	String extension		// File name extension (e.g., ".txt") or "????" for all files.
	Variable recurse		// True to recurse (do it for subfolders too).
	Variable level		// Recursion level. Pass 0 for the top level.

	Variable folderIndex, fileIndex
	String prefix

	// Build a prefix (a number of tabs to indicate the folder level by indentation)
	prefix = ""
	folderIndex = 0
	do
		if (folderIndex >= level)
			break
		endif
		prefix += "\t"		// Indent one more tab
		folderIndex += 1
	while(1)

	// Print folder
	String path
	PathInfo $pathName		// Sets S_path
	path = S_path
	Printf "%s%s\r", prefix, path

	// Print files
	fileIndex = 0
	do
		String fileName
		fileName = IndexedFile($pathName, fileIndex, extension)
		if (strlen(fileName) == 0)
			break
		endif
		Printf "%s\t%s%s\r", prefix, path, fileName
		fileIndex += 1
	while(1)

	if (recurse)			// Do we want to go into subfolder?
		folderIndex = 0
		do
			path = IndexedDir($pathName, folderIndex, 1)
			if (strlen(path) == 0)
				break		// No more folders
			endif

			String subFolderPathName = "tempPrintFoldersPath_" + num2istr(level+1)

			// Now we get the path to the new parent folder
			String subFolderPath
			subFolderPath = path

			NewPath/Q/O $subFolderPathName, subFolderPath
			PrintFoldersAndFiles(subFolderPathName, extension, recurse, level+1)
			KillPath/Z $subFolderPathName

			folderIndex += 1
		while(1)
	endif
End
```

Example: Fast Scan of Directories

Calling IndexedDir for each directory is an O(N^2) problem because to get the nth directory the OS routines underlying IndexedDir need to iterate through directories 0..n-1. This becomes an issue only if you are dealing with hundreds or thousands of directories.

This function illustrates a technique for converting this to an O(N) problem by getting a complete list of paths from IndexedDir in one call and storing them in a text wave. This approach could also be used with IndexedFile.

```
Function ScanDirectories(pathName, printDirNames)
    String pathName            // Name of Igor symbolic path
    Variable printDirNames     // True if you want to print the directory names

    Variable t0 = StopMSTimer(-2)

    String dirList = IndexedDir($pathName, -1, 0)
    Variable numDirs = ItemsInList(dirList)

    // Store directory list in a free text wave.
    // The free wave is automatically killed when the function returns.
    Make/N=(numDirs)/T/FREE dirs = StringFromList(p, dirList)

    String dirName
    Variable i
    for(i=0; i<numDirs; i+=1)
        dirName = dirs[i]
        Print i, dirName
    endfor

    Variable t1 = StopMSTimer(-2)
    Variable elapsed = (t1 - t0) / 1E6
    Printf "Took %g seconds\r", elapsed
End
```

IndexedFile

IndexedFile(*pathName*, *index*, *fileTypeOrExtStr* [, *creatorStr*, *separatorStr*])

If *index* is greater than or equal to zero, IndexedFile returns a string containing the name of the *index*th file in the folder specified by *pathName* which matches the file type or extension specified by *fileTypeOrExtStr*.

If *index* is -1, IndexedFile returns a string list of all matching files separated by a semicolon, or by separatorStr if specified.

IndexedFile returns an empty string ("") if there is no such file.

Parameters

pathName is the *name* of an Igor symbolic path. It is *not* a string.

index normally starts from zero. However, if *index* is -1, IndexedFile returns a string containing a semicolon-separated list of the names of all files in the folder associated with the specified symbolic path which match *fileTypeOrExtStr*.

fileTypeOrExtStr is either:

- A string starting with ".", such as ".txt", ".bwav", or ".c". Only files with a matching file name extension are indexed. Set *fileTypeOrExtStr* to "." to index file names that end with "." such as "myFileNameEndsWithThisDot."
- A string containing exactly four ASCII characters, such as "TEXT" or "IGBW". Only files of the specified Macintosh file type are indexed. However, if *fileTypeOrExtStr* is "????", files of any type are indexed.
- On Windows, Igor considers files with ".txt" extensions to be of type TEXT. It does similar mappings for other extensions. See **File Types and Extensions** on page III-455 for details.

creatorStr is obsolete and should no longer be used unless you need to specify the separator in which case, use "????" for *creatorStr*.

separatorStr is an optional string argument used when index is -1. If you omit *separatorStr*, file names are separated by ";". This parameter was added in Igor 9.01.

Order of Files Returned By IndexedFile

The order in which files are returned by IndexedFile is determined by the operating system. If the order matters to you, you should explicitly sort the file names.

For example, assume you have files named "File1.txt", "File2.txt" and "File10.txt". An alphabetic order gives you: "File1.txt;File10.txt;File2.txt;". Often what you really want is a combination of alphabetic and numeric sorting returning "File1.txt;File2.txt;File10.txt;". Here is a function that does that:

```
Function DemoIndexedFile()
    String pathName = "Igor"// Refers to "Igor Pro Folder"

    // Get a semicolon-separated list of all files in the folder
    String list = IndexedFile($pathName, -1, ".txt")

    // Sort using combined alpha and numeric sort
    list = SortList(list, ";", 16)

    // Process the list
    Variable numItems = ItemsInList(list)
    Variable i
    for(i=0; i<numItems; i+=1)
        String fileName = StringFromList(i, list)
        Print i, fileName
    endfor
End
```

Treatment of Macintosh Dot-Underscore Files

IndexedFile ignores "dot-underscore" files unless *fileTypeOrExtStr* is "????".

A dot-underscore file is a file created by Macintosh when it writes to a non-HFS volume, for example, when it writes to a Windows volume via SMB file sharing. The dot-underscore file stores Macintosh HFS-specific data such as the file's type and creator codes, and the file's resource fork, if it has one.

For example, if a file named "wave0.ibw" is copied via SMB to a Windows volume, Mac OS X creates two files on the Windows volume: "wave0.ibw" and "._wave0.ibw". Mac OS X makes these two files appear as one to Macintosh applications. However, Windows does not do this. As a consequence, when a Windows program sees "._wave0.ibw", it expects it to be a valid .ibw file, but it is not. This causes problems.

By ignoring dot-underscore files, IndexedFile prevents this type of problem. However, if fileTypeOrExtStr is "????", IndexedFile will return dot-underscore files on Windows.

Examples

```
NewPath/O myPath "MyDisk:MyFolder:"
Print IndexedFile(myPath,-1,"TEXT")         // all text-type files
Print IndexedFile(myPath,0,"TEXT")          // only the first text file
Print IndexedFile(myPath,-1,".dat")         // *.dat
Print IndexedFile(myPath,-1,"TEXT","IGR0")  // all Igor text files
Print IndexedFile(myPath,-1,"????")         // all files, all creators
Print IndexedFile(myPath,-1,"????","????","\r")    // CR separator
```

See **IndexedDir** for another example using IndexedFile and for a method for speeding up scanning of very large numbers of files.

See Also

The **TextFile** and **IndexedDir** functions.

IndexSort

IndexSort [/DIML] ***indexWaveName*, *sortedWaveName*** [, ***sortedWaveName***]…

The IndexSort operation sorts the values in each *sortedWaveName* wave according to the Y values of *indexWaveName*.

Flags

/DIML	Moves the dimension labels with the values (keeps any row dimension label with the row's value).

Details

indexWaveName can not be complex. *indexWaveName* is presumed to have been the destination of a previous **MakeIndex** operation.

This has the effect of putting the *sortedWaveName* waves in the same order as the wave from which the index values in *indexWaveName* was made.

All of the *sortedWaveName* waves must be of equal length.

See Also

Sorting on page III-132, **MakeIndex and IndexSort** on page III-134, **Sort**, **MakeIndex**

IndexToScale

IndexToScale(wave, index, dim)

The IndexToScale function returns the scaled coordinate value corresponding to wave element index in the specified dimension.

The IndexToScale function was added in Igor Pro 7.00.

Details

The function returns the expression:

DimOffset(*wave*,*dim*) + index*DimDelta(*wave*,*dim*)

index is an integer.

dim is 0 for rows, 1 for columns, 2 for layers or 3 for chunks.

The function returns NaN if *dim* is not a valid dimension or if *index* is greater than the number of elements in the specified dimension.

Examples

```
Make/N=(10,20,30,40) w4D
SetScale/P y 2,3,"", w4D
SetScale/P z 4,5,"", w4D
SetScale/P t 6,7,"", w4D
Print IndexToScale(w4D,1,0)
Print IndexToScale(w4D,1,1)
Print IndexToScale(w4D,1,2)
Print IndexToScale(w4D,1,3)
Print IndexToScale(w4D,1,4)
Print IndexToScale(w4D,-1,0)
Print IndexToScale(w4D,11,0)
```

See Also

ScaleToIndex, **pnt2x**, **DimDelta**, **DimOffset**

Waveform Model of Data on page II-62 for an explanation of wave scaling.

Inf

Inf

The Inf function returns the “infinity” value.

InsertPoints

`InsertPoints` [***`/M=dim /V=value`***] ***`beforePoint, numPoints, waveName`*** [, ***`waveName`***]...

The InsertPoints operation inserts *numPoints* points in front of point *beforePoint* in each *waveName*. The new points have the value zero.

Flags

/M=*dim*	Specifies the dimension into which elements are to be inserted. Values are 0 for rows, 1 for columns, 2 for layers, 3 for chunks. If /M is omitted, InsertPoints inserts in the rows dimension.
/V=*value*	When used with numeric waves, *value* specifies the value for new elements. If you omit /V, new elements are set to zero. The /V flag was added in Igor Pro 8.00.

Details

Trying to insert points into any but the rows of a zero-point wave results in a zero-point wave. You must first make the number of rows nonzero before anything else has an effect.

See Also

Lists of Values on page II-78.

InstantFrequency

`InstantFrequency [flags]` ***`srcWave`*** **`[ (`*****`startX,endX`*****`) ]`**

The InstantFrequency operation computes the instanteous frequency, and optionally the instantaneous amplitude, of srcWave. InstantFrequency was added in Igor Pro 9.00.

InstantFrequency creates an output wave whose name and location depends on the /DEST and /OUT flags as described under ***InstantFrequency Output Wave*** below.

Parameters

srcWave specifies the wave to be analyzed.

[*startX*,*endX*] is an optional subrange to analyze in point numbers.

(*startX*,*endX*) is an optional subrange to analyze in X values.

If you omit the subrange, *startX* defaults to the first point in *srcWave* and *endX* defaults to the last point in *srcWave*.

Flags for all Methods

/DEST=*destWave*	Specifies the output wave created by the operation. *destWave* can be a simple wave name, a data folder path plus wave name, or a wave reference to an existing wave. It is an error to specify the same wave as both *srcWave* and *destWave*. If you omit /DEST, the output wave name depends on /OUT. When used in a function, the InstantFrequency operation by default creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-72 for details. See ***InstantFrequency Output Wave*** below for further discussion.
/FREE	Creates a free destination wave (see **Free Waves** on page IV-91). /FREE is allowed only in functions and only if you include /DEST=destWave where destWave is a simple name or an valid wave reference.

/METH=*method* Specifies the method by which the calculation is performed.

method=0:	Osculating Circle (default)
method=1:	Gabor
method=2:	Spectogram using center of mass
method=3:	Spectogram using maximum amplitude

/OUT=*mode* Specifies the type of output to be created.

mode=1:	Frequency only (default)
mode=2:	Amplitude only
mode=3:	Frequency and amplitude
mode=4:	Signed amplitude only
mode=5:	Debugging output

If you omit /DEST, the output wave is created in the current data folder with a name determined by mode as follows:

mode=1:	W_InstantFrequency
mode=2:	W_InstantAmplitude
mode=3:	M_InstantFrequency
mode=4:	W_InstantAmplitude
mode=5:	M_InstantFrequency

Flags for Spectrogram Method (/METH=2) Only

/HOPS=*hopSize* Specifies the offset in points between centers of consecutive source segments. By default *hopSize* is 1 and the transform is computed for segments that are offset by a single points from each other.

/PAD=*newSize* Converts each segment of *srcWave* into a padded array of length *newSize*. The padded array contains the original data at the center of the array with zeros elements on both sides.

/SEGS=*segSize* Sets the length of the segment sampled from *srcWave* in points. The segment is optionally padded to a larger dimension (see /PAD) and multiplied by a window function prior to FFT. *segSize* must be 32 or greater.

Defining *n* as numpnts(*srcWave*), the default segment size, used if you omit /SEGS, is:

n/200 if *n* >= 25600

128 if 130 <= *n* <= 25599

n-2 if *n* <= 129

/WINF=*windowKind* Premultiplies a data segment with the selected window function. The default window is Hanning. See ***Window Functions*** in the documentation for **FFT** for details.

InstantFrequency Output Wave

If you use the /FREE flag then the output wave is created as a free wave using the name or wave reference specified by /DEST=*destWave*.

If you include the /DEST flag and omit /FREE then the output wave location and name is specified by the *destWave* parameter. *destWave* can be a simple wave name, a data folder path plus wave name, or a wave reference to an existing wave.

If you omit the /DEST and /FREE flags then the output wave is created in the current data folder with the default name W_InstantFrequency (/OUT=1), W_InstantAmplitude (/OUT=2 or 4), or M_InstantFrequency (/OUT=3 or 5), depending on the /OUT flag.

Gabor Method (/METH=1)

The Gabor method uses the Hilbert Transform to synthesize an "analytic" signal that is a copy of the source wave, shifted by 90 degrees, but with the constant ("DC") component removed.

A complex "phasor" wave is generated whose real part is the source wave, and the imaginary part is this analytic signal.

The phase at each time point is computed as atan2(imag(phasor[p]),real(phasor[p])). This phase calculation is affected by any constant component of the source wave. The derivative of this phase with respect to time is the instantanteous frequency.

The instantaneous amplitude is the magnitude of the phasor[p].

See the **HilbertTransform** operation example for an equivalent implementation.

The /OUT=5 debugging output for the Gabor method is:

```
destWave[][0]       // Instant frequency
destWave[][1]       // Instant amplitude
destWave[][2]       // Unwrapped phase wave
destWave[][3]       // Trajectory Y wave (Hilbert transform of input wave)
```

Osculating Circle Method (/METH=0)

The primary drawback of the Gabor method is that any sizeable constant level distorts the phase such that it isn't always increasing, which results in negative frequencies being computed. The Osculating Circle Method does not have this problem.

The Osculating Circle method constructs the same analytic phasor as the Gabor method, but computes the phase and derivative differently in a way that eliminates the constant level distortion: at each point the phasor's previous, current, and next complex values (the "trajectory") are fit to a circle in the complex plane. That circle's origin is used to measure both the phase and amplitude at that point, instead of the origin at (0+i0) that the Gabor method uses.

The /OUT=5 debugging output for the Osculating Circle method is:

```
destWave[][0]       // Instant frequency
destWave[][1]       // Signed instant amplitude
destWave[][2]       // Unwrapped phase wave
destWave[][3]       // Trajectory Y wave (Hilbert transform of input wave)
destWave[][4]       // Trajectory dY wave
destWave[][5]       // Trajectory ddY wave
destWave[][6]       // Trajectory dX wave
destWave[][7]       // Trajectory ddYX wave
destWave[][8]       // Origin Y wave
destWave[][9]       // Origin X wave
```

Spectrogram (/METH=2 or 3)

The Spectogram method for determining instant frequency and amplitude is based on measuring the Short-Time Fourier Transform, the 2D time-frequency representation for a 1D array. The scaled magnitude of the transform is known as the "spectrogram" for time series or "sonogram" in the case of sound input. Methods 2 and 3 are comprised of the following steps:

1. Compute the Short-Time Fourier Transform according to the various spectogram parameters. This results in a 2D wave, where the columns of each row comprise the scaled magnitude spectrum at one point in time.

2. For each spectrum determine where the dominant frequency lies, and its magnitude.

For /METH=2, the dominant frequency is found using the centerOfMass function.

For /METH=3, the dominant frequency is found by locating the maximum value.

The /OUT=5 debugging output for the Spectogram method is the 2D spectogram that would have been used as the output of step 1.

References

Wikipedia: https://en.wikipedia.org/wiki/Instantaneous_phase_and_frequency

Wikipedia: https://en.wikipedia.org/wiki/Gabor_transform

Wolfram: https://mathworld.wolfram.com/OsculatingCircle.html

Ming-Kuang Hsu, Jiun-Chyuan Sheu, and Cesar Hsue, "Overcoming the Negative Frequencies - Instantaneous Frequency and Amplitude Estimation using Oculating Circle Method, *Journal of Marine Science and Technology*, Vol 19, No 5, pp. 514-521, 2011.

See Also

STFT, **HilbertTransform**, **DSPPeriodogram**

Int

```
int localName
```

In a user-defined function or structure, declares a local 32-bit integer in IGOR32, a local 64-bit integer in IGOR64.

Int is available in Igor Pro 7 and later. See **Integer Expressions** on page IV-38 for details.

See Also

Int64, **UInt64**

Int64

```
int64 localName
```

Declares a local 64-bit integer in a user-defined function or structure.

Int64 is available in Igor Pro 7 and later. See **Integer Expressions** on page IV-38 for details.

See Also

Int, **UInt64**

Integrate

```
Integrate [type flags][flags] yWaveA [/X = xWaveA][/D = destWaveA]
    [, yWaveB [/X = xWaveB][/D = destWaveB][, …]]
```

The Integrate operation calculates the 1D numeric integral of a wave. X values may be supplied by the X-scaling of the source wave or by an optional X wave. Rectangular integration is used by default.

Integrate is multi-dimension-aware in the sense that it computes a 1D integration along the dimension specified by the /DIM flag or along the rows dimension if you omit /DIM.

Complex waves have their real and imaginary components integrated individually.

Flags

/DIM= *d* Specifies the wave dimension along which to integrate when *yWave* is multidimensional.

d=-1:	Treats entire wave as 1D (default).
d=0:	Integrates along rows.
d=1:	Integrates along columns.
d=2:	Integrates along layers.
d=3:	Integrates along rows.

For example, for a 2D wave, /DIM=0 integrates each row and /DIM=1 integrates each column.

/METH=*m* Sets the integration method.

m=0:	Rectangular integration (default). Results at a point are stored at the same point (rather than at the next point as for /METH=2). This method keeps the dimension size the same.
m=1:	Trapezoidal integration.
m=2:	Rectangular integration. Results at a point are stored at the next point (rather than at the same point as for /METH=0). This method increases the dimension size by one to provide a place for the last bin.

/P Forces point scaling.

/T Trapezoidal integration. Same as /METH=1.

Type Flags *(used only in functions)*

Integrate also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when it needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-73 and **WAVE Reference Type Flags** on page IV-74 for a complete list of type flags and further details.

For example, when the input (and output) waves are complex, the output wave will be complex. To get the Igor compiler to create a complex output wave reference, use the /C type flag with /D=destwave:

```
Make/O/C cInput=cmplx(sin(p/8), cos(p/8))
Make/O/C/N=0 cOutput
Integrate/C cInput /D=cOutput
```

Wave Parameters

Note: ***All*** wave parameters must follow *yWave* in the command. All wave parameter flags and type flags must appear immediately after the operation name (`Integrate`).

/D=*destWave* Specifies the name of the wave to hold the integrated data. It creates *destWave* if it does not already exist or overwrites it if it exists.

/X=*xWave* Specifies the name of corresponding X wave. For rectangular integration, the number of points in the X wave must be one greater than the number of elements in the Y wave dimension being integrated.

Details

The computation equation for rectangular integration using /METH=0 is:

$$waveOut[p]=\sum_{i=0}^{p} waveIn[i]\cdot \Delta x.$$

The computation equation for rectangular integration using /METH=2 is:

$$waveOut[0]=0$$

$$waveOut[p+1]=\sum_{i=0}^{p}(x_{i+1}-x_i)waveIn[i].$$

The inverse of this rectangular integration is the backwards difference.

Trapezoidal integration (/METH=1) is a more accurate method of computing the integral than rectangular integration. The computation equation is:

$$waveOut[0]=0$$

$$waveOut[p]=waveOut[p-1]+\frac{\Delta x}{2}\left(waveIn[p-1]+waveIn[p]\right).$$

If the optional /D = *destWave* flag is omitted, then the wave is integrated in place overwriting the source wave.

When using an X wave, the X wave must be a 1D wave with data type matching the Y wave (excluding the complex type flag). Rectangular integration (/METH=0 or 2) requires an X wave having one more point than the number of elements in the dimension of the Y wave being integrated. X waves with number points plus one are allowed for rectangular integration with methods needing only the number of points. X waves are not used with integer source waves.

Although it is mathematically suspect, rectangular integration using /METH=0 would be correct if the X scaling of the output wave is offset by ΔX.

`Differentiate/METH=1/EP=1` is the inverse of `Integrate/METH=2`, but `Integrate/METH=2` is the inverse of `Differentiate/METH=1/EP=1` only if the original first data point is added to the output wave.

Integrate applied to an XY pair of waves does not check the ordering of the X values and doesn't care about it. However, it is usually the case that your X values should be monotonic. If your X values are not monotonic, you should be aware that the X values will be taken from your X wave in the order they are found, which will result in random X intervals for the X differences. It is usually best to sort the X and Y waves first (see **Sort**).

See Also

Differentiate, **Integrate2D**, **Integrate1D**, **area** , **areaXY**

Integrate1D

Integrate1D(*UserFunctionName*, *min_x*, *max_x* [**,** ***options*** [**,** ***count*** [**,** ***pWave***]]]**)**

The Integrate1D function performs numerical integration of a user function between the specified limits (*min_x* and *max_x*).

Parameters

UserFunctionName must have this format:

```
Function UserFunctionName(inX)
   Variable inX
   ... do something
   return result
End
```

However, if you supply the optional *pWave* parameter then it must have this format:

```
Function UserFunctionName(pWave, inX)
   Wave pWave
   Variable inX
   ... do something
   return result
End
```

options is one of the following:

0:	Trapezoidal integration (default).
1:	Romberg integration.
2:	Gaussian Quadrature integration.

By default, *options* is 0 and the function performs trapezoidal integration. In this case Igor evaluates the integral iteratively. In each iteration the number of points where Igor evaluates the function increases by a factor of 2. The iterations terminate at convergence to tolerance or when the number of evaluations is 2^{23}.

The *count* parameter specifies the number of subintervals in which the integral is evaluated. If you specify 0 or a negative number for count, the function performs an adaptive Gaussian Quadrature integration in which Igor bisects the interval and performs a recursive refining of the integration only in parts of the interval where the integral does not converge to tolerance.

pWave is an optional parameter that, if present, is passed to your function as the first parameter. It is intended for your private use, to pass one or more values to your function, and is not modified by Igor. The *pWave* parameter was added in Igor Pro 7.00.

Details

You can integrate complex-valued functions using a function with the format:

```
Function/C complexUserFunction(inX)
   Variable inX
   Variable/C result
   //… do something
   return result
End
```

The syntax used to invoke the function is:

```
Variable/C cIntegralResult=Integrate1D(complexUserFunction,min_x,max_x…)
```

You can also use Integrate1D to perform a higher dimensional integrals. For example, consider the function: $F(x,y) = 2x + 3y + xy$.

In this case, the integral

$$h = \int dy \int f(x,y)\,dx$$

can be performed by establishing two user functions:

```
Function Do2dIntegration(xmin,xmax,ymin,ymax)
   Variable xmin,xmax,ymin,ymax

   Variable/G globalXmin=xmin
   Variable/G globalXmax=xmax
   Variable/G globalY

   return Integrate1d(userFunction2,ymin,ymax,1)  // Romberg integration
End

Function UserFunction1(inX)
   Variable inX

   NVAR globalY=globalY

   return (3*inX+2*globalY+inX*globalY)
End

Function UserFunction2(inY)
   Variable inY

   NVAR globalY=globalY
   globalY=inY
   NVAR globalXmin=globalXmin
   NVAR globalXmax=globalXmax

   // Romberg integration
   return Integrate1D(userFunction1,globalXmin,globalXmax,1)
End
```

This method can be extended to higher dimensions.

If the integration fails to converge or if the integrand diverges, Integrate1D returns NaN. When a function fails to converge it is a good idea to try another integration method or to use a user-defined number of intervals (as specified by the count parameter). Note that the trapezoidal method is prone to convergence problems when the absolute value of the integral is very small.

See Also

Integrate, **Integrate2D**, **SumSeries**

Integrate2D

`Integrate2D` `[`***`flags`***`] [`***`keyword = value`***` [, `***`keyword = value`***` …]]`

The Integrate2D operation calculates a two-dimensional numeric integral of a real-valued user-defined function or a wave. The result of the operation is stored in the variable V_value and the variable V_Flag is set to zero if there are no errors.

This operation was added in Igor Pro 7.00.

Flags

/OPTS=*op*	Sets the integration options. By default, both the x and the y integrations are performed using the adaptive trapezoidal method.

op is a bitwise parameter that you set to select the x and y integration methods. Set one bit for x and one bit for y:

Bit 0:	Trapezoidal in Y (1)
Bit 1:	Romberg in Y (2)
Bit 2:	Gaussian Quadrature in Y (4)
Bit 3:	Trapezoidal in X (8)
Bit 4:	Romberg in X (16)
Bit 5:	Gaussian Quadrature in X (32)

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Using these constants you can specify, for example, Romberg integration in the Y direction and Gaussian Quadrature in the X direction using /OPTS=(2 | 32).

/Q Suppress printing to the history area.

/Z=zFlag Set zFlag to 1 to suppress error reporting.

Keywords

epsilon=*ep* Specifies the convergence parameter. By default *ep*=1e-5. Smaller values lead to more accurate integration result but the tradeoff is longer computation time.

integrand=*uF* Specifies the user function to be integrated. See **The Integrand Function** below for details.

innerLowerLimit=*y1* Specifies the lower limit of the inner integral if this limit is fixed, i.e., if it is not a function of x. See the innerLowerFunc keyword if you need to specify a function for this limit.

innerUpperLimit=*y2* Specifies the upper limit of the inner integral if this limit is fixed, i.e., if it is not a function of x. See the innerUpperFunc keyword if you need to specify a function for this limit.

innerLowerFunc=*y1Func* Specifies a user-defined function for the lower limit of the inner integral. See **The Limit Functions** below.

innerUpperFunc=*y2Func* Specifies a user-defined function for the upper limit of the inner integral. See **The Limit Functions** below.

outerLowerLimit=*x1* Specifies the lower limit of the outer integral.

outerUpperLimit=*x2* Specifies the upper limit of the outer integral.

paramWave=*pWave* Specifies a wave to be passed to the integrand and limit user-defined functions as the pWave parameter. The wave may contain any number of values that you might need to evaluate the integrand or the integration limits.

If you omit paramWave then the pWave parameter to the functions will be NULL.

srcWave=*mWave* If you need to perform 2D integration of some data, you can specify the data directly instead of providing a user-defined function that returns interpolated data. mWave must be a 2D wave. Higher dimensional waves are accepted but only the first layer of the wave is used in the integration.

The Integrand Function

Integrate2D computes the general two-dimensional integral of a user-defined integrand function which you specify using the integrand keyword. The integrand function has this form:

```
Function integrandFunc(pWave,inX,inY)
    Wave/Z pWave
    Variable inX,inY
    ... do something
    return result
End
```

The function can have any name - integrandFunc is just an example. The function must take the parameters shown and must return a real numeric result. Returning a NaN terminates the integration.

pWave is a parameter wave that you specify using the paramWave keyword. The operation passes this wave on every call to the integrand function. If you omit paramWave when invoking Integrate2D then pWave will be NULL.

The Limit Functions

The limit functions provide lower and/or upper limits of integration for the inner integral if they are functions of x rather than fixed values. You specify a limit function using the innerLowerFunc and innerUpperFunc keywords. The form of the limit function is:

```
Function limitFunction(pWave,inX)
    Wave/Z pWave
    Variable inX

    ... do something
    return result
End
```

Details

The operation computes the general two-dimensional integral of the form

$$I=\int_{x1}^{x2} dx \int_{y1(x)}^{y2(x)} f(x,y)dy.$$

Here y1 and y2 are in general functions of x but could also be simple constants, and f(x,y) is real valued function. The integral is evaluated by considering the "outer" integral

$$I=\int_{x1}^{x2} G(x)dx,$$

where G(x) is the "inner" integral

$$G(x)=\int_{y1(x)}^{y2(x)} f(x,y)dy.$$

The operation allows you to specify different algorithms for integrating the inner and outer integrals. The simplest integration algorithm is the Trapezoidal method. You can typically improve on the accuracy of the calculation using Romberg integration and the performance of Gaussian quadrature depends significantly on the nature of the integrand.

Example 1: Integrating a 2D function over fixed limits

Suppose we wanted to check the normalization of the built-in two-dimensional Gauss function. The user-defined function would be:

```
Function myIntegrand1(pWave,inX,inY)
    Wave/Z pWave
    Variable inX,inY
    return Gauss(inX,50,10,inY,50,10)
End
```

To perform the integration, execute:

```
Integrate2D outerLowerLimit=0, outerUpperLimit=100, innerLowerLimit=0,
    innerUpperLimit=100, integrand=myIntegrand1
Print/D V_Value
```

Example 2: Integrating a 2D function using function limits

In this example we compute the volume of a sphere of radius 1. To do so we use symmetry and integrate the volume in one octant only. The limits of integration are [0,1] in the x direction and [0,sqrt(1-x^2)] in the y direction. In this case we need to define two user-defined functions: one for the integrand and one for the upper limit of the inner integral:

```
Function myIntegrand2(pWave,inX,inY)
   Wave/Z pWave
   Variable inx,iny
   Variable r2=inX^2+inY^2
   if(r2>=1)
      return 0
   else
      return sqrt(1-r2)
   endif
End

Function y2Func(pWave,inX)
   Wave pWave
   Variable inX
   return sqrt(1-inX^2)
End
```

To perform the integration, execute:

```
Integrate2D outerLowerLimit=0, outerUpperLimit=1, innerLowerLimit=0,
   innerUpperFunc=y2Func, integrand=myIntegrand2
Print/D 4*pi/3 -8*V_Value // Calculation error
```

Note that the integrand function tests that r2 does not exceed 1. This is because the sqrt function would return a NaN if r2>1 which can happen due to floating point rounding errors. Returning a NaN terminates the integration.

Example 3: Integrating a 2D wave using fixed limits

In this example we compute the volume between the surface defined by the wave my2DWave and the plane z=0 with integration limits [0,1] in the x-direction and [0,2] in the y-direction. For simplicity we set the wave's value to be a constant (pi).

```
Make/O/N=(5,9) my2DWave=pi
SetScale/P x 0,0.3,"", my2DWave
SetScale/P y 0,0.4,"", my2DWave
Integrate2D outerLowerLimit=0, outerupperLimit=1, innerlowerLimit=0, innerUpperLimit=2,
   srcWave=my2DWave
Print/D 2*pi-V_Value          // Calculation error
```

See Also

Integrate, **Differentiate**, **Integrate1D**, **area**, **areaXY**

IntegrateODE

IntegrateODE [***flags***] ***derivFunc*, *cwaveName*, *ywaveSpec***

The IntegrateODE operation calculates a numerical solution to a set of coupled ordinary differential equations by integrating derivatives. The derivatives are user-specified via a user-defined function, *derivFunc*. The equations must be a set of first-order equations; a single second-order equation must be recast as two first-order equations, a third-order equation to three first order equations, etc. For more details on how to write the function, see **Solving Differential Equations** on page III-322.

IntegrateODE offers two ways to specify the values of the independent variable (commonly called X or t) at which output Y values are recorded. You can specify the X values or you can request a “free-run” mode.

The algorithms used by IntegrateODE calculate results at intervals that vary according to the characteristics of the ODE system and the required accuracy. You can set specific X values where you need output (see the /X flag below) and arrangements will be made to get values at those specific X values. In between those values, IntegrateODE will calculate whatever spacing is needed, but intermediate values will not be output to you.

If you specify free-run mode, IntegrateODE will simply output all steps taken regardless of the spacing of the X values that results.

Parameters

cwaveName Name of wave containing constant coefficients to be passed to *derivFunc*.

derivFunc Name of user function that calculates derivatives. For details on the form of the function, see **Solving Differential Equations** on page III-322.

ywaveSpec Specifies a wave or waves to receive calculated results. The waves also contain initial conditions. The *ywaveSpec* can have either of two forms:

ywaveName: *ywaveName* is a single, multicolumn wave having one column for each equation in your equation set (if you have just one equation, the wave will be a simple 1D wave).

{*ywave0*, *ywave1*, …}: The *ywaveSpec* is a list of 1D waves, one wave for each equation. The ordering is important — it must correspond to the elements of the y wave and dydx wave passed to *derivFunc*.

Unless you use the /R flag to alter the start point, the solution to the equations is calculated at each point in the waves, starting with row 1. You must store the initial conditions in row 0.

Flags

/CVOP={*solver*, *jacobian*, *extendedErrors* [, *maxStep*]}

Selects options that affect how the Adams-Moulton and BDF integration schemes operate. This flag applies only when using /M = 2 or /M = 3. These methods are based on the CVODE package, hence the flag letters “CV”.

The *solver* parameter selects a solver method for each step. The values of *solver* can be:

solver=0: Select the default for the integration method. That is functional for /M=2 or Newton for /M=3.

solver=1: Functional solver.

solver=2: Newton solver.

The *jacobian* parameter selects the method used to approximate the jacobian matrix (matrix of df/dy_i where f is the derivative function).

jacobian=0: Full jacobian matrix.

jacobian=1: Diagonal approximation.

In both cases, the derivatives are approximated by finite differences.

In our experience, *jacobian* = 1 causes the integration to proceed by much smaller steps. It might decrease overall integration time by reducing the computation required to approximate the jacobian matrix.

If the *extendedErrors* parameter is nonzero, extra error information will be printed to the history area in case of an error during integration using /M=2 or /M=3. This extra information is mostly of the sort that will be meaningful only to WaveMetrics software engineers, but may occasionally help you to solve problems. It is printed out as a bug message (`BUG:...`) regardless of whether it is our bug or yours.

If *maxStep* is present and greater than zero, this option sets the maximum internal step size that the CVODE package is allowed to take. This is particularly useful with /M=3, as the BDF method is capable of taking extremely large steps if the derivatives don't change much. Use of this option may be necessary to make sure that the CVODE package doesn't step right over a brief excursion in, say, a forcing function. If you have something in your derivative function that may be step-like and brief, set *maxStep* to something smaller than the duration of the excursion.

If you want to set *maxStep* only, set the other three options to zero.

/E=*eps*

Adjusts the step size used in the calculations by comparing an estimate of the truncation error against a fraction of a scaled number. The fraction is *eps*. For instance, to achieve error less than one part in a thousand, set *eps* to 0.001. The number itself is set by a combination of the /F flag and possibly the wave specified with the /S flag. See **Solving Differential Equations** on page III-322 for details.

If you do not use the /E flag, *eps* is set to 10^{-6}.

For details, see **Error Monitoring** on page III-332.

/F=*errMethod*

Adjusts the step size used in the calculations by comparing an estimate of the truncation error against a scaled number. *errMethod* is a bitwise parameter that specifies what to include in that number:

bit 0: Add a constant from the error scaling wave set by the /S flag.

bit 1: Add the current value of the results.

bit 2: Add the current value of the derivatives.

bit 3: Multiply by the current step size (/M=0 or /M=1 only).

Each bit that you set of bits 0, 1, or 2 adds a term to the number; setting bit 3 multiplies the sum by the current step size to achieve a global error limit. Note that bit 3 has no effect if you use the Adams or BDF integrators (/M=2 or /M=3). See **Setting Bit Parameters** on page IV-12 for further details about bit settings.

If you don't include the /F flag, a constant is used. Unless you use the /S flag, that constant is set to 1.0.

For details, see **Error Monitoring** on page III-332.

/M=*m*

Specifies the method to use in calculating the solution.

m=0: Fifth-order Runge-Kutta-Fehlberg (default).

m=1: Bulirsch-Stoer method using Richardson extrapolation.

m=2: Adams-Moulton method.

m=3: BDF (Backwards Differentiation Formula, or Gear method). This method is the preferred method for stiff problems.

If you don't specify a method, the default is the Runge-Kutta method (*m*=0). Bulirsch-Stoer (*m*=1) should be faster than Runge-Kutta for problems with smooth solutions, but we find that this is often not the case. Simple experiments indicate that Adams-Moulton (*m*=2 may be fastest for nonstiff problems. BDF (*m*=3) is definitely the preferred one for stiff problems. Runge-Kutta is a robust method that may work on problems that fail with other methods.

/Q [= *quiet*]

quiet = 1 or simply /Q sets quiet mode. In quiet mode, no messages are printed in the history, and errors do not cause macro abort. The variable V_flag returns an error code. See **Details** for the meanings of the V_flag error codes.

/R=(*startX*,*endX*)

Specifies an X range of the waves in *ywaveSpec*.

/R=[*startP*,*endP*]

Specifies a point range in *ywaveSpec*.

If you specify the range as /R=[*startP*] then the end of the range is taken as the end of the wave. If /R is omitted, the entire wave is evaluated. If you specify only the end point (/R = [,*endP*]) the start point is taken as point 0.

You must store initial conditions in *startP*. The first point is *startP*+1.

/S=*errScaleWaveName*

If you set bit 0 of *errMethod* using the /F flag, or if you don’t include the /F flag, a constant is required for scaling the estimated error for each differential equation. By default, the constants are simply set to 1.0.

You provide custom values of the constants via the /S flag and a wave. Make a wave having one element for each derivative, set a reasonable scale factor for the corresponding equation, and set *errScaleWaveName* to the name of that wave.

If you don’t use the /S flag, the constants are all set to 1.0.

/STOP = {*stopWave*, *mode*}

Requests that IntegrateODE stop when certain conditions are met on either the solution values (Y values) or the derivatives.

stopWave contains information that IntegrateODE uses to determine when to stop.

mode controls the logical operations applied to the elements of stopWave:

mode=0: OR mode. If *stopWave* contains more than one condition, any one condition will stop the integration when it is satisfied.

mode=1: AND mode. If *stopWave* contains more than one condition, all conditions must be satisfied to cause the integration to stop.

See Details, below, for more information.

/U=*u* Update the display every *u* points. By default, it will update the display every 10 points. To disable updates, set *u* to a very large number.

/X=*xvaluespec* Specifies the values of the independent variable (commonly called x or t) at which values are to be calculated (see parameter *ywaveSpec*).

You can provide a wave or *x0* and *deltaX*:

/X = *xWaveName*

Use this form to provide a list of values for the independent variable. They can have arbitrary spacing and may increase or decrease, but should be monotonic.

If you use the /XRUN flag to specify free-run mode, /X = *xWaveName* is required. In this case, the X wave becomes an output wave and any contents are overwritten. See the description of /XRUN for details.

/X = {*x0, deltaX*}

If you use this form, *x0* is the initial value of the independent variable. This is the value at which the initial conditions apply. It will calculate the first result at *x0+deltaX*, and subsequent results with spacing of *deltaX*.

deltaX can be negative.

If you do not use the xValues keyword, it reads independent variable values from the X scaling of the results wave (see *ywaveSpec* parameter).

/XRUN={*dx0*, *Xmax*}

If *dx0* is nonzero, the output is generated in a free-running mode. That is, the output values are generated at whatever values if the independent variable (*x* or *t*) the integration method requires to achieve the requested accuracy. Thus, you will get solution points at variably-spaced X values.

The parameter *dx0* sets the step size for the first integration step. If this is smaller than necessary, the step size will increase rapidly. If it is too large for the requested accuracy, the integration method will decrease the step size as necessary.

If *dx0* is set to zero, free-run mode is not used; this is the same as if the XRUN flag is not used.

When using free-run mode, you must provide an X wave using /X = *xWaveName*. Set the first value of the wave (this is usually point zero, but may not be if you use the /R flag) to the initial value of X.

As the integration proceeds, the X value reached for each output point is written into the X wave. The integration stops when the latest step taken goes beyond *Xmax* or when the output waves are filled.

Details

The various waves you may use with the IntegrateODE operation must meet certain criteria. The wave to receive the results (*ywaveSpec*), and which contains the initial conditions, must have exactly one column for each equation in your system of equations, or you must supply a list of waves containing one wave for each equation. Because IntegrateODE can't determine how many equations there are from your function, it uses the number of columns or the number of waves in the list to determine the number of equations.

If you supply a list of waves for *ywaveSpec*, all the waves must have the same number of rows. If you supply a wave containing values of the independent variable or to receive X values in free-run mode (using /X=*waveName*) the wave must have the same number of rows as the *ywaveSpec* waves.

The wave you provide for error scaling via the /S flag must have one point for each equation. That is, one point for each *ywaveSpec* wave, or one point for each column of a multicolumn *ywaveSpec*.

By default, the display will update after each tenth point is calculated. If you display one of your *ywaveSpec* waves in a graph, you can watch the progress of the integration.

The display update may slow down the calculation considerably. Use the /U flag to change the interval between updates. To disable the updates entirely, set the update interval to a number larger than the length of the waves in *ywaveSpec*.

In free-run mode, it is impossible to predict how many output values you will get. IntegrateODE will stop when either your waves are filled, or when the X value exceeds *Xmax* set in the /XRUN flag. The best strategy is to make the waves quite large; unused rows in the waves will not be touched. To avoid having "funny" traces on a graph, you can prefill your waves with NaN. Make sure that you don't set the initial condition row and initial X value row to Nan!

Stopping IntegrateODE

In some circumstances it is useful to be able to stop the integration early, before the full number of output values has been computed. You can do this two ways: using the /STOP flag to put conditions on the solution, or by returning 1 from your derivative function.

When using /STOP={*stopWave*, *mode*}, *stopWave* must have one column for each equation in your system or, equivalently, a number of columns equal to the order of your system. Each column represents a condition on either the solution value or the derivatives for a given equation in your system.

Row 0 of *stopWave* contains a flag telling what sort of condition to apply to the solution values. If the flag is zero, that value is ignored. If the flag is 1, the integration is stopped if the solution value exceeds the value you put in row 1. If the flag is -1, integration is stopped when the solution value is less than the value in row 1.

Rows 2 and 3 work just like rows 0 and 1, but the conditions are applied to the derivatives rather than to the solution values.

If *stopWave* has two rows, only the solution values are checked. If *stopWave* has four rows, you can specify conditions on both solution values and derivatives.

You can set more than one flag value non-zero. If you do that, then *mode* determines how the multiple conditions are applied. If *mode* is 0, then any one condition can stop integration when it is satisfied. If *mode* is 1, all conditions with a non-zero flag value must be satisfied at the same time. If row 0 and row 2 have nothing but zeroes, then *stopWave* is ignored.

For further discussion, see **Stopping IntegrateODE on a Condition** on page III-335.

Output Variables

The IntegrateODE operation sets a variety of variables to give you information about the integration. These variables are updated at the same time as the display so you can monitor an integration in progress. They are:

V_ODEStepCompleted	Point number of the last result calculated.
V_ODEStepSize	Size of the last step in the calculation.
V_ODETotalSteps	Total number of steps required to arrive at the current result. In free-run mode, this is the same as V_ODEStepCompleted.
V_ODEMinStep	Minimum step size used during the entire calculation.
V_ODEFunctionCalls	The total number of calls made to your derivative function.
V_Flag	Indicates why IntegrateODE stopped.

The values for V_Flag are:

0:	Finished normally.
1:	User aborted the integration.
2:	Integration stopped because the step size became too small. That is, dX was so small that X + dX = X.
3:	IntegrateODE ran out of memory.
4:	In /M=2 or /M=3, the integrator received illegal inputs. Please report this to WaveMetrics (see **Technical Support** on page II-4 for contact details).
5:	In /M=2 or /M=3, the integrator stopped with a failure in the step solver. The method chosen may not be suitable to the problem.
6:	Indicates a bug in IntegrateODE. Please report this to WaveMetrics (see **Technical Support** on page II-4 for contact details).
7:	An error scaling factor was zero (see /S and /F flags).
8:	IntegrateODE stopped because the conditions specified by the /STOP flag were met.
9:	IntegrateODE stopped because the derivative function returned a value requesting the stop.

See Also

Solving Differential Equations on page III-322 gives the form of the derivative function, details on the error specifications and what they mean, along with several examples.

References

The Runge-Kutta (/M=0) and Bulirsh-Stoer (/M=1) methods are based on routines in Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992, and are used by permission.

The Adams-Moulton (/M=2) and BDF methods (/M=3) are based on the CVODE component of the SUNDIALS package developed at Lawrence Livermore National Laboratory:

SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. ACM Trans. Math. Softw. 31, 3 (September 2005), 363-396. Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. 2005. DOI=http://dx.doi.org/10.1145/1089014.1089020

Cohen, Scott D., and Alan C. Hindmarsh, *CVODE User Guide*, LLNL Report UCRL-MA-118618, September 1994.

The CVODE package was derived in part from the VODE package. The parts used in Igor are described in this paper:

Brown, P.N., G. D. Byrne, and A. C. Hindmarsh, VODE, a Variable-Coefficient ODE Solver, *SIAM J. Sci. Stat. Comput., 10*, 1038-1051, 1989.

interp

`interp(`*`x1`*`, `*`xwaveName`*`, `*`ywaveName`*`)`

The interp function returns a linearly interpolated value at the location x = *x1* of a curve whose X components come from the Y values of *xwaveName* and whose y components come from the Y values of *ywaveName*.

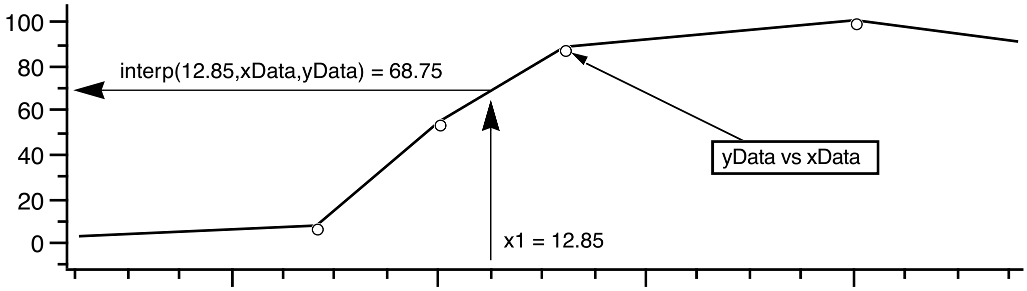
Details

interp returns nonsense if the waves are complex or if *xwaveName* is not monotonic or if either wave contains NaNs.

The interp function is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

Examples

Examples



See Also

Interpolate2

The **Loess**, **ImageInterpolate**, **Interpolate3D**, and **Interp3DPath** operations.

The **Interp2D**, **Interp3D** and **ContourZ** functions.

Interp2D

`Interp2D(`*`srcWaveName`*`, `*`xV`*`, `*`y`*`)`

The Interp2D function returns a double precision number as the bilinear interpolation value at the specified coordinates of the source wave. It returns NaN if the point is outside the source wave domain or if the source wave is complex.

Parameters

srcWaveName is the name of a 2D wave which wave must be real.

x is the X location of the interpolated point.

y is the Y location of the interpolated point.

See Also

The **ImageInterpolate** operation. **Interpolation** on page III-114.

Interp3D

`Interp3D(`*`srcWave`*`, `*`x`*`, `*`y`*`, `*`z`*` [, `*`triangulationWave`*`])`

The Interp3D function returns an interpolated value for location P=(*x*, *y*, *z*) in a 3D scalar distribution *srcWave*.

If *srcWave* is a 3D wave containing a scalar distribution sampled on a regular lattice, the function returns a linearly interpolated value for any P=(*x*, *y*, *z*) location within the domain of *srcWave*. If P is outside the domain, the function returns NaN.

To interpolate a 3D scalar distribution that is not sampled on a regular lattice, *srcWave* is a four column 2D wave where the columns correspond to *x*, *y*, *z*, f(*z*, *y*, *z*), respectively. You must also use a "triangulation" wave for *srcWave* (use `Triangulate3D/out=1` to obtain the triangulation wave). If P falls within the convex domain defined by the tetrahedra in *triangulationWave*, the function returns the barycentric linear interpolation for P using the tetrahedron where P is found. If P is outside the convex domain the function returns NaN.

Examples

```
Make/O/N=(10,20,30) ddd=gnoise(10)
Print interp3D(ddd,1,0,0)
Print interp3D(ddd,1,1,1)
```

```
Make/O/N=(10,4) ddd=gnoise(10)
Triangulate3D/OUT=1 ddd
Print interp3D(ddd,1,0,0,M_3DVertexList)
Print interp3D(ddd,1,1,1,M_3DVertexList)
```

See Also

The **Interpolate3D** operation. **Interpolation** on page III-114.

Interp3DPath

`Interp3DPath` ***`3dWave tripletPathWave`***

The Interp3DPath operation computes the trilinear interpolated values of *3dWave* for each position specified by a row of in *tripletPathWave*, which is a 3 column wave in which the first column represents the X coordinate, the second represents the Y coordinate and the third represents the Z coordinate. Interp3DPath stores the resulting interpolated values in the wave W_Interpolated. Interp3DPath is equivalent to calling the Interp3D() function for each row in *tripletPathWave* but it is computationally more efficient.

If the position specified by the *tripletPathWave* is outside the definition of the *3dWave* or if it contains a NaN, the operation stores a NaN in the corresponding output entry.

Both *3dWave* and *tripletPathWave* can be of any numeric type. W_Interpolated is always of type NT_FP64.

See Also

The **ImageInterpolate** operation and the **Interp3D** and **interp** functions. **Interpolation** on page III-114.

Interpolate2

`Interpolate2 [`*`flags`*`] [`*`xWave`*`,]` ***`yWave`***

The Interpolate2 operation performs linear, cubic spline and smoothing cubic spline interpolation on 1D waveform or XY data. It produces output in the form of a waveform or an XY pair.

The cubic spline interpolation is based on a routine from "Numerical Recipes in C".

The smoothing spline is based on "Smoothing by Spline Functions", Christian H. Reinsch, *Numerische Mathematic* 10, 177-183 (1967).

For background information, see **The Interpolate2 Operation** on page III-115.

Parameters

xWave specifies the wave which supplies the X coordinates for the input curve. If you omit it, X coordinates are taken from the X values of *yWave*.

yWave specifies the wave which supplies the Y coordinates for the input curve.

Flags

/A=*a*	Controls pre-averaging. Pre-averaging is deprecated - use the smoothing spline (/T=3) instead. If *a* is zero, Interpolate2 does no pre-averaging. If *a* is greater than one, it specifies the number of nodes through which you want the output curve to go. Interpolate2 creates the nodes by averaging the raw input data. Pre-averaging does not work correctly with the log-spaced output mode (/I=2). This is because the pre-averaging is done on linearly-spaced intervals but the input data is log-spaced.

/E=*e*

Controls how the end points are determined for cubic spline interpolation only.

e=1: Match first derivative (default)

e=2: Match second derivative (natural)

/F=*f*

f is the smoothing factor used for the smoothing spline.

f=0 is nearly identical to the cubic spline.

f>0 gives increasing amounts of smoothing as f increases.

See **Smoothing Spline Parameters** on page III-119 for details.

/FREE

Creates output waves as free waves (see **Free Waves** on page IV-91).

/FREE is allowed only in functions. If you use /FREE then the output waves specified by /X and /Y must be either simple names or valid wave references.

/FREE was added in Igor Pro 9.00.

/I[=*i*]

Determines at what X coordinates the interpolation is done.

i=0: Gives output values at evenly-spaced X coordinates that span the X range of the input data. This is the default setting if /I is omitted.

i=1: Same as *i*=0 except that the X input values are included in the list of X coordinates at which to interpolate. This is rarely needed and is not available if no X destination wave is specified. /I is equivalent to /I=1. Both are not recommended.

i=2: Gives output values at X coordinates evenly-spaced on a logarithmic scale. Ignores any non-positive values in your input X data.

This mode is not recommended. See **Interpolating Exponential Data** on page III-118 for an alternative.

i=3: Gives output values at X coordinates that you specify by setting the X coordinates of the destination wave before calling Interpolate2. You must create your destination wave or waves before doing the

If you omit /X=*xDest* then the X coordinates come from the X values of the output waveform designated by /Y=*yDest*.

If you include /X=*xDest* then the X coordinates come from the data values of the specified X output wave.

When using /I=3, the number of output points is determined by the destination wave and the /N flag is ignored.

See **Destination X Coordinates from Destination Wave** on page III-119 for further details.

/J=*j*

Controls the use of end nodes with pre-averaging (/A). Pre-averaging is deprecated - use the smoothing spline (/T=3) instead.

j=0: Turns end nodes off.

j=1: Creates end nodes by cubic extrapolation.

j=2: Creates end nodes equal to the first and last data points of the input data set, not counting points that contain NaNs or INFs.

/N=*n*

Controls the number of points in the output wave or waves. *n* defaults to the larger of 200 and the number of points in the source waves. This value is ignored if you /I=3 (X from dest mode).

/S=*s*

s is the estimated standard deviation of the noise of the Y data. It is used for the smoothing spline only. *s* is used as the estimated standard deviation for all points in the Y data.

If neither /S nor /SWAV are present, Interpolate2 arbitrarily assumes an *s* equal to .05 times the amplitude of the Y data.

/SWAV=*stdDevWave*

stdDevWave is a wave containing the standard deviation of the noise of the Y data on a point-by-point basis. It is used for the smoothing spline only. *stdDevWave* must have the same number of points as the Y data wave.

If neither /S nor /SWAV are present, Interpolate2 arbitrarily assumes an *s* equal to .05 times the amplitude of the Y data.

/T=*t* Controls the type of interpolation performed.

t=1:	Linear interpolation
t=2:	Cubic spline interpolation (default)
t=3:	Smoothing spline interpolation

/X=*xDest* Specifies the X destination wave.

If /X is present the output is an XY pair.

If /X is omitted the output is a waveform.

The X destination wave may or may not exist when Interpolate2 is called except for "X from dest" mode (/I=3) when it must exist. Interpolate2 overwrites it if it exists.

/Y=*yDest* Specifies the Y destination wave name.

If you omit /Y, a default wave name is generated. The name of the default wave is the name of the source Y wave plus "_L" for linear interpolation, "_CS" for cubic spline or "_SS" for smoothing spline.

The Y destination wave may or may not exist when Interpolate2 is called. Interpolate2 overwrites it if it exists.

See Also

The Interpolate2 Operation on page III-115

Demos

Choose Files→Example Experiments→Feature Demos→Interpolate2 Log Demo.

References

"*Numerical Recipes in C*" for cubic spline.

"Smoothing by Spline Functions", Christian H. Reinsch, *Numerische Mathematic* 10, 177-183 (1967).

Interpolate3D

```
Interpolate3D [/Z ] /RNGX={x0,dx,nx}/RNGY={y0,dy,ny}/RNGZ={z0,dz,nz}
   /DEST=dataFolderAndName, triangulationWave=tWave, srcWave=sWave
```

The Interpolate3D operation uses a precomputed triangulation of *sWave* (see **Triangulate3D**) to calculate regularly spaced interpolated values from an irregularly spaced source. The interpolated values are calculated for a lattice defined by the range flags /RNGX, /RNGY, and /RNGZ. *sWave* is a 4 column wave where the first three columns contain the spatial coordinates and the fourth column contains the associated scalar value. Interpolate3D is essentially equivalent to calling the **Interp3D** function for each interpolated point in the range but it is much more efficient.

Parameters

triangulationWave=*tWave*

Specifies a 2D index wave, *tWave*, in which each row corresponds to one tetrahedron and each column (tetrahedron vertex) is represented by an index of a row in *sWave*. Use **Triangulate3D** with /OUT=1 to obtain *tWave*.

srcWave=*sWave* Specifies a real-valued 4 column 2D source wave, *sWave*, in which columns correspond to *x*, *y*, *z*, f(*x*, *y*, *z*). Requires that the domain occupied by the set of {*x*, *y*, *z*} be convex.

Flags

/DEST=*dataFolderAndName*

Saves the result in the specified destination wave. The destination wave will be created or overwritten if it already exists. *dataFolderAndName* can include a full or partial path with the wave name.

/RNGX={*x0*,*dx*,*nx*} Specifies the range along the X-axis. The interpolated values start at *x0*. There are *nx* equally spaced interpolated values where the last value is at *x0*+(*nx*-1)*dx*. If you would like to interpolate the data for a single plane you can set the appropriate number of values to 1. For example, a YZ plane would have *nx*=1.

/RNGY={*y0*,*dy*,*ny*} Specifies the range along the Y-axis. The interpolated values start at *y0*. There are *nx* equally spaced interpolated values where the last value is at *y0*+(*ny*-1)*dy*. If you would like to interpolate the data for a single plane you can set the appropriate number of values to 1. For example, a XZ plane would have *ny*=1.

/RNGZ={*z0*,*dz*,*nz*} Specifies the range along the Z-axis. The interpolated values start at *z0*. There are *nz* equally spaced interpolated values where the last value is at *z0*+(*nz*-1)*dz*. If you would like to interpolate the data for a single plane you can set the appropriate number of values to 1. For example, a XY plane would have *nz*=1.

/Z No error reporting.

Details

The triangulation wave defines a set of tetrahedra that spans the convex source domain. If the requested range consists of points outside the domain, the interpolated values will be set to NaN. The interpolation process for points inside the convex domain consists of first finding the tetrahedron in which the point resides and then linearly interpolating the scalar value using the barycentric coordinate of the interpolated point.

In some cases the interpolation may result in NaN values for points that are clearly inside the convex domain. This may happen when the preceding Triangulate3D results in tetrahedra that are too thin. You can try using Triangulate3D with the flag /OUT=4 to get more specific information about the triangulation. Alternatively you can introduce a slight random perturbation to the input source wave before the triangulation.

Example

```
Function Interpolate3DDemo()
   Make/O/N=(50,4) ddd=gnoise(20)    // First 3 columns store XYZ coordinates
   ddd[][3]=ddd[p][2]                // Fourth column stores a scalar which is set to z
   Triangulate3D ddd                 // Perform the triangulation
   Wave M_3dVertexList
   Interpolate3D /RNGX={-30,1,80}/RNGY={-40,1,80}/RNGZ={-40,1,80}
         /DEST=W_Interp triangulationWave=M_3dVertexList,srcWave=ddd
End
```

See Also

The **Triangulate3D** operation and the **Interp3D** function. **Interpolation** on page III-114.

References

Schneider, P.J., and D. H. Eberly, *Geometric Tools for Computer Graphics*, Morgan Kaufmann, 2003.

inverseErf

inverseErf(*x*)

The inverseErf function returns the inverse of the error function.

Details

The function is calculated using rational approximations in several regions followed by one iteration of Halley’s algorithm.

See Also

The **erf**, **erfc**, **dawson**, and **inverseErfc** functions.

inverseErfc

inverseErfc(x)

The inverseErfc function returns the inverse of the complementary error function.

Details

The function is calculated using rational approximations in several regions followed by one iteration of Halley’s algorithm.

See Also

The **erf**, **erfc**, **erfcw**, **dawson**, and **inverseErf** functions.

ItemsInList

ItemsInList(*listStr* [, *listSepStr*])

The ItemsInList function returns the number of items in *listStr*. *listStr* should contain items separated by *listSepStr* which typically is ";".

Use ItemsInList to count the number of items in a string containing a list of items separated by a string (usually a single ASCII character), such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

If *listStr* is `""` then 0 is returned.

listSepStr is optional. If missing, *listSepStr* defaults to “;”.

Details

listStr is searched for item strings bound by *listSepStr* on the left and right.

An item can be empty. The lists `"abc;def;;ghi"` and `";abc;def;;ghi;"` have four items (the third item is `""`).

listStr is treated as if it ends with a *listSepStr* even if it doesn’t. The search is case-sensitive.

In Igor6, only the first byte of *listSepStr* was used. In Igor7 and later, all bytes are used.

Examples

```
Print ItemsInList("wave0;wave1;wave1#1;")         // prints 3
Print ItemsInList("key1=val1,key2=val2", ",")     // prints 2
Print ItemsInList("1 \t 2 \t", "\t")              // prints 2
Print ItemsInList(";")                            // prints 1
Print ItemsInList(";;")                           // prints 2
Print ItemsInList(";a;")                          // prints 2
Print ItemsInList(";;;")                          // prints 3
```

See Also

The **AddListItem**, **StringFromList**, **FindListItem**, **RemoveListItem**, **RemoveFromList**, **WaveList**, **TraceNameList**, **StringList**, **VariableList**, and **FunctionList** functions.

j

j

The j function returns the loop index of the 2nd innermost iterate loop in a macro. Not to be used in a function. iterate loops are archaic and should not be used.

JCAMPLoadWave

`JCAMPLoadWave [`*`flags`*`] [`*`fileNameStr`*`]`

The JCAMPLoadWave operation loads data from the named JCAMP-DX file into waves.

Parameters

If *fileNameStr* is omitted or is "", or if the /I flag is used, JCAMPLoadWave presents an Open File dialog from which you can choose the file to load.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/A	Automatically assigns arbitrary wave names using "wave" as the base name. Skips names already in use.
/A=*baseName*	Same as /A but it automatically assigns wave names of the form *baseName*0, *baseName*1.
/D	Creates double-precision waves. If omitted, JCAMPLoadWave creates single-precision waves.
/H	Reads header information from JCAMP file. If you include /W, this information is stored in the wave note. If you include /V, it is stored in header variables.
/I	Forces JCAMPLoadWave to display an Open File dialog even if the file is fully specified via /P and *fileNameStr*.
/N	Same as /A except that, instead of choosing names that are not in use, it overwrites existing waves.
/N=*baseName*	Same as /N except that it automatically assigns wave names of the form *baseName0*, *baseName1*.
/O	Overwrite existing waves in case of a name conflict.
/P=*pathName*	Specifies the folder to look in for *fileNameStr*. *pathName* is the name of an existing symbolic path.
/Q	Suppresses the normal messages in the history area.
/R	Reads data from file and creates Igor waves.
/V	Set variables from header information if /H is also present
/W	Stores header information in the wave note if /R and /H are also present.

Details

The /N flag instructs Igor to automatically name new waves "wave", or *baseName* if /N=baseName is used, plus a number. The number starts from zero and increments by one for each wave loaded from the file. If the resulting name conflicts with an existing wave, the existing wave is overwritten.

The /A flag is like /N except that Igor skips names already in use.

Output Variables

JCAMPLoadWave sets the following output variables:

`V_flag`	Number of waves loaded or -1 if an error occurs during the file load.
`S_fileName`	Name of the file being loaded.
`S_path`	File system path to the folder containing the file.
`S_waveNames`	Semicolon-separated list of the names of loaded waves.

S_path uses Macintosh path syntax (e.g., "`hd:FolderA:FolderB:`"), even on Windows. It includes a trailing colon.

When JCAMPLoadWave presents an Open File dialog and the user cancels, V_flag is set to 0 and S_fileName is set to "".

In addition, if the /V flag is used, variables are created corresponding to JCAMP-DX labels in the header. See **Variables Set By JCAMPLoadWave** on page II-168 for details.

Example

```
Function LoadJCAMP(pathName, fileName)
   String pathName     // Name of Igor symbolic path or ""
   String fileName     // Full path, partial path or simple file name

   JCAMPLoadWave/P=$pathName fileName
   if (V_Flag == 0)
      Print "No waves were loaded"
      return -1
   endif

   NVAR VJC_NPOINTS
   Printf "Number of points: %d\r", VJC_NPOINTS

   SVAR SJC_YUNITS
   Printf "Y Units: %s\r", SJC_YUNITS

   return 0
End
```

See Also

Loading JCAMP Files on page II-168

JacobiCn

`JacobiCn(`*`x`*`, `*`k`*`)`

The JacobiCn function returns the Jacobian elliptic function cn(x,k) for real x and modulus k with

$$0 < k^2 < 1.$$

The JacobiCn function was added in Igor Pro 7.00.

See Also

JacobiSn

Reference

F. W. J. Olver, D. W. Lozier, R. F. Boisvert, and C. W. Clark, editors, *NIST Handbook of Mathematical Functions*, chapter 22. Cambridge University Press, New York, NY, 2010.

JacobiSn

`JacobiSn(`*`x`*`, `*`k`*`)`

The JacobiSn function returns the Jacobian elliptic function sn(x,k) for real x and modulus k with

$$0 < k^2 < 1.$$

The JacobiSn function was added in Igor Pro 7.00.

See Also

JacobiCn

Reference

F. W. J. Olver, D. W. Lozier, R. F. Boisvert, and C. W. Clark, editors, *NIST Handbook of Mathematical Functions*, chapter 22. Cambridge University Press, New York, NY, 2010.

jlim

jlim

The jlim function returns the ending loop count for the 2nd inner most iterate loop. Not to be used in a function. iterate loops are archaic and should not be used.

JointHistogram

JointHistogram [*flags*] *wave1*, *wave2* [, *wave3*, *wave4*]

The JointHistogram computes 2D, 3D and 4D joint histograms of data provided in the input waves. The input waves must be 1D real numeric waves having the same number of points. The result of the operation is stored in the multidimensional wave M_JointHistogram in the current data folder or in the wave specified via the /DEST flag.

This operation was added in Igor Pro 7.00.

Flags

/BINS={*nx*, *ny*, *nz*, *nt*}

Specifies the number of bins along each axis. Set the number of bins for unused axes to zero. If the number of bins is non-zero, then the flags /XBMT, /YBMT, /ZBMT, and /TBMT are overridden.

/C

Sets the output wave scaling so that the values in each axis are centered in the bins. By default, wave scaling of the output wave is set with values at the left bin edges. This flag has no effect on axes where bins are specified by using /XBWV, /YBWV, /ZBWV or /TBWV.

/E

Excludes outliers. This flag is relevant only if there are one or more bin waves specified by using /XBWV, /YBWV, /ZBWV or /TBWV. By default values that might fall below the first bin or above the last bin are folded into the first and last bin respectively. These values (outliers) are excluded from the joint histogram when you use /E. See /P below for the way outliers affect the probability calculation.

/DEST=*destWave*

Specifies the output wave created by the operation. If you omit /DEST then the output wave is M_JointHistogram in the current data folder.

It is an error to specify a destination which is the same as one of the input waves.

When used in a user-defined function, the JointHistogram operation by default creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-72 for details.

/P=*mode*

Normalizes the histogram to a probability density.

Use *mode*=0 to count all points, including possible outliers but excluding non-finite values, in the probability calculation. This is the default setting.

Use *mode*=1 if you want to completely exclude outliers from the normalization.

When outliers are excluded the output wave sums to 1. When they are included the sum of the output wave is smaller by the ratio of the number of outliers to the total number of points in the histogram.

/W=*weightWave*

Creates a weighted histogram. Instead of adding a single count to the appropriate bin, the corresponding value from *weightWave* is added to the bin. *weightWave* may be any real number type.

/XBMT=*method*
/YBMT=*method*
/ZBMT=*method*
/TBMT=*method*

These flags specify which method is used to set the bins. By default method=0. These flags are overridden by /BINS if a non-zero value is specified for a given axis and by /XBWV, /YBWV, /ZBWV and /TBWV.

See **JointHistogram Binning Methods** below for details.

/XBWV=*xBinWave*
/YBWV=*yBinWave*
/ZBWV=*zBinWave*
/TBWV=*tBinWave*

Specifies the exact bins for a corresponding axis.

The wave must be 1D real numeric wave with monotonically increasing finite values and must contain a minimum of 3 data points.

The values in a bin wave specify the edges of the bins. A bin wave with N points defines n-1 bins. In the case of a 3-point bin wave, the first point corresponds to the minimum value of the first bin, the second point is the boundary between the two bins and the last point is the upper limit of the second bin.

These flags override the corresponding bin specification set via /BINS, /XBMT, /YBMT, /ZBMT and /TBMT.

/Z [=*zval*]

Suppresses error reporting.

/Z is equivalent to /Z=1 and /Z=0 is equivalent to not using the /Z flag at all.

Details

The input waves must be 1D real numeric waves. If one or more waves contain a non-finite value (a NaN or INF) the corresponding row of all waves are not counted in the joint histogram.

The optional waves that define user-specified bins must be real numeric waves and contain a monotonically increasing values. Using non-finite values in user-specified bin waves may lead to unpredictable results.

JointHistogram Binning Methods

The /XBMT, /YBMT, /ZBMT and /TBMT flags set the binning method for the X, Y, Z and T dimensions respectively.

These flags are overridden by /BINS if a non-zero value is specified for a given axis and by /XBWV, /YBWV, /ZBWV and /TBWV.

The *method* parameter is defined as follows:

method=0: 128 equally spaced bins between the minimum and maximum of the input data. This is the default setting.

method=1: The number of bins is computed using Sturges' method where

numBins=1+log2(N).

N is the number of data points in each wave. The bins are distributed so that they include the minimum and maximum values.

method=2: The number of bins is computed using Scott's method where the optimal bin width is given by

binWidth=$3.49*\sigma*N^{-1/3}$.

σ is the standard deviation of the distribution and N is the number of points. The bins are distributed so that they include the minimum and maximum values.

method=3: Freedman-Diaconis method where

binWidth=2*IQR*N-1/3,

where IQR is the interquartile distance (see **StatsQuantiles**) and the bins are evenly distributed between the minimum and maximum values.

Bin selection methods are described at: http://en.wikipedia.org/wiki/Histogram

Example: 2D Joint Histogram

```
Make/O/N=(1000) xwave=gnoise(10), ywave=gnoise(5)
JointHistogram/BINS={20,30} xwave,ywave
NewImage M_JointHistogram
```

Example: 2D Joint Histogram using one bins wave

```
Make/O/N=(1000) xwave=gnoise(10), ywave=gnoise(5)
Make/O/N=3 xBinsWave={-8,0,14}
JointHistogram/BINS={0,30}/XBWV=xBinsWave/E xwave,ywave
Display; AppendImage/T M_JointHistogram vs {xBinsWave,*}
```

Example: 3D Joint Histogram

```
Make/O/N=(1000) xwave=gnoise(10), ywave=gnoise(5), zwave=enoise(4)
JointHistogram/BINS={15,15,20,0} xwave,ywave,zwave
NewImage M_JointHistogram
ModifyImage M_JointHistogram plane=10
```

See Also

Histogram, **ImageHistogram**

JulianToDate

`JulianToDate(`*`julianDay`*`, `*`format`*`)`

The JulianToDate function returns a date string containing the day, month, and year. The input *julianDay* is truncated to an integer.

Parameters

julianDay is the Julian day to be converted.

format specifies the format of the returned date string.

format	**Date String**
0	mm/dd/year
1	dd/mm/year
2	Tuesday November 15, 2002
3	year mm dd
4	year/mm/dd

See Also

The **dateToJulian** function.

For more information about the Julian calendar see: `<http://www.tondering.dk/claus/calendar.html>`.

KillBackground

`KillBackground`

The KillBackground operation kills the unnamed background task.

KillBackground works only with the unnamed background task. New code should used named background tasks instead. See **Background Tasks** on page IV-319 for details.

Details

You can not call KillBackground from within the background function itself. However, if you return 1 from the background function, instead of the normal 0, Igor will terminate the background task.

See Also

The **BackgroundInfo**, **CtrlBackground**, **CtrlNamedBackground**, **SetBackground**, and **SetProcessSleep** operations; and **Background Tasks** on page IV-319.

KillControl

`KillControl` `[`**`/W=`*`winName`***`]` ***`controlName`***

The KillControl operation kills the named control in the top or specified graph or panel window or subwindow.

If the named control does not exist, KillControl does not complain.

Flags

/W=*winName*	Looks for the control in the named graph or panel window or subwindow. If /W is omitted, KillControl looks in the top graph or panel window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

KillDataFolder

`KillDataFolder [/Z] `*`dataFolderSpec`*

The KillDataFolder operation kills the specified data folder and everything in it including other data folders.

However, if *dataFolderSpec* is the name of a data folder reference variable that refers to a free data folder, the variable is cleared and the data folder is killed only if this is the last reference to that free data folder.

Flags

/Z	No error reporting (except for setting V_flag). Does not halt function execution.

Parameters

dataFolderSpec can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

Details

If specified data folder is the current data folder or contains the current data folder then Igor makes its parent the new current data folder.

For legacy reasons, a null data folder is taken to be the current data folder. This can happen when using a $ expression where the string might possibly evaluate to "".

It is legal to kill the root data folder. In this case the root data folder itself is not killed but everything in it is killed.

KillDataFolder generates an error if any of the waves involved are in use. In this case, nothing is killed.

KillDataFolder generates an error if any of the waves involved are in use. In this case, nothing is killed. Execution ceases unless /Z is specified.

The variable V_flag is set to 0 when there is no error, otherwise it is an error code.

Examples

```
KillDataFolder foo          // Kills foo in the current data folder.
KillDataFolder :bar:foo     // Kills foo in bar in current data folder.
String str= "root:foo"
KillDataFolder $str         // Kills foo in the root data folder.
```

See Also

Chapter II-8, **Data Folders** and the **KillStrings**, **KillVariables**, and **KillWaves** operations.

KillFIFO

`KillFIFO `*`FIFOName`*

The KillFIFO operation discards the named FIFO.

Details

FIFOs are used for data acquisition.

If there is an output or review file associated with the FIFO, KillFIFO closes the file. If the FIFO is used by an XOP, you should call the XOP to release the FIFO before killing it.

See Also

See **FIFOs and Charts** on page IV-313 for information about FIFOs and data acquisition.

KillFreeAxis

`KillFreeAxis [/W=`*`winName`*`]` ***`axisName`***

The KillFreeAxis operation removes a free axis specified by *axisName* from a graph window or subwindow.

Flags

/W=*winName*	Kills the free axis in the named graph window or subwindow. If /W is omitted, it acts on the top graph window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

Only an axis created by **NewFreeAxis** can be killed and only if no traces or images are attached to the axis.

See Also

The **NewFreeAxis** operation.

KillPath

`KillPath [/A/Z]` ***`pathName`***

The KillPath operation removes a path from the list of symbolic paths. KillPath is a newer name for the **RemovePath** operation.

Flags

/A	Kills all symbolic paths in the experiment except for the built-in paths. Omit *pathName* if you use /A.
/Z	Does not generate an error if a path to be killed is a built-in path or does not exist or is in use. To kill all paths in the experiment, use `KillPath/A/Z`.

Details

You can’t kill the built-in paths “home” and “Igor”.

A symbolic path is "in use" if the current experiment contains shared waves or notebooks loaded from the associated disk folder.

Prior to Igor Pro 9.00, the KillPath operation prevented killing a symbolic path used to a load shared wave if the shared wave file was in the current data folder. It now prevents killing a symbolic path used to load a shared wave in any data folder. Use the /Z flag to suppress error reporting if this change creates problems for you.

See Also

Symbolic Paths on page II-22, **NewPath**

KillPICTs

`KillPICTs [/A/Z] [`*`PICTName`*` [, `*`PICTName`*`]…]`

The KillPICTs operation removes one or more named pictures from the current Igor experiment.

Flags

/A	Kills all pictures in the experiment.
/Z	Does not generate an error if a picture to be killed is in use or does not exist. To kill all pictures in the experiment, use `KillPICTs/A/Z`.

Details

You can not kill a picture that is used in a graph or page layout.

Warning: You *can* kill a picture that is referenced from a graph or layout recreation macro. If you do, the graph or layout can not be completely recreated. Use the Find dialog (Edit menu) to locate references in the procedure window to a named picture you want to kill.

See Also

See **Pictures** on page III-509 for general information on how Igor handles pictures.

KillStrings

`KillStrings` `[`**`/A/Z`**`] [`***`stringName`***` [, `***`stringName`***`]…]`

The KillStrings operation discards the named global strings.

Flags

/A	Kills all global strings in the current data folder. If you use /A, omit *stringName*.
/Z	Does not generate an error if a global string to be killed does not exist. To kill all global strings in the current data folder, use `KillStrings/A/Z`.

KillVariables

`KillVariables` `[`**`/A/Z`**`] [`***`variableName`***` [, `***`variableName`***`]…]`

The KillVariables operation discards the named global numeric variables.

Flags

/A	Kills all global variables in the current data folder. If you use /A, omit *variableName*.
/Z	Does not generate an error if a global variable to be killed does not exist. To kill all global variables in the current data folder, use `KillVariables/A/Z`.

KillWaves

`KillWaves` `[`***`flags`***`]` ***`waveName`*** `[, `***`waveName`***`]…`

The KillWaves operation destroys the named waves.

Flags

/A	Kills all waves in the current data folder. If you use /A, omit *waveName*s.
/F	Deletes the Igor binary wave file from which *waveName* was loaded.
/Z	Does not generate an error if a wave to be killed is in use or does not exist.

Details

The memory the waves occupied becomes available for other uses. You can’t kill a wave used in a graph or table or which is reserved by an XOP.

XOPs reserve a wave by sending the OBJINUSE message.

For functions compiled with the obsolete rtGlobals=0 setting, you also can't kill a wave referenced from a user-defined function.

Examples

```
KillWaves/A/Z       // kill waves not in use in current data folder
```

KillWindow

`KillWindow` `[`***`flags`***`]` ***`winName`***

The KillWindow operation kills or closes a specified window or subwindow without saving a recreation macro.

Parameters

winName is the name of an existing window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Flags

/Z Does not generate an error if the specified window does not exist.

See Also

The **DoWindow** operation.

KMeans

```
KMeans [flags] populationWave
```

The KMeans operation analyzes the clustering of data in *populationWave* using an iterative algorithm. The result of KMeans is a specification of the classes which is saved in the wave M_KMClasses in the current data folder. Optional results include the distribution of class members (W_KMMembers) and the inter-class distances. *populationWave* is a 2D wave in which columns correspond to members of the population and rows contain the dimensional information.

Flags

/CAN Analyzes the clustering by computing Euclidean distances between the means of the resulting classes. The resulting distances are stored in an NxN square matrix where N is the number of classes. Self distances (along the diagonal) or distances involving classes that did not survive the iterations are filled with NaN. Also saves the wave W_KMDispersion, which contains the sum of the distances between the center of each class and all its members. Distances are evaluated using the method specified by /DIST.

/DEAD=*method* Specifies how the algorithm should handle “dead” classes, which are those that lose all members in a given iteration.

method=1: Remove the class if it looses all members.

method=2: Default; keeps the last value of the mean vector in case the class might get new members in a subsequent iteration.

method=3: Assigns the class a random mean vector.

/DIST=*mode* Specifies how the class distances are evaluated.

mode=1: Distance is evaluated as the sum of the absolute values (also known as Manhattan distance).

mode=2: Default; distance is evaluated as Euclidean distance.

/INIT=*method* Specifies the initialization method.

method=1: Random assignment of members of the population to a class.

method=2: User-specified mean values (/INW).

method=3: Default; initialize classes using values of a random selection from the population.

/INW=*iWave* Sets the initial classes. The number of rows of *iWave* equals the dimensionality of the class and the number of columns of *iWave* is the number of classes. For example, if we want to initialize 5 classes in a problem that involves position in two dimensions then *iWave* must have 2 rows and 5 columns. The number of rows must also match the number of rows in *populationWave*.

/NCLS=*num* Sets the number of classes in the data. If the initialization method uses specific means (/INIT=2) then the number of columns of *iWave* (see /INW) must match *num*. The default number of classes is 2.

/OUT=*format* Specifies the format for the results.

format=1: Output only the specification of the classes in the 2D wave M_KMClasses (default). Each column in M_KMClasses represents a class. The number of rows in M_KMClasses is equal to the number of rows in `populationWave+1`. The last row contains the number of class members. The remaining rows represent the center of the class. For example, if *populationWave* has two rows then the dimensionality of the problem is 2 and M_KMClasses has 3 rows with the first row containing the first components of each class center, the second row containing the second components of each class center and the third row containing the number of elements in each class.

format=2: Output (in addition to M_KMClasses) the class membership in the wave W_KMMembers. The rows in this 1D wave correspond to sequential members of *populationWave* and the entries correspond to the (zero based) column number in M_KMClasses.

/SEED=*val* Sets the seed for a new sequence in the pseudo-random number generator that is used by the operation. *val* must be an integer greater than zero.

By changing the sequence you may be able to find new solutions or just make the process converge at a different rate.

/TER=*method* Determines when the iterations stop.

method=1: User-specified number of iterations (/TERN).

method=2: Default; continue iterating until no more than a fixed number of elements change classes in one iteration (TERN).

/TERN=*num* Specifies the termination number. The meaning of the number is determined by /TER above. By default, the termination *method*=2 and the default value of the maximum number of elements that change classes in one iteration is 5% of the size of the population.

/Z No error reporting. If an error occurs, sets V_flag to -1 but does not halt function execution.

Details

KMeans uses an iterative algorithm to analyze the clustering of data. The algorithm is not guaranteed to find a global optimum (maximum likelihood solution) so the operation provides various flags to control when the iterations terminate. You can determine if the operation iterates a fixed number of times or loops until at most a specified maximum number of elements change class membership in a single iteration. If you are computing KMeans in more than one dimension you should pay attention to the relative magnitudes of the data in each dimension. For example, if your data is distributed on the interval [0,1] in the first dimension and on the interval [0,1e7] in the second dimension, the operation will be biased by the much larger magnitude of values in the second dimension.

Examples

Create data with 3 classes:

```
Make/O/N=(1,128) jack=4+gnoise(1)
jack[0][15,50]+=10
jack[0][60,]+=20
```

Perform KMeans looking for 5 classes:

```
KMeans/init=1/out=1/ter=1/dead=1/tern=1000/ncls=5 jack
Print M_KMClasses

  M_KMClasses[0][0]= {24.1439,68}
  M_KMClasses[0][1]= {14.1026,36}
  M_KMClasses[0][2]= {4.01537,24}
```

See Also

The **FPClustering** function.

Label

`Label [/W=`*`winName`*`/Z]` ***`axisName, labelStr`***

The Label operation labels the named axis with *labelStr*.

Parameters

axisName is the name of an existing axis in the top graph. It is usually one of “left”, “right”, “top” or “bottom”, though it may also be the name of a free axis such as “VertCrossing”.

labelStr contains the text that labels the axis.

Flags

/W=*winName*	Adds axis label in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.
/Z	No errors generated if the named axis doesn’t exist. Used for style macros.

Details

labelStr can contain escape codes which affect subsequent characters in the text. An escape code is introduced by a backslash character. In a literal string, you must enter two backslashes to produce one. See **Backslashes in Annotation Escape Sequences** on page III-58 for details.

Using escape codes you can change the font, size, style and color of text, create superscripts and subscripts, create dynamically-updated text, insert legend symbols, and apply other effects. See **Annotation Escape Codes** on page III-53 for details.

Some escape codes insert text based on axis properties. See **Axis Label Escape Codes** on page III-57 for details.

The characters “<??>” in an axis label indicate that you specified an invalid escape code or used a font that is not available.

See Also

See **Annotation Escape Codes** on page III-53. See the **Legend** operation about wave symbols.

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

laguerre

`laguerre(`*`n`*`, `*`x`*`)`

The laguerre function returns the Laguerre polynomial of degree *n* (positive integer) and argument *x*. The polynomials satisfy the recurrence relation:

$$(n+1)\mathrm{Laguerre}(n+1,x) = (2n+1-x)\mathrm{Laguerre}(n,x) - n\,\mathrm{Laguerre}(n-1,x),$$

with the initial conditions

$$\mathrm{Laguerre}(0,x) = 1$$

and

$$\mathrm{Laguerre}(1,x) = 1 - x.$$

See Also

The **laguerreA**, **laguerreGauss**, **chebyshev**, **chebyshevU**, **hermite**, **hermiteGauss**, and **legendreA** functions.

laguerreA

`laguerreA(n, k, x)`

The laguerreA function returns the associated Laguerre polynomial of degree *n* (positive integer), index *k* (non-negative integer) and argument *x*. The associated Laguerre polynomials are defined by

$$L_n^k(x) = (-1)^k \frac{d^k}{dx^k}\left[L_{n+k}(x)\right],$$

where $L_{n+k}(x)$ is the Laguerre polynomial.

See Also

The **laguerre** and **laguerreGauss** functions.

References

Arfken, G., *Mathematical Methods for Physicists*, Academic Press, New York, 1985.

laguerreGauss

`laguerreGauss(p, m, r)`

The laguerreGauss function returns the normalized product of the associated Laguerre polynomials and a Gaussian. This function is typically encountered in solutions to physical problems where it represents the radial solution with an additional factor exp(i*m*ϕ) which is not included in this case. The LaguerreGauss is given by

$$U_{pm}(r) = \sqrt{\frac{2p!}{\pi(m+p)!}}\left(r\sqrt{2}\right)^m L_p^m\left(2r^2\right)\exp\left(-r^2\right).$$

See Also

The **laguerre**, **laguerreA**, and **hermiteGauss** functions.

LambertW

`LambertW(z, branch)`

The LambertW function returns the complex value of Lambert's W function for complex *z* and integer index *branch*. The function can be defined through its inverse,

$$z = w\mathrm{e}^w .$$

Since w is multivalued, the branch parameter is used to differentiate between solutions for the equation.

The LambertW function was added in Igor Pro 7.00.

Details

IGOR's LambertW uses complex input and output. You can use LambertW in real expressions but you must make sure that you are not calling the function in a range where its imaginary part is non-zero.

The average accuracy of the function defined by cabs(*z*-w*exp(w)) in the region |real(*z*)|<10, |imag(*z*)|<10 is 5e-14. In general the accuracy decreases with increasing |branch| and with increasing distance from the origin in the z-plane.

IGOR uses a hybrid algorithm to compute the function which requires longer computation times in the presence of numerical instabilities.

References

R.M. Corless, G.H. Gonnet, D.E.G. Hare, D.J. Jeffrey and D.E. Knuth, "On Lambert W Function", Advances in Computational Mathematics 5: 329-359

Layout

`Layout` `[`***`flags`***`] [`***`objectSpec`***` [, `***`objectSpec`***`]…][`**`as`** ***`titleStr`***`]`

The Layout operation creates a page layout.

Note: The Layout operation is antiquated and can not be used in user-defined functions. For new programming, use the **NewLayout** operation instead.

Parameters

All of the parameters are optional.

Each *objectSpec* parameter identifies a graph, table, textbox or picture to be added to the layout. An object specification can also specify the location and size of the object, whether the object should have a frame or not, whether it should be transparent or opaque, and whether it should be displayed in high fidelity or not. See **Details**.

titleStr is a string expression containing the layout’s title. If not specified, Igor will provide one which identifies the objects displayed in the graph.

Flags

/A=(*rows*,*cols*) Specifies rows and columns for tiling or stacking.

/B=(*r*,*g*,*b*[,*a*]) Specifies the background color for the layout. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque white.

/C=*colorOnScreen* Obsolete. In ancient times, this flag switched the screen display of the layout between black and white and color. It is still accepted but has no effect.

/G=*g* Specifies grout, the spacing between tiled objects. Units are points unless /I, /M, or /R are specified.

/HIDE=*h* Hides (h = 1) or shows (h = 0, default) the window.

/I Specifies that coordinates are in inches. This affects subsequent /G, /W, and *objectSpec* coordinates. Coordinates are relative to the top/left corner of the paper.

/K=*k* Specifies window behavior when the user attempts to close it.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

k=3: Hides the window.

If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.

/M Specifies that coordinates are in centimeters. This affects subsequent /G, /W, and *objectSpec* coordinates. Coordinates are relative to the top/left corner of the paper.

/P=*orientation* *orientation* is either Portrait or Landscape (*e.g.*, `Layout/P= Landscape`). This controls the orientation of the page in the layout. See **Details**.

If you use the /P flag, you should make it the first flag in the Layout operation. This is necessary because the orientation of the page affects the behavior of other flags, such as /T and /G.

/R Specifies that coordinates are in percent. This affects subsequent /G, /W, and *objectSpec* coordinates. For /W, coordinates are as a percent of the main screen. For /G and *objectSpec*, coordinates are relative to the top/left corner of the printing part of the page.

/S Stacks objects.

/T Tiles objects.

/W=(*left, top, right, bottom*)

Gives the layout window a specific location and size on the screen. Coordinates for /W are in points unless /I or /M are specified.

Details

When you create a new page layout window, if preferences are enabled, the page size is determined by the preferred page size as set via the Capture Layout Prefs dialog. If preferences are disabled, as is usually the case when executing a procedure, the page is set to the factory default size.

If you use the /P flag, you should make it the first flag in the Layout operation. This is necessary because the orientation of the page affects the behavior of other flags, such as /T and /G.

The form of an *objectSpec* is:

objectName [**(*objLeft*, *objTop*, *objRight*, *objBottom*)**][**/O=*objType***][**/F=*frame***]
[**/T=*trans***][**/D=*fidelity***]

objectName can be the name of an existing graph, table or picture. It can also be the name of an object that does not yet exist. In this case it is called a “dummy object”.

objectSpec can be specified using a string by using the $ operator, but the entire *objectSpec* must be in the string.

Here are some examples of valid usage:

```
Layout Graph0

Layout/I Graph0(1, 1, 6, 5)/F=1

String s = "Graph0"
Layout/I $s

String s = "Graph0(1, 1, 6, 5)/F=1"
Layout/I $s         // Entire object spec is in string.
```

The object’s coordinates are determined as follows:

- If *objectName* is followed by a coordinates specification in (*objLeft*, *objTop*, *objRight*, *objBottom*) form then this sets the object’s coordinates. The units for the coordinates are points unless the /I or /M flag was present in which case the units are inches or centimeters respectively.
- If the object coordinates are not specified explicitly but the Layout/S flag was present then the object is stacked. If the Layout/T flag was present then the object is tiled, and if the Layout/A=(*rows*,*cols*) flag is present, tiling is performed using that number of rows and columns.
- If the object’s coordinates are not determined by these rules then the object is set to a default size and is stacked.

Each object has a type (graph, table, textbox or picture) determined as follows:

O=*objType* If the *objectName*/O=*objType* flag is present then it determines the object’s type:

objType=1:	Graph.
objType=2:	Table.
objType=8:	Picture.
objType=32:	Textbox.

If there is no /O flag and *objectName* is the name of an existing graph, table or picture, then the object type is graph, table or picture.

If the object’s type is not determined by the above rules and *objectName* contains “Table”, “PICT”, or “TextBox”, then the object type is table, picture or textbox.

If the object’s type is not specified by any of the above rules, it is taken to be a graph type object.

The remaining flags have the following meanings:

/D=*fidelity* Controls the drawing of the layout object:

fidelity=0:	Low fidelity display.
fidelity=1:	High fidelity display (default).

/F=*frame*	Controls the object frame:	
	frame=0:	No frame.
	frame=1:	Single frame (default).
	frame=2:	Double frame.
	frame=3:	Triple frame.
	frame=4:	Shadow frame.
/T=*trans*	Controls the transparency of the layout object:	
	trans=0:	Opaque (default).
	trans=1:	Transparent. For this to be effective, the object itself must also be transparent. Annotations have their own transparent/opaque settings. Graphs are transparent only if their backgrounds are white. Pictures may have been created transparent or opaque, and Igor cannot make an inherently opaque picture transparent.

See Also

The **NewLayout** and **LayoutInfo** operations. See Chapter II-18, **Page Layouts**.

Layout

`Layout`

Layout is a procedure subtype keyword that identifies a macro as being a page layout recreation macro. It is automatically used when Igor creates a window recreation macro for a layout. See **Procedure Subtypes** on page IV-204 and **Killing and Recreating a Layout** on page II-477 for details.

See Also

See Chapter II-18, **Page Layouts**.

LayoutInfo

`LayoutInfo(`*`winNameStr`*`, `*`itemNameStr`*`)`

The LayoutInfo function returns a string containing a semicolon-separated list of keywords and values that describe an object in the active page of a page layout or overall properties of the layout. The main purpose of LayoutInfo is to allow an advanced Igor programmer to write a procedure which formats or arranges objects.

winNameStr is the name of an existing page layout window or `""` to refer to the top layout.

itemNameStr is a string expression containing one of the following:

- The name (e.g., "Graph0") of a layout object in the active page to get information about that object.
- An object instance (e.g., "Graph0#0" or "Graph0#1") to get information about a particular instance of an object in the active page. This is of use only in the unusual situation when the same object appears in the active page multiple times. "Graph0#0" is equivalent to "Graph0". "Graph0#1" is the second occurrence of Graph0 in the active page.
- An integer object index starting from zero to get information about an object referenced by its position in the active page in the layout. Zero refers to the first object going from back to front in the page.
- The word "Layout" to get overall information about the layout.

Details

In cases 1, 2 and 3 above, where *itemNameStr* references an object, the returned string contains the following keywords, with a semicolon after each keyword-value pair.

Keyword	Information Following Keyword
`FIDELITY`	Object fidelity expressed as a code usable in a ModifyLayout fidelity command.
`FRAME`	Object frame expressed as a code usable in a ModifyLayout frame command.
`HEIGHT`	Object height in points.

Keyword	Information Following Keyword
INDEX	Object position in back-to-front order in the active page of the layout, starting from zero.
LEFT	Object left position in points.
NAME	The name of the object.
SELECTED	Zero if the object is not selected or nonzero if it is selected.
TOP	Object top position in points.
TRANS	Object transparency expressed as a code usable in a ModifyLayout trans command.
TYPE	Object type which is one of: Graph, Table, Picture, or Textbox.
WIDTH	Object width in points.

In case 4 above, where *itemNameStr* is "Layout", the returned string contains the following keywords, with a semicolon after each keyword-value pair.

Keyword	Information Following Keyword
BGRGB	Layout background color expressed as <red>, <green>, <blue> where each color is a value from 0 to 65535.
MAG	Layout magnification: 0.25, 0.5, 1.0, or 2.0.
NUMOBJECTS	Total number of objects in the active page of the layout.
NUMSELECTED	Number of selected objects in the active page of the layout.
PAGE	A rectangle defining the part of the paper that is inside the margins, expressed in points. The format is <left>, <top>, <right>, <bottom>.
CURRENTPAGENUM	One-based page number of the currently active page. Added in Igor Pro 7.00.
NUMPAGES	Total number of pages in the layout. Added in Igor Pro 7.00.
PAPER	A rectangle defining the bounds of the paper, expressed in points. The format is <left>, <top>, <right>, <bottom>.
SELECTED	A comma-separated list of the names of selected objects in the active page of the layout.
UNITS	Units used to display object locations and sizes. This will be one of the following: 0 for points, 1 for inches, 2 for centimeters.

LayoutInfo returns "" in the following situations:

- *winNameStr* is "" and there are no layout windows.
- *winNameStr* is a name but there are no layout windows with that name.
- *itemNameStr* is not "Layout" and is not the name or index of an existing object.

Examples

This example sets the background color of all selected graphs in the active page of a particular page layout to the color specified by `red`, `green`, and `blue`, which are numbers from 0 to 65535.

```
Function SetLayoutGraphsBackgroundColor(layoutName,red,green,blue)
   String layoutName       // Name of layout or "" for top layout.
   Variable red, green, blue

   Variable index
   String info
   Variable selected
   String indexStr
   String objectTypeStr
   String graphNameStr
```

```
    index = 0
    do
        sprintf indexStr, "%d", index
        info = LayoutInfo(layoutName, indexStr)
        if (strlen(info) == 0)
            break      // No more objects
        endif

        selected = NumberByKey("SELECTED", info)
        if (selected)
            objectTypeStr = StringByKey("TYPE", info)
            if (CmpStr(objectTypeStr,"Graph") == 0)// This is a graph?
                graphNameStr = StringByKey("NAME", info)
                ModifyGraph/W=$graphNameStr wbRGB=(red,green,blue)
                ModifyGraph/W=$graphNameStr gbRGB=(red,green,blue)
            endif
        endif

        index += 1
    while(1)
End
```

See Also

The **Layout** operation. See Chapter II-18, **Page Layouts**.

LayoutMarquee

`LayoutMarquee`

LayoutMarquee is a procedure subtype keyword that puts the name of the procedure in the layout Marquee menu. See **Marquee Menu as Input Device** on page IV-163 for details.

See Also

See Chapter II-18, **Page Layouts**.

LayoutPageAction

`LayoutPageAction [/W=`*`winName`*`]` `[`***`keyword = value`***` [, `***`keyword = value`***` …]]`

The LayoutPageAction operation adds, deletes, reorders, or adjusts the sizes of layout pages.

The LayoutPageAction operation was added in Igor Pro 7.00.

Parameters

appendPage — Appends a new page.

insertPage=*page* — Inserts a new page before *page*.

Page numbers start from 1. Pass 0 for *page* to insert before the first page.

page=*page* — Makes *page* the active page.

Page numbers start from 1.

deletePage=*page* — Deletes *page*. This action cannot be undone.

Page numbers start from 1.

reorderPages={*anchorPage*, *page1*, ...}

Reorders the pages so that *page1* and any others appear before *anchorPage*, in the same order as their appearance in the command.

Page numbers start from 1.

size=(*width*, *height*) — Sets the global page dimensions for the layout to width and height, specified in units of points.

size(*page*)=(*width*, *height*)

Sets the dimensions of *page* to *width* and *height*, specified in units of points.

Using this keyword with *page* set to -1 modifies the global page dimensions for the layout.

margins=(*leftMargin*, *topMargin*, *rightMargin*, *bottomMargin*)

Sets the global page margins for the layout to the specified values, expressed in units of points.

margins(*page*)=(*leftMargin*, *topMargin*, *rightMargin*, *bottomMargin*)

Sets the margins of specified page to these values, expressed in units of points.

Page numbers start from 1.

Passing -1 for page sets the global margins for the layout.

Flags

/W=*winName* Modifies the named layout. When omitted, the actions affect the top layout.

Details

Page numbers starts from 1. Use *page*=0 to refer to the active page.

The layout as a whole has a size and margins. These are called "global" dimensions and govern all pages by default. You can set the global dimensions using the size and margins keyword without specifying a particular page.

You can override the dimensions for a given page using size(*page*) and margins(*page*) to specify custom dimensions.

Use size(*page*)=(0,0) to revert the specified page to the global layout dimensions. This reverts both the page size and its margins.

See Also

Page Layouts on page II-475, **NewLayout**, **ModifyLayout**

LayoutSlideShow

`LayoutSlideShow [/W=winName]` `[`***`keyword = value`***` [, `***`keyword = value`***` …]]`

The LayoutSlideShow operation starts, stops, or modifies a slideshow that displays the pages of a page layout.

The LayoutSlideShow operation was added in Igor Pro 7.00.

Parameters

autoMode=*a* Controls whether the presentation will advance between slides automatically (*a*=1) or manually (*a*=0). Use the delay keyword to control the delay between automatic transitions.

delay=*d* *d* is the number of seconds to wait between slide transitions when running in auto mode.

otherScreenContents=*o*

Controls what is displayed on any additional screens that may be connected.

o=0: Other screens show the presentation.

o=1: Other screens show a presenter's view with additional information. Use the presentersView keyword to control the contents of this view.

o=2: Other screens show a presenter's view with additional information. Use the presentersView keyword to control the contents of this view.

page=*p* Causes the slideshow to start from page *p*. *p* is a page number starting from 1. This keyword has no effect unless the start keyword is also present.

presentersView=*p* Controls what is displayed on the screens that show the presenter's view.

p is a bitfield of flags:

Bit 0:	Show the next page.
Bit 1:	Show the current time.
Bit 2:	Show the elapsed time.

Setting Bit Parameters on page IV-12 for details about bit settings.

scaleMode=*s* Specifies how the pages are scaled to fit the screen.

s=0:	No scaling. Pages are drawn at actual size even if they are much larger or smaller than the screen size.
s=1:	All pages are individually scaled to the screen size.
s=2:	All pages are scaled by the same factor so that the largest page fits on the screen. This preserves the relative sizes of the pages.

screen=*s* Specifies the screen to be used for the main presentation. Use *s*=1 to use the primary screen. Use **IgorInfo** to determine the number of available screens.

start Starts the slideshow.

stop Stops the slideshow. You can also stop it by pressing the escape key.

wrapMode=*w* Controls what happens when the presentation reaches the last page in the slideshow.

w=0:	Advancing to the next page has no effect.
w=1:	Advancing to the next page causes the slideshow to wrap around to the first page.
w=2:	Advancing to the next page causes the slideshow to stop.

Flags

/W= *winName* *winName* is the name of the desired layout window. If /W is omitted or if *winName* is $"", the top layout window is used.

Details

A layout slide show can be used to present an Igor experiment to others, or to run an information kiosk.

Any changes to the layout window during a slide show are automatically reflected in the slides. For example you could use a background task to update a graph so that the slides always show the latest data.

You can control a running slide show by right-clicking on the slideshow. Alternatively, use the arrow keys or a mouse click to advance to the next slide.

Press the space bar to toggle between automatic and manual advancing of the slides. Press escape to end the slideshow.

Example

```
Function DemoSlideshow()       // Press escape to end the slideshow
   NewLayout
   TextBox/C/N=text0/F=0/A=LB/X=33.57/Y=70.81 "\\Z961"
   LayoutPageAction appendpage
   TextBox/C/N=text0/F=0/A=LB/X=33.57/Y=70.81 "\\Z962"
   LayoutPageAction appendpage
   TextBox/C/N=text0/F=0/A=LB/X=33.57/Y=70.81 "\\Z963"
   LayoutSlideShow autoMode=1,delay=1,page=1,wrapMode=1,start
End
```

See Also

Page Layouts on page II-475, **NewLayout**, **LayoutPageAction**

LayoutStyle

`LayoutStyle`

LayoutStyle is a procedure subtype keyword that puts the name of the procedure in the Style pop-up menu of the New Layout dialog and in the Layout Macros menu. See **Page Layout Style Macros** on page II-498 for details.

See Also

See Chapter II-18, **Page Layouts** and **Page Layout Style Macros** on page II-498.

leftx

`leftx(`*`waveName`*`)`

The leftx function returns the X value of point 0 (the first point) of the named 1D wave. The leftx function is not multidimensional aware. The multidimensional equivalent of this function is **DimOffset**.

Details

Point 0 contains a wave's *first* value, which is usually the leftmost point when displayed in a graph. Leftx returns the value elsewhere called *x0*. The function DimOffset returns any of x0, y0, z0, or t0, for dimensions 0, 1, 2, or 3.

See Also

The **deltax** and **rightx** functions.

For multidimensional waves, see **DimDelta**, **DimOffset**, and **DimSize**.

For an explanation of waves and X scaling, see **Changing Dimension and Data Scaling** on page II-68.

Legend

`Legend` `[`***`flags`***`] [`***`legendStr`***`]`

The Legend operation puts a legend on a graph or page layout.

Parameters

legendStr contains the text that is printed in the legend.

If *legendStr* is missing or is an empty string (`""`), the text needed for a default legend is automatically generated. Legends are automatically updated when waves are appended to or removed from the graph or when you rename a wave in the graph.

legendStr can contain escape codes which affect subsequent characters in the text. An escape code is introduced by a backslash character. In a literal string, you must enter two backslashes to produce one. See **Backslashes in Annotation Escape Sequences** on page III-58 for details.

Using escape codes you can change the font, size, style and color of text, create superscripts and subscripts, create dynamically-updated text, insert legend symbols, and apply other effects. See **Annotation Escape Codes** on page III-53 for details. However normally you leave it to Igor to automatically manage the legend.

See **Legend Text** on page III-42 for a discussion of what *legendStr* may contain.

Flags

/H=*legendSymbolWidth*

Sets the width in points of the area in which to draw the wave symbols. A value of 0 means "default". This results in a width that is based on the text size in effect when the symbol is drawn. A value of 36 gives a 0.5 inch (36 points) width which is nice in most cases.

/H={*legendSymbolWidth*, *minThickness*, *maxThickness*}

This is an additional form of the /H flag. The *legendSymbolWidth* parameter works the same as described above.

The *minThickness* and *maxThickness* parameters allow you to create a legend whose line and marker thicknesses are different from the thicknesses of the associated traces in the graph. This can be handy to make the legend more readable when you use very thin lines or markers for the traces.

minThickness and *maxThickness* are values from 0.0 to 10.0. Also, setting *minThickness* to 0.0 and *maxThickness* to 0.0 (default) uses the same thicknesses for the legend symbols as for the traces.

/J

Disables the default legend mechanism so that a default legend is not created even if *legendStr* is an empty string ("") or omitted.

Window recreation macros use /J in case *legendStr* is too long to fit on the same command line as the Legend operation itself. In this case, an AppendText command appears after the Legend command to append *legendStr* to the empty legend. For really long values of *legendStr*, there may be multiple AppendText commands.

/M[=*saMeSize*]

/M or /M=1 specifies that legend markers should be the same size as the marker in the graph.

/M=0 turns same-size mode off so that the size of the marker in the legend is based on text size.

See the **TextBox** operation for documentation for all other flags.

Examples

The command `Legend` (with no parameters) creates a default legend. A default legend in a layout contains a line for each wave in each of the graphs in the layout, starting from the bottom graph and working toward the front.

The command:

```
Legend/C/N=name ""
```

changes the named existing legend to a default legend.

You can put a legend in a page layout with a command such as:

```
Legend "\s(Graph0.wave0) this is wave0"
```

This creates a legend in the layout that shows the symbol for wave0 in Graph0. The graph named in the command is usually in the layout but it doesn’t have to be.

See Also

TextBox, **Tag**, **ColorScale**, **AnnotationInfo**, **AnnotationList**.

Annotation Escape Codes on page III-53.

Legend Text on page III-42.

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

Color as f(z) Legend Example on page II-301 for a discussion of creating a legend whose symbols match the markers in a graph that uses color as f(z).

legendreA

`legendreA(`*`n`*`, `*`m`*`, `*`x`*`)`

The legendreA function returns the associated Legendre polynomial:

$$P_n^m(x)$$

where n and m are integers such that $0 \le m \le n$ and $|x| \le 1$.

References

Arfken, G., *Mathematical Methods for Physicists*, Academic Press, New York, 1985.

limit

`limit(`*`num`*`, `*`low`*`, `*`high`*`)`

The limit function returns *num*, limited to the range from *low* to *high*:

num if *low* <= *num* <= *high*.

low if *num* < *low*.

high if *num* > *high*.

Since all comparisons with NaN return false, limit will not work as expected with NaNs. If a parameter may be NaN, use **numtype** to test it before calling limit.

See Also

SelectNumber, **min**, **max**

LinearFeedbackShiftRegister

`LinearFeedbackShiftRegister` `[`***`flags`***`]`

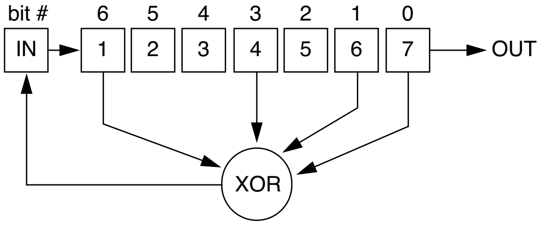
The LinearFeedbackShiftRegister operation implements a, well, linear feedback shift register, or LFSR. A LFSR is a way to produce a sequence of very bad pseudorandom numbers, or a random bit stream (that is, a random sequence of zeroes of ones that over time are nearly equal in number).

If it produces bad random numbers, why would I want to use a LFSR? A properly-configured LFSR will create a “maximal-length sequence”: a LFSR of N bits will produce 2^N-1 numbers in a quasi-random sequence without repeating. That is, it will produce all the N-bit numbers except zero. This gives the sequence good spectral properties for certain applications, and, taking the least-significant bit as the output, it creates a pseudorandom bit stream with nearly equal numbers of zeroes and ones (*nearly* means one more one than zeroes).

The LinearFeedbackShiftRegister operation generates either a wave full of the sequential states of the shift register or a wave full of ones and zeroes representing the least significant bit of the shift register.

Linear Feedback Shift Registers

A LFSR is a shift register with taps. The tap bits are XOR’ed together and the result, after the register is shifted, becomes the new most significant bit. Here is a diagram of a 7-bit LFSR:



Each successive number is generated by shifting the contents of the register (boxes 1-7) to the right, while shifting in the output of the XOR node. The XOR node samples specified bits of the register contents, generating its output ready to be shifted in. Thus, the inputs of the XOR are bits sampled *before* a shift; the output of the XOR becomes the leading bit in the register *after* a shift.

In many applications the output of interest is the stream of bits that appear in the last position. This stream of bits is a pseudorandom sequence of ones and zeroes (or ones and minus ones, or whatever other binary sequence you need).

The bits fed into the XOR node are referred to as *taps*. The taps illustrated here would be specified with the tap list 7,6,4,1. As implemented in Igor Pro, the output tap (tap 7 in the illustration) is the least significant bit, so an alternate way to express the tap list is as the binary number 1001011_2 (77_{10}).

With the right taps, a LFSR produces a maximal-length sequence. The list of sequential states in a maximal-length sequence has length 2^N-1 without repeating a state. That means that every possible N-bit nonzero number appears exactly once in the maximal-length sequence.

Maximal-length tap lists always have an even number of taps.

If you have a tap list that gives a maximal-length sequence, you can generate another tap list from it. If your tap list is (n, A, B, C) the new tap list is (n, n-C, n-B, n-A). This new tap list will generate a bit stream that is the mirror image in time of the bit stream produced by the first tap list.

Flags

/DEST=*wavename*

Specifies a wave to receive the generated sequence. With /MODE=0, the number type of the wave must have at least *nbits* bits for an integer wave, or at least an *nbit* mantissa if it is a floating-point wave. That is, /N=25 requires a double-precision wave or a 32-bit integer wave. /N=18 requires any floating-point wave or a 16- or 32-bit integer wave. If you use an integer wave, we recommend an unsigned integer wave for /N=8, 16, or 32.

If /MODE=1 is used, any number type is acceptable. See the Details for what happens if you don’t use /DEST.

If *wavename* doesn't exist, a suitable integer wave will be made.

If *wavename* already exists, LinearFeedbackShiftRegister will use it as-is. The sequence length will be taken from the wave. If the number type of the wave is not suitable, an error is issued. If the sequence length is less than the number of points in your wave, it will be truncated to match.

/FREE

In a user-defined function, makes a free wave. See **Free Waves** on page IV-91 for details.

/INIT=*initialValue*

Sets the initial value of the shift register to *initialValue*. This will also be the first value in the output for /MODE=0, or the least-significant bit of *initialValue* will be the first output for /MODE=1. You can use this initial value to restart a very long sequence from the last state of a previous run.

Default is a single 1 bit in the first position (bit *nbits*-1 for /N=*nbits*).

/LEN=*length*

Sets the length of sequence to generate. If the sequence repeats before *length* states are generated, the sequence is terminated early. If *length* is larger than the number of states in a maximal-length sequence, you will get a maximal-length sequence, or a shorter sequence if the initial value is seen again (that is, your sequence is not a maximal-length sequence).

You can specify *length* greater than the maximal-length sequence length, but it will be truncated to the maximal length.

/MAX=*index*

An internal table of tap lists gives maximal-length sequences. This table has up to 32 tap lists for each value of *nbits*. You select a tap list by setting index to a number from 0 to 31. For values of *nbits* that do not have 32 maximal-length tap lists, the table repeats. Most *nbits* values have many more than 32 possible maximal-length sequences. For each tap list in the table, another tap list can be accessed using the /MROR flag.

/MODE=*doBitStream*

Sets the output stream format.

doBitStream=0: Succession of bit register states (default).

doBitStream=1: Stream of ones and zeroes.

/MROR [=*doMirror*]

Transforms the tap list into its complementary tap list, creating a mirror-image bit stream, when you use /MROR or *doMirror*=1. Specify the tap list using /TAPS, /TAPB, or /MAX.

/N=*nbits*

Determines the number of bits in the shift register. A maximal-length sequence will have 2^{nbits}-1 states. *nbits* must be in the range of 1-32. Note that *nbits* = 1 or 2 is not very interesting.

/STOP=*stopValue*

Terminates the sequence when *stopValue* is the next shift register value. You can use this flag to generate long sequences using multiple calls to LinearFeedbackShiftRegister by storing the initial value of the first call, and setting *stopValue* to that initial value in subsequent calls.

/TAPB=*tapbits*

An alternate way to express the tap list. *tapbits* is a number in which each bit represents a tap, with bit 0 representing the tap with tap number *nbits*.

/TAPS={*t1*, *t2*, …}

Specifies the tap list. Tap numbers are in the range from 1 to *nbits*.

Details

If the /TAPS, /TAPB, or /MAX flags are absent, the maximal-length sequence corresponding to /MAX=0 is generated.

In you omit the /DEST flag, a wave named W_LFSR will be generated for you. W_LFSR is an unsigned integer wave with number type set to the minimum size for the shift register size and /MODE setting. Thus, if you set `/N=10/MODE=0`, W_LFSR will be an unsigned 16-bit integer wave.

Because W_LFSR is an unsigned integer wave, you will need to redimension the wave to a floating-point wave for many purposes. Use the **Redimension** operation or the Redimension Waves item in the Data menu.

Up to /N=18, W_LFSR will initially be created large enough to hold a maximal-length sequence, unless you request a shorter sequence using the /LEN flag. If the initial value is seen again before a maximal-length sequence is generated, it means that the tap list specified was not one that generates a maximal-length sequence, and generation is terminated. The wave is shortened to the generated sequence length.

If you set a register size greater than /N=18 and you do not use /LEN, the generated sequence will stop after 2^{18}-1 (262143) states. Note that beyond some N, it will be impossible to create a wave large enough to hold a maximal-length sequence.

Some tap lists do not generate maximal-length sequences but also do not repeat the initial value. In that case, the generated sequence will be of maximal length but will contain repeated subsequences. The V_flag variable will be set to 0 if the sequence was not a maximal-length sequence, or 1 if it was. If /LEN=*length* values were generated, V_flag is set to 2.

If you specify your own wave using /DEST, the sequence length will be the same as the length of your wave. Your wave will be resized if a shorter sequence is generated.

Generating Long Sequences in Smaller Segments

Very long maximal-length sequences will not fit in the largest wave you can make. It may also be more convenient to make multiple, small fragments of a longer sequence. You can do this using the /INIT, /STOP, and /LEN flags, along with the V_nextValue variable. Here is an example of making 1000-point subsequences from a 16-bit maximal-length sequence:

```
// Start with the first 1000 states, with initial value of 1
LinearFeedbackShiftRegister/N=16/LEN=1000/INIT=1
// Restart using the V_nextValue variable to continue the sequence
// /STOP=1 sets the stopping value to the first initial value
LinearFeedbackShiftRegister/N=16/LEN=1000/INIT=(V_nextValue)/STOP=1
// Continue…
LinearFeedbackShiftRegister/N=16/LEN=1000/INIT=(V_nextValue)/STOP=1
```

Variables

The LinearFeedbackShiftRegister operation returns information in the following variables:

`V_flag` Set to zero when a nonmaximal-length sequence was detected. This occurs if the initial value is seen again before a maximal-length sequence was generated, or if a maximal-length sequence was generated but the final state was not the same as the initial state.

Set to 1 when a maximal-length sequence was generated.

Set to 2 when the sequence was limited by /LEN=*length* or by the default limit of 2^{18}-1 (262143) states.

`V_tapValue` Set to the binary representation of the tap sequence used. That is, you can generate the same sequence using `/TAPB=V_tapValue`. If you use /MROR=1, V_tapValue reflects that setting. It will also give the actual tap value used when you specify the a maximal-length sequence using the /MAX flag.

`V_nextValue` Set to the next value beyond the last generated register state. This can be used to restart a truncated sequence.

Examples

Generate a 16-bit maximal-length sequence and reprocess the output values to be centered on zero and normalized to a maximum value of 1:

```
LinearFeedbackShiftRegister/N=16
Redimension/D W_LFSR
```

```
W_LFSR -= 2^15
W_LFSR /= 2^15-1
```

Another way to do the same thing that avoids the Redimension operation, which could lead to fragmentation of memory:

```
Make/D/N=(2^16-1) LFSR_output
LinearFeedbackShiftRegister/N=16/DEST=LFSR_output
LFSR_output -= 2^15
LFSR_output /= 2^15-1
```

Make a bit stream with random +1 and -1 instead of 0 and 1:

```
LinearFeedbackShiftRegister/N=16/MODE=1
Redimension/B W_LFSR
W_LFSR = W_LFSR*2-1
```

See Also

If you really need random numbers, we provide high-quality RNG's that return random deviates from a number of distributions. See **enoise**, **gnoise**, and others.

References

A discussion of LFSR's can be found in "Generation of Random Bits" (Section 7.4) in Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992. They refer to "primitive polynomials modulo 2" and do not use the name Linear Feedback Shift Register, but it is the same thing. We use an implementation equivalent to their Method I.

ListBox

`ListBox` `[`**`/Z`**`]` ***`ctrlName`*** `[`***`keyword = value`*** `[,` ***`keyword = value`*** `…]]`

The ListBox operation creates or modifies the named control that displays, in the target window, a list from which the user can select any number of items.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the ListBox control to be created or changed.

The following keyword=value parameters are supported:

align=*alignment*

Sets the alignment mode of the control. The alignment mode controls the interpretation of the *leftOrRight* parameter to the pos keyword. The align keyword was added in Igor Pro 8.00.

If *alignment*=0 (default), *leftOrRight* specifies the position of the left end of the control and the left end position remains fixed if the control size is changed.

If *alignment*=1, *leftOrRight* specifies the position of the right end of the control and the right end position remains fixed if the control size is changed.

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **Button** and **DefaultGUIControls** for more appearance details.

clickEventModifiers=*modifierSelector*

Selects modifier keys to ignore when processing clicks to start editing a cell or when toggling a checkbox. That is, use this keyword if you want to prevent a shift-click (for instance) from togging checkbox cells. Allows the action procedure to receive mousedown events with those modifiers without interfering with actions on the part of the listbox control.

modifierSelector is a bit pattern with a bit for each modifier key; sum these values to get the desired combination of modifiers:

modifierSelector=1:	Control key (Macintosh only)
modifierSelector=2:	Option (Macintosh) or Alt (Windows)
modifierSelector=4:	Context click (right click on Windows, control-click on Macintosh)
modifierSelector=8:	Shift key
modifierSelector=16:	Cmd key (Macintosh) or Ctrl key (Windows)
modifierSelector=32:	Caps lock key

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

col=*c*

Sets the left-most visible column (user scrolling will change this). The list is scrolled horizontally as far as possible. Sometimes this won't be far enough to actually make column *c* the first column, but it will at least be visible. Use *c* =1 to put the left edge of column 1 (the *second* column) at the left edge of the list.

colorWave=*cw*

Specifies a 3-column (RGB) or 4-column (RGBA) numeric wave. Used in conjunction with planes in selWave to define foreground and background colors for individual cells. Values range from 65535 (full on) to 0.

disable=*d*

Sets user editability of the control.

d=0:	Normal.
d=1:	Hide.
d=2:	Draw in gray state; disable control action.

editStyle=*e*

Sets the style for cells designated as editable (see selWave, bit 1).

e=0:	Uses a light blue background (default).
e=1:	Draws a frame around the cell with a white background.
e=2:	Combines the frame with the blue background. The background in all cases can be overridden using the colorWave parameter.

focusRing=*fr*

Enables or disables the drawing of a rectangle indicating keyboard focus:

fr=0:	Focus rectangle will not be drawn.
fr=1:	Focus rectangle will be drawn (default).

On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.

font="*fontName*"

Sets the font used for the list box items, e.g., `font="Helvetica"`.

frame=*f*

Specifies the list box frame style.

f=0:	No frame.
f=1:	Simple rectangle.
f=2:	3D well.
f=3:	3D raised.
f=4:	Text well style.

fsize=s

Sets list box font size.

fstyle=*fs*

fs is a bitwise parameter with each bit controlling one aspect of the font style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

hScroll=*h*

Scrolls the list to the right by *h* pixels (user scrolling will change this). *h* is the total amount of horizontal scrolling, not an increment from the current scroll position: *h* will be the value returned in the V_horizScroll variable by **ControlInfo**.

help={*helpStr*}

Sets the help for the control.

helpStr is limited to 1970 bytes (255 in Igor Pro 8 and before).

You can insert a line break by putting "\r" in a quoted string.

helpWave=*w*

Text wave containing text for tooltips for individual listbox cells. See ListBox Help below. Added in Igor Pro 8.00.

keySelectCol=*col*

Sets scan column number *col* when doing keyboard selection. Default is to scan column zero.

listWave=*w*

A 1D or 2D text wave containing the list contents.

mode=*m*

List selection mode specifying how many list selections can be made at a time.

m=0:	No selection allowed.
m=1:	One or zero selection allowed.
m=2:	One and only one selection allowed.
m=3:	Multiple, but not disjoint, selections allowed.
m=4:	Multiple and disjoint selections allowed.

When multiple columns are used, you can enable individual cells to be selected using modes 5, 6, 7, and 8 in analogy to *m*=1-4. When using *m*=3 or 4 with multiple columns, only the first column of the selWave is used to indicate selections. Checkboxes and editing mode, however, use all cells even in modes 0-4.

Modes 9 and 10 are the same as modes 4 and 8 except they use different selection rules and require testing bit 3 as well as bit 0 in selWave. In modes 4 and 8, a shift click toggles individual cells or rows, but in modes 9 and 10, the Command (*Macintosh*) or Ctrl (*Windows*) key toggles individual cells or rows whereas Shift defines a rectangular selection. T o determine if a cell is selected, perform a bitwise AND with `0x09`.

proc=*p*

Set name of user function proc to be called upon certain events. See discussion below.

pos={*leftOrRight*,*top*}

Sets the position in **Control Panel Units** of the top/left corner of the control if its alignment mode is 0 or the top/right corner of the control if its alignment mode is 1. See the align keyword above for details.

pos+={*dx*,*dy*}

Offsets the position of the list box in **Control Panel Units**.

row=*r*

r is desired top row (user scrolling will change this). Use a value of -1 to scroll to the first selected cell (if any). Combine with selRow to select a row and to ensure it is visible (modes 1 and 2).

selCol=*c*

Defines the selected column when mode is 5 or 6 and no selWave is used. To read this value, use **ControlInfo** and the V_selCol variable.

selRow=*s* Defines the selected row when mode is 1 or 2; when no selWave is used, it is defined by modes 5 or 6. Use -1 for no selection.

To read this value, use **ControlInfo** and the V_value variable.

selWave=*sw* *sw* is a numeric wave with the same dimensions as listWave. It is optional for modes 0-2, 5 and 6 and required in all other modes.

In modes greater than 2, *sw* indicates which cells are selected. In modes 1 and 2 use **ControlInfo** to find out which row is selected.

In all modes *sw* defines which cells are editable or function as checkboxes or disclosure controls.

Numeric values are treated as integers with individual bits defined as follows:

Bit 0 (0x01):	Cell is selected.
Bit 1 (0x02):	Cell is editable.
Bit 2 (0x04):	Cell editing requires a double click.
Bit 3 (0x08):	Current shift selection.
Bit 4 (0x10):	Current state of a checkbox cell.
Bit 5 (0x20):	Cell is a checkbox.
Bit 6 (0x40):	Cell is a disclosure cell. Drawn as a disclosure triangle (*Macintosh*) or a treeview expansion node (*Windows*).
Bit 7 (0x80):	Cell is disabled. Clicks on the cell are ignored and it is drawn with a grayed appearance. Added in Igor Pro 8.00. Because of technical issues, this bit currently does not prevent the selection of the cell using the arrow keys.

In modes 3 and 4 bit 0 is set only in column zero of a multicolumn listbox.

Other bits are reserved. Additional dimensions are used for color info. See the discussion for colorWave. selWave is not required for modes 5 and 6.

setEditCell={*row*,*col*,*selStart*,*selEnd*}

Initiates edit mode for the cell at *row*, *col*. An error is reported if *row* or *col* is less than zero. Nothing happens and no error is reported if *row*, *col* is beyond the limits of the listbox, or if the cell has not been made editable by setting bit 1 of *selWave*.

selStart and *selEnd* set the range of bytes that are selected when editing is initiated; 0 is the start of the text. If there are N bytes in the listbox cell, setting *selStart* or *selEnd* to N or greater moves the start or end of the selection to the point after the last character. Setting *selStart* and *selEnd* to the same value selects no characters and the insertion point is set to *selStart*. Setting *selStart* to -1 always causes all characters to be selected.

size={*width*,*height*} Sets list box size in **Control Panel Units**.

special={*kind*,*height*,*style*}

Specifies special cell formatting or contents.

kind=0: Normal text but with specified *height* (if nonzero). Use a *style* of 1 to autocalculate widths based on the entire list contents. In this case, user widths are taken to be minimums and the last is not repeated.

kind=1: Text taken to be the names of graphs or tables. Images of the graphs or tables are displayed in the cells. Use a *style* of 0 to display just the presentation portion of the graph or 1 to display it entirely. For tables, only the presentation portion is displayed.

kind=2: Text taken to be the names of pictures. Images are displayed in the cells.

kind=3: Displays a PNG, TIFF, or JPEG image. You can obtain binary picture data using SavePICT.

kind=4: Displays raw text without any escape code processing. Added in Igor Pro 8.00 to allow display of LaTeX expressions which make heavy use of backslashes.

For *kind*=1 or 2, *height* may be zero to auto-set cell height to same as width or a specific value.

titleWave=*w* Specifies a text wave containing titles for the listbox columns, instead of using the list wave dimension labels. Each row is the title for one column; if you have N columns you must have a wave with N rows.

userColumnResize=*u* Enables resizing the list columns using the mouse.

u=0: Columns are not resizable (default). The widths parameter still works, though.

u=1: User can resize columns by dragging the column dividers.

When resizing a column without Option, Alt, or Shift modifiers (a "normal" resizing), any width added to the column is subtracted from the following column (if any).

When resizing while pressing Option (*Macintosh*) or Alt (*Windows*), only columns following the dragged divider will move (the same way table columns are resized).

When pressing Shift, all columns are set to the same width as the column being resized. If the total widths of all columns is less than the width of the listbox, then each column expands to fill the available width.

userdata=*UDStr* Sets the unnamed user data to *UDStr*.

userdata(*UDName*)=*UDStr*

Sets the named user data, *UDName*, to *UDStr*.

userdata+=*UDStr* Appends *UDStr* to the current unnamed user data.

userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current named user data, *UDName*.

widths={*w1*,*w2*,…} Optional list of minimum column widths in screen pixels. If more columns than widths, the last is repeated. If total of widths is greater than list box width then a horizontal scroll bar will appear. If total is less than available width then each expands proportionally.

widths+={*w1*,*w2*,…}   Additional column widths. Because only 2500 bytes fit on a command line, lists with many columns may require multiple widths+= parameters to define all the column widths. However if all the widths are the same, widths+= is not needed; just use:

```
ListBox ctrlName widths={sameWidth}
```

win=*winName*   Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Flags

/Z   No error reporting.

Details

If the list wave has column dimension labels (see **SetDimLabel**), then those will be used as column titles. Note that a 1D wave is subtly different from a 1 column 2D wave. The former does not have any columns and therefore no column dimension labels.

Alternately, use a text wave with the titleWave keyword to specify column titles.

Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.

For instance, to make a title bold use the \f01 escape sequence:

```
SetDimLabel 1, columnNum, $"\\f01My Label", textWave
```

If you can't fit title text within the 255 byte limit (styled text can be especially long), use the *titleWave* keyword with a text wave. The wave must have as many rows as the list wave has columns. When using a title wave, there are no restrictions on the number of bytes or on what characters you can use.

This example uses a title wave to add a red up-arrow graph marker to the end of a centered title:

```
Make/O/T/N=(numColumns) columnTitles
columnTitles[colNum]="\\JCThis is the title\\K(65535,0,0)\\k(65535,0,0)\\W523"
ListBox list0 titleWave=columnTitles
```

That's a 51-byte title that results in 19 characters or symbols that you actually see. \JC requests centered text, \K sets the text color (which colors the inside of the graph marker),\k sets the marker stroke color, and \W523 inserts a down-pointing triangular graph marker.

When using modes that allow multiple selections, use Shift to extend or add to the selection.

You can specify individual cells as being editable by setting bit 1 (counting from zero on the right) in selWave. The user can start editing a cell by either clicking in it or, if the cell is selected, by pressing Enter (or Return). When finished, the user can press Enter to accept the changes or can press Escape to reject changes. The user may also press Up or Down Arrow to accept changes and begin editing the next editable cell in a column. Likewise, Tab and Shift-Tab moves to the next or previous column in a row. If bit 2 of selWave is set then a double click will be required rather than a single click. **Note**: in edit mode, Tab and Shift-Tab are used to move left and right because the Left and Right Arrow keys are used to move the text entry cursor left and right.

When the listbox has keyboard focus (either by tabbing to the list box or by clicking in the box), the keyboard arrow keys move a cell selection (or row depending on mode). When not in cell edit mode, Tab and Shift-Tab move the keyboard focus to other objects in the window. The Home, End, Page Up, and Page Down keys affect the vertical scroll bar.

When the listbox has focus, the user may type the first few chars of an entry in the list to select that entry. Only the first column is used. If a match is not found then nothing is done. The search is case insensitive.

ListBox Help

You can provide a text wave with help strings to show a tooltip for individual cells or rows of a listbox in place of the generic help string provided by the help keyword. In general, the string in helpWave[*listbox row*][*listbox column*] is used. If your help wave is a 1D wave, then the listbox column is ignored, and helpWave[*listbox row*] is used in all columns.

Note that a 1D wave (Make/N=*rows*) is not the same as a 2D wave with one column (Make/N=(*rows*,1)). A 2D wave with one column provides help for column 0 only and the generic help string is shown for column 1 and beyond. A 1D wave provides help for all columns.

If the help wave has an empty string for a given row and column then the generic help string is shown for the corresponding listbox cell. If the help wave has too few rows or columns, then the generic help string is shown for the cells outside the range of the help wave.

Listbox Action Procedure

The action procedure for a ListBox control takes a predefined structure `WMListboxAction` as a parameter to the function:

```
Function ActionProcName(LB_Struct) : ListboxControl
   STRUCT WMListboxAction &LB_Struct
   …
   return 0
End
```

The “`: ListboxControl`” designation tells Igor to include this procedure in the Procedure pop-up menu in the List Box Control dialog.

See **WMListboxAction** for details on the WMListboxAction structure.

Although the return value is not currently used, action procedures should always return zero.

You may see an old format listbox action procedure in old code:

```
Function MyListboxProc(ctrlName,row,col,event) : ListboxControl
   String ctrlName      // name of this control
   Variable row         // row if click in interior, -1 if click in title
   Variable col         // column number
   Variable event       // event code
   …
   return 0             // other return values reserved
End
```

This old format should not be used in new code.

Specifying Cell Color

The background and foreground (text) color of individual cells may be defined by providing colorWave in conjunction with specific planes in selWave. The planes in selWave are taken to be integer indexes into colorWave. The planes are defined by specific dimension labels and not by specific plane numbers. To provide foreground colors, define a plane labeled “foreColors” that contains the desired index values. Likewise define and fill a plane labeled “backColors” for background colors. The value 0 is special and indicates that the default colors should be used. Note that if you have a one column list for which you want to supply colors, the selWave needs to be three dimensional but with just one column. Here is an example:

```
Make/T/N=(5,1) tw= "row "+num2str(p)        // 5 row, 1 col text wave (2D)
Make/B/U/N=(5,1,2) sw                       // 5 row, 1 col, 2 plane byte wave
Make/O/W/U myColors={{0,0,0},{65535,0,0},{0,65535,0},{0,0,65535},{0,65535,65535}}
MatrixTranspose myColors      // above was easier to enter as 3 rows, 5 cols

NewPanel
ListBox lb,mode=3,listWave= tw,selWave= sw,size={200,100},colorWave=myColors
sw[][][1]= p                          // arbitrary index values into plane 1
```

Now, execute the following commands one at a time and observe the results:

```
SetDimLabel 2,1,backColors,sw         // define plane 1 as background colors

SetDimLabel 2,1,foreColors,sw         // redefine plane 1 as foreground colors

sw[][][%foreColors]= 4-p              // change the color index values
```

In the above example, the selWave was defined as unsigned byte. If you need more than 254 colors, you will have to use a larger number size.

The color wave may have four columns instead of three, with the fourth specifying transparency (65535=fully opaque, 0=fully transparent). Transparency of the background color is of limited utility as it just shows the white background color through the cell color, resulting in pastel shades. Transparency of the foreground color permits the cell background color to show through the text color.

Checkboxes in Cells

You can cause a cell to contain a checkbox by setting bit 5 in selWave. The title (if any) is taken from listWave and the results (selected/deselected) is bit 4 of selWave. If a checkbox cell is selected then the space bar will toggle the checkbox. (Clicking a checkbox cell does not select it — use the arrow keys.)

Errors

Your listbox may be drawn with a red X and an error code. The error codes are:

Error	Meaning
E1	The listbox is too small
E2	The listWave is invalid (missing, not text or no rows)
E3	The listWave and selWave do not match in dimensions
E4	Mode is greater than 2 but no selWave was specified
E5	The number of rows in titleWave does not match the number of columns in listWave
E6	The titleWave is not a textWave

Event Queue

It is possible for a single user action to produce more than one event. For instance, pressing the up arrow key while editing to select a cell that is not visible generates event codes 4, 8, and 6. Igor calls your action procedure separately for each code.

If your code tests for a nonzero value in eventCode2, and uses it only if it is nonzero, then the event queue method will not break your code. If you have determined empirically when you need to use eventCode2 and you do not test, your code will break.

Scroll Event Warnings

Events 8, 9, and 10 report to you that the listbox has been scrolled vertically or horizontally. These events are envisioned as allowing you to keep two listboxes synchronized (you may find other uses for these events). You might use an action procedure like this one to keep two listboxes (named list0 and list1) in sync:

```
Function ListBoxProc2(LB_Struct) : ListBoxControl
    STRUCT WMListboxAction &LB_Struct

    if (LB_Struct.eventCode == 8)
        String listname
        if (CmpStr(LB_Struct.ctrlName, "list1") == 0)
            listname = "list0"
        else
            listname = "list1"
        endif
        ControlInfo $listname
        if (V_startRow != LB_Struct.row)
            listbox $listname,row=LB_Struct.row
            ControlUpdate $listname
        endif
    endif

    return 0
End
```

It is very easy to create an infinite cascade of events feeding back between the two listboxes, especially if you use event 10. When this happens, you will see your listboxes jigging up and down endlessly. The test using ControlInfo is intended to make this unlikely.

The slow response of the old-style, nonstructure action procedure can defeat the ControlInfo test by delaying the action procedure execution. If you use events 8, 9, or 10, we recommended that you use the new-style action procedure.

Note on Keystroke Event

In a keystroke event passed to a listbox action procedure the eventCode is 12. A character code is stored in the row field of the WMListboxAction structure. This works only for ASCII characters and a few special characters such as delete (8), forward delete (127) and escape (27). It does not work for non-ASCII characters such as accented characters which are represented as UTF-8 and require multiple bytes.

As explained below, many keystroke events are not sent to the listbox action procedure because they are consumed by the internal listbox code in Igor. For this reason the keystroke event for a listbox is of limited use.

The architecture of Igor controls is such that events are passed to an action procedure only after the control has used them. In the case of a keystroke event, that means that other uses of the keystroke may consume the event before the action procedure gets a chance at it. In particular, a listbox editable cell that is actively being edited consumes keystroke events, and the action procedure is not called. The only editing-related events your action procedure will get are event codes 6 and 7.

If an arrow key is pressed, and this results in the selected row or cell changing, your action procedure will not get a keystroke event. Instead, your action procedure will receive event code 4 or 5. If the arrow key causes scrolling to occur, the action procedure will also get event code 8 or 9.

Examples

Here is a simple Listbox example:

```
Make/O/T/N=30 tjack="this is row "+num2str(p)
Make/O/B/N=30 sjack=0
NewPanel /W=(19,61,319,261)
ListBox lb1,pos={42,9},size={137,94},listWave=tjack,selWave=sjack,mode= 3
Edit/W=(367,61,724,306) tjack,sjack
ModifyTable width(tjack)=148
```

Make selections in the list and note changes in the table and vice versa. Edit one of the list text values in the table and note update of the list.

Here is an example using a titleWave and styled text in the title cells. Note that the last title isn’t very long when rendered, but requires a 63 byte specification.

```
Make/O/T/N=(4,3) ListWave="row "+num2str(p)+" col "+num2str(q)
Make/O/T/N=3 titles                    // three rows to match 3-column ListWave
titles[0] = "\f01Bold Title"
titles[1] = "title with semicolon;"
titles[2] = "Marker in Gray: \K(40000,40000,40000)\k(40000,40000,40000)\W517"
NewPanel /W=(515,542,1011,794)
ListBox list0,pos={1,2},size={391,120},listWave=ListWave
ListBox list0,titleWave=titles
```

It is often useful to be able to display a contextual menu in response to a click in a listbox cell. Here is a simple example:

```
Function ListBoxContextProc(lba) : ListBoxControl
    STRUCT WMListboxAction &lba


    Variable row = lba.row
    Variable col = lba.col
    WAVE/Z selWave = lba.selWave


    switch( lba.eventCode )
                        // Always display the contextual menu on the mouse-down event
                        // On Windows, the mouse-down is delivered only after the mouse is released
            case 1: // mouse down
                        if (row >= 0 && row < DimSize(selWave, 0))// click must be in a cell
                                if (col == 0)                   // click must be in the left
column
                                        if (lba.eventMod & 16)  // it's a contextual click
                                                String menucontents =
"Nothing;Checkbox;Disclosure;Disabled;Enabled;"
```

```
                    PopupContextualMenu menucontents
                    if (V_flag > 0)// if nothing was selected,
V_flag = 0
                        StrSwitch(S_selection)//
S_selection is the text of the menu item
                        case "Nothing":
                        SelWave[row][1] = 0
                        break
                        case "Checkbox":
                        SelWave[row][1] = 0x20
                        break
                        case "Disclosure":
                        SelWave[row][1] = 0x40
                        break
                        case "Disabled":
                        SelWave[row][1] =
SelWave[row][1] | 0x80
                        break
                        case "Enabled":
                        SelWave[row][1] =
SelWave[row][1] & ~0x80
                        break
                        endswitch
                    endif
                endif
            endif
        endif
        break
    case 13:                                    // checkbox clicked
        String msg
        // test the value of SelWave for the clicked cell to see what it is, and its state
        Variable isOn = (SelWave[row][1] & 0x10) ? 1 : 0
        if (SelWave[row][1] & 0x20)
            msg = "Checbox was turned "
            if (isOn)
                msg += "on"
            else
                msg += "off"
            endif
        else
            msg = "Disclosure was "
            if (isOn)
                msg += "opened"
            else
                msg += "closed"
            endif
        endif
        DoAlert 0, msg
        break
endswitch
return 0
End

Function ListBoxContextMenu()
Make/N=(5,2)/T/O ListWaveContext
ListWaveContext[][0] = "Column 1 should be:"
ListWaveContext[][1] = "Row "+num2str(p)
Make/N=(5,2)/O SelWaveContext
SelWaveContext=0
NewPanel /W=(150,53,450,253)
ListBox list_w_context,pos={50,30},size={200,150},proc=ListBoxContextProc,widths={3,1}
ListBox list_w_context,listWave=root:ListWaveContext,selWave=root:SelWaveContext
EndMacro
```

Copy the code above to the Procedure window and compile. On the command line, invoke the function to build the panel:

```
ListBoxContextMenu()
```

In the listbox, right-clicking (Windows) or Ctrl-clicking (Macintosh) in the left column of the listbox will present a contextual menu with choices for what to do with the right column. If you choose Checkbox or Disclosure, and the cell is enabled, when you click on the cell in the right column, an alert is displayed telling you about what happened.

An example experiment that lets you easily experiment with ListBox settings is available in “Examples:Feature Demos 2:ListBox Demo.pxp”.

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

The **GetUserData** function for retrieving named user data.

The **ControlInfo** operation for information about the control.

Setting Bit Parameters on page IV-12 for further details about bit settings.

ListBoxControl

`ListBoxControl`

ListBoxControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined listbox control. See **Procedure Subtypes** on page IV-204 for details. See **ListBox** for details on creating a listbox control.

ListMatch

ListMatch(*listStr*, *matchStr* [, *listSepStr*])

The ListMatch function returns each list item in *listStr* that matches *matchStr*.

ListStr should contain items separated by *listSepStr* which typically is ";".

You may include asterisks in *matchStr* as a wildcard character. Note that matching is case-insensitive. See **StringMatch** for wildcard details.

ListSepStr is optional. If omitted it defaults to ";".

See Also

The **GrepList**, **StringMatch**, **StringFromList**, and **WhichListItem** functions.

ListToTextWave

`ListToTextWave(`*`listStr`*`, `*`separatorStr`*`)`

The ListToTextWave function returns a free text wave containing the individual list items in *listStr*.

See **Free Waves** on page IV-91 for details on free waves.

The ListToTextWave function was added in Igor Pro 7.00.

Parameters

listStr is a string that contains any number of substrings separated by a common string separator.

separatorStr is the separator string that separates one item in the list from the next. It is usually a single semicolon character but can be any string.

Details

The ListToTextWave function returns a free wave so it can't be used on the command line or in a macro. If you need to convert the free wave to a global wave use MoveWave.

For lists with a large number of items, using ListToTextWave and then retrieving the substrings sequentially from the returned text wave is much faster than retrieving the substrings using StringFromList.

The reverse operation, converting the contents of a text wave into a string list, can be accomplished using **wfprintf** like this:

```
WAVE/T tw
String list
wfprintf list, “%s\r”, tw        // Carriage-return separated list
```

Example

```
Function Test(num, separator)
    Variable num
    String separator                        // Usually ";"

    // Build a string list using separator
    String list = ""
    Variable i
    for(i=0; i<num; i+=1)
        list += "item_" + num2str(i) + separator
    endfor

    // Convert to a text wave and print its elements
    Wave/T w = ListToTextWave(list, separator)
    Print numpnts(w)
    for(i=0; i<num; i+=1)
        Print i, w[i]
    endfor
End
```

See Also

ListToWaveRefWave, **WaveRefWaveToList**, **wfprintf**

Using Strings as Lists on page IV-172, **StringFromList**, **MoveWave**, **Free Waves** on page IV-91

ListToWaveRefWave

`ListToWaveRefWave(`*`stringList`*` [, `*`options`*`])`

The ListToWaveRefWave function returns a free wave containing a wave reference for each entry in *stringList* that corresponds to an existing wave.

The ListToWaveRefWave function was added in Igor Pro 7.00.

Parameters

stringList is a semicolon-separated string list of waves specified using full paths or partial paths relative to the current data folder.

options is a bit field that is 0 by default. Set bit 0 to if you want the function to generate an error (see **GetRTError**) if any of the list elements does not specify an existing wave. Other bits are reserved for future use and must be cleared.

Example

```
Function Test()
    NewDataFolder/O/S root:TestDF     // Create empty data folder as current DF
    Make/O aaa                        // aaa will be the first wave in list
    Make/O bbb
    String strList = WaveList("*", ";", "")
    Wave/WAVE wr = ListToWaveRefWave(strList, 0)   // Returns a free wave
    Wave w = wr[0]
    Print GetWavesDataFolder(w, 2)
    Wave w = wr[1]
    Print GetWavesDataFolder(w, 2)
End
```

See Also

WaveRefWaveToList, **ListToTextWave**, **Wave References** on page IV-71

ln

ln(*num*)

The ln function returns the natural logarithm of *num*, -INF if *num* is 0, or NaN if *num* is less than 0. In complex expressions, *num* is complex, and ln(*num*) returns a complex value.

To compute a logarithm base n use the formula:

$$\log_n(x) = \frac{\log(x)}{\log(n)}.$$

See Also

The **log** function.

LoadData

LoadData [***flags***] ***fileOrFolderNameStr***

The LoadData operation loads data from the named file or folder. “Data” means Igor waves, numeric and string variables and data folders containing them. The specified file or folder must be an Igor packed experiment file or a folder containing Igor binary data, such as an Igor unpacked experiment folder or a folder in which you have stored Igor binary wave files.

In Igor Pro 9.00 or later, LoadData can load data from a standard packed experiment file (.pxp) or from an HDF5 packed experiment file (.h5xp). Previous versions support .pxp only. The term “packed experiment file” below refers to either .pxp or .h5xp.

LoadData loads data objects into memory and they become part of the current Igor experiment, disassociated from the file from which they were loaded.

If loading from a file-system folder, the data (waves, variables, strings) in the folder, including any subfolders if /R is specified, is loaded into the current Igor data folder.

If loading from a packed Igor experiment file, the data in the file, including any packed sub-data folders if /R is specified, is loaded into the current Igor data folder.

Use LoadData to load experiment data using Igor procedures. To load experiment data interactively, use the Data Browser (Data menu).

Parameters

If you use a full or partial path for *fileOrFolderNameStr*, see **Path Separators** on page III-451 for details on forming the path.

If *fileOrFolderNameStr* is omitted you get to locate the file (if /D is omitted) or the folder (if /D is present) via a dialog.

Flags

/D If present, loads from a file-system folder (a directory). If omitted, LoadData loads from an Igor packed experiment file.

/GREP={*regExpStr*, *grepMode*, *objectTypeMask*, *grepFlags*}

Applies a regular expression to determine what objects are loaded. /GREP was added in Igor Pro 8.00.

See **Using Regular Expressions With LoadData** on page V-503 for details.

/I Interactive. Forces LoadData to present a dialog.

/J=*objectNamesStr* Loads only the objects named in the semicolon-separated list of object names.

/L=*loadFlags* Controls what kind of data objects are loaded with a bit for each data type:

loadFlags	**Bit Number**	**Loads this Object Type**
1	0	Waves
2	1	Numeric Variables
4	2	String Variables

To load multiple data types, sum the values shown in the table. For example, /L=1 loads waves only, /L=2 loads numeric variables only, and /L=3 loads both waves and numeric variables. See **Setting Bit Parameters** on page IV-12 for further details about bit settings.

If no /L is specified, all object types are loaded. This is equivalent to /L=7. All other bits are reserved and should be set to zero.

/O[=*overwriteMode*] If /O alone is used, overwrites existing data objects in case of a name conflict.

overwriteMode is defined as follows:

0: No overwrite, as if there were no /O.

1: Normal overwrite. In the event of a name conflict, objects in the incoming file replace the conflicting objects in memory. Incoming data folders completely replace any conflicting data folders in memory.

2: Mix-in overwrite. In the event of a name conflict, objects in the incoming file replace the conflicting objects in memory but nonconflicting objects in memory are left untouched.

See **Details** for more about overwriting.

/P=*pathName* Specifies folder to look in for the specified file or folder. *pathName* is the name of an existing Igor symbolic path. See **Symbolic Paths** on page II-22 for details.

/Q Suppresses the normal messages in the history area.

/R Recursively loads sub-data folders.

/S=*subDataFolderStr* Specifies a sub-data folder within a packed experiment file to be loaded. See **Loading a Specific Data Folder from a Packed Experiment** on page V-502 for details.

/T[=*topLevelName*] If /T=*topLevelName* is specified, it creates a new data folder in the current data folder with the specified name and places the loaded data in the new data folder.

If just /T is specified, it creates a new data folder in the current data folder with a name derived from the name of the unpacked experiment folder, packed experiment file or packed sub-data folder being loaded.

Details

If /T is present, LoadData loads the top level data folder and its contents. If /T is omitted, it loads just the contents of the top level data folder and not the data folder itself. This distinction has an analogy in the desktop. You can drag the contents of disk folder A into folder B or you can drag folder A itself into folder B.

Because LoadData can load from a complex packed Igor experiment file or from a complex hierarchy of file-system folders, it does not set the variables normally set by a file loader: S_path, S_fileName, and S_waveNames. The variable V_flag is set to the total number of objects loaded, or to -1 if the user cancelled the open file dialog. To find what objects were created by LoadData, you can use the **CountObjects** and **GetIndexedObjName** functions.

Overwriting Existing Data

If you do a load of a data folder, overwriting an existing data folder of the same name, the behavior of LoadData depends on whether you use /J. If you do not use /J, the entire data folder and all of its contents are replaced. If you do use /J, just the specified objects in the data folder are replaced, leaving any other preexisting objects in the data folder unchanged.

If you do not use the /O (overwrite) flag or if you use /O=0 and there is a conflict between objects or data folders in the current data folder and objects or data folders in the file or folder being loaded, LoadData presents a dialog to ask you if you want to replace the existing data. However, LoadData can not replace an object with an object of a different type and will refuse to do so.

You can overwrite an object that is in use, such as a wave that is displayed in a graph or table. You can also overwrite a data folder that contains objects that are in use. This is a powerful feature. Imagine that you define a data structure consisting of waves, variables and possibly sub-data folders. You can display the data in graphs and tables and you can display these in a layout. You can then overwrite the data with an analogous data structure from a packed experiment file and Igor will automatically update any graphs, tables or layouts that need to be updated.

Controlling Which Objects Are Loaded

LoadData provides several ways to control whether a given object is loaded or is skipped. These method, which are described in the following sections, include:

- Using /L to control whether waves, numeric variables, and string variables are loaded
- Specifying a sub-data folder using /S
- Specifying object names using /J
- Applying a regular expression using /GREP. /GREP requires Igor Pro 8.00 or later.

You can use more than one of these flags. For example, if you use /S and /J, then LoadData loads only objects in the data folder specified by /S with names listed using /J. In most cases, if you use /GREP, you will not need to combine it with /S or /J.

Loading a Specific Data Folder from a Packed Experiment

If present, /S=*subDataFolderStr* specifies the sub-data folder within the packed experiment file from which the load is to start. For example:

```
LoadData/P=Path1/S="root:Folder A:Folder B" "aPackedExpFile.pxp"
```

This starts loading from data folder "root:Folder A:Folder B" in the packed experiment file. *subDataFolderStr* is the full data folder path to the desired data folder. A trailing semicolon is allowed but not required. Since this parameter is specified as a string, you must not use single quotes to quote liberal names.

Prior to Igor Pro 7.00, the "root:" part of the data folder path was implicit and you had to omit it from *subDataFolderStr*. Now you can provide the full path including "root:" and this is recommended for clarity. For backward compatibility, if *subDataFolderStr* does not start with "root:", LoadData automatically prepends it.

/S has no effect if you are loading from an unpacked file system folder rather than from a packed experiment file.

Specifying /S="" acts like no /S at all.

Loading Specific Objects Based on Object Name

If /J=*objectNamesStr* is used, then only the objects named in objectNamesStr are loaded into the current experiment. For example, `/J="wave0;wave1;"` will load only the two named waves, ignoring any other data in the file or folder being loaded.

Assume that you have an experiment that contains 5 runs where each run, stored in a separate data folder in a packed experiment file, consists of data acquired from four channels from an ADC card. Using the /J flag, you can load just one specific channel from each run. You can use this to compare that channel's data from all runs without loading the other channels.

The list of object names used with /J must be semicolon-separated. A semicolon after the last object name in the list recommended but optional. Since objectNamesStr is specified as a string, you must not use single quotes to quote liberal names.

objectNamesStr is limited to 2000 bytes.

Specifying /J="" acts like no /J at all.

If you load a hierarchy of data folders, using the /R flag, with /J in effect, LoadData will create each data folder in the hierarchy even if it contains none of the named objects. This behavior is necessary to avoid loading a sub-data folder without loading its parent, as well as other such complications.

Using Regular Expressions With LoadData

If you include the /GREP={*regExpStr*,*grepMode*,*objectTypeMask*,*grepFlags*} flag, LoadData applies a regular expression (see **Regular Expressions** on page IV-176) to determine if a given object is to be loaded.

Before we get into the details, here are some simple examples. These examples assume that we are loading a packed experiment file named "Packed.pxp" and have an Igor symbolic path named Data.

```
// Load waves named wave1 from all data folders
LoadData/P=Data/Q/R/L=1/GREP={"(?i)^wave1$",1,1,0} "Packed.pxp"
```

```
// Load root:DataFolder0B and all objects in it
LoadData/P=Data/Q/R/L=7/GREP={"(?i)^root:DataFolder0B:$",3,8,0} "Packed.pxp"
```

```
// Load all data folders whose paths are of the form root:DataFolder<suffix>
LoadData/P=Data/Q/R/L=7/GREP={"(?i)^root:DataFolder.+:.*$",3,8,0} "Packed.pxp"
```

See the **Examples** on page V-504 for more examples.

regExpStr is the regular expression.

GREP is by default case-sensitive but Igor names are case-insensitive. For this reason, we recommend that you start *regExpStr* with "(?i)" which turns GREP's case-sensitivity off.

When looking for the appropriate value for *regExpStr*, you may find it helpful to test if a given name or path is matched by a given regular expression using **GrepString**. For example:

```
Print GrepString("root:DataFolder0A:str0", "(?i).*0$")   // Prints 1 if path is matched
```

The *grepMode* parameter selects one of three modes of using the regular expression.

If *grepMode* is 1, LoadData matches the name of each object of types specified by *objectTypeMask* against the specified regular expression. If there is a match, the object passes this test.

If *grepMode* is 2, LoadData matches the full data folder path to the object's parent data folder for objects of types specified by *objectTypeMask* against the regular expression. The full data folder path starts from "root:" and includes a trailing colon. If there is a match, the object passes this test. By definition, the parent of the root: data folder is "" (empty string) which can be expressed using the regular expression "^$".

If *grepMode* is 3, LoadData matches the full data folder path to the object against the regular expression. The full data folder path starts from "root:" and includes a trailing colon for data folder objects only. If there is a match, the object passes this test.

When loading unpacked data, modes 2 and 3 are not supported and are ignored. They apply when loading data from a packed experiment file only.

objectTypeMask is a bitwise parameter with each bit controlling whether the regular expression is applied to a given object type. Bits 0, 1, 2, and 3 control whether the regular expression is applied against waves, numeric variables, string variables and data folders respectively. You can apply a given regular expression to more than one type of object by setting more than one bit in *objectTypeMask*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

grepFlags is a bitwise parameter. Currently only bit 0 is used so its value must be 0 or 1. Normally, when *grepFlags*=0, objects that do not match the regular expression are skipped by LoadData. If *grepFlags*=1, the meaning of a match is reversed and objects that do match the regular expression are skipped by LoadData. For example:

```
// Load all objects whose name ends with "2"
LoadData/P=Data/Q/R/L=7/GREP={"(?i)[2]$",1,7,0} "Packed.pxp"    // Normal mode
```

```
// Load all objects except those whose name ends with "2"
LoadData/P=Data/Q/R/L=7/GREP={"(?i)[2]$",1,7,1} "Packed.pxp"    // Reverse mode
```

It is often not necessary, but you can use /GREP more than once in a given LoadData command, each time with a different combination of *grepMode*, object type, and reverse mode. A given object is loaded only if it passes all tests as well as any tests specified by /L, /J and /S. For example:

```
// Load root:DataFolder0B and all objects in it, except objects whose name ends with "2"
LoadData/P=Data/Q/R/L=7/GREP={"(?i)^root:DataFolder0B:$",3,8,0}/GREP={"(?i)[2]$",1,7,1} "Packed.pxp"
```

In this example, the first /GREP flag restricts which data folders are loaded and the second /GREP flag restricts which waves and variables are loaded.

If no test is applied to a particular kind of object then all objects of that type are loaded. For example, this command

```
LoadData/P=<pathName>/GREP={"(?i)^wave1$",1,1,0} <file name>
```

specifies that only waves named "wave1" should be loaded but it says nothing about numeric variables, string variables, or data folders so they are all loaded regardless of name. If you want to exclude all numeric and string variables, add the /L flag:

```
LoadData/P=<pathName>/L=1/GREP={"(?i)^wave1$",1,1,0} <file name>
```

As LoadData progresses through the data folders of the source, it has to create each data folder before it applies regular expression tests to objects in that data folder and its descendants. Consequently, it would be possible to end up with empty data folders because no data objects were loaded into them or their descendants. To eliminate the clutter that this would create, if you use /GREP, after loading all of the data, LoadData deletes any empty data folders that it created along the way. Data folders that existed before LoadData ran are not deleted even if they are empty. For backward compatibility, this pruning of empty data folders is done only if you use /GREP.

For each combination of *grepMode*, object type, and reverse mode, only one regular expression is applied. If you specify, for example, /GREP={"(?i)first",1,1,0} /GREP={"(?i)second",1,1,0}, the second /GREP flag overrides the first because they both apply a name test (*grepMode*=1) to wave objects (*objectTypeMask*=1) using normal mode (*grepFlags*=0). By contrast, /GREP={"(?i)first",1,1,0} /GREP={"(?i)second",1,2,0} results in no overriding because the first /GREP flag applies to waves (*objectTypeMask*=1) while the second applies to numeric variables (objectTypeMask=2). Similarly, /GREP={"(?i)first",1,1,0} /GREP={"(?i)second",1,1,1} results in no overriding because the first /GREP flag uses normal mode (*grepFlags*=0) while the second uses reverse mode (*grepFlags*=1).

Examples

These examples assume that we are loading a packed experiment file named "Packed.pxp" and have an Igor symbolic path named Data. The file contains this data hierarchy:

```
root:
   wave0,wave1,wave2,var0,var1,var2,str0,str1,str2
   DataFolder0A
      wave0,wave1,wave2,var0,var1,var2,str0,str1,str2
   DataFolder0B
      wave0,wave1,wave2,var0,var1,var2,str0,str1,str2
```

```
// Load everything
LoadData/P=Data/Q/R "Packed.pxp"

// Load all objects named wave1
LoadData/P=Data/Q/R/J="wave1;" "Packed.pxp"

// Load all objects whose name ends with "2"
LoadData/P=Data/Q/R/L=7/GREP={"(?i)[2]$",1,7,0} "Packed.pxp"

// Load all numeric variables whose name starts with "var"
LoadData/P=Data/Q/R/L=2/GREP={"(?i)^var",1,2,0} "Packed.pxp"

// Load all string variables whose name starts with "str" and ends with "2"
LoadData/P=Data/Q/R/L=4/GREP={"(?i)^str.*2$",1,4,0} "Packed.pxp"

// Load all objects whose path contains "DataFolder", case-insensitive
LoadData/P=Data/Q/R/L=7/GREP={"(?i)DataFolder",2,7,0} "Packed.pxp"

// Load root:DataFolder0B and all objects in it using /S
LoadData/P=Data/Q/R/L=7/S="root:DataFolder0B"/T "Packed.pxp"
```

```
// Load root:DataFolder0B and all objects in it using /GREP
LoadData/P=Data/Q/R/L=7/GREP={"(?i)^root:DataFolder0B:$",3,8,0} "Packed.pxp"

// Load all objects whose path contains "DataFolder" and whose name ends with "1"
LoadData/P=Data/Q/R/L=7/GREP={"(?i)DataFolder",2,7,0}/GREP={"(?i).*1$",1,7,0} "Packed.pxp"

// Load all objects in root:DataFolder0B and whose name ends with "0"
LoadData/P=Data/Q/R/L=7/GREP={"(?i)root:DataFolder0B:.*0$",3,7,0} "Packed.pxp"
```

See Also

SaveData, **Importing Data** on page II-126, **The Data Browser** on page II-114, **Regular Expressions** on page IV-176.

LoadPackagePreferences

LoadPackagePreferences [**/MIS=*mismatch* /P=*pathName***] ***packageName*, *prefsFileName*, *recordID*, *prefsStruct***

The LoadPackagePreferences operation loads preference data previously stored on disk by the **SavePackagePreferences** operation. The data is loaded into the specified structure.

Note: The package preferences structure must not use fields of type Variable, String, WAVE, NVAR, SVAR or FUNCREF because these fields refer to data that may not exist when LoadPackagePreferences is called.

The structure can use fields of type char, uchar, int16, uint16, int32, uint32, int64, uint64, float and double as well as fixed-size arrays of these types and substructures with fields of these types.

If the /P flag is present then the location on disk of the preference file is determined by *pathName* and *prefsFileName*. However in the usual case the /P flag will be omitted and the preference file is located in a file named *prefsFileName* in a directory named *packageName* in the Packages directory in Igor's preferences directory.

Note: You must choose a very distinctive name for *packageName* as this is the only thing preventing collisions between your package and someone else's package. If you use a name longer than 31 bytes, your package will require Igor Pro 8.00 or later.

See **Saving Package Preferences** on page IV-251 for background information and examples.

Parameters

packageName is the name of your package of Igor procedures. It is limited to 255 bytes and must be a legal name for a directory on disk. This name must be very distinctive as this is the only thing preventing collisions between your package and someone else's package. If you use a name longer than 31 bytes, your package will require Igor Pro 8.00 or later.

prefsFileName is the name of a preference file to be loaded by LoadPackagePreferences. It should include an extension, typically ".bin".

prefsStruct is the structure into which data from disk, if it exists, will be loaded.

recordID is a unique positive integer that you assign to each record that you store in the preferences file. If you store more than one structure in the file, you would use distinct *recordID*s to identify which structure you want to load. In the simple case you will store just one structure in the preference file and you can use 0 (or any positive integer of your choice) as the *recordID*.

Flags

/MIS=*mismatch* Controls what happens if the number of bytes in the file does not match the size of the structure:

0: Returns an error. Default behavior if /MIS is omitted.

1: Returns the smaller of the size of the structure and the number of bytes in the file. Does not return an error. Use this if you want to read and update old versions of a preferences structure.

/P=*pathName*	Specifies the directory to look in for the file specified by *prefsFileName*. *pathName* is the name of an existing symbolic path. See **Symbolic Paths** on page II-22 for details. `/P=$<empty string variable>` acts as if the /P flag were omitted.

Details

LoadPackagePreferences sets the following output variables:

`V_flag`	Set to 0 if no error occurred or to a nonzero error code. If the preference file does not exist, V_flag is set to zero so you must use V_bytesRead to detect that case.
`V_bytesRead`	Set to the number of bytes read from the file. This will be zero if the preference file does not exist.
`V_structSize`	Set to the size in bytes of *prefsStruct*. This may be useful in handling structure version changes.

After calling LoadPackagePreferences if V_flag is nonzero or V_bytesRead is zero then you need to create default preferences as illustrated by the example referenced below.

V_bytesRead, in conjunction with the /MIS flag, makes it possible to check for and deal with old versions of a preferences structure as it loads the version field (typically the first field) of an older or newer version structure. However in most cases it is sufficient to omit the /MIS flag and treat incompatible preference data the same as missing preference data.

Example

See the example under **Saving Package Preferences in a Special-Format Binary File** on page IV-252.

See Also

SavePackagePreferences.

LoadPICT

`LoadPICT` `[`***`flags`***`] [`***`fileNameStr`***`][,`** ***`pictName`***`]`

The LoadPICT operation loads a picture from a file or from the Clipboard into Igor. Once you have loaded a picture, you can append it to graphs and page layouts.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

If you want to force a dialog to select the file, omit the *fileNameStr* parameter.

If *fileNameStr* is “Clipboard” and /P=*pathName* is omitted, LoadPICT loads its data from the Clipboard rather than from a file.

pictName is the name that you want to give to the newly loaded picture. You can refer to the picture by its name to append it to graphs and page layouts. LoadPICT generates an error if the name conflicts with some other type of object (e.g., wave or variable) or if the name conflicts with a built-in name (e.g., the name of an operation or function).

If you omit *pictName*, LoadPICT automatically names the picture as explained in **Details**.

Flags

/M=*promptStr*	Specifies a prompt to use if LoadPICT needs to put up a dialog to find the file.

/O	Overwrites an existing picture with the same name. If /O is omitted and there is an existing picture with the same name, LoadPICT displays a dialog in which you can resolve the name conflict.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/Q	Quiet: suppresses the insertion of picture info into the history area.
/Z	Doesn't load the picture, just checks for its existence.

Details

If the picture file is not fully specified then LoadPICT presents a dialog from which you can select the file. "Fully specified" means that LoadPICT can determine the name of the file (from the *fileNameStr* parameter) and the folder containing the file (from the flag /P=*pathName* flag or from the *fileNameStr* parameter). If you want to force a dialog, omit the *fileNameStr* parameter.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-22 for details.

If you omit *pictName*, LoadPICT automatically names the picture as follows:

If the picture was loaded from a file, LoadPICT uses the file name. If necessary, it makes it into a legal name by replacing illegal characters or shortening it.

Otherwise, LoadPICT uses a name of the form "PICT_*n*".

If the resulting name is in conflict with an existing picture name, Igor puts up a name conflict resolution dialog.

LoadPICT sets the variable V_flag to 1 if the picture exists and fits in available memory or to 0 otherwise.

It also sets the string variable S_info to a semicolon-separated list of values:

Keyword	Information Following Keyword
NAME	Name of the loaded PICT, often "PICT_0", etc.
SOURCE	"Data fork" or "Clipboard".
RESOURCENAME	Obsolete - always "".
RESOURCEID	Obsolete - always 0.
TYPE	One of the following types: DIB Encapsulated PostScript Enhanced metafile JPEG PDF PNG SVG TIFF Windows bitmap Windows metafile Unknown type
BYTES	Amount of memory used by the picture.
WIDTH	Width of the picture in pixels.

Keyword	Information Following Keyword
HEIGHT	Height of the picture in pixels.
PHYSWIDTH	Physical width of the picture in points.
PHYSHEIGHT	Physical height of the picture in points.

See Also

See **Pictures** on page III-509 for general information on how Igor handles pictures.

The **ImageLoad** operation for loading PICT and other image file types into waves, and the **PICTInfo** function.

LoadWave

`LoadWave` `[`***`flags`***`] [`***`fileNameStr`***`]`

The LoadWave operation loads data from the named Igor binary, Igor text, delimited text, fixed field text, or general text file into waves. LoadWave can load 1D and 2D data from delimited text, fixed field text and general text files, or data of any dimensionality from Igor binary and Igor text files.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If LoadWave can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

If *fileNameStr* is “Clipboard” and /P=*pathName* is omitted, LoadWave takes its data from the Clipboard rather than from a file. This is not implemented for binary loads.

If *fileNameStr* is omitted or is “”, or if the /I flag is used, LoadWave presents an Open File dialog from which you can choose the file to load.

Flags

/A

“Auto-name and go” option (used with /G, /F or /J).

This skips the dialog in which you normally enter wave names. Instead it automatically assigns names of the form *wave*0, *wave*1, choosing names that are not already in use. When used with /W, it reads wave names from the file instead of automatically assigning names and /A just skips the wave name dialog. The /B flag can also override names specified by /A.

See **LoadWave Generation of Wave Names** on page II-142 for further discussion.

/A=*baseName*

Same as /A but it automatically assigns wave names of the form *baseName*0, *baseName*1.

/B=*columnInfoStr*

Specifies the name, format, numeric type, and field width for columns in the file. See **Specifying Characteristics of Individual Columns** on page II-145 for details.

/C

This is used in experiment recreation commands generated by Igor to force experiment recreation to continue if an error occurs in loading a wave.

/D

Creates double precision waves. (Used with /G, /F or /J.) The /B flag can override the numeric precision specified by the /D flag.

/E=*editCmd*

Controls table creation:

editCmd=1:	Makes a new table containing the loaded waves.
editCmd=2:	Appends the loaded waves to the top table. If no table exists, a new table is created.
editCmd=0:	Same as if /E had not been specified (loaded waves are not put in any table).

/ENCG=*textEncoding*

/ENCG={*textEncoding*, *tecOptions*}

Specifies the text encoding of the plain text file being loaded.

This flag was added in Igor Pro 7.00.

This flag is ignored when loading an Igor binary wave file.

See **Text Encoding Names and Codes** on page III-490 for a list of accepted values for *textEncoding*.

For most purposes the default value for *tecOptions*, 3, is fine and you can use /ENCG=textEncoding instead of /ENCG={*textEncoding*, *tecOptions*}.

tecOptions is an optional bitwise parameter that controls text encoding conversion. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

tecOptions is defined as follows:

Bit 0: If cleared, the presence of null bytes causes LoadWave to consider the text invalid in all byte-oriented text encodings. This makes it easier for LoadWave to identify UTF-16 and UTF-32 text which usually contains null bytes.

If set (default), null bytes are allowed in byte-oriented text encodings.

Bit 1: If cleared LoadWave does not validate text if the specified text encoding is UTF-8.

If set (default), LoadWave validates text even if the text encoding is UTF-8.

Bit 2: If cleared (default) LoadWave validates the text in the specified text encoding except that validation for UTF-8 is skipped if bit 1 is cleared.

If set, LoadWave assumes that the text is valid in the specified text encoding and does not validate it. Setting this bit makes LoadWave slightly faster but it is usually inconsequential.

Bit 3: If cleared (default), LoadWave presents the Choose Text Encoding dialog if the specified text encoding is not valid for the text in the file.

If set, LoadWave does not present the Choose Text Encoding dialog if the text is not valid in the specified text encoding and instead returns an error. Set this bit if you are running an automated procedure and do not want it interrupted by a dialog.

See **LoadWave Text Encoding Issues** on page II-149 for further discussion.

/F={*numColumns*, *defaultFieldWidth*, *flags*}

Indicates that the file uses the fixed field file format. Most FORTRAN programs generate files in this format.

numColumns is the number of columns of data in the file.

defaultFieldWidth is the default number of bytes in each column. If the columns do not all have the same number of bytes, you need to use the /B flag to provide more information to LoadWave.

flags is a bitwise parameter that controls the conversion of text to values. The bits are defined as follows:

Bit 0 : If set, any field that consists entirely of the digit “9” is taken to be blank. A field that consists entirely of the digit “9” except for a leading “+” or “-” is also taken to be blank.

All other bits are reserved and must be cleared.

/G Indicates that the file uses the general text format.

/H Loads the wave into the current experiment and severs the connection between the wave and the file. When the experiment is saved, the wave copy will be saved as part of the experiment. For a packed experiment this means it is saved in the packed experiment file. For an unpacked experiment this means it is saved in the experiment's home folder.

See **Sharing Versus Copying Igor Binary Wave Files** on page II-156.

/I Forces LoadWave to display an Open File dialog even if the file is fully specified via /P and *fileNameStr*.

/J Indicates that the file uses the delimited text format.

/K=*k* Controls how to determine whether a column in the file is numeric or text (only for delimited text and fixed field text files).

k=0:	Deduces the nature of the column automatically.
k=1:	Treats all columns as numeric.
k=2:	Treats all columns as text.

This flag as well as the ability to load text data into text waves were added in Igor Pro 3.0. The default for the LoadWave operation is /K=1, meaning that it will treat all columns as numeric. We did this so that existing procedures would behave the same in Igor Pro 3.0 as before. Use /K=0 when you want to load text columns into text waves. /K=2 may have use in a text-processing application.

For finer control, the /B flag specifies the format of each column in the file individually.

/L={*nameLine*, *firstLine*, *numLines*, *firstColumn*, *numColumns*}

Affects loading delimited text, fixed field text, and general text files only (/J, /F or /G). /L is accepted no matter what the load type but is ignored for Igor binary and Igor text loads. Line and column numbers start from 0.

nameLine is the number of the line containing column names. For general text loads, 0 means auto. See **Loading General Text Files** on page II-138 for details.

firstLine is the number of the first line to load into a wave. For general text loads, 0 means auto. See **Loading General Text Files** on page II-138 for details.

numLines is the number of lines that should be treated as data. 0 means auto which loads until the end of the file or until the end of the block of data in general text files. The *numLines* parameter can also be used to make loading very large files more efficient. See **Loading Very Large Files** on page II-150 for details.

firstColumn is the number of the first column to load into a wave. This is useful for skipping columns.

numColumns is the number of columns to load into a wave. 0 means auto, which loads all columns.

/M Loads data as matrix wave. If /M is used then it ignores the /W flag (read wave names) and follows the /U flags instead.

The wave is autonamed unless you provide a specific wave name using the /B flag.

The type of the wave (numeric or text) is determined by an assessment of the type of the first loaded column unless you override this using the /K flag or the /B flag.

See **The Load Waves Dialog for Delimited Text — 2D** on page II-133 for further information.

/N Same as /A except that, instead of choosing names that are not in use, it overwrites existing waves.

See **LoadWave Generation of Wave Names** on page II-142 for further discussion.

/N=*baseName* Same as /N except that it automatically assigns wave names of the form *baseName0*, *baseName1*.

/NAME={*namePrefix*, *nameSuffix*, *nameOptions*}

The /NAME flag overrides or combines with the other flags (/A, /N, /W, and /B) that affect the names of waves loaded by LoadWave.

/NAME was added in Igor Pro 9.00. It applies to Load General Text (/G), Load Delimited Text (/J), and Load Fixed Field Text (/F), not when loading Igor binary files or Igor Text files.

The /NAME flag is primarily intended to provide an easy way to incorporate the file name in the wave names. For example, if you are loading a single wave from a file named File.txt, you can create a single wave named File and, if you are loading three waves from a file named File.txt, you can create three waves named File0, File1, and File2.

namePrefix is "" or text to be prepended to the final wave names.

nameSuffix is "" or text to be appended to the final wave names.

If both *namePrefix* and *nameSuffix* are "", it is as if the /NAME flag was omitted altogether.

In the construction of the final wave names, LoadWave replaces ":filename:" in *namePrefix* or *nameSuffix* with the name of the file being loaded without the file name extension.

nameOptions is a bitwise parameter. The bits are defined as follows:

Bit 0: Include normal name. If set, the "normal" wave name (the name that would be used if /NAME were omitted) is included in the final name between *namePrefix* and *nameSuffix*.

Bit 1: Include normal name. If set, the "normal" wave name (the name that would be used if /NAME were omitted) is included in the final name between *namePrefix* and *nameSuffix*.

Bit 2: Include suffix number when loading multiple waves. If multiple waves, a suffix number is appended unless suppressed by bit 3.

Bit 3: Include suffix number only if there is a name conflict. If set and there are no name conflicts, this overrides bits 1 and 2 and prevents appending suffix numbers.

Bit 4: Choose suffix number to create unique names. If set, LoadWave chooses suffix numbers to avoid conflicts with existing waves. If cleared suffix numbers start from 0 and increment by one for each loaded wave.

Bit 5: Allow liberal names (not recommended).

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

See **LoadWave Generation of Wave Names** on page II-142 for further discussion.

/O Overwrite existing waves in case of a name conflict.

/P=*pathName* Specifies the folder to look in for *fileNameStr*. *pathName* is the name of an existing symbolic path.

/Q Suppresses the normal messages in the history area.

/R={*languageName*, *yearFormat*, *monthFormat*, *dayOfMonthFormat*, *dayOfWeekFormat*, *layoutStr*, *pivotYear*}

Specifies a custom date format for dates in the file. If the /R flag is used, it overrides the date setting that is part of the /V flag.

languageName controls the language used for the alphabetic date components, namely the month and the day-of-week. *languageName* is one of the following:

Default

Chinese	ChineseSimplified	Danish	Dutch
English	Finnish	French	German
Italian	Japanese	Korean	Norwegian
Portuguese	Russian	Spanish	Swedish

Default means the system language on Macintosh or the user default language on Windows.

yearFormat is one of the following codes:

1:	Two digit year.
2:	Four digit year.

monthFormat is one of the following codes:

1:	Numeric, no leading zero.
2:	Numeric with leading zero.
3:	Abbreviated alphabetic (e.g., Jan).
4:	Full alphabetic (e.g., January).

dayOfMonthFormat is one of the following codes:

1:	Numeric, no leading zero.
2:	Numeric with leading zero.

dayOfWeekFormat is one of the following codes:

1:	Abbreviated alphabetic (e.g., Mon).
2:	Full alphabetic (e.g., Monday).

layoutStr describes which components appear in the date in what order and what separators are used. *layoutStr* is constructed as follows (but with no line break):

```
"<component keyword><separator>
<component keyword><separator>
<component keyword><separator>
<component keyword>"
```

where <component keyword> is one of the following:

```
Year
Month
DayOfMonth
DayOfWeek
```

and <separator> is a string of zero to 15 bytes.

Starting from the end, parts of the layout string must be omitted if they are not used.

Extraneous spaces are not allowed in *layoutStr*. Each separator must be no longer than 15 bytes. No component can be used more than once. Some components may be used zero times.

pivotYear determines how LoadWave interprets two-digit years. If the year is specified using two digits, yy, and is less than *pivotYear* then the date is assumed to be 20yy. If the two digit year is greater than or equal to *pivotYear* then the year is assumed to be 19yy. *pivotYear* must be between 4 and 40.

See **Loading Custom Date Formats** on page II-144 for further discussion of date formats.

/T Indicates that the file uses the Igor text format.

Although LoadWave is generally thread-safe, it is not thread-safe to load an Igor text file containing an Igor command (e.g., "X <command>").

/U={*readRowLabels*, *rowPositionAction*, *readColLabels*, *colPositionAction*}

These parameters affect loading a matrix (/M) from a delimited text (/J) or a fixed field text (/F) file. They are accepted no matter what the load type is but are ignored when they don’t apply.

If *readRowLabels* is nonzero, it reads the first column of data in the file as the row labels for the matrix wave.

rowPositionAction has one of the following values:

0: The file has no row position column.

1: Uses the row position column to set the row scaling of the matrix wave.

2: Creates a 1D wave containing the values in the row position column. The name of the 1D wave will be the same as the matrix wave but with the prefix “RP_”.

The *readColLabels* and *colPositionAction* parameters have analogous meanings. The prefix used for the column position wave is “CP_”.

See Chapter II-9, **Importing and Exporting Data** for further details.

/V={*delimsStr*, *skipCharsStr*, *numConversionFlags*, *loadFlags*}

These parameters affect loading delimited text (/J) and fixed field text (/F) data and column names. They do not affect loading general text (/G). They are accepted no matter what the load type is but are ignored when they don’t apply. These parameters should rarely be needed.

delimsStr is a string expression containing the characters that should act as delimiters for delimited file loads. (When loading data as general text, the delimiters are always tab, comma and space.) The default is “\t,” for tab and comma. You can specify the space character as a delimiter, but it is always given the lowest priority behind any other delimiters contained in *delimsStr*. The low priority means that if a line of text contains any other delimiter besides the space character, then that delimiter is used rather than the space character.

skipCharsStr is a string expression containing characters that should always be treated as garbage and skipped when they appear before a number. The default is “$” for space and dollar sign. This parameter should rarely be needed.

numConversionFlags is a bitwise parameter that controls the conversion of text to numbers. The bits are defined as follows:

Bit 0: If set: dates are dd/mm/yy.
If cleared: dates are mm/dd/yy.

Bit 1: If set: decimal character is comma.
If cleared: it is period.

Bit 2: If set: thousands separators in numbers are ignored when loading delimited text only (LoadWave/J).

The thousands separator is the comma in 1,234 or, if comma is the decimal character, the dot in 1.234.

Most numeric data files do not use thousands separators and searching for them slows loading down so this bit should usually be 0.

This bit has no effect if the thousands separator (e.g., comma) is also a delimiter character as specified by delimsStr.

All other bits are reserved and must be cleared.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

If the /R flag is used to specify the date format, this overrides the setting of bit 0.

loadFlags is a bitwise parameter that controls the overall load. The bits are defined as follows:

Bit 0: If set, ignores blanks at the end of a column. This is appropriate if the file contains columns of unequal length. Set *loadFlags* = 1 to set bit 0.

Bit 1: If set, when the /W flag is specified and the line containing column labels starts with one or more space characters, the spaces are taken to be a blank column name. The resulting column will be named Blank. Use this if both the line containing column labels and the lines containing data start with leading spaces in a space-delimited file. Set loadFlags = 2 to set bit 1.

Bit 2: If set: Disables pre-counting of lines of data. See **Loading Very Large Files** on page II-150.

Bit 3: If set: Disables unescaping of backslash characters in text columns. See **Escape Sequences** on page II-150 below.

Bit 4: If set: Disables support for quoted strings.

Setting this bit improves compatibility with Igor7. It applies to delimited text only and should rarely be needed. See **Quoted Strings in Delimited Text Files** on page II-135 and **Delimited Text Compatibility With Igor7** on page II-136 for background information.

/W Looks for wave names in a file. (With /G, /F, and /J.)

LoadWave cleans up column names to create standard, not liberal, wave names. See **Object Names** on page III-501 and **CleanupName** for details.

Use /W/A to read wave names from the file and then continue the load without displaying the normal wave name dialog.

See **LoadWave Generation of Wave Names** on page II-142 for further discussion.

Details

Without the /G, /F, /J, or /T flags, LoadWave loads Igor binary wave files.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like “`hd:Folder1:`” or “`C:\\Folder1\\`”. See **Symbolic Paths** on page II-22 for details.

When loading a general text file, the delimiters are always tab, comma and space.

Output Variables

LoadWave sets the following variables:

`V_flag`	Number of waves loaded.
`S_fileName`	Name of the file being loaded.
`S_path`	File system path to the folder containing the file.
`S_waveNames`	Semicolon-separated list of the names of loaded waves.

S_path uses Macintosh path syntax (e.g., "`hd:FolderA:FolderB:`"), even on Windows. It includes a trailing colon. If LoadWave is loading from the Clipboard, S_path is set to `""`.

When LoadWave presents an Open File dialog and the user cancels, V_flag is set to 0 and S_fileName is set to `""`.

See Also

The **ImageLoad** operation.

Loading Igor Binary Data on page II-154, **Loading Delimited Text Files** on page II-129, **Loading Fixed Field Text Files** on page II-137, **Loading General Text Files** on page II-138

LoadWave Generation of Wave Names on page II-142, **Specifying Characteristics of Individual Columns** on page II-145, **Loading Custom Date Formats** on page II-144, **LoadWave Text Encoding Issues** on page II-149

See **Importing Data** on page II-126 for further information on loading waves, including loading multidimensional data from HDF, PICT, TIFF and other graphics files. Check the "More Extensions:File Loaders" folder other file-loader extensions.

Setting Bit Parameters on page IV-12 for further details about bit settings.

Loess

`Loess` `[`***`flags`***`]` **`srcWave =`** ***`srcWaveName`*** `[,` **`factors =`** ***`factorWaveName1`***
`[,` ***`factorWaveName2`*** `…]]`

The Loess operation smooths *srcWaveName* using locally-weighted regression smoothing. This algorithm is sometimes classified as a "nonparametric regression" procedure. The regression can be constant, linear, or quadratic. A robust option that ignores outliers is available. See **Basic Algorithm**, **Robust Algorithm**, and **References** for additional details and terminology.

This implementation works with waveforms, XY pairs of waves, false-color images, matrix surfaces, and multivariate data (one dependent data wave with multiple independent variable data waves).

Unlike the **FilterFIR** operation, Loess discards any NaN input values and will not generate a result that is wholly NaN.

Parameters

srcWaveName is the input data to be smoothed. It may be a one-dimensional or a two-dimensional wave, and it may contain NaNs.

When no /DEST flag is specified, Loess will overwrite *srcWaveName* with the smoothed result.

If *srcWaveName* is one-dimensional and no factors are provided, X values are derived from the X scaling of *srcWaveName*.

If *srcWaveName* is two-dimensional, the factors keyword is not permitted and the X and Y values are derived from the X and Y scaling of *srcWaveName*.

Higher dimensions of *srcWaveName* are not supported.

The optional factors parameter(s) provide the independent variable value(s) that correspond to the observed value in *srcWaveName*.

Note: Cleveland et al. (1992) use the term "multiple factors" instead of "multivariate", hence the keyword name "factors" is used to denote these waves.

Use one factors wave when *srcWaveName* is the one-dimensional Y wave of an XY data pair:

```
srcWaveName[i] = someFunction(factorWaveName1[i])
```

Use multiple factors waves when *srcWaveName* contains the values of a multivariate function. “Multivariate” means that *srcWaveName* contains the observed results of a process that combines multiple independent input variables:

```
srcWaveName[i] = someFunction(factorWaveName1[i], factorWaveName2[i],…)
```

A maximum of 10 factors waves is supported.

All factors wave(s) must be numeric, noncomplex, one-dimensional and have the same number points as *srcWaveName*.

Any NaN values in *srcWaveName*[i], *factorWaveName1*[i], *factorWaveName2*[i], … cause all corresponding values to be ignored. Factors waves may contain NaN values only when /DFCT is specified.

Loess does not support NaNs in any of *destFactorWaveName1*, *destFactorWaveName2*,… and the results are undefined.

Flags

/CONF={*confInt*, *ciPlusWaveName* [,*ciMinusWaveName*]}

confInt specifies the confidence interval (a probability value from 0 to 1). *ciPlusWaveName* and the optional *ciMinusWaveName* are the names of new or overwritten output waves to hold the fitted value ± the confidence interval.

Note: /CONF uses large memory allocations, approximately N*N*8 bytes, where N is the number of points in *srcWaveName* (see **Memory Details**).

/DEST=*destWaveName* Specifies the name of the wave to hold the smoothed data. It creates *destWaveName* if it does not already exist or overwrites it if it does. The x (and possibly y) scaling of *destWave* determines the independent (factor) coordinates unless /DFCT={*destFactorWaveName1* [,*destFactorWaveName2*...]} is also specified.

/DFCT Specifies that the /DEST wave's x (and possibly y) scaling determines the independent (factor) coordinates at which to compute the smoothed data.

/DFCT={*destFactorWaveName1* [,*destFactorWaveName2*…]}

Specifies the names of one-dimensional waves providing the independent coordinates at which to compute the smoothed data.

If /DFCT={...} is used, the same number of waves must be specified for /DFCT and for factors = {*factorWaveName1* [, *factorWaveName2*…]}, though their lengths may (and usually will) be different. The length of *destFactorWaveName* waves must be the same as that of the *destWaveName* wave.

All destination factor waves must be numeric, noncomplex, and one-dimensional. The number of destination factor waves must match the number of source factor waves (if specified), or match the dimensionality of *srcWaveName* (one destination factor wave if *srcWaveName* is one-dimensional, two destination factors waves if *srcWaveName* is two-dimensional.)

The values in the destination factor waves may not be NaN.

/E=*extrapolate* Set *extrapolate* to nonzero to use a slower fitting method that computes values beyond the domain defined by the source factors. This is the “surface” parameter named “direct” in Cleveland et al. (1992). The default is *extrapolate* = 0, which uses the “interpolate” surface parameter, instead.

/N=*neighbors* Specifies the number of values in the smoothing window.

If *neighbors* is even, the next larger odd number is used. When *neighbors* is less than two, no smoothing is done.

The default is `0.5*numpnts(`*`srcWaveName`*`)` rounded up to the next odd integer or 3, whichever is larger.

Use either /N or /SMTH, but not both.

/NORM [=*norm*]

Set *norm* to 0 when specifying multiple factors and they all have the same scale and meaning, for example multiple factors all in units of meters.

The default is *norm* = 1, which normalizes each factor independently when computing the weighting function. This is appropriate when the factors are not dimensionally related, for example one factor measures wavelength and another measures temperature.

/ORD=*order*

Specifies the regression (fitting) order, the *d* parameter in Cleveland (1979):

order=0: Fits a constant to the locally-weighted neighbors around each point.

order=1: Fits a line to the locally-weighted neighbors around each point (Lowess smoothing).

order=2: Default; fits a quadratic (Loess smoothing).

/PASS=*passes*

The number of iterations of local weighting and regression fitting performed. The minimum is 1 and the default is 4. The *passes* parameter corresponds to the *t* parameter in Cleveland (1979).

/R [=*robust*]

Set *robust* to nonzero to use a robust fitting method that uses a bisquare weight function instead of the normal tricube weight function. This corresponds to the “symmetric” family in Cleveland et al. (1992). The robust method is less affected by outliers. The default is *robust* = 0, which is the “gaussian” family in Cleveland et al. (1992).

/SMTH=*sf*

Another way to express the number of values in the smoothing window, 0 ≤ *sf* ≤ 1. The default is 0.5.

To compute *neighbors* from *sf*, use:

```
neighbors = 1+floor(sf*numpnts(srcWaveName)).
```

Use either /N or /SMTH, but not both.

/TIME=*secs*

secs is the number of seconds allowed to complete the calculation before either warning (default) or stopping.

If the stop bit (4) of /V=*verbose* is set, the caculcation stops after the alloted time. If the diagnostic bit of /V=*verbose* is also set, warnings about the calculation exceeding the allotted time are printed to the history area.

As an example, use /TIME=30/V=6 to abort calculations longer than 30 seconds and print the warning to the history area.

/V [=*verbose*]

Controls how much information to print to the history area. *verbose* is a bitwise parameter with each bit controlling one aspect:

verbose=0: Prints nothing to the history area (default).

verbose=1: Prints the number of observations, equivalent number of parameters, and residual standard error.

verbose=2: Prints diagnostic information and error messages.

verbose=4 Use with /TIME. If this bit is set, calculations that exceed *secs* seconds are aborted.

Set *verbose* to 6 to both limit the time and print diagnostic and error messages.

/V alone is the same as /V=3, which prints all information.

S_info contains all the informational messages regardless of the value of

/Z[=z]

Set *z* to nonzero to prevent an error from stopping execution. Use the V_flag variable to see if the smoothing succeeded.

Basic Algorithm

The basic locally-weighted regression algorithm fits a constant, line, or quadratic to each point of the source data, using data that falls within the given span of neighbors over the factor data. Data outside of the span is ignored (given a weight of zero), and data inside the span is given a weight that depends on the distance of the data from the point being evaluated: data closer to the point being evaluated have higher weights and have a greater affect on the fit.

The basic algorithm uses the "tricube" weighting function to emphasize near values and deemphasize far values. For the one-factor case (simple XY data), the weighting function can be expressed as:

$$w_i = \left(1 - \left|\frac{x - x_i}{\max_q(x - x_i)}\right|^3\right)^3,$$

where $\max_q(x - x_i)$ is the maximum Euclidean distance of the q factor values within the given span from the factor point (x) whose observation (y) value is being evaluated.

The weights are applied to the factor values in the span to compute the constant, linear, or quadratic regression at x.

When multiple factors are used, the Euclidean distance is computed using one dimension per factor. The default is to normalize each factor's range by the standard deviation of that factor's values before computing the Euclidean distances. When factors are dimensionally equal, use the /NORM=0 option to skip this normalization. (See /NORM, about "dimensionally equal".)

Robust Algorithm

The robust algorithm adds to the basic algorithm a method to identify and remove outliers by rejecting values that exceed a threshold related to the "median absolute deviation" of the basic regression's residuals. The remaining values are used to compute robust "bisquare" weighting values:

$$r_i = \begin{cases} \left(1 - \left|\dfrac{e_i}{6 \cdot median(|e_i|)}\right|^2\right)^2 & for \quad 0 \le |e_i| < 6 \cdot median(|e_i|) \\ 0 & for \quad |e_i| > 6 \cdot median(|e_i|) \end{cases}$$

where e_i is the difference between the observed value and the regression's fitted value, and $\mathrm{median}(|e_i|)$ is evaluated for all the observed values.

These robust weighting values are multiplied with the original weighting values and a new regression (with new residuals) is computed. This process repeats 4 times by default. Use the /PASS flag to specify a different number of repetitions.

Details

Loess sets the variable V_flag to 0 if the smoothing was successful, or to an error code if not. Unlike other operations, the /Z flag allows execution to continue even if input parameters are in error.

Information printed to the history area when /V is set is always stored in the S_info string, even if /V=0 (the default). S_Info also contains the error message text if V_flag is an error code.

The error messages are described in Cleveland et al. (1992). They are often more dire than they seem.

The error message "Span too small. Fewer data values than degrees of freedom" usually means that the /SMTH or /N values are too small. The error code returned in V_Flag for this case is 1106.

The "Extrapolation not allowed with blending" (V_Flag = 1115) error usually means that the destination factors are trying to compute observations outside of the source factors domain without specifying /E=1. This happens if the /DEST *destWaveName* already exists and has X scaling that extends beyond the X scaling of *srcWaveName*. The solution is either kill the /DEST wave, limit the X scaling to the domain of the source wave, or use /E=1.

Memory Details

Loess requires a lot of memory, especially with the /CONF flag. Even without /CONF, the memory allocations exceed this approximation:

Number of bytes allocated = number of points in *srcWaveName* * 216

With /CONF, Loess can allocate large amounts of memory, approximately N*N*8 bytes, where N is the number of points in *srcWaveName*. The 2GB memory limit of 32-bit addressing limits *srcWaveName* to approximately 10,000 points when using /CONF.

More precisely, the memory allocation may be approximated by this function:

```
Function ComputeLoessMemory(srcPoints, numFactorsWaves, doConfidence)
   Variable srcPoints         // number of points in srcWave, aka N
   Variable numFactorsWaves// 1 or number of factors (independent variables)
   Variable doConfidence      // true if /CONF is specified

   Variable doubles= 9 * srcPoints            // 9 allocated double arrays
   doubles += 5 * numFactorsWaves * srcPoints // 5 more arrays
   doubles += (1+numFactorsWaves) * srcPoints // another array
   doubles += (1+numFactorsWaves) * srcPoints // another array
   doubles += (4+5) * srcPoints               // two more arrays
   if( doConfidence )
      doubles += srcPoints*srcPoints          // one HUGE array
   endif
   Variable bytes= doubles * 8
   return bytes
End

Macro DemoLoessMemory()
   Make/O wSrcPoints={10,100,1000,2000,3000,5000,7500,10000,12500,15000,20000}
   Duplicate/O wSrcPoints, loessMemory, loessMemory3, loessMemoryConf
   SetScale d, 0,0, "Points", wSrcPoints
   SetScale d, 0,0, "Bytes", loessMemory, loessMemory3, loessMemoryConf
   loessMemory= ComputeLoessMemory(wSrcPoints[p],1, 0)// 1 factor (X) no /CONF
   loessMemory3= ComputeLoessMemory(wSrcPoints[p],3, 0)// 3 factors (X,Y,Z) no /CONF
   loessMemoryConf= ComputeLoessMemory(wSrcPoints[p],1, 1)// 1 factor (X)with /CONF
   Display loessMemory vs wSrcPoints; Append loessMemory3 vs wSrcPoints
   ModifyGraph highTrip(bottom)=1e+08, rgb(loessMemory3)=(0,0,65535)
   ModifyGraph lstyle(loessMemory3)=2
   Legend
   Display loessMemoryConf vs wSrcPoints
   AutoPositionWindow
   ModifyGraph highTrip(bottom)=1e+08
End
```

Examples

1-D, factors are X scaling, output in new wave:

```
Make/O/N=200 wv=2*sin(x/8)+gnoise(1)
KillWaves/Z smoothed                    // ensure Loess creates a new wave
Loess/DEST=smoothed srcWave=wv          // 21-point loess.
Display wv; ModifyGraph mode=3,marker=19
AppendtoGraph smoothed; ModifyGraph rgb(smoothed)=(0,0,65535)
```

1-D, output in existing wave with more points than original data:

```
Make/O/N=100 short=2*cos(x/4)+gnoise(1)
Make/O/N=300 out; SetScale/I x, 0, 99, "" out  // same X range
Loess/DEST=out/DFCT/N=30 srcWave=short
Display short; ModifyGraph mode=3,marker=19
AppendtoGraph out
ModifyGraph rgb(out)=(0,0,65535),mode(out)=2,lsize(out)=2
```

1-D Y vs X wave data interpolated to waveform (Y vs X scaling) with 99% confidence interval outputs:

```
// NOx = f(EquivRatio)
// Y wave
// Note: The next 2 Make commands are wrapped to fit on the page.
Make/O/D NOx = {4.818, 2.849, 3.275, 4.691, 4.255, 5.064, 2.118, 4.602, 2.286, 0.97,
3.965, 5.344, 3.834, 1.99, 5.199, 5.283, 3.752, 0.537, 1.64, 5.055, 4.937, 1.561};

// X wave (Note that the X wave is not sorted)
Make/O/D EquivRatio = {0.831, 1.045, 1.021, 0.97, 0.825, 0.891, 0.71, 0.801,  1.074,
1.148, 1, 0.928, 0.767, 0.701, 0.807, 0.902,   0.997, 1.224, 1.089, 0.973, 0.98, 0.665};
```

```
// Interpolate to dense waveform over X range
Make/O/D/N=100 fittedNOx
WaveStats/Q EquivRatio
SetScale/I x, V_Min, V_max, "", fittedNOx
Loess/CONF={0.99,cp,cm}/DEST=fittedNOx/DFCT/SMTH=(2/3) srcWave=NOx, factors={EquivRatio}
Display NOx vs EquivRatio; ModifyGraph mode=3,marker=19
AppendtoGraph fittedNOx, cp,cm       // fit and confidence intervals
ModifyGraph rgb(fittedNOx)=(0,0,65535)
ModifyGraph mode(fittedNOx)=2,lsize(fittedNOx)=2
```

Interpolate X, Y, Z waves as a 3D surface.

```
// Note: The next 3 Make commands are wrapped to fit on the page.
Make/O/D vels= {1769, 1711, 1538, 1456, 1608, 1574, 1565, 1692, 1538, 1505, 1764, 1723, 
1540, 1441, 1428, 1584, 1552, 1690, 1673, 1548, 1485, 1526, 1536, 1591, 1671, 1647, 1608, 
1562, 1740, 1753, 1590, 1466, 1409, 1429}
Make/O/D ews={8.46279, 3.46303, -1.51508, -6.51483, 16.597, -5.95541, -28.5078, 9.68438, 
-6.00159, -21.7557, 14.263, 6.02058, -2.25772, -10.536, -18.7785, 10.7509, -6.07024, 
1.77531, 0.767701, -0.235545, -1.24315, 21.7298, 10.3964, 0.133859, -10.1733, -20.4359, 
13.7658, -8.88429, 10.8869, 4.91318, -0.0649319, -5.06469, -10.0428, -11.0601}
Make/O/D nss={-38.1732, -15.6207, 6.83407, 29.3865, 3.67947, -1.32028, -6.32004, -
10.3852, 6.43591, 23.3302, -37.1565, -15.6842, 5.88156, 27.4473, 48.9196, 10.0254, -
5.66059, -40.6613, -17.5832, 5.39486, 28.4729, 43.5833, 20.852, 0.26848, -20.4045, -
40.988, 3.0518, -1.9696, -49.1077, -22.1619, 0.292889, 22.8453, 45.3001, 49.8887}

// Evaluate the smoothed function as interpolated image
Make/O/N=(50,50) velsImage
WaveStats/Q ews
SetScale/I x, V_Min, V_Max, "" velsImage    // destination factors
WaveStats/Q nss
SetScale/I y, V_Min, V_Max, "" velsImage    // are X and Y scaling

Loess/DEST=velsImage/DFCT/NORM=0/SMTH=0.75/E/Z srcWave=vels, factors={ews,nss}

// Display source data as a contour with x, y markers.
Display; AppendXYZContour vels vs {ews,nss}
ModifyContour vels xymarkers=1, labels=0
ColorScale

// Display interpolated surface as an image
AppendImage velsImage
ModifyImage velsImage ctab= {*,*,Grays256,0}
ModifyGraph mirror=2
```

References

Cleveland, W.S., Robust locally weighted regression and smoothing scatterplots, *J. Am. Stat. Assoc.*, *74*, 829-836, 1979.

Cleveland, W.S., E. Grosse, and M.-J. Shyu, A Package of C and Fortran Routines for Fitting Local Regression Models, Technical Report, Bell Labs, 54pp, 1992.

NIST/SEMATECH, LOESS (aka LOWESS), in *NIST/SEMATECH e-Handbook of Statistical Methods*, <http://www.itl.nist.gov/div898/handbook/pmd/section1/pmd144.htm>, 2005.

See Also

Smooth, **Interpolate2**, **interp**, **MatrixFilter**, **MatrixConvolve**, and **ImageInterpolate**.

log

log(*num*)

The log function returns the log base 10 of *num*.

It returns -INF if *num* is 0, and returns NaN if *num* is less than 0.

To compute a logarithm base n use the formula:

$$\log_n(x) = \frac{\log(x)}{\log(n)}.$$

See Also

The **ln** function.

logNormalNoise

logNormalNoise(*m*,*s*)

The logNormalNoise function returns a pseudo-random value from the lognormal distribution function whose probability distribution function is

$$f(x,m,s)=\frac{1}{xs\sqrt{2\pi}}\exp\left\{-\frac{\left[\ln(x)-m\right]^2}{2s^2}\right\},$$

with a mean $\exp\left(m+\frac{1}{2}s^2\right)$,

and variance $\exp\left(2m+s^2\right)\left[\exp(s^2)-1\right]$.

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-390.

Chapter III-12, **Statistics** for a function and operation overview.

LombPeriodogram

LombPeriodogram [***flags***] ***srcTimeWave***, ***srcAmpWave*** **[**, ***srcFreqWave*** **]**

The LombPeriodogram is used in spectral analysis of signal amplitudes specified by *srcAmpWave* which are sampled at possibly random sampling times given by *srcTimeWave*. The only assumption about the sampling times is that they are ordered from small to large time values. The periodogram is calculated for either a set of frequencies specified by *srcFreqWave* (slow method) or by the flags /FR and /NF (fast method). Unless you specify otherwise, the results of the operation are stored by default in W_LombPeriodogram and W_LombProb in the current data folder.

Flags

/DESP=*datafolderAndName*

Saves the computed P-values in a wave specified by *datafolderAndName*. The destination wave will be created or overwritten if it already exists. *dataFolderAndName* can include a full or partial path with the wave name.

Creates by default a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-72 for details.

If this flag is not specified, the operation saves the P-values in the wave W_LombProb in the current data folder.

/DEST=*datafolderAndName*

Saves the computed periodogram in a wave specified by *datafolderAndName*. The destination wave will be created or overwritten if it already exists. *datafolderAndName* can include a full or partial path with the wave name (`/DEST=root:bar:destWave`).

Creates by default a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-72 for details.

If this wave is not specified the operation saves the resulting periodogram in the wave W_LombPeriodogram in the current data folder.

/FR=fRes — Use /FR to specify the frequency resolution of the output. This flag is used together with /NF to specify the range of frequencies for which the periodogram is computed. Note that *fRes* is also the lowest frequency in the output.

/NF=numFreq — Use /NF to specify the number of frequencies at which the periodogram is computed. The range of frequencies of the periodogram is then [*fRes*, (*numFreq*-1)**fRes*].

/Q — Quiet mode; suppresses printing results in the history area.

/Z — Do not report any errors.

Details

The LombPeriodogram (sometimes referred to as "Lomb-Scargle" periodogram) is useful in detection of periodicities in data. The main advantage of this approach over Fourier analysis is that the data are not required to be sampled at equal intervals. For an input consisting of N points this benefit comes at a cost of an O(N^2) computations which becomes prohibitive for large data sets. The operation provides the option of computing the periodogram at equally spaced (output) frequencies using /FR and /NF or at completely arbitrary set of frequencies specified by *srcFreqWave*. It turns out that when you use equally spaced output frequencies the calculation is more efficient because certain parts of the calculation can be factored.

The Lomb periodogram is given by

$$LP(\omega) = \frac{1}{2\sigma^2}\left\{ \frac{\left[\sum_{i=0}^{N-1}(y_i - \bar{y})\cos\left[\omega(t_i - \tau)\right]\right]^2}{\sum_{i=0}^{N-1}\cos^2\left[\omega(t_i - \tau)\right]} + \frac{\left[\sum_{i=0}^{N-1}(y_i - \bar{y})\sin\left[\omega(t_i - \tau)\right]\right]^2}{\sum_{i=0}^{N-1}\sin^2\left[\omega(t_i - \tau)\right]} \right\}$$

Here yi is the ith point in *srcAmpWave*, ti is the corresponding point in *srcTimeWave*,

$$\bar{y} = \frac{1}{N}\sum_{i=0}^{N-1} y_i ,$$

$$\tan(2\omega\tau) = \frac{\sum_{i=0}^{N-1}\sin(2\omega t_i)}{\sum_{i=0}^{N-1}\cos(2\omega t_i)} .$$

and

$$p = 1 - \left\{1 - \exp\left[LP(w)\right]\right\}^{N_{ind}} .$$

In the absence of a Nyquist limit, the number of independent frequencies that you can compute can be estimated using:

$$N_{ind} = -6.362 + 1.193N + 0.00098N^2 .$$

This expression was given by Horne and Baliunas derived from least square fitting. Nind is used to compute the P-values as:

$$p = 1 - \left\{ 1 - \exp\left[LP(w) \right] \right\}^{N_{ind}} .$$

Note that you can invert the last expression to determine the value of LP(w) for any significance level.

See Also
The **FFT** and **DSPPeriodogram** operations.

References

1. J.H. Horne and S.L. Baliunas, *Astrophysical Journal*, 302, 757-763, 1986.

2. N.R. Lomb, *Astrophysics and Space Science*, 39, 447-462, 1976.

3. W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes*, 3rd ed., Section 13.8.

lorentzianNoise

`lorentzianNoise(`*`a`*`,`*`b`*`)`

The function returns a pseudo-random value from a Lorentzian distribution

$$f(x) = \frac{1}{\pi} \frac{(b/2)}{(x-a)^2 + (b/2)^2} .$$

Here *a* is the center and *b* is the full line width at half maximum (FWHM).

See Also
SetRandomSeed, **enoise**, **gnoise**.

Noise Functions on page III-390.

Chapter III-12, **Statistics** for a function and operation overview.

LowerStr

`LowerStr(`*`str`*`)`

The LowerStr function returns a string expression identical to *str* except that all upper-case ASCII characters are converted to lower-case.

See Also
The **UpperStr** function.

Macro

`Macro` ***`macroName`*** **`(`**`[`***`parameters`***`]`**`)`** `[`**`:`*`macro type`***`]`

The Macro keyword introduces a macro. The macro will appear in the Macros menu unless the procedure file has an explicit Macros menu definition. See Chapter IV-4, **Macros** and **Macro Syntax** on page IV-118 for further information.

MacroInfo

`MacroInfo(`*`macroNameStr`*`)`

The MacroInfo function returns a keyword-value pair list of information about the macro specified by *macroNameStr*.

MacroInfo was added in Igor Pro 9.01.

In this section, "macro" includes all types of interpreted procedures, namely procedures introduced by the **Macro**, **Proc** and **Window** keywords.

Parameters
macroNameStr is a string expression containing the name of a macro.

If *macroNameStr* is "", MacroInfo returns information about the currently executing macro or "" if no macro is executing.

Details

Because macros exist only in the ProcGlobal context, *macroNameStr* must contain a simple name, not be a double name in `<module>#<name>` format.

The returned string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

Keyword	Information Following Keyword
NAME	The name of the macro. Same as contents of *macroNameStr* in most cases.
PROCWIN	Title of procedure window containing the macro definition.
PROCLINE	Line number within the procedure window of the macro definition.
MODULE	Module containing macro definition (see **Regular Modules** on page IV-236). Blank if the procedure window lacks a #pragma moduleName definition.
KIND	Macro, Proc or Window.
N_PARAMS	Number of parameters for this macro.
SUBTYPE	The macro subtype.

See Also

Macro, **Proc**, **Window**, **MacroList**, **MacroPath**, **FunctionInfo**, **StringByKey**, **NumberByKey**

MacroList

`MacroList(`*`matchStr`*`, `*`separatorStr`*`, `*`optionsStr`*`)`

The MacroList function returns a string containing a list of the names of user-defined procedures that start with the Proc, Macro, or Window keywords that also satisfy certain criteria. Note that if the procedures need to be compiled, then MacroList may not list all of the procedures.

Parameters

Only macros having names that match *matchStr* string are listed. See **WaveList** for examples.

separatorStr is appended to each macro name as the output string is generated. *separatorStr* is usually ";" for list processing (See **Processing Lists of Waves** on page IV-198 for details on list processing).

optionsStr is used to further qualify the macros. It is a string containing keyword-value pairs separated by commas. Available options are:

KIND:*nk* Determines the kind of procedure returned.

nk=1:	List Proc procedures.
nk=2:	List Macro procedures.
nk=4:	List Window procedures.

nk can be the sum of these values to match multiple procedure kinds. For example, use 3 to list both Proc and Macro procedures.

NPARAMS:*np* Restricts the list to macros having exactly *np* parameters. Omitting this option lists macros having any number of parameters.

SUBTYPE:*typeStr* Lists macros that have the type *typeStr*. That is, you could use ButtonControl as *typeStr* to list only macros that are action procedures for buttons.

WIN:*windowNameStr*

Lists macros that are defined in the named procedure window. "Procedure" is the name of the built-in procedure window.

Note: Because *optionsStr* keyword-value pairs are comma separated and procedure window names can have commas in them, the WIN: keyword must be the last one specified.

Examples

To list all Macros with three parameters:

```
Print MacroList("*",";","KIND:2,NPARAMS:3")
```

To list all Macro, Proc, and Window procedures in the main procedure window whose names start with b:

```
Print MacroList("b*",";","WIN:Procedure")
```

See Also

The **DisplayProcedure** operation and the **FunctionList**, **OperationList**, **StringFromList**, and **WinList** functions.

For details on procedure subtypes, see **Procedure Subtypes** on page IV-204, as well as **Button**, **CheckBox**, **SetVariable**, and **PopupMenu**.

MacroPath

`MacroPath(`*`macroNameStr`*`)`

The MacroPath function returns a path to the file containing the named macro.

MacroPath was added in Igor Pro 9.01.

In this section, "macro" includes all types of interpreted procedures, namely procedures introduced by the **Macro**, **Proc** and **Window** keywords.

Parameters

If *macroNameStr* is "", MacroPath returns the path to the currently executing macro or "" if no macro is executing.

Otherwise MacroPath returns the path to the named macro or "" if no macro by that name exists.

Details

MacroPath is useful in certain specialized cases, such as if a macro needs access to a lookup table of a large number of values.

The most likely use for this is to find the path to the file containing the currently running macro. This is done by passing "" for *macroNameStr*.

The returned path uses Macintosh syntax regardless of the current platform. See **Path Separators** on page III-451 for details.

If the procedure file is a normal standalone procedure file, the returned path will be a full path to the file.

If the macro resides in the built-in procedure window the returned path will be ":Procedure". If the macro resides in a packed procedure file, the returned path will be ":<packed procedure window title>".

If MacroPath is called when procedures are in an uncompiled state, it returns ":".

See Also

Macro, **Proc**, **Window**, **MacroInfo**, **MacroList**, **FunctionPath**

magsqr

`magsqr(`*`z`*`)`

The magsqr function returns the sum of the squares of the real and imaginary parts of the complex number *z*, that is, the magnitude squared.

Examples

Assume waveCmplx is complex and waveReal is real.

```
waveReal = sqrt(magsqr(waveCmplx))
```

sets each point of waveReal to the magnitude of the complex points in waveCmplx.

You may get unexpected results if the number of points in waveCmplx differs from the number of points in waveReal because of interpolation. See **Mismatched Waves** on page II-83 for details.

See Also

The **cabs** function.

WaveMetrics provides Igor Technical Note 006, “DSP Support Macros” which uses the magsqr function to compute the magnitude of FFT data, and Power Spectral Density with options such as windowing and segmenting. See the Technical Notes folder. Some of the techniques discussed there are available as Igor procedure files in the “WaveMetrics Procedures:Analysis:” folder.

Make

Make [***flags***] ***waveName*** [**,** ***waveName***]…
Make [***flags***] ***waveName*** [**= {*n0*,*n1*,**…**}**]…
Make [***flags***] ***waveName*** [**= {{*n0*,*n1*,**…**},{*n0*,*n1*,**…**},**…**}**]…

The Make operation creates the named waves. Use braces to assign data values when creating the wave.

Flags

/B Makes 8-bit signed integer waves or unsigned waves if /U is present.

/C Makes complex waves.

/D Makes double precision waves.

/DF Wave holds data folder references.

See **Data Folder References** on page IV-78 for more discussion.

/FREE[=*nm*] Creates a free wave. Allowed only in functions and only if a simple name or wave reference structure field is specified.

See **Free Waves** on page IV-91 for further discussion.

If *nm* is present and non-zero, then waveName is used as the name for the free wave, overriding the default name '_free_'. The ability to specify the name of a free wave was added in Igor Pro 9.00 as a debugging aid - see **Free Wave Names** on page IV-95 and **Wave Tracking** on page IV-207 for details.

/I Makes 32-bit signed integer waves or unsigned waves if /U is present.

/L Makes 64-bit signed integer waves or unsigned waves if /U is present. Requires Igor Pro 7.00 or later.

/N=*n* *n* is the number of points each wave will have. If *n* is an expression, it must be enclosed in parentheses: `Make/N=(myVar+1) aNewWave`

/N=(*n1*, *n2*, *n3*, *n4*)

n1, *n2*, *n3*, *n4* specify the number of rows, columns, layers and chunks each wave will have. Trailing zeros can be omitted (e.g., `/N=(`*`n1`*`, `*`n2`*`, 0, 0)` can be abbreviated as `/N=(`*`n1`*`, `*`n2`*`))`.

/O Overwrites existing waves in case of a name conflict. After an overwrite, you cannot rely on the contents of the waves and you will need to reinitialize them or to assign appropriate values.

/R Makes real value waves (default).

/T Makes text waves.

/T=*size* Makes text waves with pre-allocated storage.

size is the number of bytes preallocated by Igor for each element in each text wave. The waves are not initialized - it is up to you to initialize them.

Preallocation can dramatically speed up text wave assignment when the wave has a very large number of points but only when all strings assigned to the wave are exactly the same size as the preallocation size.

/U Makes unsigned Integer waves.

/W Makes 16-bit signed integer waves or unsigned waves if /U is present.

/WAVE Wave holds wave references.

See **Wave References** on page IV-71 for more discussion.

/Y=*type* See ***Wave Data Types*** below.

Wave Data Types

You can use /Y=(*numType*) to set the data type instead of the /B, /C, /D, /I, /L, /R, /T, /U, and /W data type flags. See **WaveType** function for *numType* values. The /Y flag overrides other type flags. You still need to use the explicit data type flags to control the automatic wave reference created by the compiler if you use the wave in an assignment statement in the same function; see **WAVE Reference Types** on page IV-73 for details.

Details

The maximum allowed number of elements (rows*columns*layers*chunks) in a wave depends on whether you are using the 64-bit version of Igor (max is 214,700,000,000) or the 32-bit version (max is 2,147,000,000).

Unless overridden by the flags, the created waves have the default length, type, precision, units and scaling. The factory defaults are:

Note: The preferred precision set by the Miscellaneous Settings dialog only presets the Make Waves dialog checkbox and determines the precision of imported waves. It does not affect the Make operation.

Property	Default
Number of points	128
Precision	Single precision floating point
Type	Real
dimensions	1
x, y, z, and t scaling	offset=0, delta=1 (“point scaling”)
x, y, z, and t units	"" (blank)
Data Full Scale	0, 0
Data units	"" (blank)

The maximum allowed number of elements (rows*columns*layers*chunks) in a wave is 214,700,000,000.

See Also

The **SetScale**, **Duplicate**, and **Redimension** operations.

MakeIndex

`MakeIndex [/A/C/R] `*`sortKeyWaves, indexWave`*

The MakeIndex operation sets the data values of *indexWave* such that they give the ordering of *sortKeyWaves*.

For simple sorting problems, MakeIndex is not needed. Just use the **Sort** operation.

Parameters

sortKeyWaves is either the name of a single wave, to use a single sort key, or the name of multiple waves in braces, to use multiple sort keys.

indexWave must specify an existing numeric wave.

All waves must be of the same length and must not be complex.

Flags

/A	Alphanumeric. When *sortKeyWaves* includes text waves, the normal sorting places “wave1” and “wave10” before “wave9”. Use /A to sort the number portion numerically, so that “wave9” is sorted before “wave10”.
/C	Case-sensitive. When *sortKeyWaves* includes text waves, the ordering is case-insensitive unless you use the /C flag which makes it case-sensitive.
/LOC	Performs a locale-aware sort. When *sortKeyWaves* includes text waves, the text encoding of the text waves’ data is taken into account and sorting is done according to the sorting conventions of the current system locale. This flag is ignored if the text waves’ data encoding is unknown, binary, Symbol, or Dingbats. This flag cannot be used with the /A flag. See Details for more information. The /LOC flag was added in Igor Pro 7.00.
/R	Reverse the index so that ordering is from largest to smallest.

Details

MakeIndex is used in preparation for a subsequent **IndexSort** operation. If /R is used the ordering is from largest to smallest. Otherwise it is from smallest to largest.

When the /LOC flag is used, the bytes stored in the text wave at each point are converted into a Unicode string using the text encoding of the text wave data. These Unicode strings are then compared using OS specific text comparison routines based on the locale set in the operating system. This means that the order of sorted items may differ when the same sort is done with the same data under different operating systems or different system locales.

When /LOC is omitted the sort is done on the raw text without regard to the waves’ text encoding.

See Also

Sorting on page III-132, **MakeIndex and IndexSort** on page III-134, **Sort**, **IndexSort**

MandelbrotPoint

MandelbrotPoint(*x*, *y*, *maxIterations*, *algorithm*)

The MandelbrotPoint function returns a value between 0 and *maxIterations* based on the Mandelbrot set complex quadratic recurrence relation z[n] = z[n-1]^2 + c where *x* is the real component of c, *y* is the imaginary component of c and z[0] = 0.

The returned value is the number of iterations the equation was evaluated before |z[n]| > 2 (the escape radius of the Mandelbrot set), or maxIterations, whichever is less.

Parameters

algorithm=0	The "Escape Time" algorithm returns the integer n which is the number of iterations until |z[n]| > 2.
algorithm=1	The "Renormalized Iteration Count Algorithm" algorithm returns a floating point value which is a refinement of the number of iterations n by adding the quantity: `5 - ln( ln( |z[n+4]| ) ) / ln(2)` (which requires four more iterations of the recurrence relation). The returned value is clipped to maxIterations.

See Also

The “MultiThread Mandelbrot Demo” experiment.

References

http://en.wikipedia.org/wiki/Mandelbrot_set

http://linas.org/art-gallery/escape/escape.html

MarcumQ

MarcumQ(*m*, *a*, *b*)

The MarcumQ function returns the generalized Q-function defined by the integral

$$Q_m(a,b) = \int_b^\infty u\left(\frac{u}{a}\right)^{m-1} \exp\left(-\frac{(a^2+u^2)}{2}\right) I_{m-1}(au)du$$

where I_k is the modified Bessel function of the first kind and order k.

Its applications have been primarily in the fields of communication and detection theory. However, an interesting interpretation of its result with m=1 and appropriate parameter scaling is the fractional power of a two-dimensional circular Gaussian function within a displaced circular aperture.

Depending on the input arguments, the MarcumQ function may be computationally intensive but you can abort the calculation at any time.

References

Cantrell, P.E., and A.K. Ojha, Comparison of Generalized Q-Function Algorithms, *IEEE Transactions on Information Theory*, *IT-33*, 591-596, 1987.

Simon, M. K., A New Twist on the Marcum Q-Function and Its Application, *IEEE Communications Letters*, 3, 39-41, 1998.

MarkPerfTestTime

MarkPerfTestTime *idval*

Use the MarkPerfTestTime operation for performance testing of user-defined functions in conjunction with `SetIgorOption DebugTimer`. When used between `SetIgorOption DebugTimer, Start` and `SetIgorOption DebugTimer, Stop`, MarkPerfTestTime stores the ID value and the time of the call in a buffer. When `SetIgorOption DebugTimer, Stop` is called the contents of the buffer are dumped to a pair of waves: W_DebugTimerIDs will contain the ID values and W_DebugTimerVals will contain the corresponding times of the calls relative to the very first call. The timings use the same high precision mechanism as the StartMSTimer and StopMSTimer calls.

By default, `SetIgorOption DebugTimer, Start` allocates a buffer for up to 10000 entries. You can allocate a different sized buffer using `SetIgorOption DebugTimer, Start=bufsize`.

See Also

SetIgorOption, **StartMSTimer**, and **StopMSTimer**.

Additional documentation can be found in an example experiment, PerformanceTesting.pxp, and a WaveMetrics procedure file, PerformanceTestReport.ipf.

MatrixBalance

MatrixBalance [flags] *srcWave*

The MatrixBalance operation permutes and/or scales an NxN real or complex matrix to obtain a similar matrix that is more stable for subsequent calculations such as eigenvalues and eigenvectors. Balancing a matrix is useful when some matrix elements are much smaller than others.

MatrixBalance saves the resulting matrix in the wave specified by the /DSTM flag. It creates a secondary output wave specified by the /DSTS flag containing permutation and scaling information.

MatrixBalance was added in Igor Pro 9.00.

Parameters

srcWave must be single-precision or double-precision floating point, real or complex, and must contain no NaNs. MatrixBalance returns an error if these conditions are not met.

Flags

/DSTM=*dest*	Specifies the destination wave for the balanced output matrix. If you omit /DSTM, the output is saved in M_Balanced in the current data folder.

/DSTS=*dest* Specifies the destination wave for the scaling array. If you omit /DSTS, the scaling wave is saved as W_Scale in the current data folder.

/FREE Creates free destination waves when they are are specified via /DSTM or /DSTS.

/J=*job* *job* is one of the following letters:

N *srcWave* is not permuted or scaled.

P *srcWave* is permuted but not scaled.

S *srcWave* is scaled but not permuted. The scaling applies a diagonal similarity transformation to make the norms of the various columns close to each other.

B *srcWave* is both scaled and permuted (default).

/Z Do not report any errors. V_flag will be set to 0 if successful or to an error code. V_min and V_max will be set to NaN in case of an error.

Details

Matrix balancing is usually called internally by LAPACK routines when there is large variation in the magnitude of matrix elements. The /J flag supports the granularity of applying only permutation, only scaling or both.

After balancing there will be a block diagonal form that could typically be handled using Schur decomposition (see MatrixSchur). The rows/columns corresponding to this block diagonal are saved in zero based V_min and V_max. The scaling wave W_scale[i] (for i<V_min) contains the index of the row/column that was interchanged with row/col i. For i>=V_min the scaling wave contains the scaling factor use to balance row and column i.

When you balance *srcWave* using this operation the resulting eigenvectors belong to the balanced matrix. Use **MatrixReverseBalance** to obtain the eigenvectors of *srcWave*.

When solving for the Schur vectors of *srcWave* after using /J=P for permuting only, use **MatrixReverseBalance** to transform the vectors. If you are solving for the Schur vectors do not use either /J=S or /J=B because that balancing transformation is not orthogonal.

Output Variables

V_Flag Set to zero when the operation succeeds. Otherwise, when V_flag is positive the value is a standard error code. When V_flag is negative it is an indication of a wrong input parameter.

V_min Set to zero if /J=N or /J=S.

V_max Set to N-1 (where N is the number of rows or columns) if /J=N or /J=S.

When /J=B or /J=P the matrix M_Balanced[i][j]=0 if i>j and j=0...V_min-2 or i=V_max,...N-1.

Examples

```
Function DemoMatrixBalance1()
   // Generate random unbalanced data
   Make/D/O/N=(7,7) srcWave=p+10*q+enoise(5)
   MatrixBalance srcWave
   Wave M_Balanced
   Wave W_Scale

   MatrixOP/O/FREE matD=diagonal(W_Scale)
   MatrixOP/O/FREE matDInv=inv(matD)

   // Check that W_Scale produces the correct balanced matrix
   MatrixOP/O/FREE matProd=matDInv x srcWave x matD

   // Compare with M_Balanced
   MatrixOP/O/FREE/P=1 aa=sum(abs(M_Balanced-matProd))
End
Function DemoMatrixBalance2()
   Make/O/N=(3,3) mat
```

```
    mat[0][0] = {2,88,164}
    mat[0][1] = {0,1,1}
    mat[0][2] = {1e-05,0,1}

    MatrixBalance mat
    Wave W_scale,M_Balanced
    Print "EqualWaves(input,output,-1)", EqualWaves(mat,M_Balanced,-1)

    // Calculate right eigenvectors for the original matrix
    MatrixEigenV/R mat
    Wave/Z M_R_eigenVectors
    Duplicate/O M_R_eigenVectors, origEigenVectors
    Wave/Z W_eigenValues
    Duplicate/O W_eigenValues,origEigenValues

    // Calculate the right eigenvectors for the balanced matrix
    MatrixEigenV/R M_Balanced
    Wave/Z M_R_eigenVectors
    Wave/Z W_eigenValues
    MatrixOP/O/FREE tmp = sum(abs(origEigenValues-W_eigenValues))
    if (tmp[0] > 0.1)
       Print "Eigenvalues difference"
    else
       Print "Eigenvalues OK"
    endif

    // Reverse the balance and compare with original eigenvectors
    MatrixReverseBalance/J=P/SIDE=R/LH={V_min,V_max} W_Scale,M_R_eigenVectors
    Wave/Z M_RBEigenvectors
    MatrixOP/O/FREE tmp = sum(abs(M_RBEigenvectors-origEigenVectors))
    if (tmp[0] > 0.1)
       Print "Eigenvectors difference on reverse"
    else
       Print "Eigenvectors ok on reverse"
    endif
End
```

References

The operation uses the following LAPACK routines: sgebal, dgebal, cgebal, and zgebal.

See Also

MatrixReverseBalance

MatrixCondition

`MatrixCondition(`*`wave2D`*`, `*`mode`*`)`

MatrixCondition returns the estimated reciprocal of the condition number of a 2D square matrix wave2D.

The condition number is the product of the norm of the matrix with the norm of the inverse of the matrix (see details below). The type of norm is determined by the value of the mode parameter. 1-norm is used if mode is 1 and infinity-norm is used otherwise.

The MatrixCondition function was added in Igor Pro 7.00.

Details

The function uses LAPACK routines to estimate the reciprocal condition number by first obaining the norm of the input matrix and then using LU decomposition to obtain the norm of the inverse of the matrix. The estimate returned is

$$reciprocalCon = \frac{1}{\left\| wave2D \right\| * \left\| wave2D^{-1} \right\|},$$

where the norms are selected by the choice of the mode parameter. The 1-norm of matrix A with elements aij is defined as

$$\left\| A \right\|_1 = \underset{1 \le j \le n}{\max} \sum_{i=1}^{m} \left| a_{ij} \right|,$$

and the infinity-norm is defined by

$$\|A\|_\infty = \max_{1 \le i \le m} \sum_{j=1}^{n} |a_{ij}|.$$

The function returns a NaN if there is any error in the input parameters.

References

http://en.wikipedia.org/wiki/Matrix_norm

See Also

MatrixSVD provides a condition number for L2 norm using the ratio of singular values.

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-138 for more about Igor's matrix routines.

MatrixConvolve

`MatrixConvolve` `[`**`/R=`*`roiWave`***`]` ***`coefMatrix`*****`,`** ***`dataMatrix`***

The MatrixConvolve operation convolves a small coefficient matrix *coefMatrix* into the destination *dataMatrix*.

Flags

/R=*roiWave*	Modifies only data contained inside the region of interest. The ROI wave should be 8-bit unsigned with the same dimensions as *dataMatrix*. The interior of the ROI is defined by zeros and the exterior is any nonzero value.

Details

On input *coefMatrix* contains an *N*x*M* matrix of coefficients where *N* and *M* should be odd. Generally *N* and *M* will be equal. If *N* and *M* are greater than 13, it is more efficient to perform the convolution using the Fourier transform (see **FFT**).

The convolution is performed in place on the data matrix and is acausal, i.e., the output data is not shifted.

Edges are handled by replication of edge data.

When *dataMatrix* is an integer type, the results are clipped to limits of the given number type. For example, unsigned byte is clipped to 0 to 255.

MatrixConvolve works also when both *coefMatrix* and *dataMatrix* are 3D waves. In this case the convolution result is placed in the wave M_Convolution in the current data folder, and the optional /R=*roiWave* is required to be an unsigned byte wave that has the same dimensions as *dataMatrix*.

This operation does not support complex waves.

See Also

MatrixFilter and **ImageFilter** for filter convolutions.

Matrix Math Operations on page III-138 for more about Igor’s matrix routines.

The **Loess** operation.

MatrixCorr

`MatrixCorr` `[`**`/COV`**`][`**`/DEGC`**`]` ***`waveA`*** `[`**`,`** ***`waveB`***`]`

The MatrixCorr operation computes the correlation or covariance or degree of correlation matrix for the input 1D wave(s).

If we denote elements of *waveA* by $\{x_i\}$ and elements of *waveB* by $\{y_i\}$ then the correlation matrix for these waves is the vector product of the form:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} y_1 & y_2 & y_3 & \dots & y_n \end{bmatrix}^* = \begin{bmatrix} x_1y_1^* & x_1y_2^* & x_1y_3^* & \dots & x_1y_n^* \\ x_2y_1^* & x_2y_2^* & x_2y_3^* & & x_2y_n^* \\ x_3y_1^* & x_3y_2^* & x_3y_3^* & & x_3y_n^* \\ \vdots & & & & \\ x_ny_1^* & x_ny_2^* & x_ny_3^* & \cdots & x_ny_n^* \end{bmatrix}$$

where * denotes complex conjugation. If you use the optional *waveB* then the matrix is the cross correlation matrix. *waveB* must have the same length of *waveA* but it does not have to be the same number type.

Flags

The flags are mutually exclusive; only one matrix can be generated at a time.

/COV Calculates the covariance matrix.

The covariance matrix for the same input is formed in a similar way after subtracting from each vector its mean value and then dividing the resulting matrix elements by (*n*-1) where *n* is the number of elements of *waveA*.

Results are stored in the M_Corr or M_Covar waves in the current data folder.

/DEGC Calculates the complex degree of correlation. The degree of correlation is defined by:

$$\deg C = \frac{M\_Co\mathrm{var}}{\sqrt{Var(waveA) \cdot Var(waveB)}},$$

where *M_Covar* is the covariance matrix and *Var*(*wave*) is the variance of the wave.

The complex degree of correlation should satisfy: $0 \le |\deg C| \le 1.$

Examples

The covariance matrix calculation is equivalent to:

```
Variable N=1/(DimSize(waveA,0)-1)
Variable ma=mean(waveA,-inf,inf)
Variable mb=mean(waveB,-inf,inf)
waveA-=ma
waveB-=mb
MatrixTranspose/H waveB
MatrixMultiply waveA,waveB
M_product*=N
```

See Also

Matrix Math Operations on page III-138 for more about Igor’s matrix routines.

References

Hayes, M.H., *Statistical Digital Signal Processing And Modeling*, 85 pp., John Wiley, 1996.

MatrixDet

`matrixDet(`*`dataMatrix`*`)`

The MatrixDet function returns the determinant of *dataMatrix*. The matrix wave must be a real, square matrix or else the returned value will be NaN.

Details

The function calculates the determinant using LU decomposition. If, following the decomposition, any one of the diagonal elements is either identically zero or equal to 10^{-100}, the return value of the function will be zero.

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-138 for more about Igor's matrix routines.

MatrixDot

`MatrixDot(`*`waveA`*`, `*`waveB`*`)`

The MatrixDot function calculates the inner (scalar) product for two 1D waves. A 1D wave **A** represents a vector in the sense:

$$\mathbf{A} = \sum \alpha_i \hat{e}_i, \qquad \hat{e}_i \text{ is a unit vector.}$$

Given two such waves **A** and **B**, the inner product is defined as

$$ip = \sum \alpha_i \beta_i .$$

When both *waveA* and *waveB* are complex and the result is assigned to a complex-valued number MatrixDot returns:

$$ipc = \sum \alpha_i^* \beta_i .$$

If you prefer the definition where the second factor is the one that is conjugated, you can simply reverse the order of *waveA* and *waveB* in the function call.

If the result is assigned to a real number, MatrixDot returns:

$$ip = \left| \sum \alpha_i^* \beta_i \right| .$$

If either *waveA* or *waveB* is complex and the result is assigned to a real-valued number, MatrixDot returns:

$$ip = \left| \sum \alpha_i \beta_i \right| .$$

When the result is assigned to a complex-valued number MatrixDot returns:

$$ipc = \sum \alpha_i \beta_i .$$

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-138 for more about Igor's matrix routines.

MatrixEigenV

`MatrixEigenV` `[`***`flags`***`]` ***`matrixA`*** **`[,`** ***`matrixB`*** **`]`**

MatrixEigenV computes the eigenvalues and eigenvectors of a square matrix using LAPACK routines.

Flags for General Matrices

/C	Creates complex valued output for real valued input waves. This is equivalent to converting the input to complex, with zero imaginary component, prior to computing the eigenvalues and eigenvectors. The main benefit of this format is that the output has simple packing of eigenvalues and eigenvectors. See the example in the **Details for General Matrices** section below. The /C flag was added in Igor Pro 7.00.

/B=*balance* Determines how the input matrix should be scaled and or permuted to improve the conditioning of the eigenvalues.

balance=0 (default), 1, 2, or 3, corresponding respectively to N, P, S, or B in the LAPACK routines. Applicable only with the /X flag.

0:	Do not scale or permute.
1:	Permute.
2:	Do diagonal scaling.
3:	Scale and permute.

/L Calculates for left eigenvectors.

/O Overwrites *matrixWave*, requiring less memory.

/R Calculates for right eigenvectors.

/S=*sense* Determines which reciprocal condition numbers are calculated.

sense=0 (default), 1, 2, or 3, corresponding respectively to N, E, V, or B in the LAPACK routines. Applicable only with the /X flag.

0:	None.
1:	Eigenvalues only.
2:	Right eigenvectors.
3:	Eigenvalues and right eigenvectors.

If sense is 1 or 3 you must compute both left and right eigenvectors.

NOTE: /S is applicable only with the /X flag or for the generalized eigenvalue problem.

/X Uses LAPACK expert routines, which require additional parameters (see /B and /S flags). The operation creates additional waves:

The W_MatrixOpInfo wave contains in element 0 the ILO, in element 1 the IHI, and in element 2 the ABNRM from the LAPACK routines.

The wave W_MatrixRCONDE contains the reciprocal condition numbers for the eigenvalues.

The wave W_MatrixRCONDV contains the reciprocal condition number for the eigenvectors.

Flags for Symmetric Matrices

/SYM Computes the eigenvalues of an NxN symmetric matrix and stores them in the wave W_eigenValues. You must specify this flag if you want to use the special routines for symmetric matrices. The number of eigenvalues is stored in the variable V_npnts. Because W_eigenValues has N points, only the first V_npnts will contain relevant eigenvalues.

When using this flag with complex input the matrix is assumed to be Hermitian.

/EVEC Computes eigenvectors in addition to eigenvalues. Eigenvectors will be stored in the wave M_eigenVectors, which is of dimension NxN. The first V_npnts columns of the wave will contain the V_npnts eigenvectors corresponding to the eigenvalues in W_eigenValues. /EVEC must be preceded by /SYM.

/RNG={*method*,*low*,*high*}

Determines what is computed:

method=0: Computes all the eigenvalues or eigenvectors (default).

method=1: Computes eigenvalues for *low* and *high* double precision range.

method=2: Computes eigenvalues for *low* and *high* integer indices (1 based). For example, to compute the first 3 eigenvalues use: `/RNG={2,1,3}`.

/RNG must be preceded by /SYM.

Flags for Generalized Eigenvalue Problem

These are the same flags as for general (non-symmetric) matrices above except for /O and /X which are not supported for the generalized eigenvalue solution.

Common Flags

/Z No error reporting (except for setting V_flag).

Details

There are three mutually exclusive branches for the operation. The first is designed for a square matrix input *matrixA*. The operation computes the solution to the problem

$$\mathbf{Ax} = \lambda\mathbf{x},$$

where A is the input matrix, x is an eigenvector and λ is an eigenvalue.

The second branch is designed for symmetric matrices A, i.e., when

$$\mathbf{A} = \mathbf{A}^{\mathbf{T}},$$

where the superscript T denotes a transpose.

The third branch of the operation is designed to solve the generalized eigenvalue problem,

$$\mathbf{Ax} = \lambda\mathbf{Bx},$$

where A and B are square matrices, x is an eigenvector and λ is an eigenvalue.

Each branch of the operation supports its own set of flags as shown above. All branches support input of single and double precision in real or complex waves. If you specify both *matrixA* and *matrixB* then they must have the same number type.

Details for General Matrices

The eigenvalues are returned in the 1D complex wave W_eigenValues. The eigenvectors are returned in the 2D wave M_R_eigenVectors or M_L_eigenVectors.

The calculated eigenvectors are normalized to unit length.

Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having the positive imaginary part first.

If the *jth* eigenvalue is real, then the corresponding eigenvector *u*(*j*)=*M*[][*j*] is the *jth* column of M_L_eigenVectors or M_R_eigenVectors. If the *jth* and (*j*+1)*th* eigenvalues form a complex conjugate pair, then *u*(*j*) = *M*[][*j*] + *i***M*[][*j*+1] and *u*(*j*+1) = *M*[][*j*] - *i***M*[][*j*+1].

Example

```
Function TestMatrixEigenVReal()
	Make/D/N=(2,2)/O eee={{0,1},{-1,0}}
	MatrixEigenV/R eee
	Wave W_eigenvalues, M_R_eigenVectors
	MatrixOP/O firstEV=cmplx(col(M_R_eigenVectors,0),col(M_R_eigenVectors,1))
	MatrixOP/O aa=eee x firstEV - W_eigenvalues[0]*firstEV
	Print aa
End

Function TestMatrixEigenVComplex()
	Make/D/N=(2,2)/O eee={{0,1},{-1,0}}
	MatrixEigenV/R/C eee
```

```
    Wave W_eigenvalues, M_R_eigenVectors
    MatrixOP/O aa=eee x col(M_R_eigenVectors,0) - W_eigenValues[0]*col(M_R_eigenVectors,0)
    Print aa
End
```

Details for Symmetric Matrices

The LAPACK routines that compute the eigenvalues and eigenvectors of symmetric matrices claim to use the Relatively Robust Representation whenever possible. If your matrix is symmetric you should use this branch of the operation (/SYM) for improved accuracy.

Details for Generalized Eigenvalue Problem

Here the right eigenvectors (/R) are solutions to the equation

$$\mathbf{Ax} = \lambda\mathbf{Bx},$$

and the left eigenvectors (/L) are solutions to

$$\mathbf{x^H A} = \lambda\mathbf{x^H B},$$

where the superscript H denotes the conjugate transpose.

When both *matrixA* and *matrixB* are real valued, the operation creates the following waves in the current data folder:

W_alphaValues	Contains the complex alpha values.
W_betaValues	Contains the real-valued denominator such that the eigenvalues are given by $\lambda_j = \dfrac{\alpha_j}{\beta_j}$.
M_leftEigenVectors and M_rightEigenVectors	Real valued waves where columns correspond to eigenvectors of the equation.

Every point in the wave W_alphaValues corresponds to an eigenvalue. When the imaginary part of W_alphaValues[j] is zero, the eigenvalue is real and the corresponding eigenvector is also real e.g., M_rightEigenVectors[][j]. When the imaginary part is positive then there are two eigenvalues that are complex-conjugates of each other with corresponding complex eigenvectors given by

```
cmplx(M_rightEigenVectors[][j],M_rightEigenVectors[][j+1])
```

and

```
cmplx(M_rightEigenVectors[][j],-M_rightEigenVectors[][j+1])
```

When both *matrixA* and *matrixB* are complex the operation creates the complex waves:

W_alphaValues, W_betaValues, M_leftEigenVectors, M_rightEigenVectors

with the ratio W_alphaValues[j]/W_betaValues[j] expressing the generalized eigenvalue. The corresponding M_leftEigenVectors[][j] and M_rightEigenVectors[][j] are the respective left and right eigenvectors. The simplicity of the complex case suggests that when *matrixA* and *matrixB* are real it is best to convert them to complex waves prior to executing matrixEigenV.

Depending on the choice of /S the operation also calculates the reciprocal condition number for the eigenvalues (stored in W_reciprocalConditionE) and the reciprocal condition number of the eigenvectors (stored in W_reciprocalConditionV). Note that a zero entry in W_reciprocalConditionV implies that the eigenvalues could not be ordered.

See Also

Matrix Math Operations on page III-138 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

Symmetric matrices can also be decomposed using the **MatrixSchur** operation and using **MatrixOp** chol.

MatrixFactor

`MatrixFactor [ `*`flags`*` ] `*`srcWave`*

The MatrixFactor operation computes two real-valued output matrices, conceptually called matA and matB, whose matrix product minimizes the Frobenius norm of:

```
| srcWave - (matA x matB) |
```

The MatrixFactor operation was added in Igor Pro 9.00.

Flags

/COMC=*cCols* Sets the common dimension in the factorization so that, for *srcWave* with m rows and n columns, the factorization will be (m x *cCols*) matA and (*cCols* x n) matB. If you omit /COMC, *cCols* defaults to m/2.

/CORR=*cRate* Sets the correction factor for the learning rate. The default value is 0.9.

/DSTA=*wA* Specifies the output wave representing matA.

/DSTB=*wB* Specifies the output wave representing matB.

/FREE Creates output waves as free waves.

/INIA=*iwA* Specifies an initial solution matrix for matA. The wave must have the same dimensions as matA and the same numeric type as *srcWave*.

/INIB=*iwB* Specifies an initial solution matrix for matB. The wave must have the same dimensions as matB and the same numeric type as *srcWave*.

/ITER=*nIters* Sets the maximum number of iterations. By default *nIters* is 1E7.

/LRNR=*lRate* Sets the initial learning rate of the algorithm. The default value of *lRate* is 0.01.

/OUT=*type* Sets restrictions on the output. The default value for *type* is 0 and there are no restrictions on the elements of the output. Set *type*=1 for non-negative output elements or *type*=2 for positive output elements.

/TOL=*tolerance* Use /TOL to terminate iterations when the average Frobenius norm per input point falls below the tolerance value. By default *tolerance* is 1E-9 for single precision floating point input wave and 1E-15 for double precision input.

/Q Quiet - do not print diagnostic information to the history area and do not display progress bar.

/Z Errors are not fatal and do not abort procedure execution. Your procedure can inspect the V_flag variable to see if the operation succeeded. V_flag will be zero if it succeeded or nonzero if it failed.

Details

srcWave must be a single or double precision real wave.

If you specify the factorization output waves using /DSTA and /DSTB, they must have the same numeric type.

If you omit /DSTA or /DSTB, MatrixFactor produces output waves named factorAMat and factorBMat with the same numeric type as *srcWave*.

The algorithm produces a non-unique solution that may depend on the initialization of the two factors. The default initialization, used if you omit /INIA and /INIB, uses Igor's random number generator. If you want to investigate convergence you can generate your own initial values and specify them using the /INIA and /INIB flags. If you want to provide your own initial values you must provide them for both matA and matB via both /INIA and /INIB. You can run MatrixFactor once and use the outputs as initial values for a second run.

We recommend using double precision waves because the algorithm involves iterations where the Frobenius norm is computed. For large matrices or for a large number of iterations the calculation is strongly susceptible to roundoff errors which may cause it to fail to converge.

The algorithm iterates until either the tolerance is met or the number of iterations, specified by /ITER or the default value of 1E7, is exceeded. The algorithm also terminates if the the computed norm remains relatively constant:

```
(prevIterationNorm-curIterationNorm)/prevIterationNorm < 1e-16.
```

Output Variables

MatrixFactor sets these automatically created variables:

V_flag — Set to 0 if the operation succeeded or to a non-zero error code.

V_avg — The last average Frobenius norm (Frobenius sum divided by the number of points in *srcWave*).

V_iterations — The number of iterations executed before the algorithm terminated.

Example

```
Function DemoMatrixFactor()
   Make/O/N=(10,8)/D matA0 = 1+abs(enoise(3))
   Make/O/N=(8,12)/D matB0 = 1+abs(enoise(3))
   MatrixOP/O matX0=matA0 x matB0
   MatrixFactor/COMC=8/DSTA = biMatA/DSTB=biMatB matX0
   MatrixOP/O/P=1 aa = sum(sq(matX0- biMatA x biMatB))/120     // Check norm
End
```

References

Nikulin, V., Tian-Hsiang Huang, S. Ng, S. Rathnayake and G. J. McLachlan. “A Very Fast Algorithm for Matrix Factorization.” *ArXiv* abs/1011.0506 (2010).

MatrixFilter

MatrixFilter** [**flags**] **Method dataMatrix

The MatrixFilter operation performs one of several standard image filter type operations on the destination *dataMatrix*.

Note: The parameters below are also available in ImageFilter. See **ImageFilter** for additional parameters.

Parameters

Method selects the filter type. *Method* is one of the following names:

avg — *n*x*n* average filter.

FindEdges — 3x3 edge finding filter.

gauss — *n*x*n* gaussian filter.

gradN, gradNW, gradW, gradSW, gradS, gradSE, gradE, gradNE

3x3 North, NorthWest, West, … pointing gradient filter.

median — *n*x*n* median filter. You can assign values other than the median by specifying the desired rank using the /M flag.

min — *n*x*n* minimum rank filter.

max — *n*x*n* maximum rank filter.

NanZapMedian — *n*x*n* filter that only affects data points that are NaN. Replaces them with the median of the *n*x*n* surrounding points. Unless /P is used, automatically cycles through matrix until all NaNs are gone or until *cols***rows* iterations.

point — 3x3 point finding filter `8*center-outer`.

sharpen — 3x3 sharpening `filter=(12*center-outer)/4`.

sharpenmore 3x3 sharpening `filter=(9*center-outer)`.

thin Calculates binary image thinning using neighborhood maps based on the algorithm in *Graphics Gems IV*, p. 465.

Note: The thin keyword to MatrixFilter will be removed someday. The functionality will be available — just not as a part of MatrixFilter. The /R flag does not apply to the lame duck thin keyword.

Flags

/B=*b* Specifies value that is considered background. Used with thin. If object is black on white background, use 255. If object is white on a black background, use 0.

/F=*value* Specifies the value in the ROI wave that marks excluded pixels. *value* is either 0 or 1.

This flag was added in Igor Pro 7.00.

By default, and for compatibility with Igor Pro 6, *value*=0. Use /F=1 if your ROI wave contains 1 for pixels to be excluded.

/M=*rank* Assigns a pixel value other than the median when used with the median filter. Valid *rank* values are between 0 and n^2-1 (for the default median *rank*= $n^{2/2}$).

/N=*n* For any method described above as "*nxn*", you can specify that the filtering kernel will be a square matrix of size *n*. In the absence of the /N flag, the default size is 3.

/P=*p* Filter passes over the data *p* times. The default is one pass.

/R=*roiWave* Only the data outside the region of interest will be modified. *roiWave* should be an 8-bit unsigned wave with the same dimensions as the data matrix. The exterior of the ROI is defined by zeros and the interior is any nonzero value.

/T Applies the thining algorithm of Zhang and Suen with the thin parameter. The wave M_MatrixFilter contains the results; the input wave is not overwritten.

Details

This operation does not support complex waves.

See Also

ImageFilter operation for additional options. **Matrix Math Operations** on page III-138 for more about Igor's matrix routines. The **Loess** operation.

References

Heckbert, Paul S., (Ed.), *Graphics Gems IV*, 575 pp., Morgan Kaufmann Publishers, 1994.

Zhang, T. Y., and C. Y. Suen, A fast thinning algorithm for thinning digital patterns, *Comm. of the ACM*, *27*, 236-239, 1984.

MatrixGaussJ

MatrixGaussJ *matrixA*, *vectorsB*

The MatrixGaussJ operation solves matrix expression A*x=b for column vector x given matrix A and column vector b. The operation can also be used to calculate the inverse of a matrix.

Parameters

matrixA is a NxN matrix of coefficients and *vectorsB* is a NxM set of right-hand side vectors.

Details

On output, the array of solution vectors *x* is placed in M_x and the inverse of *A* is placed in M_Inverse.

If the result is a singular matrix, V_flag is set to 1 to indicate the error. All other errors result in an alert, and abort any calling procedure.

All output objects are created in the current data folder.

An error is generated if the dimensioning of the input arrays is invalid.

This routine is provided for completeness only and is not recommended for general work (use LU decomposition — see **MatrixLUD**). MatrixGaussJ does calculate the inverse matrix but that is not generally needed either.

See Also

Matrix Math Operations on page III-138 for more about Igor's matrix routines. The **MatrixLUD** operation.

MatrixGLM

```
MatrixGLM  [/Z] matrixA, matrixB, waveD
```

The MatrixGLM operation solves the general Gauss-Markov Linear Model problem (GLM) which minimizes the 2-norm of a vector y

$$\min \|y\|_2 \quad \textit{subject to} \quad d = Ax + By.$$

A is *matrixA* (an NxM wave), B is *matrixB* (an NxP wave), and d is provided by *waveD* which is a 1D wave of N rows. The vectors x and y are the results of the calculation; they are stored in output waves Mat_X and Mat_Y in the current data folder.

Flags

/Z	In the event of an error, MatrixGLM will not return the error to Igor, which would cause procedure execute to abort. Your code should use the V_flag output variable to detect and handle errors.

Details

All input waves must have the same numeric type. Supported types are single-precision and double-precision floating point, both real and complex. The output waves Mat_X and Mat_Y have the same numeric type as the input.

The LAPACK algorithm assumes that M <= N <= M+P and

$$rank(A) = M,$$

$$rank(AB) = N.$$

Under these assumptions there is a unique solution x and a minimal 2-norm solution y, which are obtained using a generalized QR factorization of A and B. If the operation completes successfully the variable V_Flag is set to zero. Otherwise it contains a LAPACK error code.

Output Variables

V_flag	Set to 0 if MatrixGLM succeeds or to a LAPACK error code.

See Also

Matrix Math Operations on page III-138 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

MatrixInverse

`MatrixInverse` `[`***`flags`***`]` ***`srcWave`***

The MatrixInverse operation calculates the inverse or the pseudo-inverse of a matrix. *srcWave* may be real or complex.

MatrixInverse saves the result in the wave M_Inverse in the current data folder.

Flags

/D Creates the wave W_W that contains eigenvalues of the singular value decomposition (SVD) for the pseudo-inverse calculation. If one or more of the eigenvalues are small, the matrix may be close to singular.

/G Calculates only the direct inverse; does not affect calculation of pseudo-inverse. By default, it calculates the inverse of the matrix using LU decomposition. The inverse is calculated using Gauss-Jordan method. The only advantage in using Gauss-Jordan is that it is more likely to flag singular matrices than the LU method.

/O Overwrites the source with the result.

/P Calculates the pseudo-inverse of a matrix using the SVD algorithm. The calculated pseudo-inverse is a unique minimal solution to the problem:

$$\min_{\mathbf{X}\in\mathbb{R}^{n\times m}} \left\|\mathbf{AX} - I_m\right\|.$$

Examples

```
Make/O/N=(2,2) mat0 = {{2,3},{1,7}}
MatrixInverse mat0                    // Creates wave M_inverse
MatrixOp/O/T=1 mat1 = M_inverse x mat0  // Check result
```

```
Make/O/D/N=(4,6) mat1 = enoise(4)
MatrixInverse/P mat1
MatrixOP/O/T=1 aa = mat1 x M_Inverse
MatrixOP/O/P=1 avgAbsErr = sum(abs(mat1 x M_Inverse - identity(4)))/12
```

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-138 for more about Igor’s matrix routines.

References

See sec. 5.5.4 of:

Golub, G.H., and C.F. Van Loan, *Matrix Computations*, 2nd ed., Johns Hopkins University Press, 1986.

MatrixLinearSolve

`MatrixLinearSolve` `[`***`flags`***`]` ***`matrixA matrixB`***

The MatrixLinearSolve operation solves the linear system *matrixA* *X=*matrixB* where *matrixA* is an N-by-N matrix and *matrixB* is an N-by-NRHS matrix of the same data type.

Flags

/M=*method*	Determines the solution method which best suites input *matrixA*.	
	method=1:	Uses simple LU decomposition (default). See also LAPACK documentation for SGESV, CGESV, DGESV, and ZGESV. Creates the wave W_IPIV that contains the pivot indices that define the permutation matrix P. Row (i) if the matrix was interchanged with row ipiv(i).
	method=2:	If *matrixA* is band diagonal, you also have to specify /D. See also LAPACK documentation for SGBSV, CGBSV, DGBSV, and ZGBSV. Creates the wave W_IPIV, which contains the pivot indices that define the permutation matrix P. Row (i) if the matrix was interchanged with row ipiv(*i*). Also note that if you are using the /O flag, the overwritten waves may have a different dimensions.
	method=4:	For tridiagonal matrix; still expecting full matrix in *matrixA*, but it will ignore the data in the elements outside the 3 diagonals. See also LAPACK documentation for SGTSV, CGTSV, DGTSV, and ZGTSV.
	method=8:	Symmetric/hermitian. See also LAPACK documentation for SPOSV, CPOSV, DPOSV, and ZPOSV.
	method=16:	Complex symmetric (complex only). See also LAPACK documentation for CSYSV and ZSYSV.
/D={*sub*,*super*}	Specifies a band diagonal matrix. The subdiagonal (*sub*) and superdiagonal (*super*) size must be positive integers.	
/L	Uses the lower triangle of *matrixA*. /L and /U are mutually exclusive flags.	
/U	Uses the upper triangle of *matrixA*. /U is the default.	
/O	Overwrites *matrixA* and *matrixB* with the results of the operation. This will save on the amount of memory needed.	
/Z	No error reporting.	

Details

If /O is not specified, the operation also creates the n-by-n wave M_A and the n-by-nrhs solution wave M_B.

The variable V_flag is created by the operation. If the operation completes successfully, V_flag is set to zero, otherwise it is set to the LAPACK error code.

See Also

Matrix Math Operations on page III-138 for more about Igor’s matrix routines and for background references with details about the LAPACK libraries.

MatrixLinearSolveTD

`MatrixLinearSolveTD [/Z]` ***`upperW, mainW, lowerW, matrixB`***

The MatrixLinearSolveTD operation solves the linear system *TDMatrix**X = *matrixB*. In the matrix product on the left hand side, *TDMatrix* is a tridiagonal matrix with upper diagonal *upperW*, main diagonal *mainW*, and lower diagonal *lowerW*. It solves for vector(s) X depending on the number of columns (NRHS) in *matrixB*.

Flags

/DSTC=*solutionWave*

Specifies the output wave when all input waves are complex. If you omit /DSTC then the output wave is M_TDLinearSolution in the current data folder. /DSTC was added in Igor Pro 9.00.

/DSTR=*solutionWave*

Specifies the output wave when all input waves are real. If you omit /DSTR then the output wave is M_TDLinearSolution in the current data folder. /DSTR was added in Igor Pro 9.00.

/FREE Creates the solution wave as a free wave. /FREE is allowed only in functions and only if the solution wave, as specified by /DSTC or /DSTR, is a simple name or wave reference structure field. /DSTC was added in Igor Pro 9.00.

/Z No error reporting.

Details

The input waves can be single or double precision (real or complex). Results are returned in the wave M_TDLinearSolution in the current data folder. The wave *mainW* determines the size of the main diagonal (N). All other waves must match it in size with *upperW* and *mainW* containing one less point and *matrixB* consisting of N-by-NRHS elements of the same data type.

MatrixLinearSolveTD should be more efficient than MatrixLinearSolve with respect to storage requirements.

MatrixLinearSolveTD creates the variable V_flag, which is zero when it finishes successfully.

See Also

Matrix Math Operations on page III-138; the **MatrixLinearSolve** and **MatrixOp** operations.

MatrixLLS

MatrixLLS [**/O/Z/M=*method***] ***matrixA matrixB***

The MatrixLLS operation solves overdetermined or underdetermined linear systems involving MxN *matrixA*, using either QR/LQ or SV decompositions. Both *matrixA* and *matrixB* must have the same number type. Supported types are real or complex single precision and double precision numbers.

Flags

/M=*method* Specifies the decomposition method.

method=0: Decomposition is to QR or LQ (default). Creates the 2D wave M_A, which contains details of the QR/LQ factorization.

method=1: Singular value decomposition. Creates the 2D wave M_A, which contains the right singular vectors stored row-wise in the first `min(m,n)` rows. Creates the 1D wave M_SV, which contains the singular values of *matrixA* arranged in decreasing order.

/O Overwrites *matrixA* with its decomposition and *matrixB* with the solution vectors. This requires less memory.

/Z No error reporting.

Details

When the /O flag is not specified, the solution vectors are stored in the wave M_B, otherwise the solution vectors are stored in *matrixB*. Let *matrixA* be *m* rows by *n* columns and *matrixB* be an *m* by NRHS (if NRHS=1 it can be omitted). If $m \geq n$, MatrixLLS solves the least squares solution to an overdetermined system:

$$Minimize\left\|matrixB - matrixA \times \mathbf{X}\right\|.$$

Here the first *n* rows of M_B contain the least squares solution vectors while the remaining rows can be squared and summed to obtain the residual sum of the squares. If you are not interested in the residual you can resize the wave using, for example:

```
Redimension/N=(n,NRHS) M_B
```

If *m*<*n*, MatrixLLS finds the minimum norm solution of the underdetermined system:

$$matrixA \times \mathbf{X} = matrixB.$$

In this case, the first *m* rows of M_B contain the minimum norm solution vectors while the remaining rows can be squared and summed to obtain the residual sum of the squares for the solution. If you are not interested in the residual you can resize the wave using, for example:

```
Redimension/N=(m,NRHS) M_B
```

Note: Here *matrixB* consists of one or more column vectors B corresponding to one or more solution vectors X that are computed simultaneously. If *matrixB* consists of a single column, M_B is a 2D matrix wave that contains a single solution column.

The variable V_flag is set to 0 when there is no error; otherwise it contains the LAPACK error code.

Examples

```
// Construct matrixA as a 4x4 matrix
Make/O/D/N=(4,4) matrixA
matrixA[0][0]= {-54,-27,-9,-38}
matrixA[0][1]= {13,60,-42,-26}
matrixA[0][2]= {-80,17,44,-35}
matrixA[0][3]= {98,-8,-72,7}

// Construct matrixB as a single column (1D wave)
Make/O/D/N=4 matrixB={-20,77,-79,-33}

// Solve for matrixA x vectorX = matrixB
MatrixLLS matrixA, matrixB// Output stored in M_B

// Verify the solution
MatrixOP/O/P=1 aa=sum(abs(matrixA x col(M_B,0) - matrixB)) aa={9.947598300641403e-14}

// Compute multiple solutions at once
// Construct matrixB as two columns (2D wave)
Redimension/N=(4,2) matrixB
matrixB[0][0]= {-20,77,-79,-33}
matrixB[0][1]= {-94,61,29,68}

// Perform the calculation
MatrixLLS matrixA, matrixB

// Verify the solutions
MatrixOP/O/P=1 aa=sum(abs(matrixA x col(M_B,0) - col(matrixB,0)))
   aa={9.947598300641403e-14}
MatrixOP/O/P=1 aa=sum(abs(matrixA x col(M_B,1) - col(matrixB,1)))
   aa={1.989519660128281e-13}

// See if we have an overdetermined system ...
Make/D/N=(5,4) matrixA
matrixA[0][0] = {-7,-62,-47,-54,-80}
matrixA[0][1] = {43,15,-9,-94,57}
matrixA[0][2] = {22,61,-95,-95,51}
matrixA[0][3] = {-54,68,81,-51,54}
Make/O/D/N=(5) matrixB
matrixB[0] = {-28,-75,-74,35,86}

// Perform the calculation
MatrixLLS matrixA, matrixB

// Remove the extra rows from M_B
Redimension/N=4 M_B

// M_B is only the least squares solution so there is no point in attempting
// to verify the solution as in the example above.
```

See Also

Matrix Math Operations on page III-138 for more about Igor’s matrix routines and for background references with details about the LAPACK libraries.

MatrixLUBkSub

MatrixLUBkSub *matrtixL*, *matrixU*, *index*, *vectorB*

The MatrixLUBkSub operation provides back substitution for LU decomposition.

Details

This operation is used to solve the matrix equation Ax=b after you have performed LU decomposition (see **MatrixLUD**). Feed this routine M_Lower, M_Upper and W_LUPermutation from MatrixLUD along with your right-hand-side vector b. The solution vector x is returned as M_x. The array b can be a matrix containing a number of b vectors and the M_x will contain a corresponding set of solution vectors.

Generates an error if the dimensions of the input matrices are not appropriate.

See Also

Matrix Math Operations on page III-138 for more about Igor’s matrix routines.

MatrixLUD

MatrixLUD [***flags***] ***matrixA***

The MatrixLUD operation computes the LU factorization of a matrix. The general form of the factorization/decomposition is expressed in terms of matrix products:

```
M_Pt x srcWave = M_Lower x M_Upper
```

M_Pt, M_Lower and M_Upper are outputs created by MatrixLUD.

M_Pt is the transpose of the permutation matrix, M_Lower is a lower triangular matrix with 1's on the main diagonal and M_Upper is an upper triangular (or trapezoidal) matrix.

The MatrixLUD operation was substantially changed in Igor Pro 7.00. See the /B flag for information about backward compatibility.

Flags

/B This flag is provided for backward compatibility only; it is not compatible with any other flag. /B makes MatrixLUD behave as it did in Igor Pro 6. This flag is deprecated and will be removed in a future version of Igor.

The input is restricted to a 2D real valued, single or double precision square matrix. The outputs (all double precision) are stored in the waves M_Upper, M_Lower and W_LUPermutation in the current data folder.

The W_LUPermutation output wave was needed for solving a linear system of equations using the back substitution routine, **MatrixLUBkSub**. For better computation methods see **MatrixLinearSolve**, **MatrixLinearSolveTD** and **MatrixLLS**.

/CMF Uses Combined Matrix Format where the upper and lower matrix factors are combined into a single matrix saved in the wave M_LUFactors in the current data folder. The upper matrix factor is constructed from the main and from the upper diagonals of M_LUFactors. The lower matrix factor is constructed from the lower diagonals of M_LUFactors and setting the main diagonal to 1.

/MIND Finds the minimum magnitude diagonal element of M_Upper and store it in V_min. This is useful for investigating the behavior of the determinant of the matrix when it is close to being singular.

/PMAT Saves the transpose of the permutation matrix in a double precision wave M_Pt in the current data folder. Note that the permutation matrix is orthogonal and so the inverse of the matrix is equal to its transpose.

/SUMP Computes the sum of the phases of the elements on the main diagonal of M_Upper and store in the variable V_Sum. V_Sum is initialized to NaN and is set only if /SUMP is specified and M_Upper is complex.

Details

The input matrix *srcWave* is an MxN real or complex wave of single or double precision. Use **MatrixLUDTD** if your input is tri-diagonal.

The main results of the factorization are stored in the waves M_Lower, M_Upper and M_Pt. Alternatively the lower and upper factors can be combined and stored in the wave M_LUFactors (see /CMF). The waves M_Lower, M_Upper and M_LUFactors have the same data type as the input wave. M_Pt is always double precision.

When the input matrix *srcWave* is square (NxN), the resulting matrices have the same dimensions (NxN). You can reconstruct the input using the MatrixOp expression:

```
MatrixOp/O rA=(M_Pt^t) x (M_Lower x M_Upper)
```

If the input matrix is rectangular (NxM) the reconstruction depends on the size of N and M. If N<M:

```
MatrixOp/O rA=(M_Pt^t) x (subRange(M_lower,0,N-1,0,N-1) x M_Upper)
```

If N>M:

```
MatrixOp/O rA=(M_Pt^t) x M_lower x subRange(M_Upper,0,M-1,0,M-1)
```

The variable V_flag is set to zero if the operation succeeds and to 1 otherwise (e.g., if the input is singular). When you use the /B flag the polarity of the matrix is returned in the variable V_LUPolarity. The variables V_Sum and V_min are also set by some of the flag options above.

See Also

MatrixLUDTD, **MatrixLUBkSub**, **MatrixLinearSolve**, **MatrixLinearSolveTD**, **MatrixLLS**, **MatrixOp**

Matrix Math Operations on page III-138 for more about Igor’s matrix routines.

MatrixLUDTD

MatrixLUDTD [***flags***] ***srcMain***, ***srcUpper***, ***srcLower***

The MatrixLUDTD operation computes the LU factorization of a tri-diagonal matrix. The general form of the factorization/decomposition is expressed in terms of matrix products:

```
M_Pt x triDiagonalMat = M_Lower x M_Upper
```

`triDiagonalMat` is the matrix defined by the main diagonal specified by *srcMain*, the upper diagonal specified by *srcUpper*, and the lower diagonal specified by *srcLower*.

M_Pt is an output wave created when the /PMAT flag is present. M_Lower and M_Upper are output waves created when the /FM flag is present. M_Pt is the transpose of the permutation matrix, M_Lower is a lower triangular matrix with 1's on the main diagonal and M_Upper is an upper triangular (or trapezoidal) matrix.

Flags

/MIND Finds the minimum magnitude diagonal element of M_Upper and stores it in V_min. This feature is useful if you want to investigate the behaviour of the determinant of the matrix when it is close to being singular.

/PMAT Saves the transpose of the permutation matrix in the wave M_Pt in the current data folder. Note that the permutation matrix is orthogonal and so the inverse of the matrix is equal to its transpose.

/SUMP Computes the sum of the phases of the elements on the main diagonal of M_Upper and store in the variable V_Sum. Note that the variable is initialized to NaN and that it is not set unless this flag is specified and M_Upper is complex.

/FM The full matrix output is stored in the waves M_Lower and M_Upper in the current data folder.

Details

You specify the tridiagonal matrix using three 1D waves of the same data type (single or double precision real or complex).

If /FM is present the output of the operation consists of two 2D waves and one 1D wave:

M_Lower is a lower triangular matrix with 1's on the main diagonal.

M_Upper is an upper triangular (or trapezoidal) matrix.

W_PIV is 1D wave containing pivot indices.
See code example below for implementation details.

If /FM is omitted the output of the operation consists of five 1D waves:

W_Diagonal is the main diagonal of matrixU.

W_UDiagonal is the first upper diagonal of M_Upper.

W_U2Diagonal is the second diagonal of M_Upper.

W_LDiagonal is the first lower diagonal of M_Lower.

W_PIV is a vector of pivot indices.

In this case M_Lower can be constructed (see below) from W_LDiagonal and the pivot index wave W_PIV.

If you are working with tridiagonal matrices you can take advantage of MatrixOp functionality to reconstruct your outputs. For example:

```
MatrixOp/O M_Upper=Diagonal(W_diagonal)
MatrixOp/O M_Upper=setOffDiag(M_Upper,1,W_UDiagonal)
MatrixOp/O M_Upper=setOffDiag(M_Upper,2,W_U2Diagonal)
```

These commands can be combined into a single command line.

The construction of M_Lower is a bit more complicated and can be accomplished for real data using the following code:

```
Function MakeLTMatrix(W_diagonal,W_LDiagonal,W_PIV)
	Wave W_diagonal,W_LDiagonal,W_PIV

	Variable i,N=DimSize(W_diagonal,0)
	MatrixOp/O M_Lower=setOffDiag(ZeroMat(N,N,4),-1,W_LDiagonal)
	M_Lower=p==q ? 1:M_Lower[p][q]       // Set the main diagonal to 1's
	MatrixOp/O index=W_PIV-1             // Convert from 1-based array
	for(i=1;i<=N-2;i+=1)
		if(index[i]!=i)
			variable j,tmp
			for(j=0;j<=i-1;j+=1)
				tmp=M_Lower[i][j]
				M_Lower[i][j]=M_Lower[i+1][j]
				M_Lower[i+1][j]=tmp
			endfor
		endif
	endfor
End
```

This code is provided for illustration only. In practice you could use the /FM flag so that the operation creates the full lower and upper matrices for you.

The variable V_flag is set to zero if the operation succeeds and to 1 otherwise (e.g., if the input is singular). The variables V_Sum and V_min are also set by some of the flag options above.

See Also

MatrixLUD, **MatrixOp**, **Matrix Math Operations** on page III-138 for more about Igor’s matrix routines.

MatrixMultiply

`MatrixMultiply` ***`matrixA`*** `[`**`/T`**`],` ***`matrixB`*** `[`**`/T`**`] [,` ***`additional matrices`***`]`

The MatrixMultiply operation calculates matrix expression *matrixA***matrixB* and puts the result in a matrix wave named M_product generated in the current data folder. The /T flag can be included to indicate that the transpose of the specified matrix should be used.

If any of the source matrices are complex, then the result is complex.

Parameters

If *matrixA* is an NxP matrix then *matrixB* must be a PxM matrix and the product is an NxM matrix. Up to 10 matrices can be specified although it is unlikely you will need more than three. The inner dimensions must be the same. Multiplication is performed from right to left.

It is legal for M_product to be one of the input matrices. Thus `MatrixMultiply A,B,C` could also be done as:

```
MatrixMultiply B,C
MatrixMultiply A,M_product
```

Details

Supports multiplication of complex matrices.

An error is generated if the dimensioning of the input arrays is invalid.

See Also

MatrixOp and **MatrixMultiplyAdd** for more efficient matrix operations.

Matrix Math Operations on page III-138 for more about Igor's matrix routines.

FastOp for additional efficient non-matrix operations.

MatrixMultiplyAdd

MatrixMultiplyAdd [/ZC or /DC] [/A=*alpha*] [/B=*beta*] *matA*[/T], *matB*[/T] *matC*

The MatrixMultiplyAdd operation calculates the matrix expression:

```
matC = alpha*matA x matB + beta*matC
```

where * indicates scalar multiplication and x indicates matrix multiplication.

MatrixMultiplyAdd uses the LAPACK library for fast computation. It was added in Igor Pro 9.00.

Use /B=0 for just matrix multiply with no addition. In this case, the *beta***matC* term is not evaluated.

Parameters

matA, *matB* and *matC* must be of the same number type and must be single or double precision real or complex.

If *matA* is an NxP matrix then *matB* must be a PxM matrix and the product is an NxM matrix. If these conditions are violated MatrixMultiplyAdd generates an error.

Include the /T flag after *matA* and/or *matB* to indicate that the transpose of *matA* and/or *matB* should be used.

If *matC* is a NULL wave reference, then a free wave is created and a reference to it is stored in the wave reference *matC*.

Flags

/A=*alpha*	*alpha* is the scalar value multiplied with *matA*. It defaults to 1.
/B=*beta*	*beta* is the scalar value multiplied with *matB*. It defaults to 1. Use /B=0 to omit the *beta***matC* term.
/DC	Duplicates the wave referenced by *matC* as a free wave, performs the calculation with the duplicate as the destination, and stores a reference to the free wave in *matC*. /DC is allowed only when calling MatrixMultiplyAdd from a user-defined function. It is an error to use /DC from the command line or in a macro.
/T	Used after *matA* and/or *matB*, /T indicates tells MatrixMultiplyAdd to use the transform of *matA* and/or *matB* in the calculation.
/ZC	Clears the input wave reference *matC*. This guarantees that *matC* is NULL which causes MatrixMultiplyAdd to create a free output wave and store a reference to it in *matC*. When using MatrixMultiplyAdd in a loop, use /ZC to clear the input wave reference to ensure that a new free wave is created each time through the loop rather than re-using the same output wave. /ZC is allowed only when calling MatrixMultiplyAdd from a user-defined function. It is an error to use /ZC from the command line or in a macro.

Details

matA and *matB* must exist but, when running in a user-defined function, *matC* may or may not exist. If it does not exist, MatrixMultiplyAdd creates a free output wave, creates a wave reference named *matC*, and stores a reference to the free output wave in *matC*.

Both *alpha* and *beta* must be real and default to 1.

Use /B=0 to omit the *beta***matC* term.

Example

```
Function Demo()     // Demonstrates various ways to call MatrixMultiplyAdd
    // Create input waves
    Make/O/N=(3,3) matA = p + 10*q
    Make/O/N=(3,3) matB = p == q

    Print "=== matC = matA x matB ==="
    Make/O/N=(3,3) matC = 0
    MatrixMultiplyAdd /B=0 matA, matB, matC
    Print matC

    Print "=== matC = 2*matA x matB ==="
    Make/O/N=(3,3) matC = 0
    MatrixMultiplyAdd /A=2 /B=0 matA, matB, matC
    Print matC

    Print "=== Free Wave Using WaveClear = matA x matB ==="
    Make/O/N=(3,3) matC = 0          // matC wave ref is cleared by WaveClear
    WaveClear matC                   // MatrixMultiplyAdd creates a free output wave
    MatrixMultiplyAdd /B=0 matA, matB, matC
    Print matC

    Print "=== Free Wave Using /ZC = matA x matB ==="
    Make/O/N=(3,3) matC = 0          // matC wave ref is cleared by /ZC
    MatrixMultiplyAdd /B=0 /ZC matA, matB, matC
    Print matC

    Print "=== Create dest wave using string variable ==="
    String dest = "matDest1"
    Make/O/N=(3,3) $dest = 0
    WAVE destWave = $dest
    MatrixMultiplyAdd /B=0 matA, matB, destWave
    Print destWave
End
```

See Also

MatrixOp and **MatrixMultiply** for more efficient matrix operations.

Matrix Math Operations on page III-138 for more about Igor's matrix routines.

FastOp for additional efficient non-matrix operations.

MatrixOp

MatrixOp [/C /FREE /NTHR=*n* /O /S] *destwave = expression*

The MatrixOp operation evaluates *expression* and stores the result in *destWave*.

expression may include literal numbers, numeric variables, numeric waves, and the set of operators and functions described below. MatrixOp does not support text waves, strings or structures.

MatrixOp is faster and in some case more readable than standard Igor waveform assignments and matrix operations.

See **Using MatrixOp** on page III-140 for an introduction to MatrixOp.

Parameters

destWave Specifies a destination wave for the assignment expression. *destWave* is created at runtime. If it already exists, you must use the /O flag to overwrite it or the operation returns an error.

When the operation is completed, *destWave* has the dimensions and data type implied by *expression*. In particular, it may be complex if *expression* evaluates to a complex quantity. If *expression* evaluates to a scalar, *destWave* is a 1x1 wave.

If you include the /FREE flag then *destWave* is created as a free wave.

By default the data type of *destWave* depends on the data types of the operands and the nature of the operations on the right-hand side of the assignment. If *expression* references integer waves only, *destWave* may be an integer wave too but most operations with a scalars convert *destWave* into a double precision wave. See **MatrixOp Data Promotion Policy** on page III-145 for further discussion.

Even if *destWave* exists before the operation, MatrixOp may change its data type and dimensionality as implied by *expression*.

You can force the number type using MatrixOp functions such as `uint16` and `fp32`.

expression *expression* is a mathematical expression referencing waves, local variables, global variables and literal numbers together with MatrixOp functions and MatrixOp operators as listed in the following sections.

You can use any combination of data types for operands.In particular, you can mix real and complex types in *expression*. MatrixOp determines data types of inputs and the appropriate output data type at runtime without regard to any type declaration such as `Wave/C`.

Operators

`+` Addition between scalars, matrix addition or addition of a scalar (real or complex) to each element of a matrix.

`-` Subtraction of one scalar from another, matrix subtraction, or subtracting a scalar from each element of a matrix. Subtraction of a matrix from a scalar is not defined.

`*` Multiplication between two scalars, multiplication of a matrix by a scalar, or element-by-element multiplication of two waves of the same dimensions.

`/` Division of two scalars, division of a matrix by a scalar, or element-by-element division between two waves of the same dimensions.

Division of a scalar by a matrix is not supported but you can use the `rec` function with multiplication instead.

`x` Matrix multiplication (lower case x symbol only).

This operator *must* be preceded and followed by a space. Matrix multiplication requires that the number of columns in the matrix on the left side be equal to the number of rows in the matrix on the right.

`.` Generalized form of a dot product. In an expression *a*.*b* it is expected that *a* and *b* have the same number of points although they may be of arbitrary numeric type. The operator returns the sum of the products of the sequential elements as if both *a* and *b* were 1D arrays.

For complex binary operand waves a and b, this operator returns the MatrixOp equivalent of sum(*a**conj(*b*)). The **MatrixDot** function returns sum(*b**conj(*a*)).

`^t` Matrix transpose. This is a postfix operator meaning that ^t appears after the name of a matrix wave.

`^h` Hermitian transpose. This is a postfix operator meaning that ^h appears after the name of a matrix wave.

`&&` Logical AND operator supports all real data types and results in a signed byte numeric token with the value of either 0 or 1. The operation acts on an element by element basis and is performed for each element of the operand waves.

`||` Logical OR operator supports all real data types and results in a signed byte numeric token with the value of either 0 or 1. The operation acts on an element by element basis and is performed for each element of the operand waves.

MatrixOp does not support operator combinations such as +=.

This table shows the precedence of MatrixOp operators:

MatrixOp Operator		Precedence
`^h`	`^t`	**Highest**
`x`	`.`	
`*`	`/`	
`+`	`-`	
`&&`	`||`	**Lowest**

You can use parentheses to force evaluation order.

Operators that have the same precedence associate from right to left. This means that `a* b / c` is equivalent to `a * (b / c)`.

Functions

These functions are available for use with MatrixOp.

`abs(w)` Absolute value of a real number or the magnitude of a complex number.

acos(*w*)
Arc cosine of *w*.

acosh(*w*)
Inverse hyperbolic cosine of *w*. Added in Igor Pro 7.00.

addCols(*w*,*dc*)
Returns a matrix where elements of the 1D wave *dc* are added to the corresponding column in *w*:

```
out = w + dc[q]
```

Added in Igor Pro 9.00.

addRows(*w*,*dr*)
Returns a matrix where elements of the 1D wave *dr* are added to the corresponding row in *w*:

```
out = w + dr[p]
```

Added in Igor Pro 9.00.

asin(*w*)
Arc sine of *w*.

asinh(*w*)
Inverse hyperbolic sine of *w*. Added in Igor Pro 7.00.

asyncCorrelation(*w*)
Asynchronous spectrum correlation matrix for a real valued input matrix wave *w*. See syncCorrelation for details.

atan(*w*)
Arc tangent (inverse tangent) of *w*.

atan2(*y*,*x*)
Arc tangent (inverse tangent) of real *y*/*x*.

atanh(*w*)
Inverse hyperbolic tangent of *w*. Added in Igor Pro 7.00.

averageCols(*w*)
Returns a (1xcolumns) wave containing the averages of the columns of matrix *w*. This is equivalent to sumCols(*w*)/numRows(*w*).

Added in Igor Pro 7.00.

axisToQuat(*ax*)
See **Functions Using Quaternions** on page V-573.

backwardSub(*U*,*c*)
Returns a column vector solution for the matrix equation Ux=c, where U is an (NxN) wave representing an upper triangular matrix and c is a column vector of N rows. If c has additional columns they are ignored. Ideally, U and c should be either SP or DP waves (real or complex). Other numeric data types are supported with a slight performance penalty.

This function is typically used in solving linear equations following a matrix decomposition into lower and upper triangular matrices (e.g., Cholesky), with an expression of the form:

```
MatrixOp/O solVector=backwardSub(U,forwardSub(L,b))
```

where U and L are the upper and lower triangular factors.

Added in Igor Pro 7.00.

beam(*w*,*row*,*col*)
When *w* is a 3D wave the beam function returns a 1D array corresponding to the data in the beam defined by w[*row*][*col*][]. In other words, it returns a 1D array consisting of all elements in the specified row and column from all layers. See also **ImageTransform** getBeam.

When *w* is a 4D wave it returns a 2D array containing w[*row*][*col*][][]. In other words, it returns a matrix consisting of all elements in the specified row and column from all layers and all chunks.

The beam function belongs to a special class in that it does not operate on a layer by layer basis. It therefore does not permit compound expressions in place of any of its parameters.

The beam function has the highest precedence.

`bitAnd(w1,w2)`

Returns an array of the same dimensions and number type as *w1* where each element corresponds to the bitwise AND operation between *w1* and *w2*.

w2 can be either a single number or a matrix of the same dimensions as *w1*. Both *w1* and *w2* must be real integer numeric types.

Added in Igor Pro 7.00.

`bitOr(w1,w2)`

Returns an array of the same dimensions and number type as *w1* where each element corresponds to the bitwise OR operation between *w1* and *w2*.

w2 can be either a single number or a matrix of the same dimensions as *w1*. Both *w1* and *w2* must be real integer numeric types.

Added in Igor Pro 7.00.

`bitReverseCol(w,c,order)`

Returns the matrix *w* with the entries of column *c* reordered.

bitReverseCol was added in Igor Pro 9.00.

order is one of the following values:

0:	Direct binary reversal of the index
1:	Hadamard order
2:	Dyadic/Paley/Gray order
3:	Unchanged order

bitReverseCol can be used for changing the order of entries in fast transform algorithms such as FFT and the fast Walsh-Hadamard transform. The number of rows in w must be a power of 2.

See ***Examples*** below for an example using bitReverseCol.

`bitShift(w1,w2)`

Returns an array of the same dimensions and number type as *w1* shifted by the amount specified in *w2*. When *w2* is positive the shifting is to the left. When it is negative the shifting is to the right.

w2 can be either a single number or a matrix of the same dimensions as *w1*. Both *w1* and w2 must be real integer numeric types.

Added in Igor Pro 7.00.

`bitXor(w1,w2)`

Returns an array of the same dimensions and number type as *w1* where each element corresponds to the bitwise XOR operation between *w1* and *w2*.

w2 can be either a single number or a matrix of the same dimensions as *w1*. Both *w1* and *w2* must be real integer numeric types.

Added in Igor Pro 7.00.

`bitNot(w)`

Returns an array of the same dimensions and number type as *w* where each each element is the bitwise complement of the corresponding element in `w`.

Added in Igor Pro 7.00.

`catCols(w1,w2)`

Concatenates the columns of *w2* to those of *w1*. *w1* and *w2* must have the same number of rows and the same number type.

Added in Igor Pro 7.00.

`catRows(w1,w2)`

Concatenates the rows of *w2* to those of *w1*. *w1* and *w2* must have the same number of columns and the same number type.

Added in Igor Pro 7.00.

`cbrt(w)`

Returns the cube root of the elements of *w*. When *w* is complex cbrt returns the principal cube root (the root with a positive imaginary number).

Added in Igor Pro 8.00.

`ceil(z)` Smallest integer larger than *z*.

`chirpZ(data,A,W,M)` Chirp Z Transform of the 1D wave data calculated for the contour defined by

$$z_k = AW^{-k},$$
$$k = 0,1,...M-1.$$

Here both *A* and *W* are complex and the standard z transform for a sequence {x(n)} is defined by

$$X(z) = \sum_{k=0}^{N-1} x(n) z^{-k}.$$

The phase of the output is inverted to match the result of the ChirpZ transform on the unit circle with that of the FFT.

`chirpZf(data,f1,f2,df)`

Chirp Z Transform except that the transform parameters are specified by real-valued starting frequency *f1*, end frequency *f2* and frequency resolution *df*. The transform is confined to the unit circle because both *A* and *W* have unit magnitude.

`chol(w)` Returns the Cholesky decomposition U of a positive definite symmetric matrix w such that w=U^t x U. Note that only the upper triangle of w is actually used in the computation.

`chunk(w,n)` Returns chunk *n* from 4D wave *w*.

Added in Igor Pro 7.00.

`clip(w,low,high)` Returns the values in the wave *w* clipped between the *low* and the *high* parameters. If *w* contains NaN or INF values, they are not modified. The result retains the same number type as the input wave *w* irrespective of the range of the *low* and *high* input parameters.

`cmplx(re,im)` Returns a complex token from two real tokens. *re* and *im* must have the same dimensionality.

Added in Igor Pro 7.00.

`col(w,c)` Returns column *c* from matrix wave *w*.

`colRepeat(w,n)` Returns a matrix that consists of *n* identical columns containing the data in the wave *w*. If *w* is a 2D wave, it is treated as if it were a single column containing all the data. Higher dimensions are supported on a layer-by-layer basis. MatrixOp returns an error if n<2.

Added in Igor Pro 7.00.

`conj(matrixWave)` Complex conjugate of the input expression.

`const(r,c,val)` Returns an (*r* x *c*) matrix where all elements are equal to *val*. The data type of the returned matrix is the same as that of *val*. See also `zeroMat` below.

Added in Igor Pro 7.00.

`convolve(`*w1*`,`*w2*`,`*opt*`)` Convolution of *w1* with *w2* subject to options opt. The dimensions of the result are determined by the largest dimensions of *w1* and *w2* with the number of rows padded (if necessary) so that they are even. Supported options include *opt*=0 for circular convolution and *opt*=4 for acausal convolution.

For fast 2D convolutions where where *w1* is an image and *w2* is a square kernel of the same numeric type, you can use *opt*=-1 or *opt*=-2. When *opt*=-1 the convolution at the boundaries is evaluated using zero padding. When *opt*=-2 the padding is a reflection of *w1* about the boundaries. When working with integer waves the kernel is internally normalized by the sum of its elements. The kernel for floating point waves remain unchanged.

To convolve an image in *w1* with a smaller point spread function in *w2* you can use:

opt=-1 if you want to pad the image boundaries with zeros or

opt=-2 if you want to pad the boundaries by reflecting image values about each boundary.

The negative options are designed for a very optimized convolution calculation which requires that *w1* and *w2* have the same numeric type. If the size of the point spread function is larger than about 13x13 it may become more efficient to compute the convolution using the positive options.

`correlate(`*w1*`,`*w2*`,`*opt*`)` Correlation of *w1* with *w2* subject to options *opt*. The dimensions of the result are determined by the largest dimensions of *w1* and *w2* with the number of rows padded (if necessary) so that they are even. Supported options include *opt*=0 for circular correlation and *opt*=4 for acausal correlation.

`cos(`*w*`)` Cosine of *w.*

`cosh(`*w*`)` Hyperbolic cosine of *w*. Added in Igor Pro 7.00.

`covariance(`*w*`)` Returns the sample covariance matrix. Added in Igor Pro 8.00.

If *w* is a rows x cols real matrix then the returned value is the cols x cols matrix Q with elements

$$q_{ij} = \frac{1}{rows-1}\sum_{i=0}^{rows-1}\left(w_{ij}-\bar{w}_j\right)\left(w_{ik}-\bar{w}_k\right),$$

where

$$\bar{w}_k = \frac{1}{rows}\sum_{i=0}^{rows-1} w_{ik}.$$

`crossCovar(`*w1*`,`*w2*`,`*opt*`)` Returns the cross-covariance for 1D waves *w1* and *w2*. The options parameter opt can be set to 0 for the raw cross-covariance or to 1 if you want the results to be normalized to 1 at zero offset. If *w1* has N rows and *w2* has M rows then the returned vector is of length N+M-1. The cross-covariance is computed by subtracting the mean of each input followed by correlation and optional normalization. See also **Correlate** with the /NODC flag.

`decimateMinMax(w,N)`

Decimates the matrix *w* on a column by column basis. Each column is divided into *N* bins and each bin is represented by its minimum and maximum values.

For an RxC real valued input *w*, the output is an array of dimensions 2*N*xC and the extrema are ordered as in {min0,max0,min1,max1...}.

Because of roundoff, sequential bins may not represent the same number of elements when R is not an integer multiple of *N*. In this case you not count on the exact form of the bin size roundoff.

The decimateMinMax keyword was added in Igor Pro 9.00.

`det(w)`

Returns a scalar corresponding to the determinant of matrix *w*, which must be real.

`diagonal(w)`

Creates a square matrix that has the same number of rows as *w*. All elements are zero except for the diagonal elements whichare taken from the first column in *w*. Use DiagRC if the input is not an existing wave (such as a result from another function).

`diagRC(w,rows,cols)`

2D matrix of dimensions *rows* by *cols*. All matrix elements are set to zero except those of the diagonal which are filled sequentially from elements of *w*. The dimensionality of *w* is unimportant. If the total number of elements in *w* is less than the number of elements on the diagonal then all elements will be used and the remaining diagonal elements will be set to zero.

`e`

Returns the base of the natural logarithm.

`equal(w1,w2)`

Returns unsigned byte result with 1 for equality and zero otherwise. The dimensionality of the result matches the dimensionality of the largest parameter.

Either or both *w1* and *w2* can be constants (i.e., one row by one column).

If *w1* and *w2* are not constants, they must have the same number of rows and columns. If w1 or w2 have multiple layers, they must either have the same number of layers or one of them must have a single layer.

Both parameters can be either real or complex. A comparison of a real with a complex parameter returns zero.

`erf(w)`

Returns the error function (see **erf**) for real values in *w*.

Added in Igor Pro 7.00.

`erfc(w)`

Returns the complementary error function (see **erfc**) for real values in *w*.

Added in Igor Pro 7.00.

`exp(w)`

Exponential function for *w* which can be real or complex, scalar or a matrix.

`expIntegralE1(w)`

Returns the exponential integral for real arguments *w*. See also **ExpIntegralE1**. Added in Igor Pro 9.00.

`expm(w)`

Returns the matrix exponential of a real-valued square matrix *w*. Added in Igor Pro 9.00.

`FCT(w,dir)`

See **Trignometric Transform Functions** on page V-571.

`fft(w,options)`

FFT of *w*.

w must have an even number of rows.

options contains a binary field flag. Set bit 1 to 1 if you want to disable the zero centering (see /Z flag in the **FFT** operation). Other bits are reserved.

MatrixOp does not support wave scaling and therefore it does not produce the same wave scaling changes as the **FFT** operation.

`floor(`*`w`*`)`

Largest integer smaller than *w*. If *w* is complex the function is separatly applied to the real and imaginary parts.

`forwardSub(`*`L`*`,`*`b`*`)`

Returns a column vector solution for the matrix equation Lx=b, where *L* is an (NxN) wave representing a lower triangular matrix and *b* is a column vector of N rows. If *b* has additional columns they are ignored.

Ideally, *L* and *b* should be either SP or DP waves (real or complex). Other numeric data types are supported with a slight performance penalty.

This function is typically used in solving linear equations following a matrix decomposition into lower and upper triangular matrices (e.g., Cholesky), with an expression of the form:

```
MatrixOp/O solVector=backwardSub(U,forwardSub(L,b))
```

where U and L are the upper and lower triangular factors.

Added in Igor Pro 7.00.

`fp32(`*`w`*`)`

Converts *w* to 32-bit single precision floating point representation. See also the /NPRM flag below for more information.

Added in Igor Pro 7.00.

`fp64(`*`w`*`)`

Converts *w* to 64-bit single precision floating point representation. See also the /NPRM flag below for more information.

Added in Igor Pro 7.00.

`Frobenius(`*`w`*`)`

Returns the Frobenius norm of a matrix defined as the square root of the sum of the squared absolute values of all elements.

`FST(`*`w`*`,`*`dir`*`)`

See **Trignometric Transform Functions** on page V-571.

`FSST(`*`w`*`,`*`dir`*`)`

See **Trignometric Transform Functions** on page V-571.

`FSCT(`*`w`*`,`*`dir`*`)`

See **Trignometric Transform Functions** on page V-571.

`FSST2(`*`w`*`,`*`dir`*`)`

See **Trignometric Transform Functions** on page V-571.

`FSCT2(`*`w`*`,`*`dir`*`)`

See **Trignometric Transform Functions** on page V-571.

`gamma(`*`w`*`)`

Returns the gamma function for real arguments *w*. See also **gamma**. Added in Igor Pro 9.00.

`gammaln(`*`w`*`)`

Returns the natural log of the gamma function for real arguments *w*. See also **gammln**. Added in Igor Pro 9.00.

`getDiag(`*`w2d`*`,`*`d`*`)`

Returns a 1D wave that contains diagonal *d* of *w2d*. *d*=0 is the main diagonal, *d*>0 correspond to upper diagonals and *d*<0 to lower diagonals.

Added in Igor Pro 7.00.

`greater(`*`a`*`,`*`b`*`)`

Returns an unsigned byte for the truth of *a* > *b*. Both *a* and *b* must be real but one or both can be constants (see `equal()` above). The dimensionality of the result matches the dimensionality of the largest parameter.

`greaterOrEqual(`*`a`*`,`*`b`*`)`

Returns an unsigned byte for the truth of *a* >= *b*. Both *a* and *b* must be real but one or both can be constants (see `equal()` above). The dimensionality of the result matches the dimensionality of the largest parameter.

Added in Igor Pro 9.00.

`hypot(`*`w1`*`,`*`w2`*`)`

Returns the square root of the sum of the squares of *w1* and *w2*.

Added in Igor Pro 7.00.

`ifft(w,options)`

IFFT of *w*.

options is a bitwise parameter defined as follows:

Bit 0: Forces the result to be real, like the **IFFT** operation /C flag.
Bit 1: Disables center-zero.
Bit 2: Swaps the results.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

MatrixOp does not support wave scaling and therefore it does not produce the same wave scaling changes as the **IFFT** operation.

`identity(n,m)`
`identity(n)`

Creates a computational object that is an identity matrix. If you use a single argument *n,* the identity created is an (*n*x*n)* square matrix with 1's for diagonal elements (the remaining elements are set to zero). If you use both arguments, the function creates an (*n*x*m*) zero matrix and fills its diagonal elements with 1's. Note that the identity is created at runtime and persists only for the purpose of the specific operation.

`imag(w)`

Returns the imaginary part of *w*.

`indexCols(w)`

Returns a token of the same dimensions as *w* where each element is equal to its column index. This is the MatrixOp equivalent of q used on righthand side of a standard wave assignment statement. The returned token is unsigned 32-bit integer if the number of columns in *w* is less than 2^32 or double-precision floating point otherwise.

Added in Igor Pro 8.00.

`indexRows(w)`

Returns a token of the same dimensions as *w* where each element is equal to its row index. This is the MatrixOp equivalent of p used on righthand side of a standard wave assignment statement. The returned token is unsigned 32-bit integer if the number of rows in *w* is less than 2^32 or double-precision floating point otherwise.

Added in Igor Pro 8.00.

`inf()`

Returns INF.

Added in Igor Pro 7.00.

`insertMat(s,d,r,c)`

Inserts matrix *s* into matrix *d* starting at row r and column c. The waves *s* and *d* must be of the same numeric data type. The inserted range is clipped to the dimensions of the wave *d*. Both *r* and *c* must be non-negative.

Added in Igor Pro 7.00.

`int8(w)`

Converts *w* to 8-bit signed integer representation. See also the /NPRM flag below for more information.

Added in Igor Pro 7.00.

`int16(w)`

Converts *w* to 16-bit signed integer representation. See also the /NPRM flag below for more information.

Added in Igor Pro 7.00.

`int32(w)`

Converts *w* to 32-bit signed integer representation. See also the /NPRM flag below for more information.

Added in Igor Pro 7.00.

`integrate(w, opt)`

Returns a running sum of `w`.

If *opt*=0 the sum runs over the entire input wave treating the columns of 2D waves as if they were part of one big column.

If *opt*=1 the running sum is computed separately for each column of a 2D wave.

Added in Igor Pro 7.00.

`intMatrix(w)`

Returns a double-precision matrix of the same dimensions as *w*.

Each element of the returned matrix is the sum of all the corresponding elements of *w* that are above and to the left of it, i.e.,

$$out_{ij} = \sum_{m=0}^{i} \sum_{n=0}^{j} w_{mn}.$$

The utility of this function is apparent in the following relationship:

$$\sum_{i=x_1}^{x_2} \sum_{j=y_1}^{y_2} w_{ij} = out[x_2, y_2] - out[x_2, y_1 - 1] - out[x_1 - 1, y_2] + out[x_1 - 1, y_1 - 1].$$

intMatrix was added in Igor Pro 7.00.

`inv(w)`

Returns the inverse of the square matrix *w*.

If w is not invertible, the operation returns a matrix of the same dimensions where all elements are set to NaN.

`inverseErf(w)`

Returns the inverse error function (see **inverseErf**) for the real values in *w*.

Added in Igor Pro 7.00.

`inverseErfc(w)`

Returns the inverse complementary error function (see **inverseErfc**) for the real values in *w*.

Added in Igor Pro 7.00.

`kronProd(u, v)`

Returns the Kronecker product of matrices *u* and *v*.

kronProd was added in Igor Pro 9.00.

If *u* has dimensions M x N and *v* has dimensions P x Q then the returned block matrix has dimensions M*P x N*Q. It is given by

$$u \otimes v = \begin{bmatrix}
u_{11}v_{11} & u_{11}v_{112} & \cdots & u_{11}v_{1q} & \cdots & \cdots & u_{1n}v_{11} & u_{1n}v_{12} & \cdots & u_{1n}v_{1q} \\
u_{11}v_{21} & u_{11}v_{22} & \cdots & u_{11}v_{2q} & \cdots & \cdots & u_{1n}v_{21} & u_{1n}v_{22} & \cdots & u_{1n}v_{2q} \\
\vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\
u_{11}v_{p1} & u_{11}v_{p2} & \cdots & u_{11}v_{pq} & \cdots & \cdots & u_{1n}v_{p1} & u_{1n}v_{p2} & \cdots & u_{1n}v_{pq} \\
\vdots & \vdots & & \vdots & & & \vdots & \vdots & & \vdots \\
\vdots & \vdots & & \vdots & & & \vdots & \vdots & & \vdots \\
u_{m1}v_{11} & u_{m1}v_{12} & \cdots & u_{m1}v_{1q} & \cdots & \cdots & u_{mn}v_{11} & u_{mn}v_{12} & \cdots & u_{mn}v_{1q} \\
u_{m1}v_{21} & u_{m1}v_{22} & \cdots & u_{m1}v_{2q} & \cdots & \cdots & u_{mn}v_{21} & u_{mn}v_{22} & \cdots & u_{mn}v_{2q} \\
\vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & & \vdots \\
u_{m1}v_{p1} & u_{m1}v_{p2} & \cdots & u_{m1}v_{pq} & & & u_{mn}v_{p1} & u_{mn}v_{p2} & & u_{mn}v_{pq}
\end{bmatrix}$$

See ***Examples*** below for an example using kronProd.

`layer(`*w*`,`*n*`,`*chunk*`)`

Returns a layer of a 3D or 4D wave.

The layer function was added in Igor Pro 7.00.

The optional *chunk* parameter was added in Igor Pro 9.00.

If you omit *chunk* and *w* is a 3D wave, the layer function returns layer *n* from *w*.

If you omit *chunk* and *w* is a 4D wave, the layer function returns layer *n* of chunk 0 from *w*.

If you include *chunk*, *w* must be a 4D wave and the layer function returns layer *n* of the specified chunk from *w*.

In all cases *w* must be a an actual wave and not an expression.

`layerStack(`*w*`,`*n*`)`

The layerStack function takes a 4D input wave w and a layer number n and returns a 3D wave (stack) that contains sequentially the data in layer n of each chunk of w. w must be a wave and not a derived token.

layerStack was added in Igor Pro 9.00.

Example:

```
// Create 3D stack from red channel of 4D RGB movie wave
MatrixOP/O redChannelStack=layerStack(myRGBMovie,0)
```

`limitProduct(`*w1*`,`*w2*`)`

Returns a partial element-by-element multiplication of waves *w1* and *w2*.

It is assumed that the dimensions of *w1* are greater or equal to that of *w2*. If *w2* is of dimensions NxM then the function returns a matrix of the same dimensions as *w1* with the first NxM elements contain the product of the corresponding elements in *w1* and *w2* and the remaining elements set to zero.

The function is designed to be used in filtering applications where the size of the kernel *w2* is much smaller than the size of the input *w1*.

Added in Igor Pro 7.00.

`log(`*w*`)`

Returns the log base 10 of a token *w* which can be real or complex, scalar or a matrix.

`log2(`*w*`)`

Returns the log base 2 of a token *w* which can be real or complex, scalar or a matrix. log2 was added in Igor Pro 9.00.

`ln(`*w*`)`

Returns the natural logarithm of a token *w2* which can be real or complex, scalar or a matrix.

`mag(`*w*`)`

Returns a real valued wave containing the magnitude of each element of *w*. This is equivalent to the `abs` function.

`magSqr(`*w*`)`

Returns a real value wave containing the square of a real *w* or the squared magnitude of complex *w*.

`maxAB(`*a*`,`*b*`)`

Returns the larger of the two real numbers *a* and *b*.

`maxAB` does not support NaN or complex inputs.

Added in Igor Pro 7.00.

`maxMagAB(`*a*`,`*b*`)`

For each data point in the tokens *a* and *b*, maxMagAB returns the one with the larger absolute value.

maxMagAB does not support complex numbers.

Added in Igor Pro 9.00.

`maxCols(`*w*`)`

Returns a 1D wave containing the maximum value of each column in the wave *w*. If *w* is complex the output contains the maximum magnitude of each column of *w*.

`maxCols` does not support NaN.

Added in Igor Pro 7.00.

`maxRows(`*w*`)`

Returns a 1D wave containing the maximum value of each row in the wave *w*. If *w* is complex the output contains the maximum magnitude of each row of *w*.

`maxRows` does not support NaN.

Added in Igor Pro 8.00.

`maxVal(`*w*`)`

Returns the maximum value of the wave *w*. If *w* is complex it returns the maximum magnitude of *w*.

When *w* is real, the returned number type is the same as *w*'s.

When *w* is complex, the result is real and represents the maximum of the magnitudes of the elements of *w*. If *w* is double precision, then the result is double precision; otherwise it is single precision.

When *w* is a 3D or 4D wave, `maxVal` returns a (1 x 1 x layers x chunks) data token.

`maxVal` does not support NaN values.

`mean(`*w*`)`

Returns the mean value *w*.

`minAB(`*a*`,`*b*`)`

Returns the smaller of the two real numbers *a* and *b*.

`minAB` does not support NaN or complex inputs.

Added in Igor Pro 7.00.

`minMagAB(`*a*`,`*b*`)`

For each data point in the tokens *a* and *b*, minMagAB returns the one with the smaller absolute value.

minMagAB does not support complex numbers.

Added in Igor Pro 9.00.

`minCols(`*w*`)`

Returns a 1D wave containing the minimum value of each column in the wave *w*. If *w* is complex the output contains the minimum magnitude of each column of *w*.

`minCols` does not support NaN.

Added in Igor Pro 8.00.

`minRows(`*w*`)`

Returns a 1D wave containing the minimum value of each row in the wave *w*. If *w* is complex the output contains the minimum magnitude of each row of *w*.

`minRows` does not support NaN.

Added in Igor Pro 8.00.

`minVal(`*w*`)`

Returns the minimum value of the wave *w*. If *w* is complex the function returns the minimum magnitude of *w*.

When *w* is real, the returned number type is the same as *w*'s.

When *w* is complex, the result is real and represents the minimum of the magnitudes of the elements of *w*. If *w* is double precision, then the result is double precision; otherwise it is single precision.

When *w* is a 3D or 4D wave, `minVal` returns a (1 x 1 x layers x chunks) data token.

`minVal` does not support NaN values.

`mod(w,b)`
Returns the remainder after dividing *w* by *b*. *b* can be a scalar or a matrix of the same dimensions as *w*.

Added in Igor Pro 7.00.

`nan()`
Returns NaN.

Added in Igor Pro 7.00.

`normalize(w)`
Normalized version of a vector or a matrix. Normalization is such that the returned token should have a unity magnitude except if all elements are zero, in which case output is unchanged.

`normalizeCols(w)`
Divides each column of the real wave *w* by the square root of the sum of the squares of all elements of the column.

`normalizeRows(w)`
Divides each row of the real wave *w* by the square root of the sum of the squares of all the elements in that row.

`normP(w,pn)`
Returns the p-norm of matrix *w* defined by

$$\left\|W\right\|_p = \left(\sum_{j=0}^{cols}\sum_{i=0}^{rows}\left|w[i][j]\right|^p\right)^{1/p}.$$

The case of *pn*=2 is equivalent to the MatrixOp Frobenius function.

Added in Igor Pro 9.00.

`numCols(w)`
Returns the number of columns in the wave *w*. When *w* is 1D the function returns 1.

`numPoints(w)`
Returns the number of points in a layer of *w*.

`numRows(w)`
Returns the number of rows in *w*.

`numType(w)`
Number the number type of *w*:

0: *w* is a normal number

1: *w* is +/-INF

2: *w* is NaN

`oneNorm(w)`
Returns the 1-norm of matrix *w* defined as the maximum absolute column sum

$$\left\|W\right\| = \max_{0\le j\le cols}\left(\sum_{i=0}^{rows}\left|w[i][j]\right|\right).$$

Added in Igor Pro 9.00.

`outerProduct(w1,w2)`
For a 1D wave *w1* containing M points and a 1D wave *w2* containing N points, outerProduct returns an M by N matrix where the (i,j) element is:

```
out[i,j] = w1[i] * conj(w2[j])
```

Added in Igor Pro 8.00.

`p2Rect(w)`
Converts each element of *w* from polar to rectangular representation.

`phase(w)`
Returns a real valued wave containing the phase of each element of *w* calculated using `phase=atan2(y,x)`.

`Pi`
Returns π.

`powC(w1,w2)`
Complex valued $w1^{w2}$ where *w1* and *w2* can be real or complex.

`powR(x,y)`
Returns *x*^*y* for real *x* and *y*.

`productCol(w,c)`

Returns the a (1 x 1) wave containing the product of the elements in column *c* of wave *w*. The output is double precision real or complex.

Added in Igor Pro 7.00.

`productCols(w)`

Returns a (1 x cols) wave where each entry is the product of all the elements in the corresponding column. The output is double precision real or complex.

Added in Igor Pro 7.00.

`productDiagonal(w,d)`

Returns a (1 x 1) wave containing the product of the elements on the specified diagonal of wave w. *d*=0 is the main diagonal, *d*>0 correspond to upper diagonals and *d*<0 to lower diagonals. The output is double precision real or complex.

Added in Igor Pro 7.00.

`productRow(w,r)`

Returns the a (1 x 1) wave containing the product of the elements in row *r* of wave *w*. The output is double precision real or complex.

Added in Igor Pro 7.00.

`productRows(w)`

Returns a (1 x rows) wave where each entry is the product of all the elements in the corresponding row. The output is double precision real or complex.

Added in Igor Pro 7.00.

`quat(arg)`

See **Functions Using Quaternions** on page V-573.

`quatToAxis(qIn)`

See **Functions Using Quaternions** on page V-573.

`quatToEuler(qIn,mode)`

See **Functions Using Quaternions** on page V-573.

`quatToMatrix(qIn)`

See **Functions Using Quaternions** on page V-573.

`r2Polar(w)`

Performs the equivalent of the **r2polar** function on each element of *w*, i.e., each complex number (x+iy) is converted into the polar representation r,theta with x+iy=r*exp(i*theta)

`real(w)`

Returns the real part of *w*.

`rec(w)`

Returns the reciprocal of each element in *w*.

`redimension(w, nr, nc)`

Returns an (nr x nc) matrix from the data in the wave w. The data in w are moved contiguously (column by column regardless of dimensionality) into the output. If the output size is larger than w the remaining points are set to zero. If w contains more than one layer the new dimensions apply on a layer by layer basis. For example:

```
Make/N=(10,20,30) ddd = p + 10*q + 100*r
MatrixOp/O aa = redimension(ddd,25,1)
```

creates a wave aa with dimensions (25,1,30).

Added in Igor Pro 7.00.

`replace(w,findVal,replacementVal)`

Replace in wave *w* every occurrence of *findVal* with *replacementVal*. The wave *w* retains its dimensionality and number type. *replacementVal* is converted to the same number type as *w* which may cause truncation.

`replaceNaNs(w,replacementVal)`

Replaces every occurrence of NaN in the wave *w* with *replacementVal*. The wave *w* retains its dimensionality. *replacementVal* is converted to the same number type as *w* which may cause truncation.

`reverseCol(w,c)` Returns array *w* with column *c* in reverse order.

Added in Igor Pro 7.00.

`reverseCols(w)` Returns array *w* with all columns in reverse order.

Added in Igor Pro 7.00.

`reverseRow(w,r)` Returns array *w* with row *r* in reverse order.

Added in Igor Pro 7.00.

`reverseRows(w)` Returns array *w* with all rows in reverse order.

Added in Igor Pro 7.00.

`rotateChunks(w,n)` Returns a matrix where the last *n* chunks of *w* are moved to chunks [0, *n*-1]. If *n* is negative then the first abs(*n*) chunks are moved to the end of the data. It is an error to pass NaN for *n*.

Added in Igor Pro 7.00.

`rotateCols(w,nc)` Rotates the columns of a 2D wave *w* so that the last *nc* columns are moved to columns [0,*nc* -1] of the data. If *nc* is negative the first abs(*nc*) columns are moved to columns [n-1-*nc* ,n-1]. Here n is the total number of columns. It is an error to pass NaN for *nc* . If *nc* is greater than the number of columns then the effective rotation is mod(*nc* ,actualCols).

`rotateLayers(w,n)` Returns a matrix where the last *n* layers of *w* are moved to layers [0, *n*-1]. If *n* is negative then the first abs(*n*) layers are moved to the end of the data. It is an error to pass NaN for *n*.

Added in Igor Pro 7.00.

`rotateRows(w,nr)` Rotates the rows of a 2D wave *w* so that the last *nr* rows are moved to rows [0,*nr* -1] of the data. If *nr* is negative the first abs(*nr*) rows are moved to rows [n-1-*nr,* n-1] where n is the total number of rows. It is an error to pass NaN for *nr*. If *nr* is greater than the number of rows then the effective rotation is mod(*nr* ,actualRows).

`round(z)` Rounds *z* to the nearest integer. The rounding method is “away from zero”.

`row(w,r)` Returns row *r* from matrix wave *w*. The returned row is a (1xC) wave where C is the number of columns in *w*. To convert it to a 1D wave use `Redimension/N=(C)`. See also **ImageTransform** `getRow`.

`rowRepeat(w,n)` Returns a matrix that consists of *n* identical rows containing the data in the wave *w*. If *w* is a 2D wave, it is treated as if it were a single column containing all the data. Higher dimensions are supported on a layer-by-layer basis. MatrixOp returns an error if n<2.

Added in Igor Pro 7.00.

`scale(w,low,high)` Returns the values in the wave *w* scaled between the *low* and the *high* parameters. If *w* contains NaN or INF values, they are not modified. The result retains the same number type as that of *w* irrespective of the range of the *low* and *high* input parameters.

`scaleCols(w1,w2)` Returns a matrix of the same dimensions as *w1* where each column of *w1* is scaled by the value in the corresponding row of the 1D wave *w2*. The number of rows in *w2* must equal the number of columns in *w1*.

Added in Igor Pro 7.00.

`scaleRows(w1,w2)` Returns a matrix of the same dimensions as *w1* where each row of *w1* is scaled by the value in the corresponding row of the 1D wave *w2*. The number of rows in *w2* must equal the number of rows in *w1*.

Added in Igor Pro 7.00.

`select(w,sr,sc)` Returns a matrix consisting of the elements of matrix w for which the row index satisfies (p%*sr*)==0 and the column index satisfies (q%*sc*)==0. The first row and column are always selected.

sr and *sc* must be real, finite, and non-negative.

Added in Igor Pro 9.00.

`setCol(w2d,c,w1d)` Returns the data in the *w2d* with the contents of *w1d* stored in column *c*.

w1d must have at least as many elements as the number of rows of *w2d*. *w2d* and *w1d* must be either both real or both complex.

Added in Igor Pro 7.00.

`setColsRange(w,low,high)`

Returns a matrix in which each column of *w* is scaled to the range [*low*,*high*]. *w* must be real-valued.

setColsRange was added in Igor Pro 9.00.

`setNaNs(w,mask)` Returns the data in the wave *w* with NaNs stored where the mask wave is non-zero. The wave *w* can be of any numeric type.

mask must have the same dimensions as *w* and must be real. It is usually the result of another expression. For example, to set all values in the destination to NaN where *w* is greater than 5:

```
MatrixOp/o ou = setNaNs(w,greater(w,5))
```

Added in Igor Pro 7.00.

`setOffDiag(w,d,w1)` Returns the data in the *w* with the contents of *w1* stored in diagonal *d*.

d=0 is the main diagonal of w, *d*>0 correspond to upper diagonals and *d*<0 to lower diagonals. *w* and *w1* can either be both real or both complex.

Added in Igor Pro 7.00.

`setRow(w2d r,w1d)` Returns the data in the *w2d* with the contents of *w1d* stored in row *r*.

w1d must have at least as many elements as the number of columns in w2d. *w2d* and *w1d* must be either both real or both complex.

Added in Igor Pro 7.00.

`sgn(w)` Returns the sign of each element in *w*. It returns -1 for negative numbers and 1 otherwise. It does not accept complex numbers.

`shiftVector(w,n,val)` Shifts the element of a 1D row-vector *w* by *n* elements and fills the displaced elements with *val,* which must match the data type of *w* and should be expressed as `cmplx(a,b)` for complex *w*.

`sin(z)` Sine of *z*.

`sinh(z)` Hyperbolic sine of *z*. Added in Igor Pro 7.00.

`slerp(qIn1,qIn2,frac)` See **Functions Using Quaternions** on page V-573.

`spliceCols(w,c,d,ic)` Returns a matrix where `ic` columns from matrix `d` are spliced into matrix `w` starting at column `c` of `w`. Matrix `d` must have the same number of rows as matrix `w` and they must both have the same number type. `ic` must be non-negative integer. If `ic` is greater than the number of columns in the matrix `d`, the function repeats as many columns of `d` as necessary.

Added in Igor Pro 9.00.

`sq(w)` Returns the square of each element of the matrix *w*.

For complex elements z the returned value is z*z, not magsqr(z).

Added in Igor Pro 9.00.

sqrt(*z*) Square root of *z*.

subRange(*w*,*rs*,*re*,*cs*,*ce*)

Returns a contiguous subset of the wave *w* from starting row *rs* through ending row *re* and from starting column *cs* through ending column *ce*. This is similar to Duplicate/R except that dimension scaling and labels not preserved.

Added in Igor Pro 7.00.

subtractMean(*w*,*opt*) Computes the mean of the real wave *w* and returns the values of the wave minus the mean value (*opt*=0). Computes the mean of each column and subtracts it from that column (*opt*=1). Subtracts the mean of each row from row values (*opt*=2).

subWaveC(*w*,*r*,*c*,*count*[,*stride*])

Returns a subset of the data that is sampled along columns of the wave *w,* containing count elements starting with the element at row *r* and column *c*. By default *stride*=1 and the sampling is continuous.

You can specify a negative stride to sample backwards from the starting element.

The operation returns an error if the sampling would exceed the array bounds in either direction.

For example:

```
Make/O/N=(22,33) ddd=x

MatrixOP/O/P=1 aa=subWaveC(ddd,4,5,10,2)
// aa={4,6,8,10,12,14,16,18,20,0}

MatrixOP/O/P=1 aa=subWaveC(ddd,4,5,5,-4)
// aa={4,0,18,14,10}
```

subWaveR(*w*,*r*,*c*,*count*[,*stride*])

Returns a subset of the data that is sampled along rows of the wave w, containing count elements starting with the element at row r and column c. By default stride=1 and the sampling is continuous.

You can specify a negative stride to sample backwards from the starting element.

The operation returns an error if the sampling would exceed the array bounds in either direction.

Examples:

```
Make/O/N=(10,20) ddd=y

// Forward sampling across right boundary
MatrixOP/O/P=1 aa=subWaveR(ddd,4,15,6,2)
// aa={15,17,19,1,3,5}

// Reverse sampling across left boundary
MatrixOP/O/P=1 aa=subWaveR(ddd,2,3,5,-1)
// aa={3,2,1,0,19}
```

sum(*z*) Returns the sum of all the elements in expression *z*.

`sumBeams(`*`w`*`)`

Returns an (n x m) matrix containing the sum over all layers of all the beams of the 3D wave *w*:

$$out_{ij} = \sum_{k=0}^{nLayers-1} w_{ijk}.$$

A beam is a 1D array in the Z-direction.

sumBeams is a non-layered function which requires that *w* be a proper 3D wave and not the result of another expression.

`sumCols(`*`w`*`)`

Returns a (1 x m) matrix containing the sums of the m columns in the nxm input wave *w*:

$$out_j = \sum_{i=0}^{nRows-1} w_{ij}.$$

`sumND(`*`w`*`)`

Returns a 1-point wave containing the sum of the token *w* regardless of its dimensionality.

sumND was added in Igor Pro 9.00.

This example illustrates the difference between sumND and the sum function which operates on a layer-by-layer basis:

```
Make/O/N=(5,6,3) w3D = z
MatrixOP/O/P=1 sumOut=sum(w3D)      // Prints 3 layer sums
MatrixOP/O/P=1 sumNDOut=sumND(w3D)// Prints single sum
```

`sumRows(`*`w`*`)`

Returns an (n x 1) matrix containing the sums of the n rows in the nxm input wave *w*:

$$out_i = \sum_{j=0}^{nCols-1} w_{ij}.$$

`sumSqr(`*`w`*`)`

Sum of the squared magnitude of all elements in *w*.

`syncCorrelation(`*`w`*`)`

Synchronous spectrum correlation matrix for a real valued input matrix wave *w*. See also `asyncCorrelation`.

The correlation matrix is computed by subtracting from each column of *w* its mean value, multiplying the resulting matrix by its transpose, and finally dividing all elements by (nrows-1) where nrows is the number of rows in *w*.

`tan(`*`w`*`)`

Tangent of *w*.

`tanh(`*`w`*`)`

Hyperbolic tangent of *w*. Added in Igor Pro 7.00.

`tensorProduct(`*`w1`*`,`*`w2`*`)`

Returns a 2D matrix that is the tensor product of the 2D matrices *w1* and *w2*. For example, the tensor product of two (2 x 2) matrices is given by:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \otimes \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{pmatrix}$$

Added in Igor Pro 7.00.

`Trace(`*w*`)`

Returns a real or complex scalar which is the sum of the diagonal elements of *w*. If *w* is not a square matrix, the sum is over the elements for which the row and column indices are the same.

`transposeVol(`*w*`,`*mode*`)`

For 3D wave *w*, transposeVol returns a transposed 3D wave depending on the value of the *mode* parameter:

mode=1:	output=w[p][r][q]
mode=2:	output=w[r][p][q]
mode=3:	output=w[r][q][p]
mode=4:	output=w[q][r][p]
mode=5:	output=w[q][p][r]

transposeVol is a non-layered function which requires that *w* be a proper 3D wave and not the result of another expression.

`triDiag(`*w1*`,`*w2*`,`*w3*`)`

Returns a tri-diagonal matrix where *w1* is the upper diagonal, *w2* the main diagonal and *w3* the lower diagonal. If *w2* has *n* points then *w1* and *w3* are expected to have *n*-1 points. The waves can be of any numeric type and the returned wave has a numeric type that accommodates the input.

`uint8(`*w*`)`

Converts *w* to 8-bit unsigned integer representation. See also the /NPRM flag below for more information. Added in Igor Pro 7.00.

`uint16(`*w*`)`

Converts *w* to 16-bit unsigned integer representation. See also the /NPRM flag below for more information. Added in Igor Pro 7.00.

`uint32(`*w*`)`

Converts *w* to 32-bit unsigned integer representation. See also the /NPRM flag below for more information. Added in Igor Pro 7.00.

`varBeams(`*w*`)`

Returns a matrix containing the variances of the beams of the real-valued 3D wave *w*.

varBeams was added in Igor Pro 9.00.

varBeams is a non-layered function which requires that *w* be a proper 3D wave and not the result of another expression.

When *w* consists of a single layer the returned variances are all NaN.

`varCols(`*w*`)`

Returns a (1 x cols) wave where each element contains the variance of the corresponding column in *w*.

`waveX(`*w*`)`

For pure wave argument *w*, waveX returns the X value for each point as determined by the wave's X scaling.

Added in Igor Pro 9.00.

`waveY(`*w*`)`

For pure wave argument *w*, waveY returns the Y value for each point as determined by the wave's Y scaling.

Added in Igor Pro 9.00.

`waveZ(`*w*`)`

For pure wave argument *w*, waveZ returns the Z value for each point as determined by the wave's Z scaling.

Added in Igor Pro 9.00.

`waveT(`*w*`)`

For pure wave argument *w*, waveT returns the T value for each point as determined by the wave's T scaling.

Added in Igor Pro 9.00.

`waveIndexSet(`*w1*`,`*w2*`,`*w3*`)`

Returns a matrix of the same dimensions as *w1* with values taken either from *w1* or from *w3* depending on values in *w2* using:

$$out[i][j] = \begin{cases} w1[i][j] & \mathit{if}\ w2[i][j] < 0 \\ w3[w2[i][j]] & \mathit{otherwise} \end{cases}.$$

w1 and *w2* must have the same number of rows and columns. *w1* and *w3* must match in number type. *w2* cannot be unsigned.

Values from *w2* are used as point number indices into *w3* which is treated like a 1D wave regardless of its actual dimensionality.

An index value from *w2* is out-of-bounds if it is greater than or equal to the number of points in *w3*. In this case, the output value is taken from *w1* as if the index value were negative.

`waveMap(`*`w1`*`,`*`w2`*`)`

Returns an array of the same dimensions as *w2* containing the values w1[*w2*[*i*][*j*]]. The data type of the output is the same as that of *w1*. Values of *w2* are taken as 1D integer indices into the *w1* array. See also **IndexSort**.

`waveChunks(`*`w`*`)`

Returns the number of chunks in the wave *w*. Added in Igor Pro 7.00.

`waveLayers(`*`w`*`)`

Returns the number of layers in the wave *w*. Added in Igor Pro 7.00.

`wavePoints(`*`w`*`)`

Returns the number of points in the wave *w*. Added in Igor Pro 7.00.

`within(`*`w`*`,`*`low`*`,`*`high`*`)`

Returns an array of the same dimensions as *w* with the value 1 where the corresponding element of w is between low and high (low <= w[i][j] < high).

Added in Igor Pro 7.00.

All parameters must be real. It is an error to pass a NaN as either *low* or *high*. It is also an error if *low* >= *high*. If *w* contains NaNs, the corresponding outputs are 0.

`zapINFs(`*`w`*`)`

Returns a one-column wave containing the sequential data of *w* with all INF elements removed. The input *w* can be 1D or 2D but higher dimensions are not supported.

Added in Igor Pro 9.00.

`zapNaNs(`*`w`*`)`

Returns a one-column wave containing the sequential data of *w* with all NaN elements removed. The input *w* can be 1D or 2D but higher dimensions are not supported.

Added in Igor Pro 9.00.

`zeroMat(`*`r`*`,`*`c`*`,`*`nt`*`)`

Returns an (*r* x *c*) matrix of number type *nt* where all entries are set to zero. See **WaveType** for supported types. See also `const` above.

Added in Igor Pro 7.00.

Trignometric Transform Functions

`FCT(w, dir)`
Computes the fast, real to real cosine transform on 1D wave *w*.

Added in Igor Pro 8.00.

The forward transform (*dir*=1) is defined by:

$$F(k)=\frac{1}{n}\left[f(0)+f(n)\cos(k\pi)\right]+\frac{2}{n}\sum_{i=1}^{n-1}f(i)\cos\left(\frac{ik\pi}{n}\right),\quad k=0,\ldots,n$$

The number of intervals is n=numpnts(*w*)-1.

The inverse transform (*dir*=-1) is defined by:

$$f(i)=\frac{1}{2}\left[F(0)+F(n)\cos(i\pi)\right]+\sum_{k=1}^{n-1}F(k)\cos\left(\frac{ik\pi}{n}\right),\quad i=0,\ldots n$$

See **Trigonometric Transforms** on page V-576 below for more information.

`FST(w, dir)`
Computes the fast, real to real sine transform on 1D wave *w*.

Added in Igor Pro 8.00.

The forward transform (*dir*=1) is defined by:

$$F(k)=\frac{2}{n}\sum_{i=1}^{n-1}f(i)\sin\left(\frac{ik\pi}{n}\right),\quad k=1,\ldots,n-1$$

where n=numpnts(*w*).

The inverse transform (*dir*=-1) is defined by:

$$f(i)=\sum_{k=1}^{n-1}F(k)\sin\left(\frac{ik\pi}{n}\right),\quad i=1,\ldots,n-1$$

See **Trigonometric Transforms** on page V-576 below for more information.

`FSST(w, dir)`
Computes the fast, real to real staggered sine transform on 1D wave *w*.

Added in Igor Pro 8.00.

The forward direction (*dir*=1) is defined by:

$$F(k)=\frac{1}{n}\sin\left(\frac{(2k-1)\pi}{2}\right)f(n)+\frac{2}{n}\sum_{i=1}^{n-1}f(i)\sin\left(\frac{i(2k-1)\pi}{2n}\right),\quad k=1,\ldots,n$$

where n=numpnts(*w*).

The inverse transform (*dir*=-1) is defined by:

$$f(i)=\sum_{k=1}^{n}F(k)\sin\left(\frac{i(2k-1)\pi}{2n}\right),\quad i=1,\ldots n$$

See **Trigonometric Transforms** on page V-576 below for more information.

`FSCT(w,dir)` Computes the fast, real to real, staggered cosine transform on 1D wave *w*.

Added in Igor Pro 8.00.

The forward direction (*dir*=1) is defined by:

$$F(k) = \frac{1}{n} f(0) + \frac{2}{n} \sum_{j=1}^{n-1} f(j) \cos\left(\frac{j\pi(2k+1)}{2n} \right), \quad k = 0, \ldots, n-1$$

The inverse transform (*dir*=-1) is defined by:

$$f(i) = \sum_{k=0}^{n-1} F(k) \cos\left(\frac{(2k+1) i\pi}{2n} \right), \quad i = 0, \ldots, n-1$$

See **Trigonometric Transforms** on page V-576 below for more information.

`FSST2(w,dir)` Computes the fast, real to real, staggered2 sine transform on 1D wave *w*.

Added in Igor Pro 8.00.

The forward direction (*dir*=1) is defined by:

$$F(k) = \frac{2}{n} \sum_{i=1}^{n} f(i) \sin\left(\frac{(2k-1)(2i-1)\pi}{4n} \right), \quad k = 1, \ldots n$$

The inverse transform (*dir*=-1) is defined by:

$$f(i) = \sum_{k=1}^{n} F(k) \cos\left(\frac{(2k-1)(2i-1)\pi}{4n} \right), \quad i = 1, \ldots, n$$

See **Trigonometric Transforms** on page V-576 below for more information.

`FSCT2(w,dir)` Computes the fast, real to real, staggered2 cosine transform on 1D wave *w*.

Added in Igor Pro 8.00.

The forward direction (dir=1) is defined by:

$$F(k) = \frac{2}{n} \sum_{i=1}^{n} f(i) \cos\left(\frac{(2k-1)(2i-1)\pi}{4n} \right), \quad k = 1, \ldots, n$$

The inverse transform (*dir*=-1) is defined by:

$$f(i) = \sum_{k=1}^{n} F(k) \cos\left(\frac{(2k-1)(2i-1)\pi}{4n} \right), \quad i = 1, \ldots, n$$

See **Trigonometric Transforms** on page V-576 below for more information.

Functions Using Quaternions

These functions create and manipulate quaternion tokens or return quaternion results. They were added in Igor Pro 8.00. See **MatrixOp Quaternion Data Tokens** on page III-146 for background information on quaternions in MatrixOp.

`quat(`*`arg`*`)`

Converts arg into a quaternion token.

arg can be:

- A scalar which is converted to a real quaternion
- A 1x3 or 3x1 wave which is converted to a pure imaginary quaternion
- The output from another quat call

Arithmetic on quaternion tokens obeys quaternion arithmetic rules.

arg must not be complex. The resulting quaternion token is not normalized by the quat function.

See **MatrixOp Quaternion Data Tokens** on page III-146 for details.

`quatToMatrix(`*`qIn`*`)`

Converts *qIn* into a quaterion token, if not already a quaterion token, normalizes it, and returns the equivalent 4x4 homogeneous rotation matrix.

`quatToAxis(`*`qIn`*`)`

Converts *qIn* into a quaterion token, if not already a quaterion token, normalizes it, and returns the equivalent axis of rotation and rotation angle. The result is a 4-element wave in which the first three elements define the rotation axis and the last element is the rotation angle in radians.

`quatToEuler(`*`qIn`*`, `*`mode`*`)`

Converts *qIn* into a quaterion token, if not already a quaterion token, and returns a 3x1 wave containing equivalent Euler angles expressed in radians.

The returned Euler angles are are phi (rotation about the X axis), theta (rotation about the Y axis), and psi (rotation about the Z axis).

The *mode* parameter defines the rotation sequence. The supported modes are: 121, 123,131, 132, 212, 213, 231, 232, 312, 313, 321 and 323. 1 designates the X axis, 2 the Y axis and 3 the Z axis. See, for example: https://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToEuler/quat_2_euler_paper_ver2-1.pdf

`axisToQuat(`*`ax`*`)`

ax is a 4-element wave with the axis of rotation in the first 3 elements and the rotation angle in radians as the last element. Returns a 4-element wave containing the quaternion components representing this rotation.

`slerp(`*`qIn1`*`,`*`qIn2`*`,`*`frac`*`)`

Performs spherical linear interpolation between quaternion *qIn1* and quaternion *qIn2*. *frac* is a value between 0 and 1 representing the amount of desired rotation from *qIn1* to *qIn2*.

The function returns a 4-element wave containing the quaternion components that represent the rotated quaternion. This is useful, for example, in animating a sequence of rotations.

Wave Parameters

MatrixOp was designed to work with 2D waves (matrices) but also works with 1D, 3D and 4D waves. A 1D wave is treated like a 1-column matrix. 3D and 4D waves are treated on a layer-by-layer basis, as if each layer were a matrix>

You can reference subsets of waves in *expression*. Only two types of subsets are supported: those that evaluation to a single element, which are treated as scalars, and those that evaluate to one or more layers. For example:

`wave1d[a]`	Scalar
`wave2d[a][b]`	Scalar
`wave3d[a][b][c]`	Scalar
`wave3d[][][a]`	Layer *a* from 3D wave
`wave3d[][][a,b]`	Layers *a* through *b* from 3D wave
`wave3d[][][a,b,c]`	Layers *a* through *b* stepping by *c* from 3D wave

You can pass waves of any dimensions as parameters to MatrixOp functions. For example:

```
Make/O/N=128 wave1d = x
MatrixOp/O outWave = powR(wave1d,2)
```

MatrixOp does not allow using the same 3D wave on both sides of the assignment:

```
MatrixOp/O wave3D = wave3D + 3    // Not allowed
```

See **MatrixOp Wave Data Tokens** on page III-141 for further discussion.

Flags

/AT=*allocationType*

/AT lets you control the method used to allocate memory in MatrixOP. It was added in Igor Pro 9.00.

Use *allocationType*=0 for generic memory allocation.

Use *allocationType*=1 for Intel scalable allocation.

Use *allocationType*=2 for Intel scalable and aligned allocation.

By default MatrixOP uses scalable and aligned allocation.

/C Provides a complex wave reference for *destWave*. If omitted, MatrixOp creates a real wave reference for *destWave*. The wave reference allows you to refer to the output wave in a subsequent statement of a user-defined function.

/FREE Creates *destWave* as a free wave. Allowed only in functions and only if a simple name or wave reference structure field is specified.

Requires Igor Pro 6.1 or later. For advanced programmers only.

See **Free Waves** on page IV-91 for more discussion.

/NTHR=*n* Sets the number of threads used to compute the results for 3D waves. Each thread computes the results for a single layer of the input.

By default (/NTHR omitted) the calculations are performed by the main thread only.

If *n*=0 the operation uses as many threads you have processors on your computer.

If *n*>0, n specifies the number of threads to use. More threads may or may not improve performance.

/NPRM Use /NPRM to restrict the automatic promotion of numeric data types in MatrixOp expressions.

By default, MatrixOp promotes numeric data types so that operations result in reasonable accuracy. In some situations you may want to keep the results as a particular data type even at the risk of truncation or overflow. If you include the /NPRM flag, MatrixOp creates the destination wave using the highest precision data type in the expression. For example, an expression A=B+C where B is 16-bit wave and C is an 8-bit wave results in a 16-bit wave A.

Unsigned number types can result only when all operands are unsigned.

/NPRM is ignored when data promotion is required. For example:

```
Make/B/U wave2
```

```
MatrixOp/O/NPRM wave1 = -wave2
```

You can use MatrixOp functions such as int8, int16, etc., to precisely control the number type of any token.

/NV=*mode* Controls use of Intel MKL vectorized functions. /NV was added in Igor Pro 9.00.

By default, *mode*=1, which uses vectorized functions where possible.

Set *mode* to 0 to request non-vectorized execution. This may improve speed when MatrixOP is called in a preemptive thread by preventing spawning additional threads.

/O Overwrites *destWave* if it already exists.

/P=*printMode*

Controls printing of the result of the MatrixOp evaluation. /P was added in Igor Pro 8.00.

/P is ignored if MatrixOp is not running in the main thread.

printMode is a value between 0 and 2:

0: No printing is done (default).

1: Prints the result from evaluating expression in layer-by-layer order to the history area of the command window and stores the result in the destination wave.

2: Prints the result from evaluating expression in layer-by-layer order to the history area of the command window but does create or store the resulting values in the destination wave. If the destination wave may already exist, you must include the /O flag, even though the destination wave is not changed by the operation.

Use /P=1 if you want MatrixOp to work normally and then to print the destination wave.

Use /P=2 if you want MatrixOp to print its result without creating or affecting any waves.

Values are printed sequentially using 16-digit precision. Output is not formatted to represent rows and columns.

When the operation's result is a 3D or 4D wave, the results are printed on a layer-by-layer basis.

See **MatrixOp Printing Examples** on page V-577 below.

/S Preserves the dimension scaling, units and wave note of a pre-existing destination wave in a MatrixOp/O command.

/T=*k* Displays results in a table. *k* specifies the behavior when the user attempts to close it.

0: Normal with dialog (default).

1: Kills with no dialog.

2: Disables killing.

Details

MatrixOp has the general form:

```
MatrixOp [flags] destWave = expression
```

destWave specifies the wave created by MatrixOp or overwritten by MatrixOp/O.

From the command line, *destWave* can be a simple wave name, a partial data folder path or a full data folder path. In a user-defined function it can be a simple wave name or, if /O is present, a wave reference pointing to an existing wave.

expression is a mathematical expression that consists of one or more data tokens combined with the built-in MatrixOp functions and MatrixOp operators listed above. MatrixOp does not support the p, q, r, s, or x, y, z, t symbols that are used in waveform assignment statements.

Data tokens include waves, variables and literal numbers.

You can use any combination of data types for operands. In particular, you can mix real and complex types in *expression*. MatrixOp determines data types of inputs and the appropriate output data type at runtime without regard to any type declaration such as `Wave/C`.

See **Using MatrixOp** on page III-140 for more information.

Trigonometric Transforms

The trigonometrics transform functions were added in Igor Pro 8.00.

FST, FCT, FSST, FSCT, FSST2 and FSCT2 are implemented using INTEL MKL library functions. The transforms may automatically execute in multiple threads.

The equations for the definitions of the forward and reverse transforms follow Intel's documentation (see https://software.intel.com/en-us/mkl-developer-reference-c-trigonometric-transforms-implemented).

In MatrixOP's implementation, all input and output arrays are zero based. This is illustrated by the following example for the staggared2 cosine:

```
Function DoStaggered2Cosine(inWave)
    Wave inWave

    Variable n = dimsize(inWave,0)
    Make/O/N=(n)/D outStaggered2CosTransform=0
    Variable i, k, theSum, frontFactor=2/n

    for(k=1; k<=n; k+=1)
        theSum=0
        for(i=1; i<=n; i+=1)
            theSum += inWave[i-1] * cos((2*k-1) * (2*i-1) * pi/(4*n))
        endfor
        outStaggered2CosTransform[k-1] = frontFactor * theSum
    endfor
End
```

Examples

In addition to these examples, see **MatrixOp Optimization Examples** on page III-148.

The following matrices are used in these examples:

```
Make/O/N=(3,3) r1=x, r2=y
```

Matrix addition and matrix multiplication by a scalar:

```
MatrixOp/O outWave = r1+r2-3*r1
```

Using the matrix `Identity` function:

```
MatrixOp/O outWave = Identity(3) x r1
```

Create a persisting identity matrix for another calculation:

```
MatrixOp/O id4 = Identity(4)
```

Using the `Trace` function:

```
MatrixOp/O outWave = (Trace(r1)*identity(3) x r1)-3*r1
```

Using matrix inverse function `Inv()` with matrix multiplication:

```
MatrixOp/O outWave = Inv(r2) x r2
```

Using the determinant function `Det()`:

```
MatrixOp/O outWave = Det(r1)+Det(r2)
```

Using the Transpose postfix operator:

```
MatrixOp/O outWave = r1^t+(r2-r1)^t-r2^t
```

Using a mix of real and complex data:

```
Variable/C complexVar = cmplx(1,2)
MatrixOp/O outWave = complexVar*r2 - Cmplx(2,4)*r1
```

Hermitian transpose operator:

```
MatrixOp/O outWave = Trace(complexVar*r2)^h -Trace(cmplx(2,4)*r1)^h
```

In-place operation and conversion to complex:

```
MatrixOp/O r1 = r1*cmplx(1,2)
```

Image filtering using 2D spatial filter filterWave:

```
MatrixOp/O filteredImage=IFFT(FFT(srcImage,2)*filterWave,3)
```

Positive shift:

```
Make/O w={0,1,2,3,4,5,6}
MatrixOp/O w=shiftVector(w,2,77)
Print w
// w[0]= {77,77,0,1,2,3,4}
```

Negative shift:

```
Make/O w={0,1,2,3,4,5,6}
MatrixOp/O w=shiftVector(w,(-2),77)
Print w
// w[0]= {2,3,4,5,6,77,77}

// Using KronProd function to generate Hadamard matrices
Function HadamardMatrix(int N)              // N must be >= 2
   Make/FREE/N=(2,2) H2={{1,1},{1,-1}}
   Duplicate/FREE H2, tmp
   int i
   for(i=2; i<N; i+=1)
      MatrixOP/O/FREE tmp=KronProd(H2,tmp)
   endfor
   Duplicate/O tmp, Hadamard
End

// bitReverseCol
Make/N=8 index = p
MatrixOP/O/P=1 out = bitReverseCol(index,0,0)     // Direct binary reversal
                                                  // Prints: out = {0,4,2,6,1,5,3,7}
MatrixOP/O/P=1 out = bitReverseCol(index,0,1)     // Hadamard order
                                                  // Prints: out = {0,4,6,2,3,7,5,1}
MatrixOP/O/P=1 out = bitReverseCol(index,0,2)     // Dyadic/Paley/Gray order
                                                  // Prints: out = {0,1,3,2,6,7,5,4}
```

MatrixOp Printing Examples

```
// Print a single-valued result
Make/O/N=(10,5) m2D=(p+1)*(q+1)
MatrixOP/O/P=1 aa=sum(m2D)
  aa={825}

// Print a real 2D result
Make/O/N=(1,4) ddd=enoise(3)
MatrixOp/O/P=1 aa=ddd
aa={-2.273916482925415,-2.327789783477783,1.286988377571106,-0.8658701777458191}

// Print higher-dimension result layer-by-layer
Make/O/N=(3,4,3) ddd=z+1
MatrixOP/O/P=1 aa=mean(ddd)
aa={1}       // Each layer results in a 1x1 value
aa={2}
aa={3}
```

References

syncCorrelation and asyncCorrelation:

Noda, I., Determination of Two-Dimensional Correlation Spectra Using the Hilbert Transform, *Applied Spectroscopy 54*, 994-999, 2000.

ChirpZ:

Rabiner, L.R., and B. Gold, *The Theory and Application of Digital Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1975.

See Also

Using MatrixOp on page III-140

Matrix Math Operations on page III-138 for more about Igor’s matrix routines

FastOp

MatrixRank

```
matrixRank(matrixWaveA [, conditionNumberA])
```

The matrixRank function returns the rank of *matrixWaveA* subject to the specified condition number.

The matrix is not considered to have full rank if its condition number exceeds the specified *conditionNumberA*.

If the optional parameter *conditionNumberA* is not specified, Igor Pro uses the value 10^{20}.

matrixRank supports real and complex single precision and double precision numeric wave data types.

The value of *conditionNumberA* should be large enough but taking into account the accuracy of the numerical representation given the numeric data type.

If there are any errors the function returns NaN.

See Also

Matrix Math Operations on page III-138 for more about Igor’s matrix routines.

MatrixReverseBalance

```
MatrixReverseBalance [flags] scaleWave, eigenvectorsWave
```

MatrixReverseBalance inverse-transforms left or right eigenvectors contained in *eigenvectorsWave* that were computed for a matrix that was balanced using MatrixBalance. The results are the eigenvectors of the pre-balanced matrix. *scaleWave* is W_scale as returned by MatrixReverseBalance.

MatrixBalance was added in Igor Pro 9.00.

Parameters

eigenvectorsWave must be single-precision or double-precision floating point, real or complex, and must contain no NaNs. MatrixReverseBalance returns an error if these conditions are not met.

Flags

/DSTM=*dest*	Specifies the destination wave for the inverse-transformed eigenvectors. If you omit /DSTM, the output is saved in M_RBEigenvectors in the current data folder.
/FREE	Create free destination wave when it is specified via /DSTM.
/J=*job*	*job* is the type of backward transformation required. It is one of the following letters: N *srcWave* is not permuted or scaled. P *srcWave* is permuted but not scaled. S *srcWave* is scaled but not permuted. The scaling applies a diagonal similarity transformation to make the norms of the various columns close to each other. B *srcWave* is both scaled and permuted (default). You should use the same value for *job* as was used in the original balancing.
/LH={*low*,*high*}	Specifies the zero-based low and high indices that were returned by MatrixBalance in V_min and V_max respectively.
/Z	Suppresses error reporting. If you use /Z, check the V_Flag output variable to see if the operation succeeded.

Details

Matrix balancing is usually called internally by LAPACK routines when there is large variation in the magnitude of matrix elements. Following matrix balancing the computed eigenvalues are expected to be more accurate but the resulting eigenvectors are not the correct eigenvectors of the original pre-balanced matrix. MatrixReverseBalance is then applied to the balanced eigenvectors in order to obtain the eigenvectors of the original matrix.

The operation uses outputs from MatrixBalance as inputs. Pass the W_scale output from MatrixBalance as the scaleWave parameter. Pass the V_min and V_max outputs from MatrixBalance as the /LH *low* and *high* parameters. See **MatrixBalance** for an example.

Output Variables

V_Flag	Set to zero when the operation succeeds. Otherwise, when V_flag is positive the value is a standard Igor error code. When V_flag is negative it is an indication of an invalid input parameter.

References

The operation uses the following LAPACK routines: sgebak, dgebak, cgebak, and zgebak.

See Also

MatrixBalance

MatrixSchur

`MatrixSchur [/Z]` ***`srcMatrix`***

The MatrixSchur operation computes for an NxN nonsymmetric srcMatrix, the eigenvalues, the real Schur form A and the matrix of Schur vectors V.

The Schur factorization has the form: S = V x A x (V^T), where V^T is the transpose (use V^H if S is complex) and x denotes matrix multiplication.

Flags

/Z	No error reporting.

Details

The operation creates:

M_A	Upper triangular matrix containing the Schur form A.
M_V	Unitary matrix containing the orthogonal matrix V of the Schur vectors.
W_REigenValues W_IEigenValues	Waves containing the real and imaginary parts of the eigenvalues when *srcMatrix* is a real wave. If *srcMatrix* is complex, the eigenvalues are stored in W_eigenValues.

The variable V_flag is set to 0 when there is no error; otherwise it contains the LAPACK error code.

Examples

You can test this operation for an N-by-N source matrix:

```
Make/D/C/N=(5,5) M_S=cmplx(enoise(1),enoise(1))
MatrixSchur M_S
MatrixOp/O unitary=(M_V^h) x M_V            // Check unitary
MatrixOp/O diff=abs(M_S-M_V x M_A x (M_V^H))   // Check decomposition
```

See Also

Matrix Math Operations on page III-138 for more about Igor’s matrix routines and for background references with details about the LAPACK libraries.

MatrixSolve

`MatrixSolve` ***`method`*****`,`** ***`matrixA`*****`,`** ***`vectorB`***

The MatrixSolve operation was superseded by MatrixLLS and is included for backward compatibility only.

Used to solve matrix equation Ax=b using the method of your choice. Choices for *method* are:

method	**Solution Method**
GJ	Gauss Jordan.
LU	LU decomposition.
SV	Singular Value decomposition.

Details

The array b can be a matrix containing a number of b vectors and the output matrix M_x will contain a corresponding set of solution vectors.

V_flag is set to zero if success, 1 if singular matrix using GJ or LU and 1 if SV fails to converge.

For normal problems you should use LU. GJ is provided only for completeness and has no practical use.

When using SV, singular values smaller than 10^{-6} times the largest singular value are set to zero before back substitution.

Generates an error if the dimensions of the input matrices are not appropriate.

See Also

The **MatrixLLS** operation. **Matrix Math Operations** on page III-138 for more about Igor's matrix routines.

MatrixSparse

MatrixSparse [***flags***] **keyword=*value***

The MatrixSparse operation provides support for basic matrix operations on sparse matrices. The MatrixSparse operation was added in Igor Pro 9.00.

The following sections use terms and concepts explained under Sparse Matrices. You will need to understand those terms and concepts to understand the following material.

Flags

/LM Limited memory hint. Specifies that the operation can allocate limited memory proportional to vector size. If this flag is omitted, the operation allocates memory aggressively up to the size of the sparse matrix.

/Q Quiet - do not print diagnostic information to the history area.

/Z Errors are not fatal and do not abort procedure execution. Your procedure can inspect the V_flag variable to see if the operation succeeded. V_flag will be zero if it succeeded or nonzero if it failed.

Keywords

alpha=*value* Specifies the real part of the alpha value for the MM, MV, and TRSV operations. The default value is 1.

alphai=*value* Specifies the imaginary part of the alpha value for the MM, MV, and TRSV operations on complex matrices. The default value is 0.

alphai should be specified only when operating on complex matrices.

beta=*value* Specifies the real part of the beta value for the MM and MV operations. The default value is 0.

betai=*value* Specifies the imaginary part of the beta value for the MM and MV operations. The default value is 0.

betai should be specified only when operating on complex matrices.

colsA=*nCols* Specifies the number of columns of the sparse matrix A.

colsA is a required keyword for all MatrixSparse commands even in the rare cases where neither cooA, cscA, nor csrA are used.

colsG=*nCols* Specifies the number of columns of the sparse matrix G.

colsG is a required keyword for operations involving sparse matrix G.

cooA={*valuesWave, rowsWave, colsWave*}

Specifies sparse matrix A in COO format using three 1D waves. See **Sparse Matrix Formats** on page III-151 for background information on cooA format.

The COO format is used only for conversion operations, not for math operations which always use CSR format.

valuesWave is an wave of length *nnz* (number of non-zero values) containing the non-zero values of A. The wave must be single or double precision real or complex and must not contain any NaNs or INFs.

rowsWave and *colsWave* are 64-bit integer 1D waves created with Make/L that contain nnz row and column indices respectively.

When using cooA, you must also specify the rowsA and colsA keywords.

cscA={*valuesWave, rowsWave, ptrWave*}

Specifies sparse matrix A in CSC format A using three 1D waves. See **Sparse Matrix Formats** on page III-151 for background information on cscA format.

The CSC format is used only for conversion operations, not for math operations which always use CSR format.

valuesWave is an wave of length *nnz* (number of non-zero values) containing the non-zero values of A. The wave must be single or double precision real or complex and must not contain any NaNs or INFs.

rowsWave and *ptrWave* are 64-bit integer 1D waves created with Make/L.

rowsWave consists of nnz entries that specify the zero-based row index of each entry in *valuesWave*.

ptrWave contains nCols or nCols+1 entries the first of which must be zero. *ptrWave*[j] gives the index in valuesWave of the first non-zero value in column j. The optional longer version of *ptrWave* stores nnz in the last wave point.

When using cscA, you must also specify the rowsA and colsA keywords.

csrA={*valuesWave, colsWave, ptrWave*}

Specifies sparse matrix A in CSR format using three 1D waves. See **Sparse Matrix Formats** on page III-151 for background information on csrA format.

The CSR format is used for conversion operations and for math operations.

valuesWave is an wave of length *nnz* (number of non-zero values) containing the non-zero values of A. The wave must be single or double precision real or complex and must not contain any NaNs or INFs.

colsWave and *ptrWave* are 64-bit integer 1D waves created with Make/L.

colsWave consists of nnz entries that specify the zero-based column index of each entry in *valuesWave*.

ptrWave contains nRows or nRows+1 entries the first of which must be zero. *ptrWave*[i] gives the index in *valuesWave* of the first non-zero value of row i. The optional longer version of *ptrWave* stores nnz in the last wave point.

When using csrA, you must also specify the rowsA and colsA keywords.

cooG={*valuesWave, rowsWave, colsWave*}

Similar to cooA but applies to sparse matrix G.

cscG={*valuesWave, rowsWave, ptrWave*}

Similar to cscA but applies to sparse matrix G.

csrG={*valuesWave*, *colsWave*, *ptrWave*}

Similar to csrA but applies to sparse matrix G.

matrixB=*wb* Designates the 2D wave *wb* as matrix B for the MM operation.

The wave *wb* must be of the same data type as sparse matrix A and must not contain INFs or NaNs.

matrixC=*wc* Designates the 2D wave *wc* as matrix C for the MM operation.

The wave *wc* must be of the same data type as sparse matrix A and must not contain INFs or NaNs.

opA=*opName* Specifies a transformation that is applied to sparse matrix A. *opName* is a single letter:

T: Transpose

H: Hermitian

N: No transformation (default)

See **MatrixSparse Transformations** on page III-156 for details.

opG=*opName* Specifies a transformation that is applied to sparse matrix G. *opName* is a single letter:

T: Transpose

H: Hermitian

N: No transformation (default)

See **MatrixSparse Transformations** on page III-156 for details.

operation=*opN* Specifies the operation name. This is a required keyword.

opN is one of the following:

ADD Adds sparse matrices. See **MatrixSparse ADD** on page III-157 for details.

MM Computes the product of a sparse matrix and a dense matrix. See **MatrixSparse MM** on page III-157 for details.

MV Computes the product of a sparse matrix and a vector. See **MatrixSparse MV** on page III-158 for details.

SMSM Computes the product of two sparse matrices. See **MatrixSparse SMSM** on page III-158 for details.

TOCOO Creates a sparse matrix in COO format from a sparse matrix or dense matrix. See **MatrixSparse TOCOO** on page III-159 for details.

TOCSC Creates a sparse matrix in CSC format from a sparse matrix or dense matrix. See **MatrixSparse TOCSC** on page III-159 for details.

TOCSR Creates a sparse matrix in CSR format from a sparse matrix or dense matrix. See **MatrixSparse TOCSR** on page III-160 for details.

TODENSE Creates a dense matrix from a sparse matrix. See **MatrixSparse TODENSE** on page III-160 for details.

TRSV Solves a system of linear equations for a triangular sparse matrix. See **MatrixSparse TRSV** on page III-161 for details.

rowsA=*nRows* Specifies the number of rows of the sparse matrix A.

rowsA is a required keyword for all MatrixSparse commands even in the rare cases where neither cooA, cscA, nor csrA are used.

rowsG=*nRows* Specifies the number of rows of the sparse matrix G.

rowsG is a required keyword for operations involving sparse matrix G.

sparseMatrixType={*smType*,*smMode*,*smDiag*}	
	Provides optional information to describe both sparse matrix A and sparse matrix G. All of the parameters are keywords. smType: GENERAL, SYMMETRIC, HERMITIAN, TRIANGULAR, DIAGONAL, BLOCK_TRIANGULAR, or BLOCK_DIAGONAL. smMode: LOWER or UPPER. smDiag: DIAG or NON_DIAG. See **Optional Sparse Matrix Information** on page III-156 for details.
vectorX=*wx*	Designates the 1D wave *wx* as vector X for the MV and TRSV operations. The wave *wx* must be of the same data type as the sparse matrix data and it must not contain INFs or NaNs.
vectorY=*wy*	Designates the 1D wave *wy* as vector Y for the MV operation. The wave *wy* must be of the same data type as the sparse matrix data and it must not contain any INFs or NaNs.

Details

MatrixSparse supports data waves with single-precision and double-precision floating point real and complex data types. See **MatrixSparse Operation Data Type** on page III-155 for details.

Index waves must be signed 64-bit integer. See **MatrixSparse Index Data Type** on page III-156 for details.

MatrixSparse does not support waves containing NaNs or INFs.

MatrixSparse math operations (ADD, MV, MM, SMSM, TRSV) require that input sparse matrices be in CSR format. The math operations that return sparse matrices (ADD, SMSM) create output sparse matrices in CSR format.

The conversion operations (TOCOO, TOCSC, TOCSR, TODENSE) accept inputs in COO, CSC, CSR, or dense formats.

Output Variables

MatrixSparse sets these automatically created variables:

V_flag	Set to 0 if the operation succeeded or to a non-zero error code.

Examples

You can find examples using MatrixSparse under **MatrixSparse Operations** on page III-156.

See Also

Sparse Matrices on page III-151 for background information about Igor's sparse matrix support.

MatrixSparse Operations on page III-156 for details on each supported operation and examples.

Matrix Math Operations on page III-138 for discussion of non-sparse Igor matrix routines.

MatrixSVBkSub

MatrixSVBkSub ***matrixU*, *vectorW*, *matrixV*, *vectorB***

The MatrixSVBkSub operation does back substitution for SV decomposition.

Details

Used to solve matrix equation Ax=b after you have performed an SV decomposition.

Feed this routine the M_U, W_W and M_V waves from **MatrixSVD** along with your right-hand-side vector b. The solution vector x is returned as M_x.

The array b can be a matrix containing a number of b vectors and the M_x will contain a corresponding set of solution vectors.

Generates an error if the dimensions of the input matrices are not appropriate.

See Also

Matrix Math Operations on page III-138 for more about Igor's matrix routines.

MatrixSVD

`MatrixSVD [`*`flags`*`]` ***`matrixWave`***

The MatrixSVD operation uses the singular value decomposition algorithm to decompose an MxN matrixWave into a product of three matrices. The default decomposition is into MxM wave M_U, min(M,N) wave W_W and NxN wave M_VT.

Flags

/B

Use this flag for backwards compatibility with Igor Pro 3. This option applies only to real valued input waves. Note that no other flag can be combined with /B. Here the decomposition is such that:

U*W*V^T = *matrixWave*

U:	MxN column-orthonormal matrix.
W:	NxN diagonal matrix of positive singular values.
V:	NxN orthonormal matrix.

/DACA

Replaces the standard LAPACK algorithm with one that is based on a divide and conquer approach. For a typical 1000x1000 matrix this provides a 6x speed improvement.

Added in Igor Pro 7.00.

/INVW

Saves the inverse of the elements in W_W. The results are then stored in wave W_InvW.

/O

Overwrites *matrixWave* with the first columns of U. Use this flag to if you need to conserve memory. See also related settings of /U and /V.

/PART =*nVals*

Performs a partial SVD computing only *nVals* singular values (stored in W_W) and the associated vectors in the matrix M_U and M_V. If you use this flag the operation ignores all other flags except /PDEL. The partial SVD is computed using the Power method of Nash and Shlien.

The /PART flag was added in Igor Pro 7.00.

/PDEL=*del*

Sets the convergence threshold which defaults to 1e-6. Larger positive values result in faster execution but may lead to less accurate results.

The /PDEL flag was added in Igor Pro 7.00.

/U =*UMatrixOptions*

UMatrixOptions can have the following values:

0:	All columns of U are returned in the wave M_U (default).
1:	The first min(m,n) columns of U are returned in the wave M_U.
2:	The first min(m,n) columns of U overwrite matrixWave (/O must be specified).
3:	No columns of U are computed.

/V=*VMatrixOptions*

VMatrixOptions can have the following values:

0:	All rows of V^T are returned in the wave M_VT (default).
1:	The first min(m,n) rows of V^T are returned in the wave M_VT.
2:	The first min(m,n) rows of V^T are overwritten on *matrixWave* (/O must be specified)
3:	No rows of V^T are computed.

/Z

No error reporting.

Details

The singular value decomposition is computed using LAPACK routines. The diagonal elements of matrix W are returned as a 1D wave named W_W. If /B is used W_W will have N elements. Otherwise the number of elements in W_W is min(M,N).

The matrix V is returned in a matrix wave named M_V if /B is used otherwise the transpose V^T is returned in the wave M_VT.

All output objects are created in the current data folder.

The variable V_flag is set to zero if the operation succeeds. It is set to 1 if the algorithm fails to converge.

The variable V_SVConditionNumber is set to the condition number of the input matrix. The condition number is the ratio of the largest singular value to the smallest.

Example

```
Make/O/D/N=(10,20) A=gnoise(10)
MatrixSVD A
MatrixOp/O diff=abs(A-(M_U x DiagRC(W_W,10,20) x M_VT))
Print sum(diff,-inf,inf)
```

References

J.C. Nash and S.Shlien "Simple Algorithms for the Partial Singular Value Decomposition", The Comp. J. (30) No. 3 1987.

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-138 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

MatrixTrace

`matrixTrace(`*`dataMatrix`*`)`

The matrixTrace function calculates the trace (sum of diagonal elements) of a square matrix. *dataMatrix* can be of any numeric data type.

If the matrix is complex, it returns the sum of the magnitudes of the diagonal elements.

See Also

Matrix Math Operations on page III-138 for more about Igor's matrix routines.

MatrixTranspose

`MatrixTranspose` `[`**`/H`**`]` ***`matrix`***

The MatrixTranspose operation Swaps rows and columns in *matrix*.

Does not take complex conjugate if data are complex. You can do that as a follow-on step.

Swaps row and column labels, units and scaling.

This works with text as well as numeric waves. If the matrix has zero data points, it just swaps the row and column scaling.

Flags

/H	Computes the Hermitian conjugate of a complex wave.

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-138 for more about Igor's matrix routines.

max

max(*num1*, *num2* [, *num3*, ... *num200*])

The max function returns the greatest value of *num1*, *num2*, ... *num200*.

If any parameter is NaN, the result is NaN.

Details

In Igor7 or later, you can pass up to 200 parameters. Previously max was limited to two parameters.

See Also

min, **limit**, **WaveMin**, **WaveMax**, **WaveMinAndMax**

mean

mean(*waveName* [, *x1*, *x2*])

The mean function returns the arithmetic mean of the wave for points from x=*x1* to x=*x2*.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

The wave values from *x1* to *x2* are summed, and the result divided by the number of points in the range.

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with `pnt2x(`*`waveName`*`,`*`pointNumber1`*`)`, and a similar expression for *x2*.

If the points nearest to *x1* or *x2* are not within the point range of 0 to numpnts(*waveName*)-1, mean limits them to the nearest of point 0 or point numpnts(*waveName*)-1.

If any values in the point range are NaN, mean returns NaN.

The function returns NaN if the input wave has zero points.

Unlike the area function, reversing the order of *x1* and *x2* does *not* change the sign of the returned value.

The mean function is not multidimensional aware. See Chapter II-6, **Multidimensional Waves**, particularly Chapter II-6, **Analysis on Multidimensional Waves** for details.

Examples

```
Make/O/N=100 data; SetScale/I x 0,Pi,data
data=sin(x)
Print mean(data,0,Pi)          // the entire point range, and no more
Print mean(data)               // same as -infinity to +infinity
Print mean(data,Inf,-Inf)      // +infinity to -infinity
```

The following is printed to the history area:

```
Print mean(data,0,Pi)          // the entire point range, and no more
  0.630201
Print mean(data)               // same as -infinity to +infinity
  0.630201
Print mean(data,Inf,-Inf)      // +infinity to -infinity
  0.630201
```

See Also

Variance, **WaveStats**, **median**, **APMath**

The figure “Comparison of area, faverage and mean functions over interval (12.75,13.32)”, in the **Details** section of the **faverage** function.

median

median(waveName [, *x1*, *x2*])

The median function returns the median value of the wave for points from x=*x1* to x=*x2*.

The median function was added in Igor Pro 7.00.

Details

If you omit *x1* and *x2*, they default to -INF and +INF, respectively.

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x1*. To use point indexing, replace *x1* with "pnt2x(waveName,pointNumber1)", and a similar expression for *x2*.

If the points nearest to *x1* or *x2* are outside the point range of 0 to numpnts(waveName)-1, median limits them to the nearest of point 0 or point numpnts(waveName)-1.

If the wave contains NaNs they are skipped.

The function returns NaN if the input wave has zero non-NaN points.

See Also

mean, **Variance**, **StatsMedian**, **StatsQuantiles**, **WaveStats**

MeasureStyledText

```
MeasureStyledText [/W=winName /A=axisName /B=baselineMode /F=fontName
   /SIZE=fontSize /STYL=fontStyle] [styledTextStr]
```

The MeasureStyledText operation takes as input a string optionally containing style codes such as are used in graph annotations and the **DrawText** operation. It sets various variables with information about the dimensions of the string.

In Igor Pro 9.00, the styledTextStr parameter was made optional, the /B flag was added, and several output variables (all but V_width and V_height) were added.

Flags

/W=*winName* Takes default text information from the window *winName*.

If you omit /W, MeasureStyledText works on the top graph window.

/A=*axisName* Takes default text information from the axis named *axisName* in the top graph or in the window specified by /W.

/B=*baseLineMode* Selects which baseline offset is returned in V_baseline if the text contains multiple lines (separated by carriage returns).

If you omit /B or specify /B=0, V_baseline is set to the offset to the last line of the text.

If you specify /B=1, V_baseline is set to the offset to the first line of the text.

The /B flag was added in Igor Pro 9.00.

/F=*fontNameStr* The name of the default font.

In Igor Pro 9.00 and later, /F="default" and /F="" are the same as omitting /F in which case the default font is defined by /W, /A, or by the overall default font specified by the **DefaultFont** operation.

/SIZE=*size* Sets default font size.

/STYL=*fontStyle* Sets default font style:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

/STYLE=0 specifies plain text.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Parameters

styledTextStr is the styled text to be measured.

In Igor Pro 9.00 and later, *styledTextStr* is an optional parameter. Previously it was required.

If you include *styledTextStr*, the default font and styled text output variables are set.

If you omit *styledTextStr*, only the default font output variables are set.

See **Default Font Output Variables** and **Styled Text Output Variables** below.

styledTextStr can contain escape codes to set the font, size, style, color and other properties. See **Annotation Escape Codes** on page III-53 for details. The text may contain multiple lines separated with carriage returns ("\r").

Details

In the absence of formatting codes within the text that set the font, font size and font style, some mechanism must be provided that sets them.

The /W flag tells MeasureStyledText to get the default font name and font size from the specified window. If /W is omitted, defaults come from the top graph window.

The /A flag specifies that the default font name and font size come from the specified axis in the graph window specified by /W or in the top graph if /W is omitted.

The /F, /SIZE and /STYL flags set defaults that override any defaults from a window or axis. If you don't use any flags, the defaults are Igor's overall defaults.

If you omit all flags, the font specified by the **DefaultFont** operation is used, the default font size is 12 points, and the style is plain.

Default Font Output Variables

In the descriptions of these variables, "default font" refers to the font, size, and style defined by /W, /A, /SIZE, and /STYLE, and not any escape codes in styledTextStr.

MeasureStyledText returns information about the default font via the following output variables:

S_font	The name of the default font (before escape codes in *styledTextStr* take effect). If a substitute font would be used, the name of the substitute font is returned.
V_ascent	Height above the baseline of the default font. This will include some blank space above even the tallest characters.
V_descent	Height below the baseline of the default font.
V_fontSize	The default font size in points.
V_fontStyle	The default font style.
V_subscriptExtraHeight	Extra height needed to accomodate any subscript when using the default font.
V_superscriptExtraHeight	Extra height needed to accomodate any superscript when using the default font.

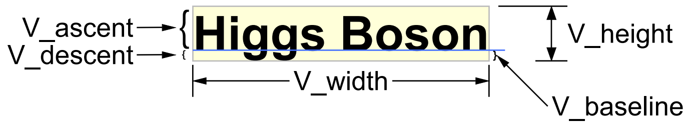
Styled Text Output Variables

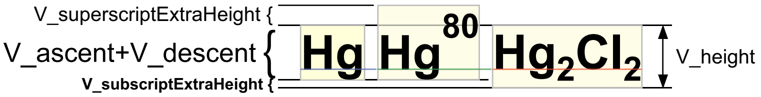
The MeasureStyledText operation returns information in the following variables:

V_width	The width in points of the text.
V_height	The height in points of the text.
V_baseline	The offset to the baseline of the styled text in points, as measured from the bottom of the text. If you omit /B or specify /B=0, this is is the offset to the last line's baseline. If you specify /B=1, this is is the offset to the first line's baseline.

MeasureStyledText Diagrams

These diagrams illustrate the meanings of the various output variables:





Example

See the Example section of the documentation for MeasureStyledText in the Igor Reference help file.

See Also

Annotation Escape Codes on page III-53 for a list of text formatting codes.

DefaultFont

Menu

```
Menu menuNameStr [, hideable, dynamic, contextualmenu]
```

The Menu keyword introduces a menu definition. You can use this to create your own menu, or to add items to a built-in Igor menu.

Use the optional *hideable* keyword to make the menu hideable using **HideIgorMenus**.

Use the optional dynamic keyword to cause Igor to re-evaluate the menu definition when the menu is used. This is helpful when the menu item text is provided by a user-defined function. See **Dynamic Menu Items** on page IV-129.

Use the optional contextualmenu keyword for menus invoked by **PopupContextualMenu**/N.

See Chapter IV-5, **User-Defined Menus** for further information.

min

```
min(num1, num2 [, num3, ... num200])
```

The min function returns the least value of *num1*, *num2*, ... *num200*.

If any parameter is NaN, the result is NaN.

Details

In Igor7 or later, you can pass up to 200 parameters. Previously min was limited to two parameters.

See Also

max, **limit**, **WaveMin**, **WaveMax**, **WaveMinAndMax**

MLLoadWave

```
MLLoadWave [flags] fileNameStr
```

The MLLoadWave operation loads data from the named Matlab MAT file into single 1D waves (vectors), multidimensional waves (matrices), numeric variables or string variables.

For background information, including configuration instructions, see **Loading Matlab MAT Files** on page II-163.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If LoadWave can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

If *fileNameStr* is omitted or is "" or the /I flag is used, MLLoadWave displays an Open File dialog in which you locate the file to be loaded.

Flags

/A[=*name*] Assign wave names using "wave" or *name*, if present, as the name or base name. Skips names already in use.

/B This flag is obsolete and is ignored. Previously it was required to tell MLLoadWave the byte order of the data in the file. MLLoadWave now determines the byte order automatically.

/C Loads columns from a Matlab matrix into an Igor 1D wave. Use /R to load rows.

/E Skips empty Matlab matrices.

/G Tells Igor to make numeric and string variables global when called from a macro. When called from a user-defined function or from the command line, variables are always created as globals.

/I Interactive. Displays the Open File dialog to get the path to the file.

/M=m m=1: Loads an entire Matlab matrix into an Igor 1D wave. This is the default if you omit /M.

m=2: Loads an entire Matlab matrix into an Igor multidimensional wave.

m=3: Loads an entire Matlab matrix into a transposed Igor multidimensional wave.

/M by itself is equivalent to /M=1.

Starting with Igor Pro 8.00, after loading a matrix that results in an Mx1 2D wave, MLLoadWave automatically redimensions the wave as an M-row 1D wave.

/N[=*name*] Assign wave names using "wave" or *name*, if present, as the name or base name. Overwrites existing waves if the name is already in use.

/O Overwrites existing waves and variables in case of a name conflict. If /O is omitted, MLLoadWave chooses names that don't conflict with existing objects.

/P=*pathName* Specifies the folder to look in for the specified file or folder. *pathName* is the name of an existing Igor symbolic path.

/Q Be quiet. Suppresses normal diagnostic messages.

/R Loads rows from a Matlab matrix into an Igor 1D wave. Use /C to load columns.

/S=s Controls how Matlab string data is loaded:

s=1 Skips Matlab string matrices.

s=2 Loads Matlab string matrices into Igor string variables. This is the default if /S is omitted.

s=3 Loads Matlab string matrices into Igor text waves.

/S by itself is equivalent to /S=1.

/T Displays the loaded waves in a new table.

/V Skips Matlab numeric variables (numeric matrices with one element).

/Y=y Specifies the number type of the numeric waves to be created. The allowed codes for y are:

2:	Single-precision floating point
4:	Double-precision floating point
32:	32-bit signed integer
16:	16-bit signed integer
8:	8-bit signed integer
96:	32-bit signed integer
80:	16-bit signed integer
72:	8-bit signed integer

/Z Interactive load. Displays a dialog presenting options for each Matlab matrix in the file.

MLLoadWave Wave Naming

If neither /A, /A[=name], /N, or N[=name] is used then the waves names are taken from the matrix name, as stored in the Matlab file.

When loading 1D waves, the /N flag instructs MLLoadWave to automatically name new waves "wave" (or *baseName* if /N=*baseName* is used) plus a number. The number starts from zero and increments by one for each wave loaded from the file. When loading multidimensional waves, *name* is used without an appended number.

The /A flag is like /N except that MLLoadWave skips names already in use.

If you specify /M=2 (load matrix into matrix) or /M=3 (load matrix into transposed matrix), MLLoadWave uses the name without appending any digits. For example, if you have a 5x3 matrix in a file and you tell MLLoadWave to load it as a matrix using the name "mat", MLLoadWave will name the matrix "mat". However, if you tell MLLoadWave to load the matrix as 3 1D waves, it will use "mat0", "mat1" and "mat2".

If the name that MLLoadWave would use when creating a wave or variable is in use for an object of the same type and if you use the overwrite flag, then it will overwrite the existing object. If you do not tell MLLoadWave to overwrite, it will choose a non-conflicting name. If the conflict is with an object of a different type or with an operation or function, MLLoadWave will also choose a non-conflicting name.

Loading Strings from Matlab Files

When loading Matlab strings into Igor, you can tell MLLoadWave to create Igor string variables or Igor text waves. For example, if you have a 2x8 string matrix, MLLoadWave can create two string variables (/S=2) or one text wave (/S=3) containing two elements.

When loading Matlab string data into an Igor wave, the Igor wave will be of dimension one less than the Matlab data set. This is because each element in a Matlab string data set is a single byte whereas each element in an Igor string wave is a string (any number of bytes).

Loading Numeric Variables from Matlab Files

MLLoadWave loads numeric matrices with one element into Igor numeric variables. It loads all other numeric matrices into Igor waves.

When called from a macro, MLLoadWave creates local numeric and string variables unless you use the /G flag which tells it to create global variables. When called from the command line or from a user-defined function, MLLoadWave always creates global variables. Macros should be avoided in new programming.

Automatic Redimensioning from 2D to 1D

Starting with Igor Pro 8.00, after loading a matrix that results in an Mx1 2D wave, MLLoadWave automatically redimensions the wave as an M-row 1D wave.

This automatic redimensioning not affect the naming of the wave. It is still named using the 2D rules explained above under MLLoadWave Wave Naming.

Loading 3D and 4D Data from Matlab Files

For a discussion of how MLLoadWave handles 3D and 4D Matlab data, see Numeric Data Loading Modes.

Specifying Loading Options for Each Matlab Matrix

The /Z flag instructs MLLoadWave to load each Matlab object (matrix, vector, variable, string) step by step. MLLoadWave presents a dialog for each Matlab object in the file. You can choose to load or skip the object. If you omit the /Z flag, MLLoadWave will load all objects in the file without presenting any dialogs.

Output Variables

MLLoadWave sets the following output variables:

`S_path`	File system path to the folder containing the file. This is a system file path (e.g., "hd:FolderA:FolderB:"), not an Igor symbolic path. The path uses Macintosh path syntax, even on Windows, and has a trailing colon.
`S_fileName`	Name of the loaded file.
`V_flag`	Number of waves created.
`V_flag1`	Number of Matlab data sets (2D, 3D, or 4D) loaded.
`V_flag2`	Number of waves created.
`V_flag3`	Number of numeric variables created.
`V_flag4`	Number of string variables created.
`S_waveNames`	Semicolon-separated list of the names of loaded waves.

Prior to MLLoadWave 5.50, the variables V_Flag1, V_Flag2, V_Flag3 and V_Flag4 were named V1_Flag, V2_Flag, V3_Flag and V4_Flag.

See Also

Symbolic Paths on page II-22

See **Loading Matlab MAT Files** on page II-163 for background information, including configuration instructions.

mod

`mod(`*`num`*`, `*`div`*`)`

The mod function returns the remainder when *num* is divided by *div*.

The mod function may give unexpected results when *num* or *div* is fractional because most fractional numbers can not be precisely represented by a finite-precision floating point value.

See Also

trunc, **gcd**

ModDate

`ModDate(`*`waveName`*`)`

The ModDate function returns the modification date/time of the wave.

Details

The returned value is a double precision Igor date/time value, which is the number of seconds from 1/1/1904. It returns zero for waves created by versions of Igor prior to 1.2, for which no modification date/time is available.

See Also

WaveModCount, **Secs2Date**, **Secs2Time**

Modify

`Modify`

We recommend that you use **ModifyGraph**, **ModifyTable**, **ModifyLayout**, or **ModifyPanel** rather than Modify. When interpreting a command, Igor treats the Modify operation as ModifyGraph, ModifyTable, ModifyLayout or ModifyPanel, depending on the target window. This does not work when executing a user-defined function.

ModifyBoxPlot

`ModifyBoxPlot [/W=`*`winName`*`] [keyword=`*`value`*`, keyword=`*`value`*`, ...]`

The ModifyBoxPlot operation modifies a box plot trace in the target or named graph. To create a box plot trace, see **AppendBoxPlot**. For a detailed discussion of box plots and the parts of a box plot, see **Box Plots** on page II-331.

ModifyBoxPlot was added in Igor Pro 8.00.

Parameters

ModifyBoxPlot parameters consist of keyword=value pairs. The trace keyword specifies the trace targeted by the subsequent keywords. For example, the command:

```
ModifyBoxPlot trace=trace0, boxFill=(49151,60031,65535)      // Light blue
```

sets the box fill for all datasets of the trace0 trace to light blue.

As of Igor Pro 9.00, you can modify a setting for a specific dataset of a specific trace by adding a zero-based dataset index in square brackets after the keyword. For example:

```
ModifyBoxPlot trace=trace0, boxFill[1]=(49151,65535,49151)  // Light green
```

This sets the box fill for the second dataset (index=1) to light green leaving the box fill for other datasets unchanged.

General Parameters

trace=*traceName*
Specifies the name of a box plot trace to be modified. An error results if the named trace is not a box plot trace. Without the trace keyword, ModifyBoxPlot uses the first trace in the graph, whether it is a box plot trace or not. But, see the instance keyword for an exception.

instance=*instanceNum*
The combination of trace and instance works the same as (*traceName#instanceNum*) for a ModifyGraph keyword.

The instance keyword without trace keyword accesses the *instanceNum*'th trace in the graph, just like [*traceNum*] used with a ModifyGraph keyword. See **Trace Names** on page II-282 and **Object Indexing** on page IV-20.

medianIsMarker[=*v*]
If *v* is omitted or is non-zero, the median is shown using a marker instead of with a line across the box part of the box plot.

notched[=*n*]
If *n* omitted or is non-zero, a notched box plot is drawn. The notches represent the 95 percent confidence limits of the median value.

outlierMethod=*m*

outlierMethod={*m*, *p1*, *p2*, *p3*, *p4*}

When the raw data values are drawn on the box plot, they are classified as normal data points, outliers and far outliers. There are four methods for classifying the values. Method 0 and 1 require no parameters and can use the outlierMethod=*m* format. Method 2 and 3 require extra parameters and require the extended format that uses curly braces:

m=0: Tukey's method (default): outliers are values outside of the inner fences and far outliers are values outside the outer fences. For details including the definition of the fences, see **Box Plots** on page II-331.

m=1: Outliers are any data values beyond the ends of the whiskers. With this method, there are no far outliers.

m=2: Outliers are values beyond the mean +- factor1*SD. Far outliers are values beyond the mean +- factor2*SD, were SD is the standard deviation. The factor1 is given by *p1* and factor2 is given by *p2*. The parameters *p3* and *p4* are not used.

m=3: Outliers and far outliers are completely specified by the parameters. *p1* sets the boundary for lower far outliers, *p2* for lower outliers, *p3* for upper outliers, and *p4* for upper far outliers. All four parameters are required.

The outlierMethod keyword was added in Igor Pro 9.00.

quartileMethod=*m*

Selects the method to be used in computing the quartiles (top and bottom of the boxes):

m=0:	Tukey's method (default)
m=1:	Minitab method
m=2:	Moore and McCabe method
m=3:	Mendenhall and Sincich method

See the discusion of **StatsQuantiles** /QM flag for details.

showData=*whatData*

Selects a subset of the raw data for each box plot in a trace to be displayed using markers. The value of *whatData* is a name:

All:	Show all data points (default)
None:	Show no data points
Outliers:	Show only outliers and far outliers
FarOutliers:	Show only far outliers

See **Box Plots** on page II-331 for more information about outliers.

By default, the marker for normal data points is a hollow circle 0.7 times the size of a normal trace marker, outliers are shown by a full-size filled circle, and far outliers are shown using a full-size filled box.

showFences[=*v*]

If *v* is omitted or non-zero, the fences are shown as dotted lines the same width as the boxes. See **Box Plots** on page II-331 for a discussion of fences. *v* defaults to 0.

showMean[=*v*]

If *v* is omitted or non-zero, the mean of the data is shown as a marker. *v* defaults to 0.

whiskerMethod=*m*

whiskerMethod={*m* [, *p1*, *p2*]}

The whiskers are drawn from the quartiles (top and bottom of the box) to some extreme value, as determined by whiskerMethod. Methods 0-5 do not take extra parameters, and can be specified using the whiskerMethod=*m* format. Methods 6 and 7 take extra parameters and require the extended format with curly braces.

The extended format was added in Igor Pro 9.00.

m=0:	The extreme data values (default)
m=1:	The inner fences
m=2:	The "adjacent" points - the last data points inside the inner fences
m=3:	One standard deviation away from the mean value
m=4:	The 9th and 91st percentiles
m=5:	The 2nd and 98th percentiles
m=6:	The lower whisker is drawn from the box to a percentile given by *p1*. The upper whisker is drawn from the box to a percentile given by *p2*. This method requires both *p1* and *p2*. Added in Igor Pro 9.00.
m=7:	Whisker ends are at values given by mean data value +- a factor times the standard deviation. The factor is given by *p1*. *p2* is not used for this method. Added in Igor Pro 9.00.

See **Box Plots** on page II-331 for a discussion of fences and percentiles.

Appearance Parameters

boxWidth=*w*

For a non-category X axis, boxWidth sets the width of the box showing the quartiles. If *w* is between zero and one, it is taken to be a fraction of the width of the plot rectangle. If *w* is greater than one, it is in points.

If the box plot is displayed using a category X axis, boxWidth is ignored. The box width is set the same as a category plot box and is affected by ModifyGraph barGap and catGap.

A new non-category box plot has box width set as boxWidth=1/(2**n*) where *n* is the number of datasets (the number of boxes) on the trace.

capWidth=*w*

The whiskers may optionally be terminated with a horizontal line of width controlled by the capWidth keyword. If *w* is between zero and one, it is a fraction of the box width. If *w* is greater than one, it is in points. If *w* is zero (default), no cap is drawn.

jitter=*j*

Applies a horizontal offset to each displayed data point in order to make it easier to see dense datasets. *j* controls the maximum offset applied to any data point, expressed as a fraction of the box width. The value of *j* may be greater than 1, but in general values less than 1 look better. The default is 0.7.

lineStyles={*boxStyle*, *whiskerStyle*, *medianStyle*[, *capStyle*]}

Set the line style used to draw the box, the whiskers, the median line, and optionally the whisker caps. See **Dashed Lines** on page III-496 for a description of line styles.

A line style of -1 uses the line style set by ModifyGraph lstyle.

All parameters default to -1.

lineThickness={*boxThickness*, *whiskerThickness*, *medianThickness*[, *capThickness*]}

Sets the line thickness used to draw the box, the whiskers, the median line, and optionally the whisker caps. A thickness less than zero uses the thickness set by the ModifyGraph lsize keyword. A zero thickness hides the corresponding element of the box plot.

All parameters default to -1.

markers={*dataMarker*, *outlierMarker*, *farOutlierMarker*[, *medianMarker*, *meanMarker*]}

Sets the marker number to be used for ordinary data points, outliers, far outliers, the median value if the medianIsMarker keyword was specified and the mean value if the showMean keyword was specified. See **Markers** on page II-291 for a complete list.

These parameters default to markers={8, 19, 16, 26, 27} (hollow circle, filled circle, filled square, X, horizontal diamond). Setting any of the parameters to -1 uses the corresponding default marker.

markersOnTop={*dataOnTop*, *medianOnTop*, *meanOnTop*}

By default, the data markers, median marker if enabled by medianIsMarker, and mean marker if enabled by showMean, are drawn below the box and whisker lines so that they don't obscure the lines. But some special effects require the markers to be on top.

Setting *dataOnTop* to 1 causes all raw data points (normal data points, outliers and far outliers) to be drawn above the box and whisker lines. *medianOnTop* and *meanOnTop* similarly control the drawing of median and mean markers.

The markersOnTop keyword was added in Igor Pro 9.00.

markerSizes={*dataSize*, *outlierSize*, *farOutlierSize*[, *medianSize*, *meanSize*]}

Sets the marker size for ordinary data points, outliers, far outliers, the median value if the medianIsMarker keyword has been specified and the mean value if the showMean keyword has been specified. A marker size of zero uses the marker size set by ModifyGraph msize times a scaling factor: the scaling factor is 2/3 for non-outlier points, 1 for outliers and the median and mean markers, and 4/3 for far outliers.

All parameters default to 0.

opaqueMarkers={*opaqueData*, *opaqueOutliers*, *opaqueFarOutliers*, *opaqueMedian*, *opaqueMean*}

Causes the interior of hollow markers to be drawn opaque, covering up items underneath. This keyword has a separate settings for normal data points, outliers, far outliers, the median marker (if medianIsMarker is enabled) and the mean marker (if showMean is enabled). Markers of a given type are opaque if the corresponding parameter is set to 1.

The opaqueMarkers keyword was added in Igor Pro 9.00.

Color Parameters

All colors are specified as (*r*,*g*,*b*[,*a*]) **RGBA Values**. Specify color=(0,0,0,0) to use the color set by ModifyGraph rgb.

boxColor=(*r*,*g*,*b*[,*a*]) Sets the outline color of the box part. The default color is black.

boxFill=(*r*,*g*,*b*[,*a*]) Sets the fill color of the box.

capColor=(*r*,*g*,*b*[,*a*]) Sets the color of the cap line, if present. The default color is black.

dataColor=(*r*,*g*,*b*[,*a*]) Sets the color of non-outlier data points. The default color is the trace color.

dataStrokeColor=(*r*,*g*,*b*[,*a*])
Sets the stroke color of non-outlier data points. The default color is black.

The dataStrokeColor keyword was added in Igor Pro 9.00.

dataFillColor=(*r*,*g*,*b*[,*a*])
Sets the fill color of non-outlier data points when the markers are hollow. The default color is white.

The dataFillColor keyword was added in Igor Pro 9.00.

farOutlierColor=(*r*,*g*,*b*[,*a*])
Sets the color of far outlier points. The default color is the trace color.

farOutlierStrokeColor=(*r*,*g*,*b*[,*a*])
Sets the stroke color of far outlier data points. The default color is black.

The farOutlierStrokeColor keyword was added in Igor Pro 9.00.

farOutlierFillColor=(*r*,*g*,*b*[,*a*])
Sets the fill color of far outlier data points when the markers are hollow. The default color is white.

The farOutlierFillColor keyword was added in Igor Pro 9.00.

meanColor=(*r*,*g*,*b*[,*a*])
Sets the color of the mean marker. The default color is the trace color.

meanStrokeColor=(*r*,*g*,*b*[,*a*])
Sets the stroke color of the mean marker, if showMean is enabled. The default color is black.

The meanStrokeColor keyword was added in Igor Pro 9.00.

meanFillColor=(*r*,*g*,*b*[,*a*])
Sets the fill color of the mean marker, if showMean is enabled. Applies when the marker is hollow. The default color is white.

The meanFillColor keyword was added in Igor Pro 9.00.

medianLineColor=(*r*,*g*,*b*[,*a*])
Sets the color of the median line. You can see the effect only if medianIsMarker is not set. The default color is black.

medianMarkerColor=(*r*,*g*,*b*[,*a*])
Sets the color of the median marker. You can see the effect only if medianIsMarker is set. The default color is the trace color.

medianStrokeColor=(*r*,*g*,*b*[,*a*])
Sets the stroke color of the median marker which is displayed if medianIsMarker is enabled. The default color is black.

The medianStrokeColor keyword was added in Igor Pro 9.00.

medianFillColor=(*r*,*g*,*b*[,*a*])
Sets the fill color of the median marker which is displayed if medianIsMarker is enabled. Applies when the marker is hollow. The default color is white.

The medianFillColor keyword was added in Igor Pro 9.00.

outlierColor=(*r*,*g*,*b*[,*a*])
Sets the color of the markers for normal outliers. The default color is the trace color.

outlierStrokeColor=(*r*,*g*,*b*[,*a*])
Sets the stroke color of outlier data points. The default color is black.

The outlierStrokeColor keyword was added in Igor Pro 9.00.

outlierFillColor=(*r*,*g*,*b*[,*a*])
Sets the fill color of outlier data points when the markers are hollow. The default color is white.

The outlierFillColor keyword was added in Igor Pro 9.00.

whiskerColor=(*r*,*g*,*b*[,*a*])
Sets the color the whisker lines. The default color is black.

Box Plot Per-Data-Point Marker Settings

You can override the basic settings for data point marker color, marker style and marker size using marker settings waves containing per-data-point settings. This feature was added in Igor Pro 9.00.

You can apply per-data-point marker settings to an entire trace (to all datasets comprising a trace) or to a specific dataset of a trace. For example:

```
Function DemoBoxPlotPerPointMarkerSettings()
   Make/O box0={1,2,3,4,5}, box1={2,3,4,5,6}, box2={3,4,5,6,7}
   String title = "Box Plot Per Point Marker Settings"
   Display/W=(557,99,948,310)/N=BoxPlotPerPointPlot as title

   // Create a trace named trace0 with three datasets: box0, box1, box2
   AppendBoxPlot/TN=trace0 box0,box1,box2

   // Set the marker and marker size for all datasets of trace trace0
   ModifyBoxPlot trace=trace0, markers={18,-1,-1}    // 18=Diamond
   ModifyBoxPlot trace=trace0, markerSizes={5,5,5}

   // Set the per-data-point marker for all datasets of trace0
   Make/O boxMarkers = {15,16,17,18,19}
   ModifyBoxPlot trace=trace0, dataMarkerWave=boxMarkers

   // Set the per-data-point marker for dataset box1 only
   Make/O boxMarkersForBox1 = {32,33,34,35,36}
   ModifyBoxPlot trace=trace0, dataMarkerWave[1]=boxMarkersForBox1
End
```

Usually a marker settings wave will have the same number of rows as there are data points in a given dataset, but that is not required. If there are fewer rows in the settings wave than in the dataset, the extra data points retain their basic settings. If there are more rows in the settings wave than in the dataset, the extra settings wave points are not used.

See **Making Each Data Point Look Different** on page II-343 for more information and examples.

dataColorWave [=*colorWave*]

Sets *colorWave* to override data point marker color on a point-by-point basis. The wave must be a 3 or 4 column wave containing red, green, blue and optionally alpha values.

If you omit "=*colorWave*", any previous marker color wave setting is cleared.

The dataColorWave keyword was added in Igor Pro 9.00.

dataMarkerWave [=*markerWave*]

Sets *markerWave* to override data point markers on a point-by-point basis. The values in *markerWave* are standard graph marker numbers. See **Markers** on page II-291 for a table of the markers and the associated marker numbers. The marker numbers are clipped to a valid range.

If you omit "=*markerWave*", any previous marker wave setting is cleared.

The dataMarkerWave keyword was added in Igor Pro 9.00.

dataSizeWave [=*markerSizeWave*]

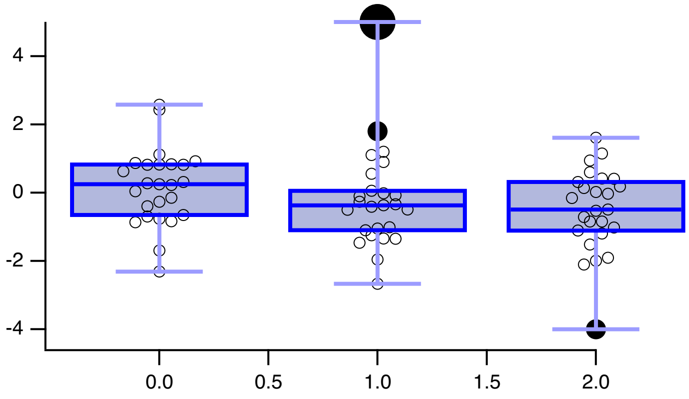
Sets *markerSizeWave* to override data point marker size on a point-by-point basis. The values in *markerSizeWave* are clipped to the range [0,200].

If you omit "=*markerSizeWave*", any previous marker size wave setting is cleared.

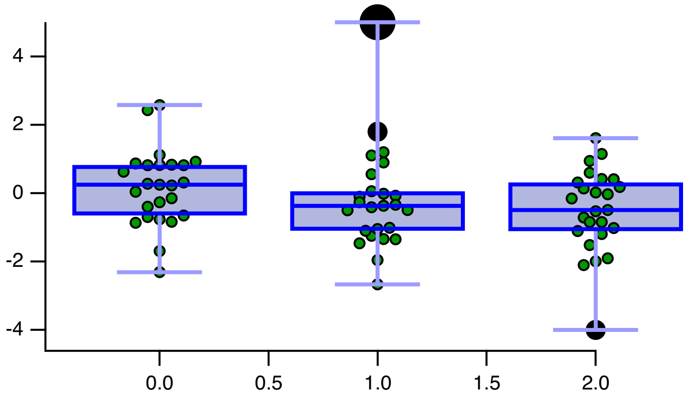
The dataSizeWave keyword was added in Igor Pro 9.00.

Example

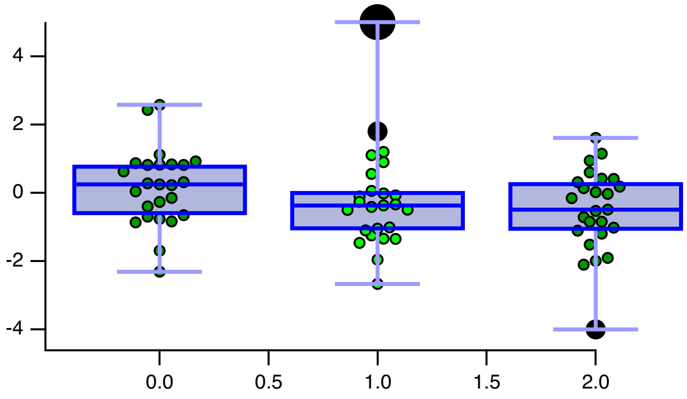
```
Make/O/N=(25,3) multicol            // A three-column wave with 25 rows
SetRandomSeed(.4)                   // For reproducible "randomness"
multicol = gnoise(1)                // Three normally-distributed datasets
multicol[20][1] = 5                 // A "far" outlier
multicol[13][2] = -4                // An outlier
Display; AppendBoxPlot multicol
ModifyGraph lSize=2
ModifyBoxPlot markers={8,19,19}
ModifyBoxPlot markerSizes={3,5,9}
ModifyBoxPlot capWidth=0.5
ModifyBoxPlot boxColor=(0,0,65535)
ModifyBoxPlot medianLineColor=(0,0,65535)
ModifyBoxPlot whiskerColor=(40000,40000,65535)
ModifyBoxPlot capColor=(40000,40000,65535)
ModifyBoxPlot boxFill=(0,0,65535,20000)
ModifyBoxPlot dataColor=(0,0,0)
ModifyBoxPlot outlierColor=(0,0,0)
ModifyBoxPlot farOutlierColor=(0,0,0)
ModifyBoxPlot jitter=0.75
```



```
ModifyBoxPlot trace=multicol,markerThick={1,0,0},markersFilled={1,0,0,0,0}
ModifyBoxPlot trace=multicol,markerThick={1,0,0},dataFillColor=(2,39321,1)
```



```
ModifyBoxPlot trace=multicol,dataFillColor[1]=(0,65535,0)
```



See Also

AppendBoxPlot, **AddWavesToBoxPlot**, **ModifyGraph (traces)**

Box Plots on page II-331

ModifyBrowser

`ModifyBrowser [/M]` `[`***`keyword = value`***` [, `***`keyword = value`***` …]]`

The ModifyBrowser operation modifies the state of the Data Browser according to the specified keywords.

Documentation for the ModifyBrowser operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "ModifyBrowser"
```

ModifyCamera

`ModifyCamera` `[`***`flags`***`]` **`[`*****`keywords`***`]`

The ModifyCamera operation modifies the properties of a camera window.

Documentation for the ModifyCamera operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "ModifyCamera"
```

ModifyContour

`ModifyContour` `[`**`/W=`*****`winName`***`]`***`contourInstanceName`***`, `***`keyword`*****`=`*****`value`***
`[, `***`keyword`*****`=`*****`value`***`…]`

The ModifyContour operation modifies the number, Z value and appearance of the contour level traces associated with *contourInstanceName*.

contourInstanceName is a name derived from the name of the wave that provides the Z data values. It is usually just the name of the wave, but may have #1, #2, etc. added to it in the unlikely event that the same Z wave is contoured more than once in the same graph.

contourInstanceName can also take the form of a null name and instance number to affect the instanceth contour plot. That is,

```
ModifyContour ''#1
```

modifies the appearance of the second contour plot in the top graph, no matter what the contour plot names are. Note: Two single quotes, not a double quote.

The number of contour level traces and their Z values are set by the *autoLevels*, *manLevels,* and *moreLevels* keywords, described in the **Parameters** section. Normally, you will use either *autoLevels* or *manLevels,* and then optionally generate additional levels using *moreLevels*.

Parameters

Each parameter has the syntax

```
keyword = value
```

and is applied to all of the contour level traces associated with *contourInstanceName*.

To modify an individual contour level trace, use **ModifyGraph**.

allocationFactor=*factor*	In rare situations that usually involve spatial degeneracy, the Voronoi interpolation algorithm may need additional memory. You can use the allocationFactor keyword with an integer factor greater than 1 to increase the memory allocation. allocationFactor was added in Igor Pro 9.00.
autoLevels= {*minLevel*, *maxLevel*, *numLevels*}	

Controls automatic determination of contour levels.

If *numLevels* is zero, no automatic levels are generated. If it is nonzero, it specifies the desired number of automatic contour levels.

minLevel specifies the minimum contour level and *maxLevel* specifies the maximum contour level. The values that you specify are an approximate guide for Igor to use in determining the actual levels.

However, if *minLevel* or *maxLevel* is * (asterisk symbol), Igor uses the minimum or maximum value of the Z data for the corresponding contour level.

Using the autoLevels keyword cancels the effect of any previous autoLevels or manLevels keyword.

When you first append a contour plot to a graph, default contour levels are generated by the default setting `autoLevels={*,*,11}`.

boundary=*b* Draws an outline around the XY domain of the contour data. For a matrix, this draws a rectangle showing the minimum and maximum X and Y values. For XYZ triples, the outline is a polygon enclosing the outside edges of the Delaunay Triangulation. Like the contour lines, the boundary is drawn using a graph trace, whose name is usually something like "`contourInstanceName = boundary`".

b=0: Hides the data boundary (default).

b=1: Shows the data boundary.

cIndexFill= *matrixWave* Sets contour fills to use a color index wave when automatic fill is on (see the fill keyword).

cIndexFill works the same as the cIndexLines keyword which controls the colors of the contour level traces.

See **Contour Fills** on page II-373 for more information.

cIndexFill was added in Igor Pro 7.00.

cIndexLines= *matrixWave*

Sets the Z value mapping mode such that contour line colors are determined by doing a lookup in the specified matrix wave.

matrixWave is a 3 column wave that contains red, green, and blue values from 0 to 65535. (The matrix can actually have more than three columns. Any extra columns are ignored.)

The color for a the contour line at Z=*z* is determined by finding the RGB values in the row of *matrixWave* whose scaled X index is *z*. In other words, the red value is *matrixWave*(*z*)[0], the green value is *matrixWave*(*z*)[1] and the blue value is *matrixWave*(*z*)[2].

If *matrixWave* has default X scaling, where the scaled X index equals the point number, then row 0 contains the color for Z=0, row 1 contains the color for Z=1, etc.

If you use cIndexLines, you must not use ctabLines or rgbLines in the same command.

cTabFill= {*zMin*, *zMax*, *ctName*, *mode*}

Sets contour fills to use a color table when automatic fill is on (see the fill keyword).

cTabFill works the same as the ctabLines keyword which controls the colors of the contour level traces.

See **Contour Fills** on page II-373 for more information.

cTabFill was added in Igor Pro 7.00.

ctabLines={*zMin*, *zMax*, *ctName*, *mode*}

Sets the Z value mapping mode such that contour line colors are determined by doing a lookup in the specified color table. *zMin* is mapped to the first color in the color table. *zMax* is mapped to the last color. Z values between the min and max are linearly mapped to the colors between the first and last in the color table.

You can enter * (an asterisk) for *zMin* and *zMax*, which uses the minimum and maximum Z values of the data. The default is {*,*,Rainbow}.

Set parameter *mode* to 1 to reverse the color table; zero or missing does not reverse the color table.

ctName can be any color table name returned by the **CTabList** function, such as Grays or Rainbow (see **Image Color Tables** on page II-392) or the name of a 3 column or 4 column color table wave (see **Color Table Waves** on page II-399).

A color table wave name supplied to `ctabLines` must not be the name of a built-in color table (see **CTabList**). A 3 column or 4 column color table wave must have values that range between 0 and 65535. Column 0 is red, 1 is green, and 2 is blue. In column 3 a value of 65535 is opaque, and 0 is fully transparent.

If you use ctabLines, you must not use cIndexLines or rgbLines in the same command.

equalVoronoiDistances=*e*

Normally the x range and y range of the data are each normalized to a 0-1 range separately to generate the Voronoi triangulation. Voronoi triangulation is a distance-based ("nearest neighbor") algorithm that may benefit from scaling the X and Y ranges together to avoid numerical problems that occur when the triangles become very thin because of widely differing x and y ranges.

e=0: The x and y ranges are scaled individually to the 0-1 range (default).

e=1: The x and y ranges are scaled so that that maximum range of x or y is scaled to the 0-1 range, and the other is proportionally smaller. For example, if yMax-yMin = 1000 and xMax-xMin = 5, then the y range is scaled to 0-1 and the y range is scaled to 5/1000 = 0 - 0.005.

The equalVoronoiDistances keyword is allowed only for XYZ contour plots.

fill=*f* Controls the automatic filling of contour levels.

f=0: Turns automatic fill off. Default.

f=1: Turns automatic fill on.

See **Contour Fills** on page II-373 for more information.

fill was added in Igor Pro 7.00.

interpolate=*i*

XYZ contours can be interpolated to increase the apparent resolution, resulting in smoother contour lines.

This keyword is allowed only for XYZ contours, created by **AppendXYZContour**.

i=0: Linear interpolation (default). This means that only the original Delaunay triangulation generates contour lines.

i=1: Four times the resolution generates a smoother set of contour lines. As expected, this takes longer than Linear interpolation.

i=2: Sixteen times the resolution generates a much smoother set of contour lines. This is rather slow.

The interpolate parameter can be up to 8. Each time you increase *i* by one, you quadruple the apparent resolution and get smoother contour lines at the expense of computation time. Values of *i* greater than two are impractical because of the computation time required.

labelBkg=(*r*,*g*,*b*[,*a*])

Sets the background color for all contour level labels to the specified color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

labelBkg=*b*

Controls the background color of contour labels.

b=0: Uses each label's individual background color, as set via the Modify Annotation dialog.

b=1: Makes all contour level labels transparent.

b=2: Uses the plot area background color as the label background color (default).

b=3 Uses the window background color as the label background color.

labelDigits=*d*

d is the number of digits after the decimal point when using labelFormat=3 or labelFormat=5.

labelFont=*fontName*

Default; specifies the font to use for contour level labels. If you pass "" for *fontName*, it will use the graph font (set via the Modify Graph dialog) for contour labels.

labelFormat=*l*

Controls the formatting of contour labels. See the **printf** operation for a discussion of formatting.

l=0: Uses general format that is suitable for most data. This is equivalent to "%<sigDigits>g".

l=1: Uses integer format, equivalent to "%<sigDigits>d". This rounds fractional values.

l=3: Uses fixed point format, equivalent to "%<decimalDigits>f".

l=5 Uses exponential format, equivalent to "%<decimalDigits>e".

labelFSize=*s*

Specifies the font size of contour labels in points. For example, use labelSize=12 for 12 point type. The default value is 0, which chooses the size automatically based on the size of the graph.

labelFStyle=*n*

n is a bitwise parameter with each bit controlling one aspect of the font style for the contour level labels. The default is 0, plain text.

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

labelHV=*hv* Specifies the contour label orientation.

If *hv* is 3, 4, 5, or 6, the contour label's text rotates whenever it is redrawn, usually when the underlying contour data changes, the graph is resized, or the label is reattached to a new contour trace point.

hv=0: Horizontal contour level labels.

hv=1: Vertical contour level labels.

hv=2: Horizontal or vertical contour level labels, depending on the slope of the contour line.

hv=3: Tangent to the contour line.

hv=4: Tangent to the contour line, snaps to vertical or horizontal if within 2 degrees of vertical or horizontal (default).

hv=5: Perpendicular to the contour line.

hv=6: Perpendicular to the contour line, snaps to vertical or horizontal if within 2 degrees of vertical or horizontal.

labelRGB=(*r*,*g*,*b*[,*a*]) Sets the text color for all contour level labels. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

labels=*l* Controls the display of contour labels.

l=0: Hides contour level labels.

l=1: Leaves any contour level labels in place but stops updating them and stops generation of new labels.

l=2: Generates or updates labels for the existing contour levels and window size when the command executes, but disables further updating of labels when window size or contour plot changes. This is the recommended setting if updating the labels takes long enough to annoy you.

l=3: Default; generates labels for all contour levels whenever the contoured data changes but not when the window size changes. If you resize the graph, the labels may overlap or be too sparse.

l=4: Generates labels for all contour levels whenever the contoured data, contour levels, axis range, or the graph size changes. (Actually, there are too many causes to list here. If all this update annoys you, use labels=2 “update once, now”.)

labelSigDigits=*d* *d* is the number of significant digits when labelFormat=0 is used.

logLines= 1 or 0 0 sets the default linearly-spaced contour line colors.

1 turns on logarithmically-spaced line colors. This requires that the contour levels values be greater than 0 to display correctly.

Affects line color only when the cIndexLines or ctabLines parameter is used.

logLines does not affect the contour levels. To assign logarithmically-spaced contour levels, use the moreLevels parameter and disable autoLevels, for example:

```
ModifyContour ''#0, autoLevels={*,*,0} // No auto levels
ModifyContour ''#0, moreLevels=0
ModifyContour ''#0, moreLevels={1e-07,1e-06,1e-05,1e-04}
```

manLevels= {*firstLevel*, *increment*, *numLevels*}

Explicitly specifies contour levels. ModifyContour will generate *numLevels* contour levels, evenly spaced starting from *firstLevel* and stepping by *increment*.

manLevels cancels the effect of any previous manLevels or autoLevels settings.

manLevels= *manLevelsWave*

Explicitly specifies contour levels. ModifyContour will generate contour levels at the values in *manLevelsWave*.

manLevels cancels the effect of any previous manLevels or autoLevels settings.

moreLevels= {*level*, *level* …}

Explicitly specifies contour levels. ModifyContour will generate a contour trace for each of the listed levels. The maximum number of levels that you can specify in a single command is the 50. However, you can concatenate any number of ModifyContour moreLevels commands. moreLevels adds levels in addition to any specified by manLevels or autoLevels. It does not override other parameters.

moreLevels=0: Removes all levels generated by previous moreLevels settings.

nullValue=*zValue*

This keyword only affects the behavior of the **ContourZ** function. It is allowed only for XYZ contours, created by **AppendXYZContour**.

By default, ContourZ treats data outside the domain of the contour as NaN and so returns NaN if you ask for a contour value outside that domain.

The nullValue keyword allows you to change the default behavior to make ContourZ treat values outside the domain as the specified zValue.

nullValueAuto

This keyword only affects the behavior of the **ContourZ** function. It is allowed only for XYZ contours, created by **AppendXYZContour**.

nullValueAuto acts like nullValue=*zValue* with *zValue* automatically set to the minimum value in the Z wave minus 1.

See the nullValue keyword for details.

To turn nullValueAuto off and return the contour to the default state, execute:

```
ModifyContour <contourInstanceName>, nullValue=NaN
```

perturbation=*p*

Enable or disable perturbation (alteration) of the x and y values by a miniscule amount to improve the natural neighbor triangulation of XYZ contours.

p=0: Disables perturbation, preserving the original x and y values unchanged.

p=1: Enables x/y perturbation (default). The values are shifted by random values less than +/-0.000005 times the x and y domain extents.

You can observe the perturbed x/y coordinates in the triangulation trace added by ModifyContour triangulation=1.

The perturbation keyword is allowed only for XYZ contour plots.

rgbFill=(*r*,*g*,*b*[,*a*])

Specifies red, green, and blue values for all contour fills. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

If you use rgbFill, you must not use cIndexFill or ctabFill in the same command.

rgbLines=(*r*,*g*,*b*[,*a*])

Specifies red, green, and blue values for all contour lines. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

If you use rgbLines, you must not use cIndexLines or ctabLines in the same command.

triangulation=*t*

Draws the Delaunay Triangulation. As part of the XYZ contouring algorithm, the XY domain is subdivided into triangles in a process called Delaunay Triangulation. Like the contour lines, the triangulation is drawn using a graph trace, whose name is usually something like “contourInstanceName =triangulation”.

The triangulation keyword is allowed only for XYZ contours, created by **AppendXYZContour**.

t=0: Hides the Delaunay triangulation (default).

t=1: Shows the Delaunay triangulation.

update=*u* Sets the type of updating of contour traces when the data or contour settings change.

u=0: Turns off dynamic updates, which might be advisable if updates take a long time.

u=1: Updates the contours only once, or until you next execute an update=1 command.

u=2: Updates are automatic (default).

u=3 Marks the contour plot as having been updated once (*u*=1) already. This option is used in recreation macros to prevent an extra redraw of a graph saved with *u*=1 update mode in effect.

If you use it in a command, the result is similar to *u*=0, but the Modify Contour Appearance dialog will automatically select “update once, now” from the Update Contours pop-up menu.

xymarkers=*x* Controls the visibility of XY markers.

x=0: Hides markers showing XY coordinates of the Z data (default).

x=1: Displays markers showing XY coordinates of Z data. Initially, this uses marker number zero. You can change this using the Modify Trace Appearance dialog.

Flags

/W=*winName* Applies to contours in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

See Also

AppendMatrixContour and **AppendXYZContour**.

References

Watson, David F., *nngridr - An Implementation of Natural Neighbor Interpolation*, Dave Watson Publisher, Claremont, Australia, 1994.

ModifyControl

`ModifyControl [/Z] ctrlName [keyword = value [, keyword = value …]]`

The ModifyControl operation modifies the named control. ModifyControl works on any kind of existing control. To modify multiple controls, use **ModifyControlList**.

Parameters

ctrlName specifies the name of the control to be created or changed. The control must exist.

Keywords

The following keyword=value parameters are supported:

activate	align	appearance	bodywidth	disable	fColor
focusRing	font	fSize	fStyle	help	labelBack
noproc	pos	proc	rename	size	title
userdata	valueBackColor	valueColor	win		

Coordinates are in **Control Panel Units**.

For details on these keywords, see the documentation for **SetVariable** on page V-854.

The following keywords are not supported:

mode popmatch popvalue value variable

Flags

/Z No error reporting.

Details

Use ModifyControl to move, hide, disable, or change the appearance of a control without regard to its kind

Example

Here is a **TabControl** procedure that shows and hides all controls in the tabs appropriately, without knowing what kind of controls they are.

The "trick" here is that all controls that are to be shown within particular tab *n* have been assigned names that end with "_tab*n*" such as "_tab0" and "_tab1":

```
Function TabProc(ctrlName,tabNum) : TabControl
   String ctrlName
   Variable tabNum

   String curTabMatch= "*_tab"+num2istr(tabNum)

   String controls= ControlNameList("")
   Variable i, n= ItemsInList(controls)
   for(i=0; i<n; i+=1)
      String control= StringFromList(i, controls)
      Variable isInATab= stringmatch(control,"*_tab*")
      if( isInATab )
         Variable show= stringmatch(control,curTabMatch)
         ControlInfo $control                    // gets V_disable
         if( show )
            V_disable= V_disable & ~0x1          // clear the hide bit
         else
            V_disable= V_disable | 0x1           // set the hide bit
         endif
         ModifyControl $control disable=V_disable
      endif
   endfor
   return 0
End

// Action procedures which enable or disable the buttons
Function Tab1CheckProc(ctrlName,enableButton) : CheckBoxControl
   String ctrlName
   Variable enableButton

   ModifyControl button_tab1, disable=(enableButton ? 0 : 2 )
End

Function Tab0CheckProc(ctrlName,enableButton) : CheckBoxControl
   String ctrlName
   Variable enableButton

   ModifyControl button_tab0, disable=(enableButton ? 0 : 2 )
End

// Panel macro that creates a TabControl using TabProc
Window TabbedPanel() : Panel
   PauseUpdate; Silent 1             // building window...
   NewPanel /W=(381,121,614,237) as "Tab Demo"
   TabControl tab, pos={12,9},size={205,91},proc=TabProc,tabLabel(0)="Tab 0"
   TabControl tab, tabLabel(1)="Tab 1",value= 0
   Button button_tab0, pos={54,39},size={110,20},disable=2
   Button button_tab0, title="Button in Tab0"
   Button button_tab1, pos={54,63},size={110,20},disable=1
   Button button_tab1, title="Button in Tab1"
   CheckBox check1_tab1, pos={51,41}, size={117,14}, disable=1, value= 1
   CheckBox check1_tab1, proc=Tab1CheckProc, title="Enable Button in Tab 1"
   CheckBox check0_tab0, pos={51,73}, size={117,14}, proc=Tab0CheckProc
   CheckBox check0_tab0, value= 0, title="Enable Button in Tab 0"
EndMacro
```

Run TabbedPanel to create the panel. Then click on "Tab 0" and "Tab 1" to run TabProc.

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

Related functions **ModifyControlList** and **ControlNameList**.

The **Button**, **Chart**, **CheckBox**, **GroupBox**, **ListBox**, **PopupMenu**, **SetVariable**, **Slider**, **TabControl**, **TitleBox**, and **ValDisplay** controls.

ModifyControlList

`ModifyControlList [/Z] listStr [, keyword = value]…`

The ModifyControlList operation modifies the controls named in the *listStr* string expression. ModifyControlList works on any kind of existing control.

Parameters

listStr is a semicolon-separated list of names in a string expression. The expression can be an explicit list of control names such as "button0;checkbox1;" or it can be any string expression such as a call to the ControlNameList string function:

```
ModifyControlList ControlNameList("",";","*_tab0") disable=1
```

The controls must exist.

Keywords

The following keyword=value parameters are supported:

activate	align	appearance	bodywidth	disable	fColor
focusRing	font	fSize	fStyle	help	labelBack
noproc	pos	proc	rename	size	title
userdata	valueBackColor	valueColor	win		

Coordinates are in **Control Panel Units**.

For details on these keywords, see the documentation for **SetVariable** on page V-854.

The following keywords are not supported:

mod	popmatch	popvalue	value	variable

Flags

/Z	No error reporting.

Details

Use ModifyControlList to move, hide, disable, or change the appearance of multiple controls without regard to their kind.

If *listStr* contains the name of a nonexistent control, an error is generated.

if *listStr* is `""` (or any list element in *listStr* is `""`), it is ignored and no error is generated.

Example

Here is the **TabControl** procedure example from **ModifyControl** rewritten to use ModifyControlList. It shows and hides all controls in the tabs appropriately, without knowing what kind of controls they are, but the code is simpler. This method does not, however, preserve the enable bit when a control is hidden.

The “trick” here is that all controls that are to be shown within particular tab *n* have been assigned names that end with “_tab*n*” such as “_tab0” and “_tab1”:

```
// Action procedure
Function TabProc2(ctrlName,tabNum) : TabControl
	String ctrlName
	Variable tabNum

	String controlsInATab= ControlNameList("",";","*_tab*")
	String curTabMatch= "*_tab"+num2istr(tabNum)
	String controlsInCurTab= ListMatch(controlsInATab, curTabMatch)
	String controlsInOtherTabs=ListMatch(controlsInATab,"!"+curTabMatch)

	ModifyControlList controlsInOtherTabs disable=1			// hide
	ModifyControlList controlsInCurTab disable=0			// show

	return 0
End

// Panel macro that creates a TabControl using TabProc2():
Window TabbedPanel2() : Panel
	PauseUpdate; Silent 1				// building window…
	NewPanel /W=(35,208,266,374) as "Tab Demo"
	TabControl tab,pos={12,9},size={205,140},proc=TabProc2
	TabControl tab,tabLabel(0)="Tab 0"
	TabControl tab,tabLabel(1)="Tab 1",value= 0
	Button button_tab0,pos={26,43},size={110,20},title="Button in Tab0"
	Button button2_tab0,pos={26,74},size={110,20},title="Button in Tab0"
	Button button3_tab0,pos={26,106},size={110,20},title="Button in Tab0"
	Button button_tab1,pos={85,43},size={110,20},title="Button in Tab1"
	Button button2_tab1,pos={85,75},size={110,20},title="Button in Tab1"
	Button button3_tab1,pos={84,108},size={110,20},title="Button in Tab1"
	ModifyControlList ControlNameList("",";","*_tab1") disable=1
EndMacro
```

Run TabbedPanel2 and then click on "Tab 0" and "Tab 1" to run TabProc2.

See Also

See Chapter III-14, **Controls and Control Panels** for details about control panels and controls.

Related functions **ModifyControl** and **ControlNameList**.

The **Button**, **Chart**, **CheckBox**, **GroupBox**, **ListBox**, **PopupMenu**, **SetVariable**, **Slider**, **TabControl**, **TitleBox**, and **ValDisplay** controls.

ModifyFreeAxis

ModifyFreeAxis [**/W=*winName***] ***axisName***, **master=*mastName***
[, **hook=*funcName***]

The ModifyFreeAxis operation designates the free axis (created with **NewFreeAxis**) to follow a controlling axis from which it gets axis range and units information. The free axis updates whenever the controlling axis changes. The axis limits and units can be modified by a user hook function.

Parameters

axisName is the name of the free axis (which must have been created by **NewFreeAxis**).

masterName is the name of the master axis controlling *axisName*.

funcName is the name of the user function that modifies the limits and units properties of the axis. If *funcName* is `$""`, the named hook function is removed.

Flags

/W=*winName*	Modifies *axisName* in the named graph window or subwindow. If /W is omitted the command affects the top graph window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The free axis can also be designated to call a user-defined hook function that can modify limits and units properties of the axis. The hook function must be of the following form:

```
Function MyAxisHook(info)
	STRUCT WMAxisHookStruct &info

	<code to modify graph units or limits>
	return 0
End
```

where WMAxisHookStruct is a built-in structure with the following members:

WMAxisHookStruct Structure Members

Member	Description
char win[MAX_WIN_PATH+1]	Host (sub)window
char axName[MAX_OBJ_NAME+1]	Name of the axis
char mastName[MAX_OBJ_NAME+1]	Name of controlling axis or ""
char units[MAX_UNITS+1]	Axis units
double min, max	Axis range minimum and maximum values

The constants used to size the char arrays are internal to Igor and are subject to change in future versions.

The hook function is called when refreshing axis range information (generally early in the update of a graph). Your hook must never kill a graph or an axis.

Example

This example demonstrates how to program a free axis hook function, whose most important task is to change the values of info.min and info.max to alter the axis range of the free axis. The example free axis displays Fahrenheit values for data in Celsius.

```
Function CentigradeAndFahrenheit()
    Make/O/N=20 temperatures = -2+p/3+gnoise(0.5)  // sample data
    Display temperatures// default left axis will indicate data's centigrade range
    String graphName = S_name
    Label/W=$graphName left "°C"
    ModifyGraph/W=$graphName zero(left)=1
    Legend/W=$graphName

    // make a right axis whose range will be Fahrenheit
    NewFreeAxis/R/O/W=$graphName fahrenheit
    ModifyGraph/W=$graphName freePos(fahrenheit)={0,kwFraction},lblPos(fahrenheit)=43
    Label/W=$graphName fahrenheit "°F"

    ModifyFreeAxis/W=$graphName fahrenheit, master=left, hook=CtoF_FreeAxisHook
    // NOTE master=left part which makes the "free" axis
    // actually a "slave" to the left ("master") axis.
End

Function CtoF_FreeAxisHook(info)
    STRUCT WMAxisHookStruct &info

    GetAxis/Q/W=$info.win $info.mastName    // get master axis range in V_min, V_Max
    Variable minF = V_min*9/5+32
    Variable maxF = V_max*9/5+32

//  SetAxis/W=$info.win $info.axName, minF, maxF
//  SetAxis here is fruitless. These values get overwritten by Igor
//  after reading info.min and info.max, which we now set:
    info.min = minF           // new min for free axis
    info.max= maxF            // new max for free axis
    return 0
End
```

See Also

The **SetAxis**, **KillFreeAxis**, and **NewFreeAxis** operations.

The **ModifyGraph (axes)** operation for changing other aspects of a free axis.

ModifyGizmo

ModifyGizmo [***flags***] ***keyword*** [***=value***]

The ModifyGizmo operation changes Gizmo properties.

Documentation for the ModifyGizmo operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "ModifyGizmo"
```

ModifyGraph *(general)*

ModifyGraph [**/W=*winName*/Z**] ***key=value*** [, ***key=value***]…

The ModifyGraph operation modifies the target or named graph. This section of ModifyGraph relates to general graph window settings.

Parameters

expand=*e*
Specifies the onscreen expansion (or magnification) factor of a graph. *e* may be zero or 0.125 to 8 times expansion.

Graph magnification affects only base graphs (not subwindowed graphs), and it affects only the onscreen display; it has **no** effect on graph exporting or printing.

When magnification changes, the graph window will automatically resize except for negative values, which are used in recreation macros where the size is already correct.

frameInset= *i*
Specifies the number of pixels by which to inset the frame of the graph subwindow.

frameStyle= *f*
Specifies the frame style for a graph subwindow.

f=0:	None.
f=1:	Single.
f=2:	Double.
f=3:	Triple.
f=4:	Shadow.
f=5:	Indented.
f=6:	Raised.
f=7:	Text well.

The last three styles are fake 3D and will look good only if the background color of the enclosing space and the graph itself is a light shade of gray.

gfMult=*f*
Multiplies font and marker size by *f* percent. Clipped to between 25% and 400%; it is applied after all other font and marker size calculations.

gFont=*fontStr*
Specifies the name of the default font for the graph, overriding the normal default font. The normal default font for a subgraph is obtained from its parent while a base graph uses the value set by the **DefaultFont** operation.

gfSize=*gfs*
Sets the default size for text in the graph. Normally, the default size for text is proportional to the graph size; gfSize will override that calculation as will the gfRelSize method. Use a value of -1 to make a subgraph get its default font size from its parent.

gfRelSize=*pct*
Specifies the percentage of the graph size to use in calculating a default size for text in the graph. This overrides the normal method for setting default font size as a function of graph size. When used, the default marker size is set to one third the font size. Use a value of 0 to revert to the default method.

gmSize=*gms*
Sets the default size for markers in the graph. Use a value of -1 to make a subgraph get its default marker size from its parent.

height=*heightSpec*
Sets the height for the graph area. See the **Examples**.

swapXY=*s* Sets the orientation of the X and Y axes.

s=0: Normal orientation of X and Y axes.

s=1: Swap X and Y values to plot Y coordinates versus the horizontal axes and X coordinates versus the vertical axes. The effect is similar to mirroring the graph about the lower-left to upper-right diagonal.

useComma=*uc* Controls the decimal separator used in tick mark labels.

uc=0: Use period as decimal separator and comma as thousands separator (default) when displaying numbers in graph labels and annotations.

uc=1: Use comma as decimal separator and period as the thousands separator. This does not alter the presentation of numbers in \{*expression*} constructs in annotations.

UIControl=*f* Disables certain aspects of the user interface for graphs. The UIControl keyword, added in Igor Pro 7.00, is for use by advanced Igor programmers who want to disable user actions.

This is a bitwise setting. **Setting Bit Parameters** on page IV-12 for details about bit settings.

f is defined as follows:

Bit 0: Disable axis click. Prevents moving or otherwise modifying an axis.

Bit 1: Disable cursor click. Prevents moving a graph cursor.

Bit 2: Disable trace drag. Prohibits the click-and-hold action to offset a trace on the graph.

Bit 3: Disable marquee. When set, you can't make a marquee on the graph, which in turn prevents changing the range of the graph using the marquee.

Bit 4: Disable draw mode.

Bit 5: Disable double click. Prohibits any double-click action. In general, double-clicks in a graph bring up dialogs to modify the graph's appearance.

Bit 6: Disable clicks on annotations. Prevents modification of annotations.

Bit 7: Disable tool tips.

Bit 8: Disable contextual menus.

Bit 9: Disable marquee menu. With this set, you can still have a marquee and use it for, i.e., selecting some portion of the graph, but you can't use the maruqee menu to change the graph's range. Note that if bit 3 is set, this bit is moot.

Bit 10: Disable mouse wheel events. This will prevent axis scaling using the mouse wheel.

Bit 11: Disable option-drag. Prevents offsetting the graph by holding down the option (Macintosh) or Alt (Windows) key and then dragging in the plot area.

To disable items in the Graph menu use **SetIgorMenuMode**.

useDotForX=*u* In the display of axis labels, if *u*=1 then, in instances of exponential notation such as “$5x10^3$”, the normal “x” is replaced with a dot giving “$5x10^3$”. useDotForX was added in Igor Pro 9.00.

useLongMinus=*m* Uses a normal (*m*=0; default) or long dash (*m*=1) for the minus sign.

width=*widthSpec* Sets the width of the graph area. See the examples.

Flags

/W=*winName* Modifies the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/Z Does not generate an error if the indexed trace, named wave, or named axis does not exist in a style macro.

Examples

The following code creates a graph where all the text expands and contracts directly in relation to the window size:

```
Make jack=sin(x/8);display jack
ModifyGraph mode=4,marker=8,gfRelSize= 5.0
TextBox/N=text0/A=MC "Some \\Zr200big\\]0 and \\Zr050small\\]0\rtext"
```

The *widthSpec* and *heightSpec*s set the width and height mode for the top graph. The following examples illustrate how to specify the various modes.

```
ModifyGraph width=0, height=0
```

Set to auto height, width mode. The width, height of horizontal and vertical axes are automatically determined based on the overall size of the graph and other factors such as axis offset setting and effect of exterior textboxes. This is the normal, default mode.

```
Variable n=72*5
ModifyGraph width=n
```

Five inches as points absolute width mode, horizontal axis width constrained to `n` points.

```
ModifyGraph height=n
```

Absolute height mode, `n` is in points. The height of the vertical axes is constrained to `n` points.

```
Variable n=2
ModifyGraph
width={perUnit,n,bottom}
```

Per unit width mode. The width of the horizontal axes is `n` points times the range of the bottom axis.

```
ModifyGraph height={Aspect,n}
```

Aspect height mode, `n = aspect ratio`. The height of the vertical axes is `n` times the width of the horizontal axes.

```
ModifyGraph
width={Plan,n,bottom,left}
```

Plan width mode. The width of the horizontal axes is `n` times the height of the vertical axes times range of the bottom axis divided by the range of the left axis.

ModifyGraph *(traces)*

`ModifyGraph` `[`***`/W=winName/Z`***`]` ***`key`*** `[`***`(traceName)`***`]` ***`= value`***
`[,` ***`key`*** `[`***`(traceName)`***`]` ***`= value`***`]…`

This section of ModifyGraph relates to modifying the appearance of wave “traces” in a graph. A trace is a representation of the data in a wave, usually connected line segments.

Parameters

Each *key* parameter may take an optional *traceName* enclosed in parentheses. Usually *traceName* is simply the name of a wave displayed in the graph, as in “`mode(myWave)=4`”. If “(*traceName*)” is omitted, all traces in the graph are affected. For instance, “`ModifyGraph lSize=0.5`” sets the lines size of all traces to 0.5 points.

For multiple trace instances, *traceName* is followed by the “#” character and instance number. For example, “`mode(myWave#1)=4`”. See **Instance Notation** on page IV-20.

A string containing a trace name can be used with the $ operator to specify *traceName*. For example, `String MyTrace="myWave#1"; mode($MyTrace)=4`.

Though not shown in the syntax, the optional “(*traceName*)” may be replaced with “[*traceIndex*]”, where *traceIndex* is zero or a positive integer denoting the trace to be modified. “[0]” denotes the first trace appended to the graph, “[1]” denotes the second trace, etc. This syntax is used for style macros, in conjunction with the /Z flag.

For certain modes and certain properties, you can set the conditions at a specific point on a trace by appending the point number in square brackets after the trace name. For more information, see the **Customize at Point** on page V-625.

The parameter descriptions below omit the optional "(*traceName*)". When using ModifyGraph from a user-defined function, be careful not to pass wave references to ModifyGraph. ModifyGraph expects trace names, not wave references. See **Trace Name Parameters** on page IV-88 for details.

arrowMarker=0

arrowMarker={*aWave*, *lineThick*, *headLen*, *headFat*, *posMode* [, *barbSharp*=*b*, *barbSide*=*s*, *frameThick*=*f*]}

Draws arrows instead of conventional markers at each data point in a wave. Arrows are not clipped to the plot area and will be drawn wherever a data point is within the plot area.

aWave contains arrow information for each data point. It is a two (or more) column wave containing arrow line lengths (in points) in column 0 and angles (in radians measured counterclockwise) in column 1. Zero angle is a horizontal arrow pointing to the right. If an arrow is below the minimum length of 4 points, a default marker is drawn.

You can change arrow markers into standard meteorological wind barbs by adding a column to *aWave* and giving it a column label of windBarb. Values are integers from 0 to 40 representing wind speeds up to 4 flags. Use positive integers for clockwise barbs and negative for the reverse. Use NaN to suppress the drawing. See **Wind Barb Plots** on page II-329 for an example.

Additional columns may be supplied in *aWave* to control parameters on a point by point basis. These optional columns are specified by dimension label and not by specific column numbers. The labels are *lineThick*, *headLen*, and *headFat* that correspond to the same parameters listed above.

lineThick is the line thickness in points.

headLen is the arrow head length in points.

headFat controls the arrow fatness. It is the width of the arrow head divided by the length.

posMode specifies the arrow location relative to the data point.

posMode=0:	Start at point.
posMode=1:	Middle on point.
posMode=2:	End at point.

In addition to the wave specification, *aWave* can also be the literal `_inline_` to draw lines and arrows between points on the trace (see **Examples**). If *aWave* is `_inline_`, *posMode* values are:

posMode=0:	Arrow at start.
posMode=1:	Arrow in middle.
posMode=2:	Arrow at end.
posMode=3:	Arrow in middle pointing backwards.

You can also enable inline mode even if *aWave* is not `_inline_` by setting posMode to values between 4 and 7. These are the same as modes 0-3 above.

Optional parameters must be specified using *keyword* = *value* syntax and can only be appended after *posMode* in any order.

barbSharp is the continuously variable barb sharpness between -1.0 and 1. 0:

barbSharp=1:	No barb; lines only.
barbSharp=0:	Blunt (default).
barbSharp=-1:	Diamond.

barbSide specifies which side of the line has barbs relative to a right-facing arrow:

barbSide=0: None.
barbSide=1: Top.
barbSide=2: Bottom.
barbSide=3: Both (default).

frameThick specifies the stroke outline thickness of the arrow in points. The default is *frameThick* = 0 for solid fill.

aWave can contain columns with data for each optional parameter using matching column names.

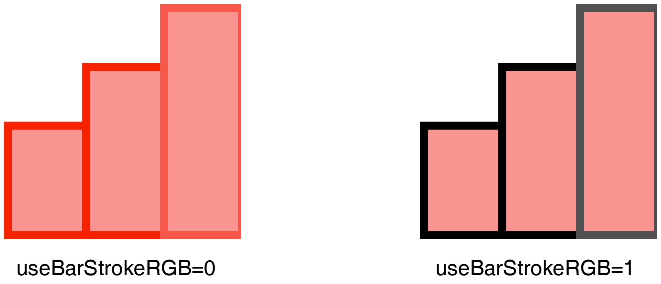
barStrokeRGB=(*r*,*g*,*b*[,*a*])

Specifies a separate color for bar strokes (outlines) if useBarStrokeRGB is 1. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

Applies only to Histogram Bars drawing mode (mode=5).

The bar fill color continues to be set with the rgb=(*r*,*g*,*b*[,*a*]), zColor={...}, usePlusRGB, plusRGB=(*r*,*g*,*b*[,*a*]), useNegRGB, and negRGB=(*r*,*g*,*b*[,*a*]) parameters.

Use barStrokeRGB and useBarStrokeRGB to put a differently-colored outline around Histogram Bars:



cmplxMode=*c* Display method for complex waves.

c=0: Default mode displays both real and imaginary parts (imaginary part offset by dx/2).
c=1: Real part only.
c=2: Imaginary part only.
c=3: Magnitude.
c=4: Phase (radians).

cmplxMode=0 does not work when the trace is a subrange of a multidimensional wave.

column=*n* Changes the displayed column from a matrix. Out of bounds values are clipped.

gaps=*g* Controls treatment of NaNs:

g=0: No gaps (ignores NaNs).
g=1: Gaps (shows NaNs as gaps).

gradient See **Gradient Fills** on page III-498 for details.

gradientExtra See **Gradient Fills** on page III-498 for details.

hBarNegFill=*n* Fill kind for negative areas if useNegPat is true. *n* is the same as for the hbFill keyword.

hbFill=*n*	Sets the fill pattern.
	n=0: No fill.
	n=1: Erase.
	n=2: Solid black.
	n=3: 75% gray.
	n=4: 50% gray.
	n=5: 25% gray.
	n>=6: See **Fill Patterns** on page III-498.
hideTrace=*h*	Removes a trace from the graph display.
	h=0: Shows the trace if it is hidden.
	h=1: Hides the trace and removes it from autoscale calculations.
	h=2: Hides the trace.
	When using *h*=1 to hide a graph trace, the hidden trace symbol and following text in annotations are also hidden. The amount of hidden text is the lesser of: the remaining text on the same line or the text up to but not including another trace symbol "`\s(traceName)`".
lHair=*lh*	Sets the hairline factor for traces printed on a PostScript® printer.
lineJoin={*j*, *ml*}	Sets the line join style and miter limit.
	Line join:
	j=0 Miter joins
	j=1 Round joins
	j=2 Bevel joins (default)
	Miter limit:
	ml >= 1 Sets miter limit to *ml*
	ml = INF Sets miter limit to unlimited
	ml = 0 Leaves miter limit unchanged
	ml = -1 Sets miter limit to default (10)
	The miter limit applies only to miter joins (*j*=0) and is ignored otherwise.
	See **Line Join and End Cap Styles** on page III-496 for further information.
	The lineJoin keyword was added in Igor Pro 8.00.
live=*lv*	*lv* is a bitwise parameter defined as follows:
	Bit 0: Live mode (see Live Mode below)
	Bit 1: Fast line drawing (see Fast Line Drawing)
	See **Setting Bit Parameters** on page IV-12 for details about bit settings.
logZColor=*lzc*	Controls the interpretation of the zColor parameter.
	lzc=0: Sets the default linearly-spaced zColors.
	lzc=1: Turns on logarithmically-spaced zColors. This requires that the zWave values be greater than 0 to display correctly.
	Affects trace line color only when the zColor parameter is used with a color table or color index wave - it has no effect if rgb=(*r*,*g*,*b*) parameter or zColor={...,directRGB} are used.

lOptions=cap Sets the line end cap style:

cap=0: Flat caps

cap=1: Round caps

cap=2: Square caps (default)

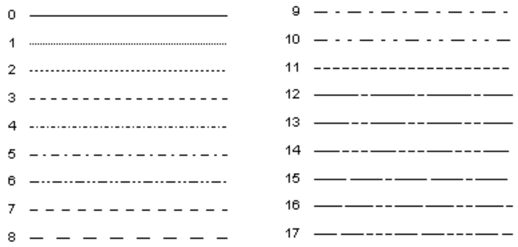
See **Line Join and End Cap Styles** on page III-496 for further information.

lSize=*l* Sets the line thickness, which can be fractional or zero, which hides the line.

lSmooth=*ls* Sets the smoothing factor for traces printed on a PostScript® printer.

lStyle=*s* Sets trace line style or dash pattern.

s=0 for solid lines. *s*=1 to s=17 for various dashed line styles.



marker=*n* n =0 to 62 designates various markers if mode=3 or 4.

You can also create custom markers. See the **SetWindow** markerHook keyword.

See **Markers** on page II-291 for a table of marker values.

mask={*maskwave*,*mode*,*value*} or 0

Specifies individual points for display by comparing values in *maskWave* with *value* as specified by *mode*.

mode=0: Exclude if equal.

mode=1: Include if equal.

mode=2: Include if bitwise AND is true.

mode=3: Include if bitwise AND is false.

maskwave can be specified using subrange notation. The length of *maskwave* (or subrange) must match the size of specified trace’s wave (or subrange.) Bitwise modes should be used with integer waves with the intent of using one mask wave with multiple traces. See **Examples**.

mode=*m* Sets trace display mode.

m=0: Lines between points.

m=1: Sticks to zero.

m=2: Dots at points.

m=3: Markers.

m=4: Lines and markers.

m=5: Histogram bars.

m=6: Cityscape.

m=7: Fill to zero.

m=8: Sticks and markers.

mrkFillRGB=(*r*,*g*,*b*[,*a*])

Sets the background fill color for hollow markers. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

This setting takes effect only if opaque is set to non-zero. See the opaque keyword below.

The mrkFillRGB keyword was added in Igor Pro 9.00.

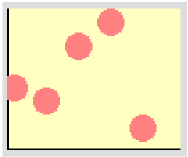
mrkStrokeRGB=(*r*,*g*,*b*[,*a*])

Specifies the color for marker stroked lines if useMrkStrokeRGB = 1. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

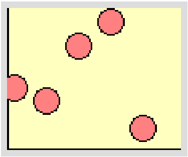
The marker fill color continues to be set with the rgb=(*r*,*g*,*b*[,*a*]) or zColor={…} parameters.

Applies only to the nontext and nonarrow marker modes.

Use mrkStrokeRGB and useMrkStrokeRGB to put a colored outline around filled markers, such as marker=19:



useMrkStrokeRGB=0



useMrkStrokeRGB=1

Note: The stroke color of unfilled markers such as marker 8 is also affected by mrkStrokeRGB, but their fill color is only affected by the opaque parameter (and the opaque fill color is always white, so if you want a color-filled marker, don't use unfilled markers).

mrkThick=*t* Sets the thickness of markers in points, which can be fractional.

msize=*m* Specifies the marker size in points.

m=0:	Autosize markers.
m>0:	Sets marker size.

m can be fractional, which will only make a difference when the graph is printed because fractional points can not be displayed on the screen.

mskip=*n* Puts a marker on only every *n*th data point in Lines and Markers mode (mode=4). Useful for displaying many data points when you want to identify the traces with markers. The maximum value for *n* is 32767.

mstandoff=*s* Prevents lines from touching markers in lines and markers mode if *s* is greater than zero. *s* is in units of points. This feature was added in Igor Pro 8.00.

muloffset={*mx*,*my*} Sets the display multiplier for X (*mx*) and Y (*my*). The effective value for a given X or Y data point then becomes *muloffset*data+offset*. A value of zero means "no multiplier" — not multiply by zero.

negRGB=(*r*,*g*,*b*[,*a*]) Specifies the color for negative values represented by the trace if useNegRGB is 1. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

offset={*x*,*y*} Sets the display offset in horizontal (X) and vertical (Y) axis units.

opaque=*o* Displays transparent (*o*=0) or opaque (*o*=1) markers.

If *o* is zero (default) hollow markers such as unfilled circles and boxes have transparent backgrounds. If *o* is non-zero, the background is filled with color. The color defaults to white, but you can change it using the mrkFillRGB keyword.

patBkgColor= 0, 1, 2 or (*r*,*g*,*b*[,*a*])

Specifies the background color for fill patterns.

0, the default, is white, 1 is graph background, 2 is transparent.

Use (*r*,*g*,*b*[,*a*]) for a specific RGB color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

plotClip=*p* — *p* =1 clips the trace by the operating system (not by Igor) to the plot rectangle. This trims overhanging markers and thick lines. On Windows, this may not be supported for certain printers or by certain applications when importing.

plusRGB=(*r*,*g*,*b*[,*a*]) — Specifies the color for positive values represented by the trace if usePlusRGB is 1. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

quickdrag=*q* — Controls dragging of traces.

q=0:	Normal traces.
q=1:	Traces that can be instantly dragged without the normal one second delay. See the **Quickdrag** section below.
q=2:	Causes the mouse cursor to change to 4 arrows when over the trace and a reduced search is used.

removeCustom=*r* — Removes per-point trace customizations. See **Customize at Point** on page II-306 for background information. The removeCustom keyword was added in Igor Pro 9.00.

The parameter *r* is required by the ModifyGraph syntax, but its value is immaterial. We recommend using 1 for *r* but any number will work.

These examples show how to use the removeCustom keyword given a graph with a trace named wave0:

```
// Remove customizations for point 3 of trace wave0
ModifyGraph removeCustom(wave0[3]) = 1

// Remove customizations for all points of trace wave0
ModifyGraph removeCustom(wave0) = 1

// Remove customizations for all points of all traces
ModifyGraph removeCustom = 1
```

rgb=(*r*,*g*,*b*[,*a*]) — Specifies the color of the trace. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

textMarker={<*char* or *wave*>,*font*,*style*,*rot*,*just*,*xOffset*,*yOffset*} or 0

Uses the specified character or text from the specified wave in place of the marker for each point in the trace.

If the first parameter is a quoted string or a string expression of the form `""+strexpr` in a user function, ModifyGraph uses the first three bytes of the string as the marker for all points. Three bytes are supported mainly for non-ASCII characters but can be used for 3 separate single-byte characters. Otherwise, it interprets the first parameter as the name of a wave. If the wave is a text wave, it uses the value of each point in the text wave as the marker for the corresponding point in the trace. If the wave is a numeric wave, the value for each point is converted into text and the result is used as the marker for the corresponding point in the trace.

xOffset and *yOffset* are offsets in fractional points. Each marker will be drawn offset from the location of the corresponding point in the trace by these amounts.

style is a font style code as used with the ModifyGraph fstyle keyword.

rot is a text rotation between -360 and 360 degrees.

just is a justification code as used in the **DrawText** operation except the X and Y codes are combined as y*4+x. Use 5 for centered.

The font size is 3*marker size. Note that marker size and color can be dynamically set via the zColor and zmrkSize keywords.

toMode=*t* Modifies the behavior of the display modes as determined by the mode parameter.

t=0: Fill to zero.

t=1: Fill to next trace. Applies to Sticks to zero (mode=1), histogram bars (mode=5), and fill to zero (mode=7).

t=2: Add the current trace's Y values to the next trace's Y values. Works with all display modes.

t=3: Stack on next and is the same as *t*=2 except that the added value is clipped to zero. Works with all display modes.

t=-1: This mode is used only with category plots and means "keep with next" (i.e., put in the same subcategory as the next trace). It is used for special effects only.

For modes 1, 2 and 3, both Y-waves must have the same number of points and must use the same X values. Igor uses the X values from the first wave for both Y-

traceName=*name* Sets, changes, or removes a custom trace name.

The traceName keyword was added in Igor Pro 9.00.

By default Igor assigns trace names based on the name of the displayed wave. You can assign a custom trace name when appending a wave to a graph using the /TN flag with the Display and AppendToGraph operations. See **Trace Names** on page II-282 for details.

The traceName keyword allows you to change the name of a trace after it has been added to a graph. For example:

```
Make/O wave0; Display wave0
ModifyGraph traceName(wave0)=Custom0
```

Use $"" for name to remove the custom name and revert to the default trace name:

```
ModifyGraph traceName(Custom0)=$""
```

Renaming a trace can change the names of other traces in the graph from the same wave. For example, if you display two instances of wave0 in one graph, the first trace name is wave0 and the second is wave0#1. If you now rename trace wave0, the name of the second instance of wave0 changes from wave0#1 to wave0. See **Instance Notation** on page IV-20 for further discussion.

useBarStrokeRGB=*u*

If *u*=1 then bar stroked lines use the color specified by the barStrokeRGB keyword.

Applies only to Histogram Bars drawing mode (mode=5).

The bar fill color continues to be set with the rgb, zColor, usePlusRGB, plusRGB, useNegRGB, and negRGB keywords.

If *u*=0 then the bar stroked line colors are set with the rgb=(*r*,*g*,*b*[,*a*]) or zColor={...} parameters, just like the bar fill color.

useMrkStrokeRGB=*u*

If *u* =1 then marker stroked lines use the color specified by the mrkStrokeRGB keyword. The marker fill color continues to be set with the rgb=(*r*,*g*,*b*[,*a*]) or zColor={…} parameters.

Applies only to the nontext and nonarrow marker modes.

If *u*=0 then the marker stroked line colors are set with the rgb=(*r*,*g*,*b*[,*a*]) or zColor={…} parameters, just like the marker fill color.

useNegPat=*u* If *u*=1, negative fills use the mode specified by the hBarNegFill keyword. Applies to the fill-to-zero, fill-to-next and histogram bar modes.

useNegRGB=*u* If *u* =1, negative fills use the color specified by the negRGB keyword. Applies to the fill-to-zero, fill-to-next and histogram bar modes.

usePlusRGB=*u* If *u* =1, positive fills use the color specified by the plusRGB keyword. Applies to the fill-to-zero, fill-to-next and histogram bar modes.

userData={*udName*, *doAppend*, *data*}

Attaches arbitrary data to a trace. You should specify a trace name (userData(<traceName>)={...}). Otherwise copies of the data will be attached to every trace, which is most likely not what you intend.

Use the GetUserData function to retrieve the data, with the trace name as the object ID.

udName: The name of your user data. Use $"" for unnamed user data.

doAppend=0: Do not append. Any pre-existing data is replaced.

doAppend=1: Append the data. Data is added to the end of any pre-existing data.

data: A string expression containing the data you wish to attach to the trace.

zColor={*zWave*,*zMin*,*zMax*,*ctName* [,*reverseMode* [,*cWave*]]} or 0

Dynamically sets color based on the values in *zWave* and color table name or mode specified by *ctName*.

zWave may be a subrange expression such as `myZWave[2,9]` when *zWave* has more points than the trace, in which case `myZWave[2]` provides the Z value for the first point of the trace, and autoscaled *zMin* or *zMax* is determined over only the *zWave* subrange.

If a value in the *zWave* is NaN then a gap or missing marker will be observed. If a value is out of range it will be replaced with the nearest valid value. See also the zColorMax and zColorMin keywords.

ctName can be the name of a built-in color table such as returned by the **CTabList** function, such as Grays or Rainbow, for color table mode, `ctableRGB` for color table wave mode, `cindexRGB` for color index wave mode, or `directRGB` for direct color wave mode.

zColor for Built-in Color Table Mode

This mode uses *zWave* to select a color from a built-in color table specified by *ctName*. See **Image Color Tables** on page II-392 for details.

zWave contains values that are used to select a color from the built-in color table specified by *ctName*.

zMin is the *zWave* value that maps to the first entry in the color table. Use * for *zMin* to autoscale it to the smallest value in zWave.

zMax is the *zWave* value that maps to the last entry in the color table. Use * for *zMax* to autoscale it to the largest value in zWave.

ctName is the name of a built-in color table such as Grays or Rainbow. See the **CTabList** function for a list of built-in color tables.

Set *reverseMode* to 1 to reverse the color table lookup or to 0 to use the normal lookup. If you omit *reverseMode* or specify -1, the reverse mode is unchanged.

cWave must be omitted.

Normally the colors from the color table are linearly distributed between *zMin* and *zMax*. Use `logZColor=1` to distribute them logarithmically.

```
// Example zColor command using built-in color table
ModifyGraph zColor(data)={zWave,*,*,Rainbow}
```

zColor for Color Table Wave Mode

This mode is like Built-in Color Table except that the colors are stored in a color table wave that you have created. A color table wavey can be a 3 column RGB wave or a 4 column RGBA wave. See **Color Table Waves** on page II-399 for details.

zWave contains values that are used to select a color from the color table wave specified by *cWave*.

zMin is the *zWave* value that maps to the first entry in the color table wave. Use * for *zMin* to autoscale it to the smallest value in zWave.

zMax is the *zWave* value that maps to the last entry in the color table wave. Use * for *zMax* to autoscale it to the largest value in zWave.

ctName is `ctableRGB`.

Set *reverseMode* to 1 to reverse the color table lookup or to 0 to use the normal lookup. If you omit *reverseMode* or specify -1, the reverse mode is unchanged.

cWave is a reference to your color table wave.

Normally the colors from the color table are linearly distributed between *zMin* and *zMax*. Use `logZColor=1` to distribute them logarithmically.

Example `ctableRGB zColor` command:

```
ColorTab2Wave Rainbow   // Creates M_Colors wave
Rename M_Colors, MyColorTableWave
ModifyGraph zColor(data)={zWave,*,*,ctableRGB,0,MyColorTableWave}
```

zColor for Color Index Wave Mode

This mode is like Color Table Wave except that the values in *zWave* represent X indices with respect to *cWave*. You must create the RGB or RGBA color index wave and set its X scaling appropriately. See **Color Index Wave** on page II-372 for details.

zWave contains values that are used to select a color from the color index wave specified by *cWave*.

zMin and *zMax* are not used and should be set to `*`.

ctName is `cindexRGB`.

Set *reverseMode* to 1 to reverse the color table lookup or to 0 to use the normal lookup. If you omit *reverseMode* or specify -1, the reverse mode is unchanged. Normally the zWave values select the color from the row of *cWave* whose X value is closest to the zWave value. *reverseMode*=1 reverses the colors.

cWave is a reference to your color index wave.

Normally the colors from the color index wave are linearly distributed between the minimum and maximum X values of the color index wave. Use `logZColor`=1 to distribute them logarithmically.

```
// Example cindexRGB zColor command
zColor(data)={myZWave,*,*,cindexRGB,0,M_colors}
// M_colors is generated by ColorTab2Wave
```

zColor for Direct Color Wave Mode

In direct color mode, *zWave* is an RGB or RGBA wave that directly specifies the color for each point in the trace. If *zWave* is 8-bit unsigned integer, then color component values range from 0 to 255. For other numeric types, color component values range from 0 to 65535. See **ColorTab2Wave**, which generates RGB waves, and **Direct Color Details** on page II-401.

zWave is an RGB or RGBA wave that directly specifies the color for each point of the trace.

zMin and *zMax* are not used and should be set to `*`.

ctName is `directRGB`.

reverseMode is not applicable and should be omitted or set to 0.

cWave must be omitted.

```
// Example directRGB zColor command
zColor(data)={zWaveRGB,*,*,directRGB}
```

Turning zColor Off

zColor = 0 turns the zColor modes off.

zColorMax=(*red*, *green*, *blue*)

Sets the color of the trace for zColor={*zWave*, …} values greater than the zColor's *zMax*. Also turns on zColorMax mode.

The *red*, *green*, and *blue* color values are in the range of 0 to 65535.

zColorMax=1, 0, or NaN

Turns zColorMax mode off, on, or transparent. These modes affect the color of zColor={*zWave*, …} values greater than the zColor's *zMax*.

1:	Turns on zColorMax mode. The color of the affected trace pixels is black or the last color set by zColorMax=(*red*, *green*, *blue*).
0:	Turns off zColorMax mode (default). The color of the affected trace pixels is the last color in the zColor's *ctname* color table.
NaN:	Transparent zColorMax mode. Affected trace pixels are not drawn.

zColorMin=(*red*, *green*, *blue*)

Sets the color of the trace for zColor={*zWave*, …} values less than the zColor's *zMin*. Also turns zColorMin mode on.

The *red*, *green*, and *blue* color values are in the range of 0 to 65535.

zColorMin=1, 0, or NaN

Turns zColorMin mode off, on, or transparent. These modes affect the color of zColor={*zWave*, …} values less than the zColor's *zMin*.

1:	Turns on zColorMin mode. The color of the affected image pixels is black or the last color set by zColorMin=(*red*, *green*, *blue*).
0:	Turns off zColorMin mode (default). The color of the affected trace pixels is the first color in the zColor's *ctname* color table.
NaN:	Transparent zColorMin mode. Affected trace pixels are not drawn.

zmrkNum={*zWave*} or 0

Dynamically sets the marker number for each point to the corresponding value in *zWave*. The values in *zWave* are the marker numbers (as used with the marker keyword). If a value in the *zWave* is NaN then no marker will be drawn at the corresponding point. If a value is out of range it will be replaced with the nearest valid value.

zmrkNum=0 turns this mode off.

zmrkSize={*zWave*,*zMin*,*zMax*,*mrkMin*,*mrkMax*,[*axis*]} or 0

Dynamically sets marker size based on values in *zWave*. See **Marker Size as f(z)** on page II-299 for background information.

zmrkSize = 0 turns this mode off.

Use * or a missing parameter for *zMin* and *zMax* to autoscale. *mrkMin* and *mrkMax* can be fractional.

mrkMin is the marker size to use when z equals *zMin* and *mrkMax* is the marker size to use when z equals *zMax*. *mrkMin* and *mrkMax* are not limits; they just control the linear mapping of z values to marker size.

If a value in the *zWave* is NaN then the corresponding marker is not drawn.

The marker size is clipped to 400 on the high end and to 1 point on the low end.

axis is an optional parameter specifying an existing axis on the graph. It is supported in Igor Pro 9.00 and later. Passing $"" for *axis* is the same as omitting it.

If *axis* is specified, the values in *zWave* are interpreted as in units of the specified axis and specify the half-width of the marker to be drawn. For a circle marker, this is the radius of the marker.

If *axis* is specified, the *zMin*, *zMax*, *mrkMin*, and *mrkMax* parameters have no impact on the marker size unless you remove the specified axis in which case those parameters control the trace's marker sizes.

See **Marker Size as f(z) in Axis Units** on page II-300 for further information.

zpatNum={*zWave*} or 0

Dynamically sets the positive fill type/pattern number for each point to the corresponding value in *zWave*. The values in *zWave* are the pattern numbers (as used with the hbFill keyword). If a value in the *zWave* is NaN then the corresponding point will not be drawn. If a value is out of range it will be replaced with the nearest valid value.

zpatNum=0 turns this mode off.

Flags

/W=*winName* Modifies the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the Command Line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/Z Does not generate an error if the indexed trace, named wave, or named axis does not exist in a style macro.

Details

Waves supplied with zmrkSize, zmrkNum, and zColor may use **Subrange Display Syntax** on page II-321.

Live Mode

Bit 0 of live=*lv* controls live mode. Live mode improves graph update performance when one or more of the waves displayed in the graph is frequently modified, for example, if the waves are being acquired from a data acquisition system. Live Mode traces do not autoscale axes.

Fast Line Drawing

Bit 1 of live=*lv* controls fast line drawing. When enabled, Igor draws traces in lines-between-points mode using custom fast code instead of the normal system drawing code. The result is not as esthetically pleasing but can be much faster and may be critical in data acquisition applications. The custom code is used only for drawing to the screen, not for exporting graphics.

Fast line drawing was added in Igor Pro 8.00.

Quickdrag

Quick drag mode (quickdrag=1) is a special purpose mode for creating cross hair cursors using a package of Igor procedures. (See the Cross Hair Demo example experiment.) Normally you would have to click and hold on a trace for one second before entering drag mode. When quickdrag is in effect, there is no delay. If a trace is in quickdrag mode it should also be set to live mode. With this combination you can click a trace and immediately drag it to a new XY offset. In addition to quick drag mode, the cross hair package relies on Igor to store information about the drag in a string variable if certain conditions are in effect. The string variable name (that you have to create) is S_TraceOffsetInfo, which must reside in a data folder that has the same name as the graph (not title!) which in turn must reside in root:WinGlobals:. If these conditions are met, then after a trace is dragged, information will be stored in the string using the following key-value format: `GRAPH:<name of graph>;XOFFSET:<x offset value>;YOFFSET:<y offset value>;TNAME:<trace name>;`

Customize at Point

You can customize the appearance of individual points on a trace in a graph for bar, marker, dot and lines to zero modes using `key(tracename[pnt])=value` syntax. The point number must be a literal number and the trace name is required.

To remove a customization, use `key(tracename[-pnt-1])=value` where value is not important but must match the syntax for the keyword. The offset of -1 is needed because point numbers start from zero.. You can also remove customizations using the removeCustom keyword described above.

Although the syntax is allowed for all trace modifiers, it has meaning only for the following: rgb, marker, msize, mrkThick, opaque, mrkFillRGB, mrkStrokeRGB, barStrokeRGB, hbFill, patBkgColor and lSize.

Note that useBarStrokeRGB and useMrkStrokeRGB are not needed. The act of using barStrokeRGB or mrkStrokeRGB is enough to customize the point. But as a convenience, since these are generated by the modify graph dialog, they are ignored if used with [pnt] syntax.

Also note that legend symbols can use [pnt] syntax like so:

```
\s(<tracename>[pnt])
```

Automatically generated legends automatically include symbols for customized points.

For example:

```
Make/O/N=10 jack=sin(x); Display jack
ModifyGraph mode=5,hbFill=6,rgb=(0,0,0)
ModifyGraph hbFill(jack[2])=7,rgb(jack[2])=(0,65535,0)
ModifyGraph rgb(jack[3])=(65535,0,0)
Legend/C/N=text1/F=0/A=MC
```

Examples

Arrow markers.

```
Make/N=10 wave1= x; Display wave1
Make/N=(10,2) awave
awave[][0]= p*5              // length
awave[][1]= pi*p/9           // angle
ModifyGraph mode=3,arrowMarker(wave1)={awave,1,10,0.3,0}

// Now add an optional column to control headLen
Redimension/N=(-1,3) awave
awave[][2]= 7+p              // will be head length

// Note: nothing changes until the following is executed
SetDimLabel 1,2,headLen,awave
```

Create meteorological wind barb symbols.

```
Make/O/N=50 jack= floor(x/10),jackx= mod(x,10)
Display jack vs jackx
Make/O/N=(50,3) jackbarb
```

```
jackbarb[][0]= 40              // length of stem
jackbarb[][1]= 45*pi/180       // angle (45deg)
jackbarb[][2]= p               // wind speed code
SetDimLabel 1,2,windBarb,jackbarb
ModifyGraph mode=3,arrowMarker(jack)={jackbarb,1,10,0.5,0}
ModifyGraph margin(top)=62,margin(right)=84
```

See also **Wind Barb Plots** on page II-329.

Inline arrows and barb sharpness.

```
Make/O/N=20 wavex=cos(x/3),wavey=sin(x)
Display wavey vs wavex
ModifyGraph mode=3,arrowMarker={_inline_,1,20,.5,0,barbSharp= 0.2}
```

Use direct color mode to individually color each point in a trace:

```
Make jack=sin(x/8)
Make/N=(128,3)/B/U jackrgb
Display jack
ModifyGraph mode=3,marker=19
jackrgb= enoise(128)+128
ModifyGraph zColor(jack)={jackrgb,*,*,directRGB}
```

Use masking.

```
Make/N=100 jack= (p&1) ? sin(x/8) : cos(x/8)
Display jack

Make/N=100 mjack= (p&1) ? 0 : NaN       // just to show NaN can be used
ModifyGraph mask(jack)={mjack,0,NaN}

// now switch which points are shown
mjack= (p&1) ? NaN : 0
```

See Also

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

ModifyGraph *(axes)*

ModifyGraph [**/W=*winName*/Z**] ***key*** [**(*axisName*)**] **= *value***
[**,** ***key*** [**(*axisName*)**] **= *value***]…

This section of ModifyGraph relates to modifying the appearance of axes in a graph.

Parameters

Each *key* parameter may take an optional *axisName* enclosed in parentheses.

axisName is “left”, “right”, “top”, “bottom” or the name of a free axis such as “vertCrossing”. For instance, “`ModifyGraph axThick(left)=0.5`” sets the axis thickness for only the left axis.

If “(*axisName*)” is omitted, all axes in the graph are affected. For instance, “`ModifyGraph standoff=0`” disables axis standoff for all axes in the graph.

The parameter descriptions below omit the optional “(*axisName*)”.

axisClip= *c* Specifies one of three clipping modes for traces.

c=0: Clips traces to a plot rectangle as defined by the pair of axes used by a given trace (default).

c=1: Plots traces on an axis with a restricted range (as set by axisEnab) to extend to the full range of the normal plot rectangle.

c=2: Traces extend outside the normal plot rectangle to the full extent of the graph area.

axisEnab={*lowFrac*,*highFrac*}

Restricts the length of an axis to a subrange of normal. The axis is drawn from *lowFrac* to *highFrac* of graph area height (vertical axis) or width (horizontal axis). For instance, {0.1,0.75} specifies that the axis is drawn from 10% to 75% of the graph area height/width, instead of the normal 0% to 100%. AxisEnab is discussed in **Creating Split Axes** on page II-347 and **Creating Stacked Plots** on page II-324.

axisOnTop=*t* Specifies drawing level of axis and associated grid lines.

t=0: Draws axis before traces and images (default).

t=1: Draws the axis after all traces and images.

axOffset=*a* Specifies the distance from default axis position to actual axis position in units of the width of a zero character (0) in a tick mark label. Unlike margin, axOffset adjusts to changes in the size of the graph.

axThick=*t* Specifies the axis thickness in points.

barGap=*fraction* Sets the fraction of the width available for bars to be used as gap between bars.

barGap sets the gap between bars within a single category while catGap sets the gap between categories.

btLen=*p* Sets the length of major ("big") tick marks to *p* points. If *p* is zero, it uses the default length. *p* may be fractional.

btThick=*p* Sets the thickness of major ("big") tick marks to *p* points. If *p* is zero, it uses the default thickness. *p* may be fractional.

catGap=*fraction* The value for catGap is the fraction of the category width to be used as gap. The gap is divided equally between the start and end of the category width. A value of 0.2 would use 20% of the available space for the gap and leave 80% of the available space for the bars.

catGap sets the gap between categories while barGap sets the gap between bars within a single category.

dateFormat={*languageName*, *yearFormat*, *monthFormat*, *dayOfMonthFormat*, *dayOfWeekFormat*, *layoutStr*, *commonFormat*}

Sets the custom date format used in the active graph.

Note: Use a custom date format only if you turn it on via a ModifyGraph dateInfo command. The last parameter to the ModifyGraph dateInfo command must be -1 to turn on the custom date format.

Parameters are the same as for the LoadWave/R flag except for the last one.

commonFormat selects the common date format to use in the Modify Axis dialog. The legal values correspond to the choices in the Common Format pop-up menu of the Modify Axis dialog. They are:

Value	Date Format	Value	Date Format
1	mm/dd/yy	16	mm/yy
2	mm-dd-yy	17	mm.yy
3	mm.dd.yy	18	Abbreviated month and year
4	mmddyy	19	Full month and year
6	dd/mm/yy	21	mm/dd
7	dd-mm-yy	22	dd.mm
8	dd.mm.yy	23	Abbreviated month and day
9	ddmmyy	24	Full month and day
11	yy/mm/dd	26	Abbreviated date without day of week
12	yy-mm-dd	27	Abbreviated date with day of week
13	yy.mm.dd	28	Full date without day of week
14	yymmdd	29	Full date with day of week

If the *commonFormat* parameter is negative, then it will select the Use Custom Format radio button in the Modify Axis dialog rather than Use Common Format and will then use the absolute value of *commonFormat* to determine which item to select in the Common Format pop-up menu.

dateInfo={*sd*,*tm*,*dt*} Controls formatting of date/time axes.

sd=0: Show date in the date&time format.

The date is always suppressed if *tm*=2 (elapsed time).

The time is always suppressed if there are fewer than one ticks per day and *tm*=0 (12-hour time) or *tm*=1 (24-hour time).

sd=1: Suppress date.

The date is always suppressed if *tm*=2 (elapsed time).

sd=2: Suppress time.

The time is always shown if *tm*=2 (elapsed time).

sd=2 requires Igor Pro 9.00 or later.

tm=0: 12 hour (AM/PM) time.

tm=1: 24 hour (military) time.

tm=2: Elapsed time.

dt=-1: Custom date as specified via the dateFormat keyword.

dt=0: Short dates (2/22/90).

dt=1: Long dates (Thursday, February 22, 1990).

dt=2: Abbreviated dates (Thurs, Feb 22, 1990).

font="*fontName*" Sets the axis label font, e.g., `font(left)="Helvetica"`.

freePos(*freeAxName*)=*p*

Sets the position of the *free* axis relative to the edge of the plot area to which the axis is anchored. *p* is in points. i.e., if the axis was made via /R=*axName* then the axis is placed *p* points from the right edge of the plot area. Positive is away from the central plot area. *freeAxName* may not be any of the standard axes: “left”, “bottom”, “right” or “top”.

freePos(*freeAxName*)={*crossAxVal*,*crossAxName*}

Positions the *free* axis so it will cross the perpendicular axis *crossAxName* where it has a value of *crossAxVal*. *freeAxName* may not be any of the standard axis names “left”, “bottom”, “right”, or “top”, though *crossAxName* may.

You can position a free axis as a fraction of the distance across the plot area by using `kwFraction` for *crossAxName*. *crossAxVal* must then be between 0 and 1; any values outside this range are clipped to valid values.

fsize=*s* Autosizes (*s*=0) tick mark labels and axis labels.

If *s* is between 3 and 99 then the labels are fixed at *s* points.

fstyle=*f*

f is a bitwise parameter with each bit controlling one aspect of the font style for the axis and tick mark labels as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

ftLen=*p*

Sets the length of 5th (or emphasized minor) tick marks to *p* points. If *p* is zero, it uses the default length. *p* may be fractional.

ftThick=*p*

Sets the thickness of 5th (or emphasized minor) tick marks to *p* points (fractional). If *p* is zero, it uses the default thickness.

grid=*g*

Controls grid lines.

g=0: Grid off.

g=1: Grid on.

g=2: Grid on major ticks only.

gridEnab={*lowFrac*,*highFrac*}

Restricts the length of axis grid lines to a subrange of normal. The grid is drawn from *lowFrac* to *highFrac* of graph area height (if axis is horizontal) or width (if axis is vertical).

gridHair=*h*

Sets the grid hairline thickness (*h* =0 to 3; 0 for thicker lines, 3 for thinner; default is 2). If *h*=0, the thickness of grid lines on major tick marks is the same as the axis thickness, half for a minor tick and one tenth for a subminor tick (log axis only). As *h* increases these thicknesses decrease by a factor of 2^*h*. If you want to see the effect of different values of gridHair, you will need to print a sample graph because you generally can't see the effect of thin lines on the screen. Also see the example experiment "Examples:Graphing Techniques:Graph Grid Demo".

gridStyle=*g*

Sets the grid style to various combinations of solid and dashed lines. In the following discussion, major, minor and subminor refer to grid lines the corresponding tick marks. Subminor ticks are used only on log axes when there is a small range and sufficient room (they correspond to hundredths of a decade). The different grid styes are solid, dotted, dashed, and blank. The possible grids are as follows:

g=0: Same as mode 1 if graph background is white else uses mode 5.

g=1: Major dotted, minor and subminor dashed.

g=2: All dotted.

g=3: Major solid, minor dotted, subminor blank.

g=4: Major and minor solid, subminor dotted.

g=5: All solid.

Also see the example experiment "Examples:Graphing Techniques:Graph Grid Demo".

highTrip=*h*

If the extrema of an axis are between its lowTrip and its highTrip then tick mark labels use fixed point notation. Otherwise they use exponential (scientific or engineering) notation.

lblLatPos=*p*

Sets a lateral offset for the axis label. This is an offset parallel to the corresponding axis. *p* is in points. Positive is down for vertical axes and to the right for horizontal axes.

lblLineSpacing=*linespace*

Specifies an adjustment to the normal line spacing for multi-line axis labels. linespace is points of extra (plus or minus) line spacing. For negative values, a blank line may be necessary to avoid clipping the bottom of the last line.

lblLineSpacing affects all lines of the axis label. See also the \sa and \sb line spacing escape codes described under **General Escape Codes** on page III-54. They can be used to affect the spacing between individual lines.

lblMargin=*l* Specifies the distance from the edge of graph to a label in points.

lblPos=*p* Sets the distance from an axis to the corresponding axis label in points. If *p*=0, it automatically picks an appropriate distance.

This setting is used only if the given graph edge has at least one free axis. Otherwise, the lblMargin setting is used to position the axis label.

lblPosMode= *m* Affects the meaning and usage of lblPos, lblLatPos, and lblMargin parameters. Mainly for use when you have multiple axes on a side and you need axis labels to be properly positioned even as you make graph windows dramatically larger or smaller.

m=0: Default compatibility mode (Margin or Axis absolute depending on presence of free axis).

m=1: Margin absolute.

m=2: Margin scaled.

m=3: Axis absolute.

m=4: Axis scaled.

The absolute modes are measured in points whereas scaled modes have similar values but automatically expand or contract as the axis font height changes. Mode 0 is the default and results in no change relative to previous versions of Igor Pro that used lblMargin unless a given side used a free axis in which case it used lblPos in absolute mode. The margin modes measure relative to an edge of the graph while the axis modes measure relative to the position of the axis. When using stacked axes, use either margin modes. With multiple nonstacked axes, use Axis scaled if the graph edge is not using a fixed margin or use axis absolute if it is.

lblRot=*r* Rotates the axis label by *r* degrees. *r* is a value from -360 to 360. Rotation is counterclockwise and starts from the label's normal orientation.

linTkLabel=*tl* *tl*=1 attaches the data units with any exponent or prefix to each tick label on a normal axis. *tl*=0 removes them.

In Igor Pro 9.00 and later, *tl*=2 suppresses the space character normally displayed before units.

log=*l* Controls axis log mode.

g=0: Normal axis.

g=1: Log base 10.

g=2: Log base 2.

logHTrip=*h* Same as highTrip but for log axes.

logLabel=*l* Sets the maximum number of decades in a log axis before minor tick labels are suppressed.

logLTrip=*l* Same as lowTrip but for log axes.

loglinear=*l* Switches to a linear tick method (*l*=1) on a log axis if the number of decades of ranges is less than 2. It switches to a linear tick exponent method if the number of decades is greater than five.

logmtlOffset=*o* Offsets the minor tick mark labels on a log axis by *o* fractional points relative to the default tick mark label position. Positive is away from the axis. Added in Igor Pro 9.00.

logTicks=*t* Sets the maximum number of decades in log axis before minor ticks are suppressed.

lowTrip=*l* If the extrema of an axis are between its lowTrip and its highTrip then tick mark labels use fixed point notation. Otherwise they use exponential (scientific or engineering) notation.

manminor={*number*, *emphasizeEvery*}

Specifies how to draw minor ticks in manual tick mode. There will be *number* ticks between each major (labeled) tick. You will usually want to set this to 4 to make 5 divisions, or 9 to make 10 divisions. A medium-sized tick (an emphasized minor tick) will be drawn every *emphasizeEvery* minor tick.

manTick={*cantick*, *tickinc*, *exp*, *digitsrt* [, *timeUnit*]}

Turns on manual tick mode. The tick from which all other ticks are calculated is the cononic tick (*cantick*). The numerical spacing between ticks is set by *tickinc*. *cantick* and *tickinc* are multiplied by 10*exp*. The number of digits to the right of the decimal point displayed in the tick labels is set by *digitsrt*.

The optional parameter *timeUnit* is used with Date/Time axes to specify the units of tickinc. In this case, tickinc must be an integer. The value of *timeUnit* is one of the following keywords:

```
second, minute, hour, day, week, month, year
```

On a date/time axis, the *exp* and *digitsrt* keywords are ignored, but must be present. You can set them to zero.

manTick=0 Turns off manual tick mode.

margin=*m* Sets a fixed margin from the edge of the window to the axis in points. Used principally to make axes of multiple graphs on a page line up when “stacked”. You can use the left, right, bottom, and top axis names (even if an axis with that name doesn’t exist) to adjust the graph plot area. See **Types of Axes** on page II-279.

m=0: Sets “automatic” margin size (dependent on the length and height of tick marks and labels).

m=-1: Sets the margin to “none”, or 0. The axis is drawn at the graph window’s edge.

minor=*m* Disables (*m*=0) or enables (*m*=1) minor ticks.

mirror=*m* Controls axis mirroring.

m=1: Right axis mirroring left or top mirroring bottom.

m=2: Mirror axis without tick marks.

m=3: Mirror axis with tick marks and tick labels.

m=0: No mirroring.

mirrorPos=*pos* Specifies the position of the mirror axis relative to the normal position. *pos* is a value between 0 and 1.

noLabel=*n* Controls axis labeling.

n=0: Normal labels.

n=1: Suppresses tick mark labels.

n=2: Suppresses tick mark labels and axis labels.

notation=*n* Uses engineering (*n*=0) or scientific (*n*=1) notation for tick mark labels.

Affects tick mark labels displayed exponentially. See highTrip and lowTrip. Does not affect log axes.

nticks=*n* Specifies the approximate number of ticks marks (*n*) on axis.

prescaleExp=*exp* Multiplies axis range by 10^*exp* for tick labeling and *exp* is subtracted from the axis label exponent. In other words, the exponent is moved from the tick labels to the axis label. (This affects the display only, not the source data.)

sep=*s* Specifies the minimum number of screen points (s) between minor ticks.

standoff=*s* Suppresses (*s*=0) or enables (*s*=1) axis standoff.

Axis standoff prevents waves or markers from covering the axis.

stLen=*p* Sets the length of minor (“small”) tick marks to *p* points. If *p* is zero, it uses the default length. *p* may be fractional.

stThick=*p* Sets the thickness of minor (“small”) tick marks to *p* points. If *p* is zero, it uses the default thickness. *p* may be fractional.

tick=*t* Sets tick position.

t=0: Outside axis.

t=1: Crossing axis.

t=2: Inside axis.

t=3: None.

In a category plot, adding 4 to the usual values for the tick keyword will place the tick marks in the center of each category rather than at the edges.

In a normal (non-category) plot specifying *t*=4 or *t*=5 replaces tick marks and axis lines with rectangular regions with alternating fills. This feature, which was added in Igor Pro 8.00, is used for cartographic plots.

t=4 gives even fills and *t*=5 gives odd fills. The btLen keyword determines the thickness of the fill area.

It is recommended that you use mirror axes and axisOnTop=1 as shown in this example:

```
Make jack=sin(x/8)
Display jack
ModifyGraph axisOnTop=1
ModifyGraph mirror=1
ModifyGraph standoff=0
ModifyGraph tick=4
```

tickEnab={*lowTick*,*highTick*}

Restricts axis ticking to a subrange of normal. Ticks are drawn and labelled only if they fall within this inclusive numerical range.

tickExp=*te* *te*=1 forces tick labels to exponential notation when labels have units with a prefix. *te*=0 turns this off.

tickUnit=*tu* Suppresses (*tu* =1) or turns on (*tu* =0) units labels attached to tick marks.

tickZap={[*v1* [,*v2* [,*v3*]]]}

Suppresses drawing of the tick mark label for values given in the list. This is useful when you have crossing axes to prevent tick mark labels from overlapping. The list may contain zero, one, two or three values. The values must be exact to suppress the label.

tkLblRot=*r*
Rotates the tick mark labels by *r* degrees. *r* is a value from -360 to 360. Rotation is counterclockwise and starts from the label's normal orientation.

tlOffset=*o*
Offsets the tick mark labels by *o* fractional points relative to the default tick mark label position. Positive is away from the axis.

ttLen=*p*
Sets the length of subminor (“tiny”) tick marks to *p* points. If *p* is zero, it uses the default length. *p* may be fractional. Subminor ticks are used only in log axes.

ttThick=*p*
Sets the thickness of subminor (“tiny”) tick marks to *p* points. If *p* is zero, it uses the default thickness. *p* may be fractional.

userticks={*tickPosWave*, *tickLabelWave*}

Draws axes with purely user-defined tick mark positions and labels, overriding other settings. *tickPosWave* is a numeric wave containing the desired positions of the tick marks, and *tickLabelWave* is a text wave containing the labels. See **User Ticks from Waves** on page II-313 for an example.

The tick mark labels can be multiline and use styled text. For more details, see **Fancy Tick Mark Labels** on page II-358.

tickPosWave need not be monotonic. Igor will plot a tick if a value is in the range of the axis. Both linear and log axes are supported.

Graph margins will adjust to accommodate tick labels. This will not prevent overlap between labels, which you will need fix yourself.

userticks=0
Removes the userticks setting returning the axis to the previously-set mode.

useTSep=*t*
t=1 displays a thousand's separator character between every group of three digits in the tick mark label (e.g., "1,000" instead of "1000"). The default is *t*=0.

zapLZ=*t*
Removes (*t*=1) leading zeros from tick mark labels. For example 0.5 becomes .5 and -0.5 becomes -.5. Default is *t*=0.

zapTZ=*t*
Removes (*t*=1) trailing zeros from tick mark labels. The the radix point will also be removed if all digits are zero. Default is *t*=0.

zero=*z*
Controls the zero line.

z=0: A zero line at x=0 or y=0.

The line style is set to *z*-1. See **ModifyGraph (traces)** on page V-613, lStyle keyword, for details on line styles.

z=1: No zero line.

zeroThick=*zt*
Sets the thickness of the zero line in points, from 0.0 to 5.0 points. *zt*=0.0 means the zero line thickness automatically follows the thickness of the axis; this is the default. You can use 0.1 for a thin zero line thickness.

ZisZ=*t*
t=1 uses the single digit 0 as the zero tick mark label (if any) regardless of the number of digits used for other labels. Default is *t*=0.

Flags

/W=*winName*	Modifies the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.
/Z	Does not generate an error if the named axis does not exist in a style macro.

Details

With the prescaleExp parameter, you can force tick and axis label scaling to values different from the defaults. For example, if you have data whose X scaling ranges from 9pA to 120pA and you display this on a log axis, the tick marks will be labelled 10pA and 100pA. But if you really want the tick marks labeled 10 and 100 with pA in the axis label, you can set the prescaleExp to 12. To see this, execute the following commands:

```
Make/O jack=x
Display jack
SetScale x,9e-12,120e-12,"A",jack
ModifyGraph log(bottom)=1
```

then execute:

```
ModifyGraph prescaleExp(bottom)=12
```

The tickExp parameter applies to units that do not traditionally use SI prefix characters. For example, one usually speaks of 10^{-3} Torr and not mTorr. To see how this feature works, execute the following example commands:

```
Make/O jack=x
Display jack
SetScale x,1E-7,1E-5,"Torr",jack
ModifyGraph log(bottom)=1
```

then execute:

```
ModifyGraph tickExp(bottom)=1
```

at this point, the tick mark labels have Torr in them. If you want to eliminate the units from the tick marks, execute:

```
ModifyGraph tickUnit(bottom)=1
```

and if you now want Torr in the label string, use the \U escape in the label string:

```
Label bottom "\\U"
```

To see the effect of linTkLabel, execute these commands:

```
Make/O jack=x
Display jack
SetScale x,1E-7,1E-5,"Torr",jack
```

then execute:

```
ModifyGraph linTkLabel(bottom)=1
```

and then try:

```
ModifyGraph tickExp(bottom)=1
```

and finally:

```
ModifyGraph tickUnit(bottom)=1
```

ModifyGraph *(colors)*

ModifyGraph [/W=winName/Z] *key* [(*axisName*)]=(*r*,*g*,*b*,[*a*])
[, *key* [(*axisName*)]=(*r*,*g*,*b*[,*a*])]…

This section of ModifyGraph relates to modifying the use of colors in a graph.

Parameters

Most of the *key* parameters may take an optional *axisName* enclosed in parentheses. *axisName* is “left”, “right”, “top”, “bottom” or the name of an free axis such as “vertCrossing”.

Where the parameter descriptions indicate an “(*axisName*)”, it may be omitted to change all axes in the graph.

r, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. (0, 0, 0) specifies opaque black and (65535, 65535, 65535) specifies opaque white.

Parameter Specification	Object Colored
`alblRGB(axisName)=(r,g,b[,a])`	Axis labels
`axRGB(axisName)=(r,g,b[,a])`	Axis
`cbRGB=(r,g,b[,a])`	Control bar background
`gbRGB=(r,g,b[,a])`	Graph background
`gbGradient`	See **Gradient Fills** on page III-498 for details.
`gbGradientExtra`	See **Gradient Fills** on page III-498 for details.
`gridRGB(axisName)=(r,g,b[,a])`	Axis grid lines
`tickRGB(axisName )=(r,g,b[,a])`	Axis Tick marks
`tlblRGB(axisName)=(r,g,b[,a])`	Axis Tick labels
`wbRGB=(r,g,b[,a])`	Window background
`wbGradient`	See **Gradient Fills** on page III-498 for details.
`wbGradientExtra`	See **Gradient Fills** on page III-498 for details.

Flags

/W=*winName* Modifies the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/Z Does not generate an error if the named axis does not exist in a style macro.

Details

On Windows, use maximum white to set the control bar background color to track the 3D Objects color in the Appearance Tab of the Display Properties control panel:

```
ModifyGraph cbRGB=(65535,65535,65535)
```

See Also

See **Instance Notation** on page IV-20.

ModifyImage

ModifyImage [/W=*winName*] *imageInstance, keyword = value*
[, *keyword = value*]…

The ModifyImage operation changes properties of the given image in the top graph (or the specified graph if /W is used). *imageInstance* is the name of the image to be altered. This name is usually simply the name of the matrix wave containing the image data. If the same matrix wave is displayed more than once, you must append #0, #1 etc. to the name to distinguish which is which.

imageInstance can also take the form of a null name with an instance number to affect the instanceth image. That is,

```
ModifyImage ''#1
```

modifies the appearance of the second image that was appended to the top graph, no matter what the image names are. Note: two single quotes are used, not a double quote.

Parameters

Here are the keyword-value pairs. These apply to false color images in which the data in the matrix is used as an index into a color table. They do not apply to direct color images in which the data in the matrix specifies the color directly.

cindex=*matrixWave* Sets the Z value mapping mode such that image colors are determined by doing a lookup in the specified matrix wave.

matrixWave is a 3 column wave that contains red, green, and blue values from 0 to 65535. (The matrix can actually have more than three columns. It ignores any extra columns.)

The color at Z=z is determined by finding the RGB values in the row of *matrixWave* whose scaled X index is z. In other words, the red value is *matrixWave*(z)[0], the green value is *matrixWave*(z)[1] and the blue value is *matrixWave*(z)[2].

If *matrixWave* has default X scaling, where the scaled X index equals the point number, then row 0 contains the color for Z=0, row 1 contains the color for Z=1, etc.

If you use cindex, you should not use ctab in the same command.

ctab={*zMin*, *zMax*, *ctName*, *reverse*}

Sets the z mapping mode by which values in the matrix are mapped linearly into the color table specified by *ctName*.

zMin and *zMax* set the range of z values to map. Omit *zMin* or *zMax* to leave as is or use * to autoscale.

The color table name can be missing if you want to leave it as is.

ctName can be any color table name returned by the CTabList function, such as Grays or Rainbow (see **Image Color Tables** on page II-392) or the name of a 3 column or 4 column color table wave (**Color Table Waves** on page II-399).

A color table wave name supplied to `ctab` must not be the name of a built-in color table (see **CTabList**). A 3 column or 4 column color table wave must have values that range between 0 and 65535. Column 0 is red, 1 is green, and 2 is blue. In column 3 a value of 65535 is opaque, and 0 is fully transparent.

Set *reverse* to 1 to reverse the color table. Setting it to 0 or omitting it leaves the color table unreversed.

ctabAutoscale=*autoBits*

Sets the range of data used for autoscaling ctab * values.

Bit 0:	Autoscales only the XY subset being displayed.
Bit 1:	Autoscales only the current plane being displayed.

If neither bit is set (if *autoBits* = 0, the default), then all of the data in the image wave is used to autoscale the *'d *zMin*, *zMax* values for ctab.

eval={*value*, *red*, *green*, *blue*, [*alpha*]}

If the red, green, and blue values are in the valid range for a color value (0 to 65535) the explicit value-color pair is added (or updated if value already exists). If the color values are out of range (-1 is suggested) then the value is removed from the list if it is present (no error if it is not).

alpha is optional: a value of 65535 is opaque, and 0 is fully transparent.

explicit=1 *or* 0 Turns explicit (monochrome) mode on (1) or off (0). Meant to be used with unsigned byte data but will do the best it can for other types. If value of data is equal to one of the defined explicit values then its defined color is used otherwise the pixel will be blank. The default predefined values are:

255:	black
0:	white

You can add, change, or delete explicit values with the eval keyword.

genericNotRGB=*v* Controls the auto-detection of three and four plane images:

v=1: Suppresses the auto-detection of three and four plane images as direct color (RGB or RGBA), like AppendImage/G=1.

v=0: Three and four plane images are treated as direct color (RGB or RGBA). This is the default if genericNotRGB is omitted.

The genericNotRGB keyword was added in Igor Pro 9.00.

imCmplxMode=*m* Sets complex data display mode.

m=0: Magnitude (default).

m=1: Real only.

m=2: Imaginary only.

m=3: Phase in radians.

interpolate= *mode* *mode* = 1 turns on smoothing of the boundaries between pixels. Since this is implemented via system graphics calls and not by Igor actually doing the interpolation, it will not affect EPS or EMF export on Windows and will not affect EPS export on Mac. Although this may create a more esthetically pleasing display, it is not clear that it is appropriate for scientific data.

mode = -1 forces pixels to be drawn as individual rectangles. This is sometimes needed when a third-party program improperly interpolates PDF or EPS exported images.

log= 1 or 0 0 sets the default linearly-spaced false-image colors.

1 turns on logarithmically-spaced false-image colors. This requires that the image values be greater than 0 to display correctly.

Affects the image colors for color table and color index images only (see **Color Table Details** on page II-395 and **Indexed Color Details** on page II-400).

lookup= *waveName* Specifies an optional 1D wave that can be used to modify the mapping of scaled z values into the color table specified with the ctab parameter. Values should range from 0.0 to 1.0. A linear ramp from 0 to 1 would have no effect while a ramp from 1 to 0 would reverse the image. Used to apply gamma correction to grayscale images or for special effects. Use a NULL wave (`$""`) to remove the option.

maxRGB=(*red*, *green*, *blue*, [*alpha*])

Sets the color of image values greater than the ctab *zMax* or greater than the cindex of the *matrixWave* maximum X scaling value. Also turns max color mode on.

The *red*, *green,* and *blue* color values are in the range of 0 to 65535.

alpha is optional: a value of 65535 is opaque, and 0 is fully transparent.

maxRGB=1 *or* 0 *or* NaN

Turns max color mode off, on, or transparent. These modes affect the display of image values greater than the ctab *zMax* or greater than the cindex of the *matrixWave* maximum X scaling value.

1: Turns on max color mode. The color of the affected image pixels is black or the last color set by maxRGB=(*red*, *green*, *blue*).

0: Turns off max color mode (default). The color of the affected image pixels is the last color table or color index color.

NaN: Transparent max color mode. The affected image pixels are not drawn.

minRGB=(*red*, *green*, *blue*, [*alpha*])

Sets the color of image values less than the ctab *zMin* or less than the cindex of the *matrixWave* minimum X scaling value. Also turns min color mode on.

The *red*, *green*, and *blue* color values are in the range of 0 to 65535.

alpha is optional: a value of 65535 is opaque, and 0 is fully transparent.

minRGB=1 *or* 0 *or* NaN

Turns min color mode off, on, or transparent. These modes affect the display of image values less than the ctab *zMin* or less than the cindex of the *matrixWave* minimum X scaling value.

1: Turns on min color mode. The color of the affected image pixels is black or the last color set by minRGB=(*red*, *green*, *blue*).

0: Turns off min color mode (default). The color of the affected image pixels is the first color table or color index color.

NaN: Transparent min color mode. The affected image pixels are not drawn.

plane=*p* Determines which part of a 3D or 4D image wave to display.

The meaning of *p* depends on the nature of the image wave. If the size of the layer dimension of the image wave is exactly three then the wave is treated as RGB data with R, G, and B data in the three layers. If the size of the layer dimension is exactly four, then the wave is treated as RGBA data, with A in the fourth layer. Otherwise each layer of the wave is treated as a separate grayscale image.

Plane=p With RGB Data

If the wave is 3D, plane=*p* has no effect.

If the wave is 4D, each chunk contains a different set of R, G and B layers and *p* selects which chunk to display.

Plane=p With Grayscale Data

If the wave is 3D, *p* selects which layer to display.

If the wave is 4D, plane=*p* acts as if all of the chunks were combined into a virtual 3D wave and *p* selects which layer of this virtual 3D wave to display.

rgbMult=*m* If *m* is non-zero, direct color values in a 3-plane RGB or 4-plane RGBA image are multiplied by *m*. (Alpha values are not multiplied.) This would typically be used for 10, 12 or 14 bit integers in a 16 bit word. For example, if your image data is 14 bits, use rgbMult=4.

Flags

/W=*winName* Directs action to a specific window or subwindow rather than the top graph window. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

See Also

AppendImage and **RemoveImage**.

ModifyLayout

`ModifyLayout [`***`flags`***`]` ***`key`***` [`**`(`*`objectName`*`)`**`] `***`=value`***` [, `***`key`***` [`**`(`*`objectName`*`)`**`] `***`=value`***`]`…

The ModifyLayout operation modifies objects in the top layout or in the layout specified by the /W flag.

Parameters

Each *key* parameter may take an optional *objectName* enclosed in parentheses. If "(*objectName*)" is omitted, all objects in the layout are affected.

Though not shown in the syntax, the optional "(*objectName*)" may be replaced with "[*objectIndex*]", where *objectIndex* is zero or a positive integer denoting the object to be modified. "[0]" denotes the first object appended to the layout, "[1]" denotes the second object, etc. This syntax is used for style macros, in conjunction with the /Z flag.

The parameter descriptions below omit the optional "(*objectName*)".

The "units", "mag" and "bgRGB" keywords apply to the layout as a whole, not to a specific object and do not accept an *objectName*.

bgRGB=(*r*,*g*,*b*[,*a*]) Specifies the background color for the layout. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

columns=*c* Specifies the number of columns for a table object.

fidelity=*f* Controls the drawing of layout objects.

f=0: Low fidelity.

f=1: High fidelity.

frame=*f* Specifies the type of frame enclosing the object.

f=0: No frame.
f=1: Single frame (default).
f=2: Double frame.
f=3: Triple frame.
f=4: Shadow frame.

gradient See **Gradient Fills** on page III-498 for details.

gradientExtra See **Gradient Fills** on page III-498 for details.

height=*h* Sets the height of the object.

left=*l* *l* is the horizontal coordinate of the left edge of the object relative to the left edge of the paper.

mag=*m* Sets the on screen layout magnification where *m* is a value between 0.5 and 10.

m=1 corresponds to 100%.

Factors of two, such as *m*=.25, *m*=.5, *m*=1, *m*=2, tend to produce the best on screen graphics.

rows=*r* Specifies the number of rows for table object.

top=*t* *t* is the vertical coordinate of the top edge of the object relative to the top edge of the paper.

trans=*t* Controls the transparency of the layout object:

t=0: Opaque (default).

t=1: Transparent. For this to be effective, the object itself must also be transparent. Annotations have their own transparent/opaque settings. Graphs are transparent only if their backgrounds are white. PICTs may have been created transparent or opaque, and Igor cannot make an opaque PICT transparent.

units=*u* Sets dimension units in the layout info panel and in the Modify Objects dialog.

u=0: Points.

u=1: Inches.

u=2: Centimeters.

width=*w* Sets the object width.

Flags

/I Dimensions in inches.

/M Dimensions in centimeters.

/W=*winName* *winName* is the name of the page layout window to be modified. If /W is omitted or if *winName* is `$""`, the top page layout is modified.

/Z Does not generate an error if the indexed or named object does not exist in a style macro.

The /I and /M flags affect the units of the parameters for the left, top, width and height keywords only. If neither /I nor /M is present then the parameters for the left, top, width and height keywords are points.

Details

Note that the units keyword affects only the units used in the layout info panel and in the Modify Objects dialog. It has nothing to do with the units used for the left, top, width and height keywords. Those units are points unless the /I or /M flags is present.

See Also

NewLayout, **AppendLayoutObject**, **RemoveLayoutObjects**, **LayoutPageAction**

ModifyPanel

`ModifyPanel` `[`***`/W=winName`***`]` ***`keyword = value`*** `[,` ***`keyword = value`*** `…]`

The ModifyPanel operation modifies properties of the top or named control panel window or subwindow.

Parameters

keyword is one of the following:

cbRGB=(*r*,*g*,*b*[,*a*]) Specifies the background color of the entire control panel or the graph's control bar area. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

drawInOrder=*d* Determines the drawing order of controls in the control panel.

d=0: Draw tab controls before any other controls (default).

d=1: Draw all controls in the order in which they appear in the recreation macro, which is also the creation order.

The drawInOrder keyword was added in Igor Pro 8.00.

Prior to Igor Pro 8.00 tab controls (see **TabControl**) were always drawn before all other controls. This prevented placing a tab control inside a groupbox with a colored interior. drawInOrder=1 forces Igor to draw controls in creation order, which is the same as the order in which they appear in a recreation macro.

expand=*e* Sets the expansion factor of the panel. *e* is a number between 0.25 to 8.0. Values of *e* greater than 1.0 make the panel, its controls, and its subwindows, appear larger than normal.

Though rarely needed, using the expand keyword, you can set the expansion of panel subwindows independent of the main panel window's expansion.

When you change the expansion of a top-level control panel window, the panel window automatically resizes itself.

The expand keyword was added in Igor Pro 9.00.

See **Control Panel Expansion** on page III-443 for further discussion.

fixedSize=*f* Controls the resizing of the panel window.

f=0: Panel can be resized (default).

f=1: Panel cannot be resized by adjusting the size box or frame (nor maximized on Windows), but the window can be minimized (on Windows) and the MoveWindow operation can still change the size.

The `fixedSize` keyword overrides any previous size limit set using the `SetWindow sizeLimit` command. If you try to use `SetWindow sizeLimit` on a window with `fixedSize=1`, Igor generates an error.

frameInset= *i* Specifies the number of pixels by which to inset the frame of the panel subwindow. Mostly useful for overlaying panels in graphs to give a fake 3D frame a better appearance.

frameStyle= *f* Specifies the frame style for a panel subwindow.

f=0: None.

f=1: Single.

f=2: Indented.

f=3: Raised.

f=4: Text well.

The last three styles are fake 3D and will look good only if the background color of the enclosing space and the panel itself is a light shade of gray.

noEdit= *e* Sets the editability of the panel.

e=0: Editable (default).

e=1: Not editable. For a panel window, the Panel menu item is not present and the ShowTools command is ignored. For a panel subwindow, it can not be activated by clicking.

Flags

/W= *winName* Modifies the control panel in the named graph or control panel window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

On Windows, set *r*, *g*, and *b* to 65535 (maximum white) to set the background color of the control panel to track the 3D Objects color in the Appearance Tab of the Display Properties control panel.

See Also

The **NewPanel** operation.

Controls in Graphs on page III-441.

ModifyProcedure

ModifyProcedure [/A[=*all*] /W=*procWinTitleListStr*] /Z[=*z*]
[procedure=*functionOrMacroNameStr*, hide=*h*, lock=*ro*, writeProtect=*wp*,
userCanOverride=*ovr*]

The ModifyProcedure operation modifies one or more procedure windows, depending on the /A and /W flags and on the procedure keyword.

The ModifyProcedure operation was added in Igor 8.03.

Use /W="Procedure" to modify the built-in procedure window.

Use /W=procWinTitleListStr to modify a procedure window specified by its title, such as /W="MyProc" for packed procedure files or /W="MyProc.ipf" for standalone procedure files.

Use procedure=*functionOrMacroNameStr* to modify a procedure window containing a particular function.

See **Specifying Which Procedure Windows to Modify** below for other usages.

If the procedure window is associated with a standalone file, as opposed to being packed, the file's read-only setting may be set or cleared, as described below for the lock and writeProtect keywords.

Parameters

procedure=*functionOrMacroNameStr*

functionOrMacroNameStr is a string containing the name of a function or macro in the procedure window to modify.

functionOrMacroNameStr may be a simple name or may include independent module and/or module name prefixes to designate static functions. For use with independent modules, see **Using ModifyProcedure With Independent Modules** below.

lock=*ro* Sets or clears the read-only state of the targeted procedure windows.

ro=-2: Clears the read-only state, like *ro*=0.

In addition, if the procedure window is associated with a standalone file, as opposed to being packed, the file's read-only setting is set to unlocked, as with **SetFileFolderInfo**/RO=0.

ro=-1: The read-only state is not changed.

ro=0: Clears the read-only state. This allows the user to change the text if the procedure window is also not write-protected.

ro=1: Sets the read-only state. The procedure window shows a lock icon in the lower left corner.

ro=-2: Sets the read-only state, like *ro*=1.

In addition, if the procedure window is associated with a standalone file, as opposed to being packed, the file's read-only setting is set to locked, as with **SetFileFolderInfo**/RO=1.

hide=*h* Hides or shows the targeted procedure windows. Added in Igor 9.00.

h=-1: The procedure window visibility is not changed.

h=0: Shows the procedure window.

h=1: Hides the procedure window.

writeProtect=*wp* Sets or clears the write-protect state of the targeted procedure windows.

wp=-2: Clears the write-protect state, like *wp*=0.

In addition, if the procedure window is associated with a standalone file, as opposed to being packed, the file's read-only setting is set to unlocked, as with **SetFileFolderInfo**/RO=0.

wp=-1: The write-protect state is not changed.

wp=0: Clears the write-protect state. The user can change the text if the procedure window is also not locked.

wp=1: Sets the write-protect state. The procedure window shows a write-protect icon (a pencil with a red circle-and-line indicator).

wp=2: Sets the write-protect state, like *wp*=1.

In addition, if the procedure window is associated with a standalone file, as opposed to being packed, the file's read-only setting is set to locked, as with **SetFileFolderInfo**/RO=1.

userCanOverride=*ovr*

Controls the ability of the user to change the write-protect state of the targeted procedure windows.

ovr=0: Prevents the user from changing the write-protect state by clicking the write-protect icon.

ovr=1: Allows the user to change the write-protect state by clicking the write-protect icon.

Flags

/A[=*all*] Specifies all procedure windows as the targets of the ModifyProcedure operation.

Use /A=1 to target all procedure windows in the ProcGlobal module. /A alone is the same as /A=1.

Use /A=2 to target all procedure windows in independent modules. See **Using ModifyProcedure With Independent Modules** for details.

/W=*procWinTitleListStr* Specifies a particular procedure window or a list of procedure windows as the targets of the ModifyProcedure operation.

If *procWinTitleListStr* is the title of a single procedure window, this specifies that procedure window as the single target.

If *procWinTitleListStr* contains a semicolon-separated list of procedure window titles, this specifies those procedure windows as targets.

See **Specifying Which Procedure Windows to Modify** below for details.

/Z[=*z*] Prevents procedure execution from aborting if an error occurs. Use /Z or /Z=1 if you want to handle errors from ModifyProcedure in your procedures rather than having execution abort.

Details

When changing the lock or write-protect states for a standalone file (lock=+/-2 or writeProtect=+/-2), there are scenarios where the file's locked state can't be changed. This would occur, for example, if the file is open in another program or if you do not have sufficient privileges to modify the file.

Specifying Which Procedure Windows to Modify

The simplest way to modify one procedure window whose title you know is to use /W:

```
ModifyProcedure/W="My Procedure.ipf" writeProtect=2   // 2 for standalone file
ModifyProcedure/W="Procedure" writeProtect=1      // Modifies built-in procedure window
```

Use /A=1 to target all procedure windows in the ProcGlobal module:

```
ModifyProcedure/A=1 writeProtect=1
```

Don't specify both /A and /W as this is ambiguous and generates an error.

Use procedure=*functionOrMacroNameStr* to modify all procedure windows in the ProcGlobal module containing a procedure with the specified name:

```
ModifyProcedure procedure=MyFunction, writeProtect=1
```

If you omit procedure=*functionOrMacroNameStr*, /A, and /W then the built-in procedure window is modified.

Use *functionOrMacroNameStr* only when you are not certain which procedure window contains the function or macro. If a procedure window has syntax errors that prevent Igor from determining where functions and macros start and end, then ModifyProcedure may not be able to locate the correct procedure window in which case it returns an error.

You can use /W and procedure=*functionOrMacroNameStr* to search for a function or macro in one or more procedure windows. This is useful when *functionOrMacroNameStr* is the name of a static function in a procedure window that uses #pragma moduleName.

Using ModifyProcedure With Independent Modules

To work with independent modules, you must enable independent module development. This is for advanced programmers only. See **Independent Modules** on page IV-238 for details. The material in this section assumes that independent module development is enabled.

/W=*procWinTitleListStr* can be a list of procedure window titles each followed by a space and an independent module name in brackets. Then procedure=*functionOrMacroNameStr* applies to the specified procedure windows and independent modules.

For example, if any procedure file contains these statements:

```
#pragma IndependentModule=myIM
#include <Axis Utilities>
```

then the commands

```
ModifyProcedure/W="Axis Utilities.ipf [myIM]" procedure="HVAxisList"
Print S_windowList, V_isReadOnly, V_writeProtect
```

report the title, read-only state, and write-protect state of the procedure window that contains the HVAxisList function, which is in the "Axis Utilities.ipf" file and the independent module myIM.

Similarly, procedure=*functionOrMacroNameStr* may also specify an independent module prefix followed by the # character. The preceding ModifyProcedure command can be rewritten as:

```
ModifyProcedure/W="Axis Utilities.ipf" procedure="myIM#HVAxisList"
```

or simply:

```
ModifyProcedure procedure="myIM#HVAxisList"
```

You can use the same syntax to modify the procedure window containing a static function in a non-independent module procedure file using a module name instead of, or in addition to, the independent module name.

procWinTitleListStr can also be just an independent module name in brackets to target all procedure windows that belong to the named independent module containing the specified function:

```
ModifyProcedure/W="[myIM]" procedure="HVAxisList", writeProtect=0
```

Output Variables

The ModifyProcedure operation returns information in the following variables for the last procedure window targeted by the parameters. The information returned reflects the state of affairs after the ModifyProcedure command performs any requested modifications except for V_wasHidden.

`V_wasHidden`	If the procedure window was previously hidden, V_wasHidden is set to 1, otherwise to 0.
`V_isReadOnly`	If the procedure's lock icon is showing, V_isReadOnly is set to 1, otherwise to 0.
`V_writeProtect`	If the pencil with the red line icon is showing, V_writeProtect is set to 1, otherwise to 0.
`V_userCanOverride`	Set to 1 (the default) if the user can change the write-protect state to writable.

`S_windowList`	Set to a semicolon-separated list of procedure window titles that match the parameters, with an appended independent module name in brackets if necessary. If S_windowList is empty, then no procedure windows matched the parameters, and no modifications were performed.

You can obtain the information provided by the output variables without modifying any procedure windows by omitting the lock, writeProtect, and userCanOverride keywords.

Examples

```
// Completely unlock main Procedure window
ModifyProcedure lock=0, writeProtect=0, userCanOverride=1; Print S_windowList

// Completely unlock the procedure window containing the function named MyFunction
ModifyProcedure procedure="MyFunction", lock=0, writeProtect=0, userCanOverride=1
Print S_windowList            // Print procedure window title

// Lock all procedure windows in the myIM independent module, even those included into it,
// and print the titles of the matching procedure windows
Execute "SetIgorOption IndependentModuleDev=1"
ModifyProcedure/W="[myIM]" lock=1, writeProtect=1, userCanOverride=0; Print S_windowList

// Unlock all procedure windows in the ProcGlobal module
ModifyProcedure/A=1 lock=0, writeProtect=0, userCanOverride=1

// Hide all procedures in ProcGlobal and all independent modules
ModifyProcedure/A=2 hide=1    // Equivalent to HideProcedures
```

See Also

Independent Modules on page IV-238

HideProcedures, **DisplayProcedure**, **ProcedureText**, **ProcedureVersion**

DoWindow, **WinList**

MacroList, **FunctionList**

ModifyTable

ModifyTable [**/W=*winName*/Z**] ***key*** [**(*columnSpec*)**] ***=value*** [, ***key*** [**(*columnSpec*)**] ***=value***]…

The ModifyTable operation modifies the appearance the top or named table window or subwindow.

Parameters

Many of the parameter keywords take an optional *columnSpec* enclosed in parentheses. Usually *columnSpec* is simply the name of a wave displayed in the table. All table columns are affected when you omit `(`*columnSpec*`)`.

More precisely, column specifications are wave names for waves in the current data folder or data folder paths leading to waves in any data folder optionally followed by the suffixes .i, .l, .d, .id or .ld to specify dimension indices, dimension labels, data values, dimension indices and data values, or dimension labels and data values of the wave. For example, `ModifyTable font(myWave.i)="Helvetica"`. If the wave is complex, the column specification may be followed by `.real` or `.imag` suffixes.

One additional *columnSpec* is Point, which refers to the first column containing the dimension index numbers. If multidimensional waves are displayed in the table, this column may have the title “Row”, “Column”, “Layer”, “Chunk” or “Element”, but the *columnSpec* for this column is always Point. See **Column Names** on page II-241 for details.

Though not shown in the syntax, the optional `(`*columnSpec*`)` may be replaced with `[`*columnIndex*`]`, where *columnIndex* is zero or a positive integer denoting the column to be modified. `[0]` denotes the Point column, `[1]` denotes the first column appended to the table, `[2]` denotes the second appended column, etc. This syntax is used for style macros, in conjunction with the /Z flag.

You can use a range of column numbers instead of just a single column number, for example `[0,3]`.

The parameter descriptions below omit the optional (*columnSpec*).

alignment=*a* Sets the alignment of table cell text.

a=0: Left aligned.

a=1: Center aligned.

a=2: Right aligned.

autosize={*mode*, *options*, *padding*, *perColumnMaxSeconds*, *totalMaxSeconds*}

Autosizes the specified column or columns.

mode=0: Sets width of each data column from a given multidimensional wave individually.

mode=1: Sets width of all data columns from a given multidimensional wave the same.

options is a bitwise parameter. Usually 0 is the best choice.

Bit 0: Ignores column names.

Bit 1: Ignores horizontal indices.

Bit 2: Ignores data cells.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

padding specifies extra padding for each column in points. Use -1 to get the default amount of padding (16 points).

perColumnMaxSeconds specifies the maximum amount of time to spend autosizing a single column. Use 0 to get the default amount of time (one second).

totalmaxSeconds specifies the maximum amount of time for autosizing the entire table. Use 0 to get the default amount of time (ten seconds).

digits=*d* Specifies the number of digits after decimal point or, for hexadecimal and octal columns, the number of total digits.

elements=(*row*, *col*, *layer*, *chunk*)

Selects the view of a multidimensional wave in the table. The values given to *row*, *col*, *layer*, and *chunk* specify how to change the view.

-1: No change from current view.

-1: Display this dimension vertically.

-3: Display this dimension horizontally.

≥0: For waves with 3 or 4 dimensions, display this element of the other dimensions.

See **ModifyTable Elements Command** on page II-263 for a detailed discussion of

entryMode=*m* Queries or sets the table’s entry line mode.

m=0: Just queries.

m=1: Accepts any entry that was started if possible.

m=2: Cancels any entry that was started if possible.

If *m* is 0 then the entry line state is not changed but is returned via V_flag as follows:

0: No entry is in progress.

-1: An entry is in progress and is valid.

Other: An entry is in progress and is invalid.

In Igor Pro 8.03 and later, S_value is set to contain the text showing in the entry line.

If *m* is 1 then the entry is accepted if it is valid and its state is returned via V_flag as follows:

0:	No entry is in progress.
-1:	The entry was accepted.
Other:	The entry is invalid and was not accepted.

In Igor Pro 8.03 and later, S_value is set to contain the text showing in the entry line whether or not an entry was in progress and whether or not i was accepted.

If *m* is 2 then the entry is cancelled if possible and its state is returned via V_flag as follows:

0:	No entry is in progress.
-1:	The entry was cancelled.

In Igor Pro 8.03 and later, S_value is set to contain the text showing in the entry line after the entry was cancelled.

font="*fontName*"   Sets font used in the table, e.g., `font="Helvetica"`.

format=*f*   Sets the data format for the table.

f=0:	General.
f=1:	Integer.
f=2:	Integer with thousands (e.g., "1,234").
f=3:	Fixed point (e.g., "1234.56").
f=4:	Fixed point with thousands (e.g., "1,234.56").
f=5:	Exponential (scientific only).
f=6:	Date format.
f=7:	Time format (always 24 hour time).
f=8:	Date&time format (date followed by time).
f=9:	Octal.
f=10:	Hexadecimal.

You cannot apply date or date&time formats to a wave that is not double-precision (see **Date, Time, and Date&Time Units** on page II-69). To avoid this error, use **Redimension** to change the wave to double-precision.

frameInset= *i*   Specifies the number of pixels by which to inset the frame of the table subwindow.

frameStyle= *f*   Specifies the frame style for a table subwindow.

f=0:	None.
f=1:	Single.
f=2:	Double.
f=3:	Triple.
f=4:	Shadow.
f=5:	Indented.
f=6:	Raised.
f=7:	Text well.

The last three styles are fake 3D and will look good only if the background color of the enclosing space and the table itself is a light shade of gray.

horizontalIndex=*h* Controls what is displayed in the horizontal index row when multidimensional waves are displayed.

h=0: Displays dimension labels if the multidimensional wave's label column is displayed, otherwise displays numeric indices (default).

h=1: Always displays numeric indices for multidimensional waves.

h=2: Always displays dimension labels for multidimensional waves.

The horizontal index row appears below the row of column names if the table contains a multidimensional wave. Use horizontalIndex to override the default behavior in order to display labels for the horizontal dimension while displaying numeric indices for the vertical dimension or vice versa.

The horizontalIndex keyword controls the horizontal index row only. To control what is displayed vertically, use **AppendToTable** to append a numeric index or dimension label column.

rgb=(*r*,*g*,*b*[,*a*]) Sets color of text. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

selection=(*firstRow*, *firstCol*, *lastRow*, *lastCol*, *targetRow*, *targetCol*)

Sets the selected cells in the table.

If any of the parameters have the value -1 then the corresponding part of the selection is not changed.

Otherwise they set the first and last selected cell and the target cell. Row and column values are 0 or greater. The Point column can not be selected.

The proposed parameters are clipped to avoid invalid combinations, such as the last selected row being before the first selected row.

With one exception, it does not support selecting unused cells. Therefore the proposed selection is clipped to prevent this. The exception is that, if the parameters call for selecting the first cell in the first unused column, then this is permitted.

showFracSeconds=*s* Shows (*s*=1) or hides (*s*=0; default) fractional seconds.

showParts=*parts* Specifies what elements of the table should be visible. Other elements are hidden.

parts is a bitwise parameter specifying what to show.

bit 0:	Entry line and other top line controls.
bit 1:	Name row.
bit 2:	Horizontal index row.
bit 3:	Point column.
bit 4:	Horizontal scroll bar.
bit 5:	Vertical scroll bar.
bit 6:	Insertion cells.
bit 7:	Insertion cells.

All other bits are reserved and must be set to zero except that you can pass -1 to indicate that you want to show all parts of the table.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Presentation tables in subwindows in graphs and page layouts do not have an entry line or scroll bars and therefore never show these items.

See **Parts of a Table** on page II-235 and **Showing and Hiding Parts of a Table** on page II-237 for further information.

sigdigits=*d* *d* is the number of significant digits when the numeric format is general.

size=*s* Font size, e.g., `size=14`.

style=*n* — *n* is a bitwise parameter with each bit controlling one aspect of the column's font style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

For example, bold underlined is $2^0 + 2^2 = 1 + 4 = 5$. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

title="*title*" — Sets the title of a column to *title*.

topLeftCell=(*row*, *column*)

Scrolls the table contents so that the cell identified by (*row*, *column*) is the top left visible data cell, or as close as possible.

If *row* is -1 then the table's vertical scrolling is not changed. If *column* is -1 then the table's horizontal scrolling is not changed.

If they are positive, *row* and *column* are zero-based numbers which are clipped to valid values before being used. *row*=0 refers to the first row of data in the table, *column*=0 refers to the first column of data.

The Point column can not be scrolled horizontally.

trailingZeros=*t* — *t*=1 shows trailing zeros. This affects the general numeric format only.

viewSelection — Scrolls the table contents so that the target cell and selection are maximally in view. The target cell will always be visible. The selection may overflow the visible area.

See also the topLeftCell and selection keywords.

viewSelection was added in Igor Pro 9.00.

width=*w* — Sets column width to *w* points.

You will not always get the exact number of points that you request. This is because a column must have an even number of screen pixels, so that grid lines look good. Igor will modify your requested number of points to meet this requirement.

Flags

/W= *winName* — Modifies the named table window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/Z — No errors generated if the indexed or specified column does not exist in a style macro.

Examples

```
ModifyTable size(myWave)=14       // change font size of myWave column
ModifyTable width(Point)=0                  // hide Point column
ModifyTable style(cmplxWave.imag)=32        // condensed= bit 5 = 2^5 = 32
```

See Also

See **Column Names** on page II-241 and **ModifyTable Elements Command** on page II-263.

ModifyViolinPlot

ModifyViolinPlot [/W=*winName*] [keyword=*value*, keyword=*value*, ...]

The ModifyViolinPlot operation modifies a violin plot trace in the target or named graph.

ModifyViolinPlot was added in Igor Pro 8.00.

For a detailed discussion of violin plots and the parts of a violin plot, see **Violin Plots** on page II-337.

Parameters

ModifyViolinPlot parameters consist of keyword=value pairs. The trace keyword specifies the trace targeted by the subsequent keywords. For example, the command:

```
ModifyViolinPlot trace=trace0, markerSize=5
```

sets the marker size for all datasets of the trace0 trace.

As of Igor Pro 9.00, you can modify a setting for a specific dataset of a specific trace by adding a zero-based dataset index in square brackets after the keyword. For example:

```
ModifyViolinPlot trace=trace0, markerSize[1]=7
```

This sets the marker size for the second dataset (index=1) to 7 leaving the marker size for other datasets unchanged.

General Parameters

trace=*traceName*

Specifies the name of a violin plot trace to be modified. An error results if the named trace is not a violin plot trace. Without the trace keyword, ModifyViolinPlot uses the first trace in the graph, whether it is a violin plot trace or not but see the instance keyword for an exception.

instance=*instanceNum*

The combination of trace and instance works the same as (*traceName#instanceNum*) for a ModifyGraph keyword.

The instance keyword without trace keyword accesses the *instanceNum*'th trace in the graph, just like [*traceNum*] used with a ModifyGraph keyword. See **Trace Names** on page II-282 and **Object Indexing** on page IV-20.

bandwidth[(*ds*)]=*bw*

Sets the bandwidth used in computing the KDE curves. If you include *ds*, the bandwidth is set to *bw* for dataset *ds*. If you omit *ds*, the bandwidth is set for all datasets. *bw* must be greater than or equal to 0. 0 uses the bandwidth estimated by the method set by the bandwidthMethod keyword.

bandwidthMethod=*m*

Sets the method used to estimate the appropriate bandwidth used in computing the KDE curves. The methods are

m=0:	Silverman
m=1:	Scott (default)
m=2:	Bowman and Azzalini

See StatsKDE for details.

You can override the estimated bandwidth using the bandwidth keyword.

boxWidth=*w*

For a non-category X axis, boxWidth sets the width of the space reserved for displaying the KDE curves. If *w* is between zero and one, it is taken to be a fraction of the width of the plot rectangle. If *w* is greater than one, it is in points.

All the KDE curves are normalized such that the maximum density value of the KDEs over all datasets fills the box width so only one of the violin plots will be as wide as *w*. You can control the normalization using the maxDensity keyword.

If the violin plot is displayed using a category X axis, boxWidth is ignored. The space reserved for each dataset is controlled by the category axis and is affected by ModifyGraph barGap and catGap.

A new non-category violin plot has box width set as boxWidth=1/(2**n*) where *n* is the number of datasets (the number of violin plots) on the trace.

closeOutline[=*close*]

If *close* is 1 or omitted, the ends of the KDE curves are connected by a straight line. Closed curves are more visible if side=1 or side=2 in which case the KDE curves are closed by a straight line at the midline between the curves.

curveExtension=*ext*

Sets the range of the KDE curves. The KDE curves are computed for a range of y=minData-*ext* to y=maxData+*ext*, where *ext* is in units of bandwidth. *ext* defaults to 1.

dataMarker=marker

Sets the marker number used to display the raw data. If marker is -1, the marker as set by ModifyGraph marker is used. The default is 8 (hollow circle).

jitter=*j*

In order to separate closely-spaced raw data points, the data points can be displaced horizontally. *j* controls the maximum offset applied to any data point, expressed as a fraction of the box width. The value of *j* may be greater than 1, but in general values less than 1 look better. The default value is 0.5.

kernel=*k*

Sets the kernel function to be used in computing the KDE curves. Supported values are

k=1:	Epanechnikov
k=2:	Bi-weight
k=3:	Tri-weight
k=4:	Triangular
k=5:	Gaussian (default)
k=6:	Rectangular

It is unusual for a violin plot to use anything other than Gaussian, which is the default. See StatsKDE for details.

lineStyle=*style*

Sets the line style used to draw the KDE curves. See **Dashed Lines** on page III-496 for a description of line styles.

A line style of -1 uses the line style set by ModifyGraph lstyle.

style defaults to -1.

lineThickness=*thick*

Sets the thickness of the line used to plot the KDE curves. A thickness less than zero uses the line width set by the ModifyGraph lsize keyword. *thick*=0 hides the KDE curve. The default value is -1.

markerSize=*size*

Sets the size of the marker used to display the raw data. A marker size of zero uses one-half of the marker size as set by ModifyGraph msize. *size* defaults to zero.

markerThick=*t*

Sets the thickness in points of the strokes of markers used to display the raw data.

The markerThick keyword was added in Igor Pro 9.00.

maxDensity=*d*

Sets the normalization for the width of the KDE curves to *d*. All the KDE curves are normalized such that the maximum density value of all KDEs fills the box width. The width of each curve gives an indication of the relative maximum density. maxDensity allows you to set the same normalization for multiple violin plots so that the width for all is relative to the same value.

plotSide=*side*

By default, a violin plot is symmetrical with the KDE curve plotted twice in mirror image. You can control which side is plotted:

side=0:	Both sides in mirror image (default)
side=1:	The left side
side=2:	The right side

Using 1 or 2 allows you to combine two violin plots in an asymmetric plot.

showData[=*sd*]

Shows or hides the raw data points. *sd* is 0 or 1. showData by itself is equivalent to showData=1. The default is 1.

Mean and Median Parameters

You can display markers showing the mean and the median of each dataset using the keywords documented in this section. See **Markers** on page II-291 for a list of available markers. For both mean and median, a marker size of zero uses the marker size set by ModifyGraph msize.

showMean[=*show*]
Shows or hides the marker representing the mean value. *show* is 0 or 1. showMean by itself is equivalent to showMean=1. The default is 0.

meanMarker=*marker*
Sets the marker to use for the mean value. The default for the mean marker is 27 (hollow horizontal diamond with dot).

meanMarkerSize=*size*
Sets the size of the marker used to display the mean value. A value of zero uses the trace marker size as set by ModifyGraph msize, or the normal trace default marker size. The default is zero.

meanMarkerThick=*t*
Sets the thickness in points of the stroke of markers used to display the mean value if showMean is enabled.

The meanMarkerThick keyword was added in Igor Pro 9.00.

showMedian[=*show*]
Shows or hides the marker representing the median value. *show* is 0 or 1. showMedian by itself is equivalent to showMedian=1. The default is 0.

medianMarker=*marker*
Sets the marker to use for the median value. The default for the median marker is 26 (filled horizontal diamond).

medianMarkerSize=*size*
Sets the size of the marker used to display the median value. A value of zero uses the trace marker size as set by ModifyGraph msize, or the normal trace default marker size. The default is zero.

medianMarkerThick=*t*
Sets the thickness in points of the stroke of markers used to display the median value if showMedian is enabled.

The medianMarkerThick keyword was added in Igor Pro 9.00.

Color Parameters

All colors are specified as (*r*,*g*,*b*[,*a*]) **RGBA Values**.

fillColor=(*r*,*g*,*b*[,*a*])
Fills the area between the distribution curves with the specified color. Specifying fillColor=(0,0,0,0) removes the fill. If side=1 or side=2, the fill is between the curve and the mid line.

lineColor=(*r*,*g*,*b*[,*a*])
Sets the color of the KDE curves.

Specify lineColor=(0,0,0,0) to use the trace color set using ModifyGraph rgb as the line color. The default is black.

markerColor=(*r*,*g*,*b*[,*a*])
Sets the color of markers used to display the raw data.

Specify markerColor=(0,0,0,0) to use the trace color set using ModifyGraph rgb as the marker color. This is the default behavior.

markerStrokeColor=(*r*,*g*,*b*[,*a*])
Sets the color of the strokes of markers used to display the raw data.

Specify markerStrokeColor=(0,0,0,0) to use the to use the trace color set using ModifyGraph rgb as the marker color. This is the default behavior.

The markerStrokeColor keyword was added in Igor Pro 9.00.

markerFilled=*f*
If *f* is non-zero, markers are filled with color. Normally, hollow markers have transparent fill. Default fill color is white. Applies only to hollow markers such as marker 8, the hollow circle marker.

The markerFilled keyword was added in Igor Pro 9.00.

markerFillColor=(*r*,*g*,*b*[,*a*])
Sets the fill color of markers used to display the raw data. Applies only to hollow markers such as marker 8, the hollow circle marker.

The markerFillColor keyword was added in Igor Pro 9.00.

meanMarkerColor=(*r*,*g*,*b*[,*a*])

Sets the color of markers used to display the mean value.

Specify meanMarkerColor=(0,0,0,0) to use the trace color set using ModifyGraph rgb as the mean marker color. This is the default behavior.

meanMarkerStrokeColor=(*r*,*g*,*b*[,*a*])

Sets the color of the strokes of markers used to display the mean of the data if showMean is enabled.

Specify meanMarkerStrokeColor=(0,0,0,0) to use the to use the trace color set using ModifyGraph rgb as the marker color. This is the default behavior.

The meanMarkerStrokeColor keyword was added in Igor Pro 9.00.

meanMarkerFilled=*f*

If *f* is non-zero, the mean value marker is filled with color. Normally, hollow markers have transparent fill. Default fill color is white. Applies only to hollow markers such as marker 8, the hollow circle marker.

The meanMarkerFilled keyword was added in Igor Pro 9.00.

meanMarkerFillColor=(*r*,*g*,*b*[,*a*])

Sets the fill color of markers used to display the mean value if showMean is enabled. Applies only to hollow markers such as marker 8, the hollow circle marker.

The meanMarkerFillColor keyword was added in Igor Pro 9.00.

medianMarkerColor=(*r*,*g*,*b*[,*a*])

Sets the color of markers used to display the median value.

Specify medianMarkerColor=(0,0,0,0) to use the trace color set using ModifyGraph rgb as the median marker color. This is the default behavior.

medianMarkerStrokeColor=(*r*,*g*,*b*[,*a*])

Sets the color of the strokes of markers used to display the mean of the data if showMedian is enabled.

Specify medianMarkerStrokeColor=(0,0,0,0) to use the to use the trace color set using ModifyGraph rgb as the marker color. This is the default behavior.

The medianMarkerStrokeColor keyword was added in Igor Pro 9.00.

medianMarkerFilled=*f*

If *f* is non-zero, the median value marker is filled with color. Normally, hollow markers have transparent fill. Default fill color is white. Applies only to hollow markers such as marker 8, the hollow circle marker.

The medianMarkerFilled keyword was added in Igor Pro 9.00.

medianMarkerFillColor=(*r*,*g*,*b*[,*a*])

Sets the fill color of markers used to display the median value if showMedian is enabled. Applies only to hollow markers such as marker 8, the hollow circle marker.

The medianMarkerFillColor keyword was added in Igor Pro 9.00.

Violin Plot Per-Data-Point Marker Settings

You can override the basic settings for data point marker color, marker style and marker size using marker settings waves containing per-data-point settings.

You can apply per-data-point marker settings to an entire trace (to all datasets comprising a trace) or to a specific dataset of a trace. For example:

```
Function DemoViolinPlotPerPointMarkerSettings()
   Make/O violin0={1,2,3,4,5}, violin1={2,3,4,5,6}, violin2={3,4,5,6,7}
   String title = "Violin Plot Per Point Marker Settings"
   Display/W=(557,99,948,310)/N=ViolinPlotPerPointPlot as title

   // Create a trace named trace0 with three datasets: violin0, violin1, violin2
   AppendViolinPlot/TN=trace0 violin0,violin1,violin2
```

```
    // Set the marker and marker size for all datasets of trace trace0
    ModifyViolinPlot trace=trace0, dataMarker=18      // 18=Diamond
    ModifyViolinPlot trace=trace0, markerSize=5

    // Set the per-data-point marker for all datasets of trace trace0
    Make/O violinMarkers = {15,16,17,18,19}
    ModifyViolinPlot trace=trace0, dataMarkerWave=violinMarkers

    // Set the per-data-point marker for dataset violin1 only
    Make/O violinMarkersForViolin1 = {32,33,34,35,36}
    ModifyViolinPlot trace=trace0, dataMarkerWave[1]=violinMarkersForViolin1
End
```

Usually a marker settings wave will have the same number of rows as there are data points in a given dataset, but that is not required. If there are fewer rows in the settings wave than in the dataset, the extra data points retain their basic settings. If there are more rows in the settings wave than in the dataset, the extra settings wave points are not used.

See **Making Each Data Point Look Different** on page II-343 for more information and examples.

dataColorWave [=*colorWave*]

Sets *colorWave* to override data point marker color on a point-by-point basis. The wave must be a 3 or 4 column wave containing red, green, blue and optionally alpha values.

If you omit "=*colorWave*", any previous marker color wave setting is cleared.

The dataColorWave keyword was added in Igor Pro 9.00.

dataMarkerWave [=*markerWave*]

Sets *markerWave* to override data point markers on a point-by-point basis. The values in *markerWave* are standard graph marker numbers. See **Markers** on page II-291 for a table of the markers and the associated marker numbers. The marker numbers are clipped to a valid range.

If you omit "=*markerWave*", any previous marker wave setting is cleared.

The dataMarkerWave keyword was added in Igor Pro 9.00.

dataSizeWave [=*markerSizeWave*]

Sets *markerSizeWave* to override data point marker size on a point-by-point basis. The values in *markerSizeWave* are clipped to the range [0,200].

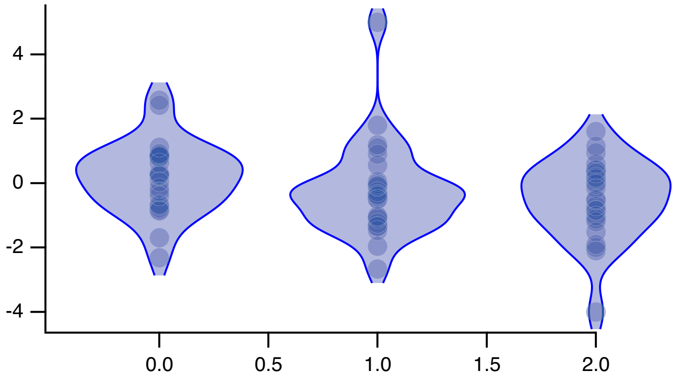
If you omit "=*markerSizeWave*", any previous marker size wave setting is cleared.

The dataSizeWave keyword was added in Igor Pro 9.00.

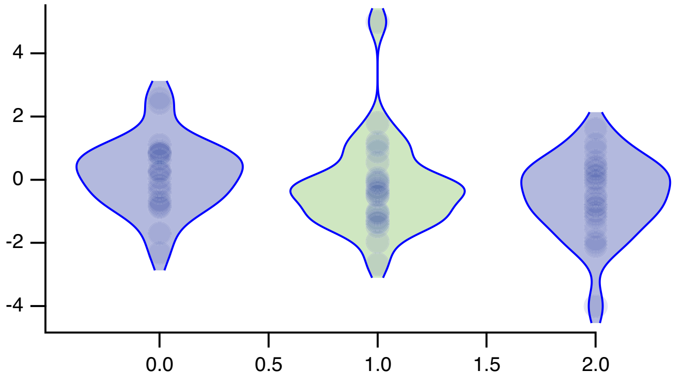
Examples

A violin plot with transparent marker color used to give a visual indication of the density of data points.

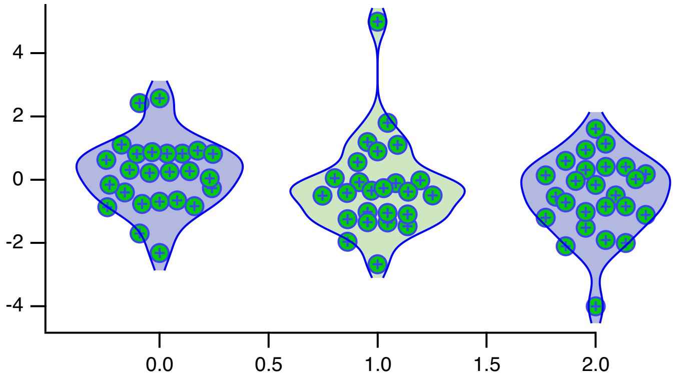
```
// Create a violin plot with transparent marker color used to give a visual
// indication of the density of data points
Make/O/N=(25,3) multicol                    // A three-column wave with 25 rows
SetRandomSeed(.4)                           // For reproducible "randomness"
multicol = gnoise(1)                        // Three normally-distributed datasets
multicol[20][1] = 5                         // A "far" outlier
multicol[13][2] = -4                        // An outlier
Display; AppendViolinPlot multicol
ModifyViolinPlot lineColor=(0,0,65535)
ModifyViolinPlot fillColor=(0,0,65535,19661)
ModifyViolinPlot jitter=0
ModifyViolinPlot showData=1
ModifyViolinPlot dataMarker=19
ModifyViolinPlot markerSize=6
ModifyViolinPlot markerColor=(0,0,65535,6554)
```



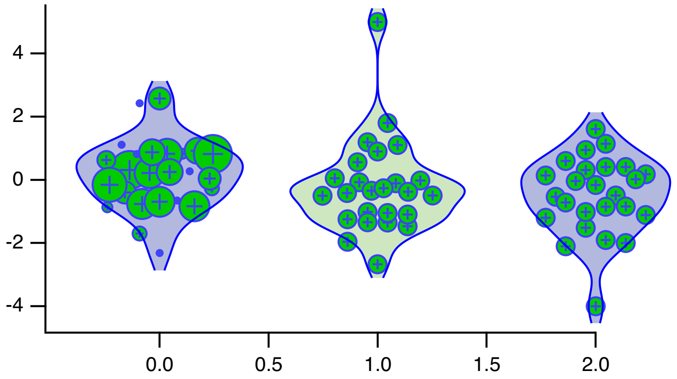
```
ModifyViolinPlot fillColor[1]=(0,65535,0,19661)    // Second dataset transparent green
```



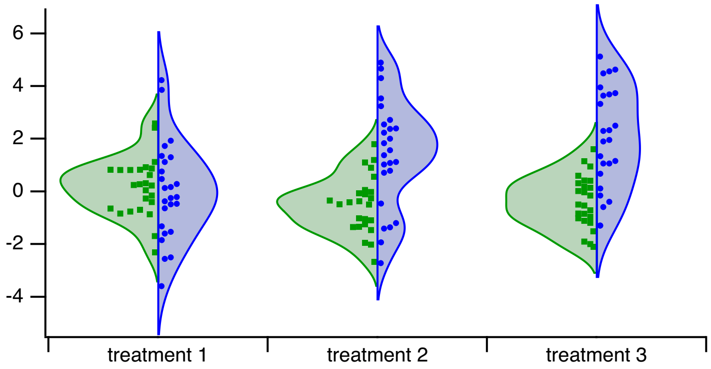
```
// Add jitter, choose the circle-with-plus marker
ModifyViolinPlot trace=multicol,Jitter=0.7
ModifyViolinPlot trace=multicol,DataMarker=42,MarkerSize=5
// Set marker color to blue with a one-point outline
ModifyViolinPlot trace=multicol,MarkerColor=(16385,16388,65535)
ModifyViolinPlot trace=multicol,MarkerThick=1
// Set the marker to filled with dark green fill color
ModifyViolinPlot trace=multicol,MarkerFilled=1,MarkerFillColor=(3,52428,1)
```



```
// Set a marker size wave for dataset 0 (the first dataset)
Make/N=25/O sizeWave = enoise(5)+7
ModifyViolinPlot DataSizeWave[0] = sizeWave
```



```
// Create a violin plot with asymmetric curves plotted on a category axis
Make/O/N=(25,3) ds1, ds2
SetRandomSeed(.4)                                    // For reproducible "randomness"
ds1 = gnoise(1)
ds2 = gnoise(2)+q

// We need a text wave to make a category plot
Make/N=3/T/O labels={"treatment 1", "treatment 2", "treatment 3"}
Display
AppendViolinPlot ds1 vs labels
AppendViolinPlot ds2 vs labels

// Keep plots together in a single category space
ModifyGraph toMode(ds1)=-1

// Display ds1 on the left, ds2 on the right
ModifyViolinPlot trace=ds1,plotSide=1
ModifyViolinPlot trace=ds2,plotSide=2

// Extend the KDE curves
ModifyViolinPlot trace=ds1,CurveExtension=2
ModifyViolinPlot trace=ds2,CurveExtension=2

// Close the curves with line at the midline
ModifyViolinPlot trace=ds1,closeOutline=1
ModifyViolinPlot trace=ds2,closeOutline=1

// Use the same normalization for both curves
ModifyViolinPlot trace=ds1,maxDensity=0.36
ModifyViolinPlot trace=ds2,maxDensity=0.36

// Apply jitter to the data points
ModifyViolinPlot trace=ds1,jitter=0.5
ModifyViolinPlot trace=ds2,jitter=0.5

// Set markers and colors
ModifyViolinPlot trace=ds1,showData=1,dataMarker=16,markerColor=(2,39321,1),lineColor=(2,39321,1)
ModifyViolinPlot trace=ds2,showData=1,dataMarker=19,markerColor=(0,0,65535),lineColor=(0,0,65535)
ModifyViolinPlot trace=ds1,fillColor=(2,39321,1,19661)
ModifyViolinPlot trace=ds2,fillColor=(0,0,65535,19661)
```



See Also

AppendViolinPlot, **AddWavesToViolinPlot**, **ModifyGraph (traces)**

Violin Plots on page II-337

ModifyWaterfall

ModifyWaterfall [**/W=*winName***] ***keyword* = *value*** [**,** ***keyword* = *value*** …]

The ModifyWaterfall operation modifies the properties of the waterfall plot in the top or named graph.

Parameters

keyword is one of the following:

angle= *a* — Angle in degrees from horizontal of the angled Y axis (*a* =10 to 90).

axlen= *len* — Relative length of angled Y axis. *len* is a fraction between 0.1 and 0.9.

hidden= *h* — Controls the hidden line algorithm.

- *h*=0: Turns hidden lines off.
- *h*=1: Uses painter’s algorithm.
- *h*=2: True hidden.
- *h*=3: Hides lines with bottom removed.
- *h*=4: Hides lines using a different color for the bottom. When specified, the top color is the normal color for lines and the bottom color is set using `ModifyGraph negRGB=(r,g,b[,a])`.

Hidden lines are active only when the mode is lines between points.

Flags

/W= *winName* — Modifies waterfall plot in the named graph window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

Painter’s algorithm draws the traces from back to front and erases hidden lines while modes 2, 3 and 4 detect which line segments are hidden and suppresses the drawing of these segments.

See Also

Waterfall Plots on page II-326.

The **NewWaterfall** and **ModifyGraph** operations.

ModuleName

`#pragma ModuleName = `*`modName`*

The ModuleName pragma assigns a name, which must be unique among all procedure files, to a procedure file so that you can use static functions and Proc Pictures in a global context, such as in the action procedure of a control or on the Command Line.

Using the ModuleName pragma involves at least two steps. First, within the procedure file assign it a name using `#pragma ModuleName=`*`modName`*, and then access objects in the named file by preceding the object name with the name of the module and the # character, such as or example: `ModName#StatFuncName()`.

See Also

The **Regular Modules** on page IV-236, **Static**, **Picture**, and #**pragma**.

MoveDataFolder

`MoveDataFolder [ /O=`*`options`*` /Z ] `*`sourceDataFolderSpec`*`, `*`destDataFolderPath`*

The MoveDataFolder operation removes the source data folder (and everything it contains) and places it at the specified location with the original name.

Parameters

sourceDataFolderSpec can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

destDataFolderPath can be a partial path (relative to the current data folder) or an absolute path (starting from root).

Flags

/O=*options*	Overwrites the destination data folder if it already exists. The /O flag was added in Igor Pro 8.00. *options*=1: Completely overwrites the destination data folder. If the destination data folder exists, MoveDataFolder first deletes it if possible. If it can not be deleted, for example because it contains a wave that is in use, MoveDataFolder generates an error. If the deletion succeeds, MoveDataFolder then copies the source to the destination. Then the source data folder is deleted. *options*=2: Merges the source data folder into the destination data folder. If an item in the source data folder exists in the destination data folder, MoveDataFolder overwrites it. Otherwise it copies it. Items in the destination data folder that do not exist in the source data folder remain in the destination data folder. Then the source data folder is deleted. *options*=3: Merges the source data folder into the destination data folder and then deletes items that are not in the source data folder if possible. If an item in the source data folder does not exist in the destination data folder, MoveDataFolder attempts to delete it from the destination data folder. If it can not be deleted because it is in use, no error is generated. Then the source data folder is deleted.
/Z	Errors are not fatal - if an error occurs, procedure execution continues. You can check the V_flag output variable to see if an error occurred.

Details

MoveDataFolder generates an error if a data folder of the same name already exists at the destination unless the /O flag is used. When the /O flag is non-zero and the destination data folder already exists, MoveDataFolder is equivalent to DuplicateDataFolder followed by a KillDataFolder on the source.

Output Variables

MoveDataFolder sets the following output variable:

`V_flag`	0 if the operation succeeded or a non-zero error code. The V_flag output variable was added in Igor Pro 8.00.

Examples

```
MoveDataFolder root:DF0, root:archive      // Move DF0 into archive
```

See Also

See the **DuplicateDataFolder** operation. Chapter II-8, **Data Folders**.

MoveFile

`MoveFile` `[`***`flags`***`][`***`srcFileStr`***`] [`**`as`** ***`destFileOrFolderStr`***`]`

The MoveFile operation moves or renames a file on disk. A file is renamed by “moving” it to the same folder it is already in using a different name.

Parameters

srcFileStr can be a full path to the file to be moved or renamed (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*.

If Igor can not determine the location of the file from *srcFileStr* and *pathName*, it displays an Open File dialog allowing you to specify the source file.

destFileOrFolderStr is interpreted as the name of (or path to) an existing folder when /D is specified, otherwise it is interpreted as the name of (or path to) a possibly existing file.

If *destFileOrFolderStr* is a partial path, it is relative to the folder associated with *pathName*.

If /D is specified, the source file is moved inside the folder using the source file’s name.

If Igor can not determine the location of the destination file from *pathName*, *srcFileStr*, and *destFileOrFolderStr*, it displays a Save File dialog allowing you to specify the destination file (and folder).

If you use a full or partial path for either *srcFileStr* or *destFileOrFolderStr*, see **Path Separators** on page III-451 for details on forming the path.

Folder paths should not end with single Path Separators. See the **Details** section for **MoveFolder**.

Flags

/D	Interprets *destFileOrFolderStr* as the name of (or path to) an existing folder (or directory). Without /D, *destFileOrFolderStr* is the name of (or path to) a file. If *destFileOrFolderStr* is not a full path to a folder, it is relative to the folder associated with *pathName*.
/I [=*i*]	Specifies the level of interactivity with the user. /I=0: Interactive only if *srcFileStr* or *destFileOrFolderStr* is not specified or if the source file is missing. (Same as if /I was not specified.) /I=1: Interactive even if *srcFileStr* is specified and the source file exists. /I=2: Interactive even if *destFileOrFolderStr* is specified. /I=3: Interactive even if *srcFileStr* is specified and the source file exists. Same as /I only.
/M=*messageStr*	Specifies the prompt message in the Open File dialog. If /S is not specified, then *messageStr* will be used for both Open File and for Save File dialogs.
/O	Overwrite existing destination file, if any. Without /O, the user is asked if replacing the existing file is to be allowed.
/P=*pathName*	Specifies the folder to look in for the source file, and the folder into which the file is copied. *pathName* is the name of an existing symbolic path. Using /P means that both *srcFileStr* and *destFileOrFolderStr* must be either simple file or folder names, or paths relative to the folder specified by *pathName*.
/S=*saveMessageStr*	Specifies the prompt message in the Save File dialog.

/Z[=z] Prevents procedure execution from aborting if it attempts to move a file that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

/Z=0:	Same as no /Z.
/Z=1:	Moves a file only if it exists. /Z alone is equivalent to /Z=1.
/Z=2:	Moves a file if it exists or displays a dialog if it does not exist.

Variables

The MoveFile operation returns information in the following variables:

`V_flag`	Set to zero if the file was moved, to -1 if the user cancelled either the Open File or Save File dialogs, and to some nonzero value if an error occurred, such as the specified file does not exist.
`S_fileName`	Stores the full path to where the file was moved from. If an error occurred or if the user cancelled, it is set to an empty string.
`S_path`	Stores the full path where the file was moved to. If an error occurred or if the user cancelled, it is set to an empty string.

Examples

Rename a file, using full paths:

```
MoveFile "HD:folder:aFile.txt" as "HD:folder:bFile.txt"
```

Rename a file, using a symbolic path:

```
MoveFile/P=myPath "aFile.txt" as "bFile.txt"
```

Move a file into a subfolder (the subfolder must exist):

```
MoveFile/D "Macintosh HD:folder:aFile.txt" as ":subfolder"
```

Move a file into an unrelated folder (the subfolder must exist):

```
MoveFile/D "Macintosh HD:folder:afile.txt" as "Server:archive"
```

Move a file from one folder to another and rename it:

```
MoveFile "Macintosh HD:folder:afile.txt" as "Server:archive:destFile.txt"
```

Move user-selected file into a particular folder:

```
MoveFile/D as "C:My Data:Selected Files Folder"
```

Move user-selected file in any folder as `bFile.txt` in same folder:

```
MoveFile as "bFile.txt"
```

Move user-selected file in any folder as `bFile.txt` in any folder:

```
MoveFile/I=2 as "bFile.txt"
```

See Also

The **Open**, **MoveFolder**, **CopyFolder**, **NewPath**, and **CreateAliasShortcut** operations. The **IndexedFile** function. **Symbolic Paths** on page II-22.

MoveFolder

`MoveFolder [`*`flags`*`][`*`srcFolderStr`*`] [as `*`destFolderStr`*`]`

The MoveFolder operation moves or renames a folder on disk. A folder is renamed by "moving" it into the same folder it is already in, but with a different name.

Warning: *The `MoveFolder` command can destroy data* by overwriting another folder and its contents!

If you overwrite an existing folder on disk, MoveFolder will do so only if permission is granted by the user. The default behavior is to display a dialog asking for permission. The user can alter this behavior via the Miscellaneous Settings dialog's Misc category.

If permission is denied, the folder will not be moved and V_Flag will return 1088 (Command is disabled) or 1275 (You denied permission to overwrite a folder). Command execution will cease unless the /Z flag is specified.

Parameters

srcFolderStr can be a full path to the folder to be moved or renamed (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a folder within the folder associated with *pathName*.

If the location of the source folder cannot be determined from *srcFolderStr* and *pathName*, it displays a Select Folder dialog allowing you to specify the source.

If /P=*pathName* is given, but *srcFolderStr* is not, then the folder associated with *pathName* is moved or renamed.

destFolderStr specifies the final location of the folder or, if /D is used, the parent of the final location of the folder.

destFolderStr can be a full path to the output (destination) folder (in which case /P is not needed), or a partial path relative to the folder associated with *pathName*.

If the location of the destination folder cannot be determined from *destFolderStr* and *pathName*, it displays a Save Folder dialog allowing you to specify the destination.

If you use a full or partial path for either file, see **Path Separators** on page III-451 for details on forming the path.

Flags

/D
Interprets *destFolderStr* as the name of (or path to) an existing folder (or "directory") to move the source folder into. Without /D, it interprets *destFolderStr* as the name of (or path to) the moved folder.

If *destFolderStr* is not a full path to a folder, it is relative to the source folder.

/I [=*i*]
Specifies the level of interactivity with the user.

/I=0: Interactive only if *srcFolderStr* or *destFolderStr* is not specified or if the source folder is missing. (Same as if /I was not specified.)

/I=1: Interactive even if *srcFolderStr* is specified and the source folder exists.

/I=2: Interactive even if *destFolderStr* is specified.

/I=3: Interactive even if *srcFolderStr* is specified and the source folder exists. Same as /I only.

/M=*messageStr*
Specifies the prompt message in the Open File dialog. If /S is not used, then *messageStr* will be used for both Open File and for Save File dialogs.

/O
Overwrite existing destination folder, if any. This deletes the existing destination folder. When /O is specified, the source folder can't be moved into an existing folder without specifying the name of the moved folder in *destFolderStr*.

/P=*pathName*
Specifies the folder for relative paths in *srcFolderStr* and *destFolderStr*. *pathName* is the name of an existing symbolic path.

If *srcFolderStr* is omitted, the folder associated with *pathName* is moved. If *destFolderStr* is omitted, the source folder is moved into the folder associated with *pathName*.

Using /P means that *srcFolderStr* (if specified) and *destFolderStr* must be either simple folder names or paths relative to the folder specified by *pathName*.

/S=*saveMessageStr*	Specifies the prompt message in the Save File dialog.
/Z[=z]	Prevents procedure execution from aborting if it attempts to move a folder that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort. /Z=0: Same as no /Z. /Z=1: Moves a folder only if it exists. /Z alone is equivalent to /Z=1. /Z=2: Moves a folder if it exists or displays a dialog if it does not exist.

Variables

The MoveFolder operation returns information in the following variables:

`V_flag`	Set to zero if the file was moved, to -1 if the user cancelled either the Open File or Save File dialogs, and to some nonzero value if an error occurred, such as the specified file does not exist.
`S_fileName`	Stores the full path to the folder that was moved, with a trailing colon. If an error occurred or if the user cancelled, it is set to an empty string.
`S_path`	Stores the full path of the moved folder, with a trailing colon. If an error occurred or if the user cancelled, it is set to an empty string.

Details

You can use only /P=*pathName* (omitting *srcFolderStr*) to specify the source folder to be moved.

A folder path should not end with single Path Separators. For example:

```
MoveFolder "Macintosh HD:folder" as "Macintosh HD:Renamed Folder:"
MoveFolder "Macintosh HD:folder:" as "Macintosh HD:Renamed Folder"
MoveFolder "Macintosh HD:folder:" as "Macintosh HD:Renamed Folder:"
```

will do weird, unexpected things (and probably damaging things when /O is also used). Instead, use:

```
MoveFolder "Macintosh HD:folder" as "Macintosh HD:Renamed Folder"
```

Beware of PathInfo and other command which return paths with an ending path separator. (They can be removed with the **RemoveEnding** function.)

A folder may not be moved into one of its own subfolders.

Conversely, the command:

```
MoveFolder/O/P=myPath "afolder"
```

which attempts to overwrite the folder associated with myPath with a folder that is inside it (namely “afolder”) is not allowed. Instead, use:

```
MoveFolder/O/P=myPath "::afolder"
```

On Windows, renaming or moving a folder never updates the value of any Igor Symbolic Paths that point to a moved folder:

```
// Create a folder
NewPath/O/C myPath "C:\\My Data\\My Work"

// Move the folder
MoveFolder/P=myPath as "C:\\My Data\\Moved"

// Display the path's value
PathInfo myPath               // (or use the Path Status dialog)
Print S_Path
• C:My Data:My Work
```

You can use PathInfo to determine if a folder referred to by an Igor symbolic path exists and where it is on the disk. Use NewPath/O to reset the path’s value.

On the Macintosh, however, renaming or moving a folder on the same volume does alter the value of symbolic path. This is because MoveFolder uses a Mac OS alias to keep track of the folder. A folder renamed or moved on the same volume retains the original “volume refnum” and “directory ID” stored in the alias mechanism, so that the alias (and hence Igor’s symbolic path) remains pointing to the moved folder. After moving the folder, using the unchanged volume refnum and directory ID (in PathInfo or when you use /P=pathName) returns the updated path.

Moving the folder to a different volume actually creates a new folder with new volume refnum and directory IDs, and symbolic paths pointing to or into the moved folder aren't updated. They will be pointing at a deleted folder (they're probably invalid).

Examples

Rename a folder ("move" it to the same folder):

```
MoveFolder "Macintosh HD:folder" as "Macintosh HD:Renamed Folder"
```

Rename a folder referred to by only a path:

```
NewPath/O myPath "Macintosh HD:folder"
MoveFolder/P=myPath as "::Renamed Folder"
```

Move a folder from one volume to another. This moves "Macintosh HD:My Folder" inside "Server:My Folder" if "Server:My Folder" already exists:

```
MoveFolder "Macintosh HD:My Folder" as "Server:My Folder"
```

Move a folder from one volume to another. This overwrites "Server:My Folder" (if it existed) with the moved "Macintosh HD:My Folder":

```
MoveFolder/O "Macintosh HD:My Folder" as "Server:My Folder"
```

Move user-selected folder in any folder as "Renamed Folder" into a user-selected folder (possibly the same one):

```
MoveFolder as "Renamed Folder"
```

Move user-selected file in any folder as "Moved Folder" in any folder:

```
MoveFolder/I=3 as "Moved Folder"
```

See Also

MoveFile, **CopyFolder**, **DeleteFolder**, **IndexedDir**, **PathInfo**, and **RemoveEnding**. **Symbolic Paths** on page II-22.

MoveString

MoveString *sourceString*, *destDataFolderPath* [***newname***]

The MoveString operation removes the source string variable and places it in the specified location optionally with a new name.

Parameters

sourceString can be just the name of a string variable in the current data folder, a partial path (relative to the current data folder) and variable name or an absolute path (starting from root) and variable name.

destDataFolderPath can be a partial path (relative to the current data folder) or an absolute path (starting from root).

Details

An error is issued if a variable or wave of the same name already exists at the destination.

Examples

```
MoveString :foo:s1,:bar:      // Move string s1 into data folder bar
MoveString :foo:s1,:bar:ss1   // Move string s1 into bar with new name ss1
```

See Also

The **MoveVariable**, **MoveWave**, and **Rename** operations; andChapter II-8, **Data Folders**.

MoveSubwindow

MoveSubwindow [**/W=*winName***] ***key* = (*values*)**[**, *key* = (*values*)**]…

The MoveSubwindow operation moves the active or named subwindow to a new location within the host window. This command is primarily for use by recreation macros; users should use layout mode for repositioning subwindows.

Parameters

fguide=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the frame guide name(s) to which the outer frame of the subwindow is attached inside the host window.

The frame guides are identified by the standard names or user-defined names as defined by the host. Use * to specify a default guide name.

When the host is a graph, additional standard guides are available for the outer graph rectangle and the inner plot rectangle (where traces are plotted).

See **Details** for standard guide names.

fnum=(*left*, *top*, *right*, *bottom*)

Specifies the new location of the subwindow. The location coordinates of the subwindow sides can have one of two possible meanings:

When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.

When any value is greater than 1, coordinates are taken to be fixed locations measured in points, or **Control Panel Units** for control panel hosts, relative to the top left corner of the host frame.

pguide=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the guide name(s) to which the plot rectangle of the graph subwindow is attached inside the host window.

Guides are identified by the standard names or user-defined names as defined by the host. Use * to specify a default guide name.

See **Details** for standard guide names.

Flags

/W= *winName* Moves the subwindow in the named window or subwindow. When omitted, action will affect the active subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

When moving an exterior subwindow, only the fnum keyword may be used. The values are the same as the **NewPanel** /W flag for exterior subwindows.

The names for the built-in guides are as defined in the following table:

	Left	**Right**	**Top**	**Bottom**
Subwindow Frame	FL	FR	FT	FB
Outer Graph Rectangle	GL	GR	GT	GB
Inner Plot Rectangle	PL	PR	PT	PB

The frame guides apply to all window and subwindow types. The graph rectangle and plot rectangle guide types apply only to graph windows and subwindows.

See Also

The **MoveWindow** operation. Chapter III-4, **Embedding and Subwindows** for further details and discussion.

MoveVariable

MoveVariable *sourceVar*, *destDataFolderPath* [***newname***]

The MoveVariable operation removes the source numeric variable and places it in the specified location optionally with a new name.

Parameters

sourceVar can be just the name of a numeric variable in the current data folder, a partial path (relative to the current data folder) and variable name or an absolute path (starting from root) and variable name.

destDataFolderPath can be a partial path (relative to the current data folder) or an absolute path (starting from root).

Details

An error is issued if a variable or wave of the same name already exists at the destination.

Examples

```
MoveVariable :foo:v1,:bar:          // Move v1 into data folder bar
MoveVariable :foo:v1,:bar:vv1       // Move v1 into bar with new name vv1
```

See Also

The **MoveString**, **MoveWave**, and **Rename** operations; and Chapter II-8, **Data Folders**.

MoveWave

MoveWave *sourceWave*,[***destDataFolderPath*:**] [***newName***]

The MoveWave operation removes the source wave and places it in the specified location optionally with a new name.

If you want to rename a wave without moving it, use **Rename** instead.

Parameters

sourceWave can be just the name of a wave in the current data folder, a partial path (relative to the current data folder) and wave name, an absolute path (starting from root) and wave name, or a wave reference variable in a user-defined function.

destDataFolderPath can be a partial path (relative to the current data folder), an absolute path (starting from root), or a data folder reference (DFREF) in a user-defined function. If the destination is a null DFREF, the wave is moved to the current data folder.

Details

An error is issued if a variable or wave of the same name already exists at the destination.

MoveWave Destination

Depending on the syntax you use, MoveWave may move sourceWave to another data folder, rename sourceWave, or both. To explain this, we show examples below which call this setup function:

```
Function Setup()
    SetDataFolder root:
    KillDataFolder root:            // clear out any previous stuff
    NewDataFolder/O root:DF0
    Make/O root:wave0
End

The Setup function gives you a data hierarchy like this:
root (<- current data folder)
    wave0
    DF0

// 1. Simple dest name: Renames wave0 as wave1
Function Demo1()
    Setup()
    Wave w = root:wave0
    MoveWave w, wave1               // Use Rename instead
End

// 2. Dest path with trailing colon: Moves wave0 without renaming
Function Demo2()
    Setup()
    Wave w = root:wave0
    MoveWave w, root:DF0:
End

// 3. Dest path with trailing colon and name: Moves wave0 and renames as wave1
Function Demo3()
    Setup()
    Wave w = root:wave0
    MoveWave w, root:DF0:wave1
End

// 4. DFREF dest without trailing colon: Moves wave0 without renaming
Function Demo4()
    Setup()
    Wave w = root:wave0
    DFREF dfr = root:DF0
    MoveWave w, dfr
End

// 5. DFREF dest with trailing colon: Generates error
Function Demo5()
    Setup()
    Wave w = root:wave0
    DFREF dfr = root:DF0
    MoveWave w, dfr:// Error - trailing colon not allowed
End

// 6. Dest path with trailing colon and name: Moves wave0 and renames as wave1
Function Demo6()
    Setup()
    Wave w = root:wave0
    DFREF dfr = root:DF0
    MoveWave w, dfr:wave1
End

// 7. Null DFREF as destination; moves to current data folder
Function Demo7()
    Setup()
    Wave w = root:wave0
    SetDataFolder root:DF0           // make DF0 the current data folder
    DFREF noDF = $"Doesnotexist"
    MoveWave w, noDF                 // moves to current DF
end

// 8. Use MoveWave to make a wave free (move it to no data folder)
Function Demo8()
    Setup()
    Wave w = root:wave0
    // Doesn't do it:
    //DFREF noDF = $"Doesnotexist"
    //MoveWave w, noDF               // moves to current DF
    DFREF freedf = NewFreeDataFolder()
    MoveWave w, freedf
    KillDataFolder freedf
    // as long as wave reference w remains,
    // the wave will continue to exist, and is a free wave
end
```

See Also

The **MoveString**, **MoveVariable**, and **Rename** operations; and Chapter II-8, **Data Folders**.

MoveWindow

`MoveWindow [`***`flags`***`]` ***`left, top, right, bottom`***

The MoveWindow operation moves the target or specified window to the given coordinates.

Flags

/C	Moves Command window instead of the target window.
/F	*Windows*: Moves the Igor Pro application “frame” and the frame is then adjusted so that no part is offscreen. *Macintosh*: Moves nothing.

/I	Coordinates are in inches.
/M	Coordinates are in centimeters.
/P=*procedureTitleAsName*	Moves the specified procedure window instead of the target window.
/W=*winName*	Moves the named window.

Details

Note that neither *winName* nor *procedureTitleAsName* is a string but is the actual window name or procedure window title. If the procedure window's title (procedure windows don't have names) has a space in it, use $ and quotes:

```
MoveWindow/P=$"Log Histogram" 0,0,600,400
```

If /W, /F, /C, and /P are omitted, MoveWindow moves the target window.

The coordinates are in points if neither /I nor /M is used.

In Igor Pro 7.00 or later, to move the window without changing its size, pass -1 for both *right* and *bottom*.

You can use the MoveWindow operation to minimize, restore, or maximize a window by specifying 0, 1, or 2 for all of the coordinates, respectively, as follows:

```
MoveWindow 0, 0, 0, 0      // Minimize target window.
MoveWindow 1, 1, 1, 1      // Restore target window.
MoveWindow 2, 2, 2, 2      // Maximize target window.
```

On Macintosh, "maximize" means to move and resize the window so that it fills the screen. "Minimize"means to minimize to the dock.

If the window size has been constrained by `SetWindow sizeLimit`, those limits are silently applied to the size set by MoveWindow.

See Also

The **MoveSubwindow** and **DoWindow** operations.

MPFXEMGPeak

`MPFXEMGPeak(cw, yw, xw)`

The MPFXEMGPeak function implements a single exponentially modified Gaussian peak with no Y offset in the format of an all-at-once fitting function. The exponentially modified Gaussian peak shape is a convolution of an exponential and a Gaussian peak.

Parameters

cw Coefficient wave, which must be a double-precision wave.

The Gaussian peak shape is defined by the coefficients as follows:

cw[0]: Peak location. This is actually the location of the underlying Gaussian peak. There is no analytic expression for the actual peak location.

cw[1]: Standard deviation of the Gaussian portion.

cw[2]: Amplitude-related parameter.

cw[3]: Decay constant of the exponential portion.

cw must be a double precision wave.

yw Y wave into which values are stored.

yw may be either double precision or single precision.

xw X wave containing the X values at which the peak function is to be evaluated.

xw may be either double precision or single precision.

Details

There is no analytic expression for peak parameters of common interest like the amplitude, location and width. The Multpeak Fitting package uses numerical techniques to get approximations.

This function is primarily intended to support the Multipeak Fitting package.

To use MPFXEMGPeak as a fitting function, wrap it in an all-at-once user-defined fitting function:

```
Function FitVoigtPeak(Wave cw, Wave yw, Wave xw) : FitFunc
   Variable dummy = MPFXEMGPeak(cw, yw, xw)
End
```

The assignment to "dummy" is required because you must explicitly do something with the return value of a built-in function.

The implementation of this function involves the **erfcx** function.

If the waves do not satisfy the number type requirements, the function returns NaN. A successful invocation returns zero.

See Also

All-At-Once Fitting Functions on page III-256

MPFXExpConvExpPeak

`MPFXExpConvExpPeak(cw, yw, xw)`

The MPFXExpConvExpPeak function implements a single peak with no Y offset in the format of an all-at-once fitting function. The peak shape is that of an exponential convolved with another exponential.

MPFXExpConvExpPeak is similar to the MPFXEMGPeak, but it has a sharp onset. It fills the wave yw with peak values as if a simple wave assignment was executed.

Parameters

cw Coefficient wave. The Gaussian peak shape is defined by the coefficients as follows:

cw[0]:	Peak location.
cw[1]:	Peak height.
cw[2]:	Inverse of the decay constant of one exponential.
cw[3]:	Inverse of the decay constant of the other exponential.

cw must be a double precision wave.

yw Y wave into which values are stored.

yw may be either double precision or single precision.

xw X wave containing the X values at which the peak function is to be evaluated.

xw may be either double precision or single precision.

Details

The following equations and discussion use these definitions:

```
c0 = cw[0], c1 = cw[1], c2 = cw[2], c3 = cw[3]
```

The peak location given by c0 is not the actual peak location; it is simply a parameter that offsets the peak in the X direction. The actual location is given by

$$loc = \frac{ln(\frac{c2}{c3})}{c2 - c3} + c0$$

The actual peak height is given by

$$h = \frac{c1 \cdot c2 \left\{ \left(\frac{c2}{c3}\right)^{-\frac{c2}{c2-c3}} - \left(\frac{c2}{c3}\right)^{-\frac{c3}{c2-c3}} \right\}}{c3 - c2}$$

The peak area is given by c1/c3.

We are not aware of an analytic expression for the full width at half maximum (FWHM).

This function is primarily intended to support the Multi-peak Fitting package.

To use MPFXExpConvExpPeak as a fitting function, wrap it in an all-at-once user-defined fitting function:

```
Function FitVoigtPeak(Wave cw, Wave yw, Wave xw) : FitFunc
   Variable dummy = MPFXExpConvExpPeak(cw, yw, xw)
End
```

The assignment to "dummy" is required because you must explicitly do something with the return value of a built-in function.

If the waves do not satisfy the number type requirements, the function returns NaN. A successful invocation returns zero.

See Also

All-At-Once Fitting Functions on page III-256

MPFXGaussPeak

`MPFXGaussPeak(cw, yw, xw)`

The MPFXGaussPeak function implements a single Gaussian peak with no Y offset in the format of an all-at-once fitting function. It fills the wave *yw* with values defined by a Gaussian peak as if this wave assignment statement was executed:

```
yw = cw[2] * exp( -((xw - cw[0])/cw[1])^2 )
```

Parameters

cw Coefficient wave. The Gaussian peak shape is defined by the coefficients as follows:

cw[0]:	Peak location.
cw[1]:	Peak width: sqrt(2) * (standard deviation).
cw[2]:	Amplitude.

cw must be a double precision wave.

yw Y wave into which values are stored.

yw may be either double precision or single precision.

xw X wave containing the X values at which the peak function is to be evaluated.

xw may be either double precision or single precision.

Details

This function is primarily intended to support the Multipeak Fitting package. To use MPFXGaussPeak as a fitting function, wrap it in an all-at-once user-defined fitting function:

```
Function FitGaussPeak(Wave cw, Wave yw, Wave xw) : FitFunc
   Variable dummy = MPFXGaussPeak(cw, yw, xw)
End
```

The assignment to "dummy" is required because you must explicitly do something with the return value of a built-in function.

If the waves do not satisfy the number type requirements, the function returns NaN. A successful invocation returns zero.

See Also

All-At-Once Fitting Functions on page III-256

MPFXLorentzianPeak

`MPFXLorentzianPeak(cw, yw, xw)`

The MPFXLorentzianPeak function implements a single Lorentzian peak with no Y offset in the format of an all-at-once fitting function. It fills the wave yw with values defined by a Lorentzian peak as if this wave assignment statement was executed:

```
yw = 2*cw[2]/pi * cw[1]/(4*(xw-cw[0])^2 + cw[1]^2)
```

Parameters

cw Coefficient wave. The Lorentzian peak shape is defined by the coefficients as follows:

cw[0]:	Peak location.
cw[1]:	Peak width as full width at half maximum.
cw[2]:	Peak area.

cw must be a double precision wave.

yw Y wave into which values are stored.

yw may be either double precision or single precision.

xw X wave containing the X values at which the peak function is to be evaluated.

xw may be either double precision or single precision.

Details

This function is primarily intended to support the Multipeak Fitting package. To use MPFXLorentzianPeak as a fitting function, wrap it in an all-at-once user-defined fitting function:

```
Function FitLorentzianPeak(Wave cw, Wave yw, Wave xw) : FitFunc
   Variable dummy = MPFXLorentzianPeak(cw, yw, xw)
End
```

The assignment to "dummy" is required because you must explicitly do something with the return value of a built-in function.

If the waves do not satisfy the number type requirements, the function returns NaN. A successful invocation returns zero.

See Also

All-At-Once Fitting Functions on page III-256

MPFXVoigtPeak

`MPFXVoigtPeak(cw, yw, xw)`

The MPFXVoigtPeak function implements a single Voigt peak with no Y offset in the format of an all-at-once fitting function. It fills the wave yw with values defined by a Voigt peak as if this wave assignment statement was executed:

```
yw = cw[2]*VoigtFunc(cw[1]*(xw-cw[0]), cw[3])
```

The **VoigtFunc** function here is a basic Voigt peak shape, a convolution of a Gaussian and Lorentzian peak shapes. The first parameter of VoigtFunc controls the shape. A value of zero results in a peak shape that is 100% Gaussian. As the first parameter approaches infinity the shape transitions to 100% Lorentzian. At a value of sqrt(ln(2)) ≅ 0.832555 the mix is 50/50.

Parameters

cw Coefficient wave. The Gaussian peak shape is defined by the coefficients as follows:

cw[0]: Peak location.

cw[1]: Affects the width; the actual width is a complicated function of cw[1], cw[2], and cw[3]

cw[2]: Amplitude factor; the actual amplitude is affected by the other parameters.

cw[3]: Shape factor. Zero results in pure Gaussian, infinity results in pure Lorentzian, one is 50% Gaussian and 50% Lorentzian.

cw must be a double precision wave.

yw Y wave into which values are stored.

yw may be either double precision or single precision.

xw X wave containing the X values at which the peak function is to be evaluated.

xw may be either double precision or single precision.

Details

This function is primarily intended to support the Multipeak Fitting package. For other purposes we recommend the **VoigtPeak** function which has more convenient parameters.

To use MPFXVoigtPeak as a fitting function, wrap it in an all-at-once user-defined fitting function:

```
Function FitVoigtPeak(Wave cw, Wave yw, Wave xw) : FitFunc
   Variable dummy = MPFXVoigtPeak(cw, yw, xw)
End
```

The assignment to "dummy" is required because you must explicitly do something with the return value of a built-in function.

If the waves do not satisfy the number type requirements, the function returns NaN. A successful invocation returns zero.

References

The code used to compute VoigtPeak was written by Steven G. Johnson of MIT. You can learn more about it at http://ab-initio.mit.edu/Faddeeva.

See Also

All-At-Once Fitting Functions on page III-256, **VoigtPeak**, **VoigtFunc**

MultiTaperPSD

`MultiTaperPSD [`*`flags`*`]` ***`srcWave`***

The MultiTaperPSD operation estimates the power spectral density of *srcWave* using Slepian (DPSS) tapers.

The MultiTaperPSD operation was added in Igor Pro 7.00.

Flags

/A — Uses Thomson's adaptive algorithm. In this case the operation also creates the wave W_MultiTaperDF that contains the effective degrees of freedom. For each frequency of the PSD the algorithm is expected to converge within few iterations. When it fails to converge, the operation prints in the history the total number of frequencies where it did not converge while the actual output contains the last iteration estimate.

/DB — Scale the PSD results as `10*log10(spectralEst(f))`.

/DBF=*f0* — Scale the PSD results as `10*log10(spectralEst(f)/spectralEst(`*`f0`*`))` where *f0* must be in the range [0,0.5/DimDelta(*srcWave*,0)].

/DEST=destWave — Saves the PSD estimate in a wave specified by destWave. The destination wave is created or overwritten if it already exists.

Creates a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-72 for details.

If you omit /DEST the operation saves the resulting spectral estimate in the wave W_MultiTaperPSD in the current data folder.

/F — Computes F-test statistic for each output frequency. The results are stored in the wave W_MultiTaperF.

If /DEST is also used then the F-test results are stored in the same data folder as destWave. Otherwise W_MultiTaperF is created in the current data folder.

The statistic is a variance ratio, of the background and the power at the specific frequency. Since the PSDs of the background and the line are assumed to be distributed as Chi-squared with 2 and 2*nTapers-2 degrees of freedom respectively, the relevant critical value for computing confidence intervals can be obtained from:

```
StatsInvFCdf(percentSignificance/100,2,2*nTapers-2)
```

/NOR=*N* — Sets the normalization factor that is used to multiply each element of the output. For example, if you want to normalize the output such that the sum of the PSD estimate matches the variance of the input use /NOR=2/(np*np) where np is the number of points in *srcWave*.

/NTPR=*nTapers* — Specifies the number of Slepian tapers to be used. If you do not specify a number of tapers, the operation uses 2*nw(twice the time-bandwidth product).

/NW=*nw* — Specifies the time-bandwidth product. This value should typically be in the range [2,6]. Given a time-bandwidth product nw it is recommended to use no more than 2**nw* tapers in order to maximize variance efficiency.

/Q — Quiet mode; suppresses printing in the history area.

/R=[*startPoint*,*endPoint*]

Calculates the PSD estimate for a specified input range. *startPoint* and *endPoint* are expressed in terms of point numbers of the source wave.

/R=(*startX*,*endX*) — Calculates the PSD estimate for a specified input range. *startX* and *endX* are expressed in terms of X values. Note that this option converts your X specifications to point numbers and some roundoff may occur.

/Z — Do not report errors.

Details

The MultiTaperPSD operation estimates the PSD of *srcWave* by computing a set of discrete prolate spheroidal functions (Slepian **DPSS**) and using them as optimal window functions. The window functions/tapers are applied to the input signal and squares of the resulting Fourier transforms are weighted together to produce the PSD estimate.

srcWave must be a real-valued numeric wave of single or double precision and must not contain any INFs or NaNs.

The mean value of the input is subtracted prior to multiplication by the tapers. Like DSPPeriodogram, the MultiTaperPSD operation leaves the normalization to the user.

The default PSD estimate is calculated by combining the Fourier transforms of the tapered signal with weights from the DPSS calculation. You can use the /A flag to improve the PSD estimate for increasing tapers. Thomson's adaptive algorithm is reasonably efficient and also provides an estimate of the effective degrees of freedom as a function of frequency.

The operation sets the variable V_Flag to zero if successful or to a -1 if it encounters an error. If you are using Thomson's adaptive algorithm (/A) V_Flag is set to the number of frequencies at which the algorithm failed to converge.

See Also

FFT, **DSPPeriodogram**, **DPSS**, **ImageWindow**, **Hanning**, **LombPeriodogram**

Demos

See the “MultiTaperPSD Demo” example experiment.

References

D.J. Thomson: "Spectrum Estimation and Harmonic Analysis", Proc. IEEE 70 (9) 1982 pp. 1055.

D. Slepian, "Prolate Spheroidal Wave Functions, Fourier Analysis, and Uncertainty -- V: the Discrete Case", Bell System Tech J. Vol 57 (5) May-June 1978.

Lees, J. M. and J. Park (1995). Multiple-taper spectral analysis: A stand-alone C-subroutine: Computers & Geosciences: 21, 199-236.

MultiThread

MultiThread `[ /N=`*`numThreads`* `]` *wave* = *expression*

In user-defined functions, the MultiThread keyword can be inserted in front of wave assignment statements to speed up execution on multiprocessor computer systems.

Warning: Misuse of this keyword can result in a performance penalty or even a crash. Be sure to read **Automatic Parallel Processing with MultiThread** on page IV-323 before using MultiThread.

The expression must be thread-safe. This means that if it calls a function, the function must be thread-safe. This goes for both built-in and user-defined functions.

Not all built-in functions are thread-safe. Use the Command Help tab in the Igor Help Browser to see which functions are thread-safe.

User-defined functions are thread-safe if they are defined using the **ThreadSafe** keyword. See **ThreadSafe Functions** on page IV-106 for details.

You can specify the number of threads using the /NT flag. The default value, which takes effect if you omit /NT or specify /NT=0, uses the number returned by **ThreadProcessorCount**. /NT=1 specifies a single thread and is equivalent to omitting the MultiThread keyword; you might want to turn off threading to avoid its overhead when the wave has few points. *numThreads* values greater than one specify the desired number of threads to be used. Igor may use fewer threads depending on how it is able to partition the task. The /NT flag was added in Igor Pro 8.00.

See Also

Automatic Parallel Processing with MultiThread on page IV-323.

Waveform Arithmetic and Assignments on page II-74.

The “MultiThread Mandelbrot Demo” experiment.

MultiThreadingControl

`MultiThreadingControl` ***`keyword`*** **`[=`*`value`*`]`**

The MultiThreadingControl operation allows you to control how automatic multithreading works with those IGOR operations that support it. Automatic multithreading is described below under Details.

For most purposes you will not need to use this operation.

The MultiThreadingControl operation was added in Igor Pro 7.00.

Keywords

getMode — Writes the current mode value into the variable V_autoMultiThread.

getThresholds — Creates the wave W_MultiThreadingArraySizes in the current data folder. See **Automatic Multithreading Thresholds** below for details.

setMode=*m* — Sets the mode for automatic multithreading. The mode controls the circumstances in which automatic multithreading is enabled.

m=0: Disables automatic multithreading unconditionally.

m=1: Enables automatic multithreading based on operation-specific thresholds for operations called from the main thread only. This is the default setting.

m=4: Enables automatic multithreading based on operation-specific thresholds for operations called from the main thread and from user-created explicit threads.

m=8: Enables automatic multithreading unconditionally - regardless of thresholds or the type of the calling thread.

You can not combine modes by ORing. The only valid values for *m* are those shown above.

setThresholds=*tWave* — Sets the thresholds for automatic multithreading. See **Automatic Multithreading Thresholds** below for details.

Details

Some IGOR operations and functions have internal code that can execute calculations in parallel using multiple threads. These operations are marked as "Automatically Multithreaded" in the Command Help pane of the Igor Help Browser.

Running on multiple threads reduces the time required for number-crunching tasks on multi-processor machines when the benefit of using multiple processors exceeds the overhead of running in multiple threads. This is usually the case only for large-scale jobs.

By default Igor uses automatic multithreading in operations that support it when the number of calculations exceeds a threshold value. This is called "automatic multithreading" to distinguish it from the explicit multithreading that you can instruct Igor to do. Explicit multithreading is described under **ThreadSafe Functions and Multitasking** on page IV-329. You don't need to do anything to benefit from automatic multithreading.

By default automatic multithreading is enabled for operations called from the main thread and disabled for operations called from explicit threads that you create (mode=1). You can change this using the setMode keyword described above.

The state of automatic multithreading is not saved with the experiment. It is initialized to mode=1 with default thresholds every time you start IGOR.

Automatic Multithreading Thresholds

Executing these commands

```
MultiThreadingControl getThresholds
Edit W_MultiThreadingArraySizes.ld
```

creates a wave named W_MultiThreadingArraySizes and displays it in a table. This shows you the current threshold for each operation that supports automatic multithreading. The wave includes dimension labels so you can see which row represents which operation's threshold.

The meaning of a given threshold value depends on the operation. For most operations the threshold is in terms of the number of points in the input wave. For some operations the threshold depends on the complexity of the calculation. For example, the threshold for the CurveFit operation takes the complexity of the fitting function into account.

You can change the threshold for a given operation by setting the data for the appropriate row of W_MultiThreadingArraySizes and passing it back to the MultiThreadingControl operation using the setThresholds keyword. For example:

```
W_MultiThreadingArraySizes[%ICA]=5000      // Set the ICA threshold
MultiThreadingControl setThresholds=W_MultiThreadingArraySizes  // Apply
```

You should not change any aspect of the wave other than the threshold values.

Multithreading and Roundoff Error

Every floating point operation can potentially lead to roundoff error. Normally, if you run the exact same code on the the exact same data, you get the exact same roundoff error and the exact same result.

This is not necessarily the case with multithreaded calculations Because the partitioning of work among threads depends on circumstances, such as how many threads are free, different calculation runs on the same data may partition work differently. This can lead to differences in roundoff error and therefore differences in results.

You can obtain reproducibility at the expense of performance by turning off automatic multithreading:

```
MultiThreadingControl setMode=0
```

Although this provides reproducibility, it does not necessarily reduce the size of roundoff error.

Examples

```
MultiThreadingControl setMode=0         // Disable automatic multithreading
MultiThreadingControl setMode=8         // Always multithread regardless of wave size
```

See Also

Automatic Parallel Processing with TBB on page IV-323

Automatic Parallel Processing with MultiThread on page IV-323

ThreadSafe Functions on page IV-106, **ThreadSafe Functions and Multitasking** on page IV-329

NameOfWave

NameOfWave(*wave*)

The NameOfWave function returns a string containing the name of the specified wave.

In a user-defined function that has a parameter or local variable of type WAVE, NameOfWave returns the actual name of the wave identified by the WAVE reference. It can also be used with wave reference functions such as **WaveRefIndexedDFR**.

NameOfWave does not return the full data folder path to the wave. Use **GetWavesDataFolder** for this information.

A null wave reference returns a zero-length string. This might be encountered, for instance, when using WaveRefIndexedDFR in a loop to act on all waves in a data folder, and the loop has incremented beyond the highest valid index.

Examples

```
Function ZeroWave(w)
   Wave w
   w = 0
   Print "Zeroed the contents of", NameOfWave(w)
End
```

See Also

See **WAVE**; the **GetWavesDataFolder** and **WaveRefIndexed** functions; and **Wave Reference Functions** on page IV-197.

NaN

`NaN`

The NaN function returns the “Not a Number” value according to the IEEE standards.

Comparison operators do not work with NaN parameters because, by definition, NaN compared to anything, even another NaN, is false. Use **numtype** to test if a value is NaN.

NeuralNetworkRun

`NeuralNetworkRun` `[`**`/Q/Z`**`]` **`Input=`*`testWave`*`, WeightsWave1=`*`w1`*`, WeightsWave2=`*`w2`***

The NeuralNetworkRun operation uses the interconnection weights generated by NeuralNetworkTrain, and saved in the waves M_Weights1 and M_Weights2, to execute the network for a given input. The input can contain a single run represented by a 1D wave or M runs represented by M columns of a 2D wave. The output of the calculation is saved in the wave W_NNResults or M_NNResults depending on the dimensionality of the input wave. The structure of the network is completely specified by the two weights waves and must match the number of rows in the input wave.

Flags

/Q	Suppresses printing information in the History area.
/Z	No error reporting.

Parameters

Input=*testWave*	Specifies the input to the neural network. *testWave* must be a single or double precision wave containing entries in the range [0,1] and have the correct number of rows to match the weights. Execute the network for multiple runs by using a 2D input wave where each column corresponds to a single run. For a 2D input, the result will be stored in M_NNResults with a corresponding column structure.
WeightsWave1=*w1*	Specifies the interconnection weights between the input and the hidden layer.
WeightsWave2=*w2*	Specifies the interconnection weights between the hidden layer and the output.

See Also

The **NeuralNetworkTrain** operation.

NeuralNetworkTrain

`NeuralNetworkTrain [/Q/Z]` `[`***`keyword = value`***`]`...

The NeuralNetworkTrain operation trains a three-layer neural network. The training produces two 2D waves that store the interconnection weights between the network neurodes. Once you obtain the weights, you can use them with NeuralNetworkRun.

Flags

/Q	Suppresses printing information in the History area.
/Z	No error reporting.

Parameters

keyword is one of the following:

Input=*inWave*	Specifies the input patterns for training. *inWave* is a 2D wave where each row corresponds to a single training event and each column corresponds to the input values. The number of rows in *inWave* (the number of training sets) and in the output wave must be equal. *inWave* must be single or double precision and all entries must be in the range [0,1].
Iterations=*num*	Specifies the number of iterations. Default is 10000.
MinError=*val*	Terminates training when the total error drops below *val* (default is 1e-8). The total error is normalized, and is defined as the sum of the squared errors divided by the number of training sets times outputs.

Momentum=*val* Specifies a coefficient for the back-propagation algorithm. This coefficient adds to the change in a particular weight a contribution proportional to the error in a previous iteration. Default momentum is 0.075.

NHidden=*num* Specifies the number of hidden neurodes. You do not need to use the Structure keyword with NHidden because the network is completely specified by the training waves and NHidden.

NReport=*num* Specifies over how many iterations (default is 1000) to print the global RMS error to the history area. Ignored with /Q.

Output=*outWave* Specifies the expected outputs corresponding to the entries in the input wave. The number of rows in *outWave* (the number of training sets) and in the input wave must be equal. *outWave* must be single or double precision and all entries must be in the range [0,1].

LearningRate=*val* Sets the network learning rate, which is used in the backpropagation calculation. Default is 0.15.

Restart Allows specification of your own set of weights as the starting values. Use this to run the training and feed the output weights of one training session as the input for the next.

Structure={*Ni*, *Nh*, *No*}

Specifies the structure of the network. *Ni* is the number of neurodes at the input, *Nh* is the number of hidden neurodes, and *No* is the number of output neurodes. Structure is unnecessary when using NHidden is because the remaining numbers are determined by the sizes of the input and output waves.

WeightsWave1=*w1* Specifies the weights for propagation from the first layer to the second. The 2D wave must be double precision and the dimensions must match the specified neurodes with the same numbers of rows and inputs and with matching numbers of columns and hidden neurodes.

WeightsWave2=*w2* Specifies the weights for propagation from the second to the third layer. The 2D wave must be double precision and the dimensions must match the specified neurodes with the same numbers of rows and hidden neurodes and with matching numbers of columns and outputs.

Details

NeuralNetworkTrain is the first half of the implementation of a three-layer neural network in which both in inputs and outputs are taken as normalized quantities in the range [0,1]. Network training is based on back-propagation to iteratively minimize the error between the output and the expected output for any given training set. Training creates in two 2D waves that contain the interconnection weights between the neurodes. M_Weights1 contains the weights between the input layer and the hidden layer and M_Weights2 contains the weights between the hidden layer and the output layer. During the iteration stage, global error information can be printed in the history area.

The algorithm computes the output of the *k*th neurode by

$$V_k = \left[1+\exp\left(-\sum_{i=1}^{n} w_i s_i\right)\right]^{-1},$$

where w_i is the weight corresponding to input *i*, s_i is the signal corresponding to that input, and *n* is the number of inputs connected to the neurode.

The total error is defined as the sum (over all training sets and all outputs) of the squared differences between the network outputs and the expected values. The sum is normalized by the product of the number of training sets and the number of outputs. The history reports (see NReport parameter) the square root of the total error (RMS error). The square root of the error computed at the end of the last iteration is stored in the variable V_rms.

See Also

The **NeuralNetworkRun** operation.

NewCamera

`NewCamera [`*`flags`*`] [`*`keywords`*`]`

The NewCamera operation creates a new camera window.

Documentation for the NewCamera operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "NewCamera"
```

NewDataFolder

`NewDataFolder [/O/S]` ***`dataFolderSpec`***

The NewDataFolder operation creates a new data folder of the given name.

Parameters

dataFolderSpec can be just a data folder name, a partial path (relative to the current data folder) with name or a full path (starting from root) with name. If just a data folder name is used then the new data folder is created in the current data folder. If a full or partial path is used, all data folders except for the last in the path must already exist.

Flags

/O	No error if a data folder of the same name already exists.
/S	Sets the current data folder to dataFolderSpec after creating the data folder.

Examples

```
NewDataFolder foo          // Creates foo in the current data folder
NewDataFolder :bar:foo     // Creates foo in bar in current data folder
NewDataFolder root:foo     // Creates foo in the root data folder
```

See Also

Chapter II-8, **Data Folders**.

NewFIFO

`NewFIFO` ***`FIFOName`***

The NewFIFO operation creates a new FIFO.

Details

Useless until channel info is added with **NewFIFOChan**.

An error is generated if a FIFO of same name already exists. *FIFOName* needs to be unique only among FIFOs. You can not overwrite a FIFO.

See Also

FIFOs are used for data acquisition. See **FIFOs and Charts** on page IV-313 and the **NewFIFOChan** operation for more information.

NewFIFOChan

`NewFIFOChan [`*`flags`*`]` ***`FIFOName, channelName, offset, gain, minusFS, plusFS,`***
`unitsStr` **`[,`** ***`vectPnts`*** **`]`**

The NewFIFOChan operation creates a new channel for the named FIFO.

Parameters

channelName must be unique for the specified FIFO.

The *offset*, *gain*, *plusFS*, *minusFS* and *unitsStr* parameters are used when the channel's data is displayed in a chart or transferred to a wave. If given, *vectPnts* must be between 1 and 65535.

Flags

The flags define the type of data to be stored in the FIFO channel:

/B	8-bit signed integer. Unsigned if /U is present.
/C	Complex.
/D	Double precision IEEE floating point.
/I	32-bit signed integer. Unsigned if /U is present.
/S	Single precision IEEE floating point (default).
/U	Unsigned integer data.
/W	16-bit signed integer. Unsigned if /U is present.
/Y=*type*	Specifies wave data type. See details below.

Wave Data Types

As a replacement for the above number type flags you can use /Y=*numType* to set the number type as an integer code. See the **WaveType** function for code values. Do not use /Y in combination with other type flags.

Details

You can not invoke NewFIFOChan while the named FIFO is running.

If you provide a value for *vectPnts*, you will create a channel capable of holding a vector of data rather than just a single data value. When such a channel is used in a Chart, it is displayed as an image using one of the built-in color tables.

Igor scales values in the FIFO channel before displaying them in a chart or transferring them to a wave as follows:

```
scaled_value = (FIFO_value - offset) * gain
```

Igor uses the *plusFS* and *minusFS* parameters (plus and minus full scale) to set the default display scaling for charts.

The *unitsStr* parameter is limited to a maximum of three bytes.

When you transfer a channel's data to a wave, using the **FIFO2Wave** operation, Igor stores the *plusFS* and *minusFS* values and the *unitsStr* in the wave's Y scaling.

See Also

FIFOs are used for data acquisition. See **FIFOs and Charts** on page IV-313 and the **NewFIFO** and **FIFO2Wave** operations for more information.

The **Chart** operation for displaying FIFO data.

NewFreeAxis

`NewFreeAxis``[`***`flags`***`]` ***`axisName`***

The NewFreeAxis operation creates a new free axis that has no controlling wave.

Parameters

axisName is the name for the new free axis.

Flags

/L/R/B/T	Specifies whether to attach the free axis to the Left, Right, Bottom, or Top plot edge, respectively. The Left edge is used by default.
/O	Replaces *axisName* if it already exists, which means any existing axis is marked as truly free.
/W=*winName*	Draws in the named graph window. *winName* may also be the name of a subwindow. *winName* must not conflict with other axis names except when using the /O flag. If /W is omitted, it creates a new axis in the active graph window or subwindow.

Details

A truly free axis does not use any scaling or units information from any associated waves (which need not exist.) You can set the properties of a free axis using **SetAxis** or **ModifyFreeAxis**.

Example

Copy this function to your Procedure window and compile:

```
Function axhook(s)
   STRUCT WMAxisHookStruct &s

   Variable t= s.max
   s.max= s.min
   s.min= t
   return 0
End
```

Now execute this code on the Command line:

```
Make jack=x
Display jack
NewFreeAxis fred
ModifyFreeAxis fred, master=left, hook=axhook
```

See Also

The **SetAxis**, **KillFreeAxis**, and **ModifyFreeAxis** operations.

NewFreeDataFolder

`NewFreeDataFolder()`

The NewFreeDataFolder function creates a free data folder and then returns its data folder reference.

Recommended for advanced programmers only.

Details

Free data folders are those that are not a part of the normal data folder hierarchy and can not be located by name.

See Also

Chapter II-8, **Data Folders**, **Free Data Folders** on page IV-96 and **Data Folder References** on page IV-78.

NewFreeWave

NewFreeWave(*type*, *numPoints* [,*nameStr*])

The NewFreeWave function creates a free 1D wave of the given type and number of points and then returns its wave reference.

Recommended for advanced programmers only.

Details

By default, NewFreeWave creates a free wave named '_free_'. You can specify another name via the optional *nameStr* input. The ability to specify the name of a free wave was added in Igor Pro 9.00 as a debugging aid - see **Free Wave Names** on page IV-95 and **Wave Tracking** on page IV-207 for details.

You can also create free waves using **Make**/FREE and **Duplicate**/FREE. These are preferable for creating multidimensional free waves and also fine for general use.

The *type* parameter can be either a code as documented for **WaveType** or can be 0x100 to create a data folder reference wave or 0x200 to create a wave reference wave.

You can redimension free waves as desired but, for maximum efficiency, you should create the wave with the desired type and total number of points and then use the /E=1 flag with **Redimension** to simply reshape without moving data.

A free wave is automatically discarded when the last reference to it disappears.

See Also

Free Waves on page IV-91, **Make**, **Duplicate**.

NewGizmo

`NewGizmo` `[`***`flags`***`]`

The NewGizmo operation creates a new Gizmo display window.

Documentation for the NewGizmo operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "NewGizmo"
```

NewImage

`NewImage` `[`***`flags`***`]` ***`matrix`***

The NewImage operation creates a new image graph much like “`Display;AppendImage matrix`” except the graph is prepared using a style more appropriate for images. Rather than using preferences, NewImage provides several discrete styles to choose from.

Parameters

matrix is usually an MxN matrix containing image data. See **AppendImage** for details.

Flags

/F	By default, the image is flipped vertically to correspond to normal image orientation. if /F is present then the image is not flipped.
/G=*g*	Controls treatment of three-plane images as direct (RGB) color. *g*=1: Suppresses the autodetection of three-plane images as direct (RGB) color. *g*=1: Same as no /G flag (default).
/HIDE=*h*	Hides (h = 1) or shows (h = 0, default) the window.
/HOST=*hcSpec*	Embeds the new image plot in the host window or subwindow specified by *hcSpec*. When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.
/K=*k*	Specifies window behavior when the user attempts to close it. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. *k*=3: Hides the window. If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.
/N=*name*	Requests that the created graph have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen graph name.

/S=*s* Specifies one of several window styles.

s=0: Fills entire window with image. No axes. However, this can result in the lower-right corner not being visible due to the target icon or grow icon (*Macintosh*).

s=1: Like *s*=0 but insets image to avoid corner icon.

s=2: Provides minimalist axes (default).

Details

The graph is sized to make the image pixels a multiple of the screen pixels with the graph size constrained to be not too small and not too large.

If *matrix* appears to fit Igor's standard monochrome category, then explicit mode is set (See ModifyImage explicit). To be considered monochrome the wave must be unsigned byte and contain only values of 0, 64 or 255.

Once the graph is created it is a normal graph and has no special properties other than the settings it was created with. Specifically, it will not autosize itself if the dimensions of *matrix* are changed. NewImage is just a shortcut for creating a graph window with a style appropriate for images.

This operation is limited in scope by design. If you need to specify the position, size or title, then use the operations Display and AppendImage.

If the styles provided are not what you desire, touch up an image graph to meet your needs and then use Capture Graph Prefs from the Graphs menu. Then use "`Display;AppendImage`" rather than NewImage.

See Also

The **Display**, **DoWindow**, **AppendImage**, and **ModifyImage** operations.

NewLayout

`NewLayout [`*`flags`*`] [as `*`titleStr`*`]`

The NewLayout operation creates a page layout.

Unlike the Layout operation, NewLayout can be used in user-defined functions. Therefore, NewLayout should be used in new programming instead of Layout.

NewLayout just creates the layout window. Use **AppendLayoutObject** to add objects to the window.

Parameters

The optional *titleStr* parameter is a string expression containing the layout's title. If not specified, Igor will provide one which identifies the objects displayed in the graph.

Flags

/B=(*r*,*g*,*b*[,*a*]) Specifies the background color for the layout. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque white.

/C=*colorOnScreen* Obsolete. In ancient times, this flag switched the screen display of the layout between black and white and color. It is still accepted but has no effect.

/HIDE=*h* Hides (h = 1) or shows (h = 0, default) the window.

/K=*k* Specifies window behavior when the user attempts to close it.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

k=3: Hides the window.

If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.

/N=*name* Requests that the layout have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen layout name.

If /N is not used, a name of the form "Layout*n*", where *n* is some integer, is assigned. In a function or macro, the assigned name is stored in the S_name string. This is the name you can use to refer to the page layout window from a procedure. Use the **RenameWindow** operation to rename the window.

/P=*orientation* Sets the orientation of the page in the layout to either Portrait or Landscape (e.g., `Layout/P=Landscape`). See **Details**.

/W=(*left*,*top*,*right*,*bottom*)

Gives the layout window a specific location and size on the screen. Coordinates for /W are in points.

Details

When you create a new page layout window, if preferences are enabled, the page size is determined by the preferred page size as set via the Capture Layout Prefs dialog. If preferences are disabled, as is usually the case when executing a procedure, the page is set to the factory default size.

See Also

AppendLayoutObject, **DoWindow**, **RemoveLayoutObjects**, and **ModifyLayout**.

NewMovie

`NewMovie` `[`***`flags`***`] [`**`as`** ***`fileNameStr`***`]`

The NewMovie operation opens a movie file in preparation for adding frames.

By default, NewMovie creates MP4 movies on both Macintosh and Windows. Prior to Igor Pro 8, it created QuickTime movies on Macintosh and AVI movies on Windows. That older technology is still available using the /A flag on Windows, but it is deprecated and may not be available in future operating systems. QuickTime is no longer available as of Mac OS 10.15 (Catalina) and the /A flag results in an error in Igor Pro 9 on all Mac operating system versions.

Parameters

The file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If NewMovie can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/A Windows: NewMovie/A creates movie files using the deprecated AVI technology.

Macintosh: NewMovie/A returns an error.

/CF=*factor* Specifies a compression factor relative to the theoretical uncompressed value. The default compression factor of 200 is used if you omit /CF. /CF was added in Igor Pro 8.00. It is ignored if you use the /A flag.

/CTYPE=*typeStr* Specifies the compression codec to use.

Windows: *typeStr* can be "WMV3" to create .wmv files or "mp4v" to create MP4 files (default).

Macintosh: /CTYP is ignored and an MP4 file is created.

/F=*frameRate* Frames per second between 1 and 60. *frameRate d*efaults to 30.

/I Obsolete.

/L[=*flatten*]	Obsolete.
/O	Overwrite existing file, if any.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/PICT=*pictName*	Uses the specified picture (see **Pictures** on page III-509) rather than the top graph.
/S=*soundWave*	Adding a sound track is not currently supported. If you would like this feature, let us know.
/Z	No error reporting; an error is indicated by nonzero value of the output variable V_flag. If the user clicks the cancel button in the Save File dialog, V_flag is set to -1.

Details

If either the path or the file name is omitted then NewMovie displays a Save File dialog to let you create a movie file. If both are present, NewMovie creates the file automatically.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-22 for details.

There can be only one open movie at a time.

The target window at the time you invoke NewMovie must be a graph, page layout or Gizmo plot unless the /PICT flag is present. The window size should remain constant while adding frames to the movie. The window and optional sound wave are used to determine the size and sound properties only; they do not specify the first frame.

In Igor7 or later, the target window at the time you call NewMovie is remembered and is used by AddMovieFrame even if it is not the target window when you call AddMovieFrame.

The /PICT flag allows you to create a movie from a page layout in conjunction with the SavePICT/P=_PictGallery_ method. See **SavePICT** on page V-826. This allows creation of a movie from a source other than a graph, page layout or Gizmo window, but is rarely needed.

See Also

Movies on page IV-245.

The **AddMovieFrame**, **AddMovieAudio**, **CloseMovie**, **PlayMovie**, **PlayMovieAction** and **SavePICT** operations.

NewNotebook

`NewNotebook` `[`***`flags`***`] [`**`as`** ***`titleStr`***`]`

The NewNotebook operation creates a new notebook document.

Parameters

The optional *titleStr* is a string containing the title of the notebook window.

Flags

/FG=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the frame guide to which the outer frame of the subwindow is attached inside the host window.

The standard frame guide names are `FL`, `FR`, `FT`, and `FB`, for the left, right, top, and bottom frame guides, respectively, or user-defined guide names as defined by the host. Use `*` to specify a default guide name.

Guides may override the numeric positioning set by /W.

/HOST=*hcSpec* Embeds the new notebook in the host window or subwindow specified by *hcSpec*. The host window or subwindow must be a control panel. Graphs and page layouts are not supported as hosts for notebook subwindows.

When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

See **Notebooks as Subwindows in Control Panels** on page III-91 for more information.

/ENCG=*textEncoding*

textEncoding specifies the text encoding for the new notebook. This determines the text encoding used for later saving the notebook to a file.

See **Text Encoding Names and Codes** on page III-490 for a list of accepted values for textEncoding.

This flag was added in Igor Pro 7.00.

This flag is relevant for plain text notebooks only and has no effect for formatted notebooks because formatted text notebooks can contain multiple text encodings. See **Plain Text File Text Encodings** on page III-466 and **Formatted Text Notebook File Text Encodings** on page III-472 for details.

If you omit /ENCG or pass 0 (unknown) for *textEncoding*, the notebook's text encoding is determined by the default text encoding - see **The Default Text Encoding** on page III-465 for details.

For most purposes, UTF-8 (*textEncoding*=1) is recommended. Other values are available for compatibility with software that requires a specific text encoding. This includes Igor Pro 6 which uses MacRoman (*textEncoding*=2), Windows-1252 (*textEncoding*=3) or Shift-JIS (*textEncoding*=4) depending on the operating system and localization.

This flag has an optional form that allows you to control whether the byte order mark is written when the notebook is later saved to disk. It applies to Unicode text encodings also. The form is:

```
/ENCG = {textEncoding, writeBOM }
```

If you use the simpler form or omit /ENCG entirely, the notebook's writeBOM property defaults to 1.

See **Byte Order Marks** on page III-471 for background information.

/F=*format* Specifies the format of the notebook:

format=0:	Plain text.
format=1:	Formatted text.
format=-1:	Displays a dialog in which the user can choose plain text or formatted text (default).

/K=*k* Specifies window behavior when the user attempts to close it.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.
k=3:	Hides the window.

If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.

/N=*winName* Sets the notebook’s window name to *winName*.

/OPTS=*options* Sets special options. *options* is a bitwise parameter interpreted as follows:

Bit 0:	Hide the vertical scroll bar.
Bit 1:	Hide the horizontal scroll bar.
Bit 2:	Set the write-protect icon initially to on.
Bit 3:	Sets the changeableByCommandOnly bit. When set, the user can not make any modifications.

All other bits are reserved and must be set to zero.

If /OPTS is omitted, all bits default to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/V=*visible* Specifies whether the notebook window is visible (*visible*=1; default) or invisible (*visible*=0).

/W=(*left*,*top*,*right*,*bottom*)

Sets window location. Coordinates are in points for normal notebook windows.

When used with the /HOST flag, the specified location coordinates can have one of two possible meanings:

When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.

When any value is greater than 1, coordinates are taken to be fixed locations measured in points, or **Control Panel Units** for control panel hosts, relative to the top left corner of the host frame.

Details

A notebook has a file name, a window name, *and* a window title. In the simplest case these will all be the same.

The file name is the name by which the operating system identifies the notebook once it is saved to disk. When you initially create a notebook, it is not associated with any file. However it still has a file name. This is the name that will be used when the file is saved to disk.

The window name is the name by which Igor identifies the window and therefore the name you specify in operations that act on the notebook.

The window title is what appears in the window's title bar. If you omit the title, NewNotebook uses a default title that is the same as the window name.

If you specify the window name and the notebook format and omit the window title, this is the simplest case. NewNotebook creates the document with no user interaction. The file name, window name and window title will all be the same. For example:

```
NewNotebook/N=Notebook1/F=0
```

If you omit the window name, NewNotebook chooses a default name (e.g., "Notebook0") and presents the standard New Notebook dialog.

If you omit the format or specify a format of -1 (either plain or formatted text), NewNotebook presents the standard New Notebook dialog. For example:

```
NewNotebook/N=Notebook1        // no format specified
```

See Also

The **Notebook** and **OpenNotebook** operations, and Chapter III-1, **Notebooks**.

Notebooks as Subwindows in Control Panels on page III-91.

NewPanel

NewPanel [***flags***] [**as *titleStr***]

The NewPanel operation creates a control panel window or subwindow, which may contain Igor controls and drawing objects.

Flags

/EXP=*e* Sets the expansion of the panel. *e* is a number between 0.25 to 8.0. Values of *e* greater than 1.0 make the panel, its controls, and its subwindows, appear larger than normal.

The expansion factor affects the size of the window specified by /W.

If you omit /EXP or specify /EXP=0, the expansion defaults to the value set in the Panel section of the Miscellaneous Settings dialog.

To change the expansion after the panel is created, use ModifyPanel expand=e or the Expansion submenu of the Panel menu.

The /EXP flag was added in Igor Pro 9.00.

See **Control Panel Expansion** on page III-443 for further discussion.

/EXT=*e* Creates an exterior subwindow in combination with /HOST. *e* specifies the host window side location:

e=0:	Right.
e=1:	Left.
e=2:	Bottom.
e=3:	Top.

/FG=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the frame guide to which the outer frame of the subwindow is attached inside the host window.

The standard frame guide names are FL, FR, FT, and FB, for the left, right, top, and bottom frame guides, respectively, or user-defined guide names as defined by the host. Use * to specify a default guide name.

Guides may override the numeric positioning set by /W.

/FLT[=*f*] /FLT or /FLT=1 makes the panel a floating panel.

/FLT=2 makes it a floating panel with no close box.

/FLT=0 is the same as omitting /FLT and creates a regular (non-floating) control panel.

You must execute the following after the NewPanel command:

```
SetActiveSubwindow _endfloat_
```

See **Floating Panels** below for further information.

/FLTH=h Selects whether a floating panel is automatically hidden when Igor is deactivated.

h=0:	The panel is not automatically hidden.
h=1:	The panel is automatically hidden.

The default is platform specific and matches the behavior prior to Igor Pro 9. On Macintosh floating panels are hidden when Igor is deactivated. On Windows they are not hidden when Igor is deactivated.

/HIDE=*h* Hides (h = 1) or shows (h = 0, default) the window.

/HOST=*hcSpec* Embeds the new control panel in the host window or subwindow specified by *hcSpec*.

When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/I Sets coordinates to inches.

/K=*k* Specifies window behavior when the user attempts to close it.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

k=3: Hides the window.

If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.

Exterior subwindows never display a dialog when killed.

/M Sets coordinates to centimeters.

/N=*name* Requests that the created panel have this name, if it is not in use. If it is in use, then *name*0, `name`1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen panel name.

Note that a function or macro with the same name will cause a name conflict.

/NA= *n* Sets panel no-activate mode.

n=0: Normal (default).

n=1: Button click doesn't activate window but click outside of any control does.

n=2: No activation even if click is outside controls. Title bar clicks still activate.

/W=(*left*,*top*,*right*,*bottom*)

Sets the initial coordinates of the panel window. See **Interpretation of NewPanel Coordinates** on page III-444 for a discussion of the *left*, *top*, *right*, and *bottom* parameters.

When used with the /HOST flag, the specified location coordinates of the sides can have one of two possible meanings:

When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size. This applies to interior panels only, not to exterior panels.

When any value is greater than 1, coordinates are taken to be fixed locations measured in points, or **Control Panel Units** for control panel hosts, relative to the top left corner of the host frame.

When the subwindow position is fully specified using guides (using the /HOST or /FG flags), the /W flag may still be used although it is not needed.

Details

If /N is not used, NewPanel automatically assigns to the panel a window name of the form "Panel*n*", where *n* is some integer. In a function or macro, the assigned name is stored in the S_name string. This is the name you can use to refer to the panel from a procedure. Use the **RenameWindow** operation to rename the panel.

On Windows there are special considerations relating to screen resolution and control panels. See **Control Panel Resolution on Windows** on page III-456 for details.

Floating Panels

Floating control panels float above all other windows except dialogs. Because floating panels cover up other windows, you should use them sparingly and you should take care to make them small and unobtrusive.

Floating panels are not resizable by default. To allow panel resizing use

```
ModifyPanel fixedSize=0
```

Because floating panels always act as if they are on top, the standard rules for target windows and keyboard focus do not apply.

Normally, a floating panel is never the target window and control procedures will need to explicitly designate the target. But a newly-created floating panel is the default target and will remain so until you execute

```
SetActiveSubwindow _endfloat_
```

It also becomes the default target when the tools are showing and in any non-Operate mode. Similarly, a floating panel with tools not in Operate mode has keyboard focus. To avoid confusion, do not attempt to work on other windows when a floating panel is the default target.

When working with a floating panel, you can show or hide tools or create a recreation macro by Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the panel.

A floating panel does not have keyboard focus. However, a floating panel gains keyboard focus when a control that needs focus is clicked. Focus remains until you press Enter or Escape for a text entry in a setvariable, press Tab until no control has the focus, or until you click outside a focusable control.

On Macintosh, if a floating panel has focus and you activate another window, focus will leave the panel. However on Windows, if a floating panel has focus and you activate another window, the activate sequence will be fouled up leaving the windows in an indeterminate state. Consequently, it is important that you always finish any keyboard interaction started in a floating panel before moving on to other windows. If this can cause confusion, you should not use controls such as SetVariable and ListBox in a floating panel.

On Macintosh, floating panels are hidden when dialogs are up or when Igor Pro is not the front application.

Exterior Subwindows

Exterior subwindows are automatically positioned along the designated side of a host window. The host window can be a graph, table, panel or Gizmo plot. You can designate fixed sizes or automatic size with minima. Subwindows are stacked beside the designated side in their creation order with the first one closest.

Subwindow dimensions have various meanings depending on their location. Interior values are taken to be additional grout, exterior values are taken to be sizes. For left or right panels, top is taken to be the minimum height and bottom, if not zero, is height. For top and bottom, left is taken to be the minimum width and right, if not zero, is width. Zero values default to 50 for width and height or size of host.

Exterior subwindows are nonresizable by default. Use `ModifyPanel fixedSize=0` to allow manual resizing. If you resize a panel, the original window dimensions are lost. You can also use **MoveSubwindow** to resize the subwindow.

Unlike normal subwindows, exterior subwindows have a tools palette. Click in the window and then choose the Show Tools or Hide Tools menu item.

Exterior subwindows have hook functions independent of the host window.

Examples

In a new experiment, execute these commands on the command line to create two exterior subwindows:

```
Display
// Create panel on right with min height of 200 points, width of 100.
NewPanel/HOST=Graph0/EXT=0/W=(0,200,100,0)
// Create another panel on right with grout of 10 and height= width= 100.
NewPanel/HOST=Graph0/EXT=0/W=(10,0,100,100)
```

Now try resizing and moving the graph.

For a demonstration of how the various exterior panels work, copy the following code to the procedure window in a new experiment:

```
Function bpNewExSw(ba) : ButtonControl
    STRUCT WMButtonAction &ba

    switch( ba.eventCode )
        case 2:                              // mouse up
            ControlInfo/W=$ba.win ckUseRect
            Variable useR= V_Value
            ControlInfo/W=$ba.win popSide
            Variable side= V_Value-1
            ControlInfo/W=$ba.win ckResizeable
            Variable resizeable= V_Value
            WAVE w=root:epsizes
            if( useR )
                NewPanel/HOST=$ba.win/EXT=(side)/W=(w[0],w[1],w[2],w[3])
            else
                NewPanel/HOST=$ba.win/EXT=(side)
            endif
            if( resizeable )
                ModifyPanel fixedSize=0 // default is 1 for floating and exterior sw
            endif
            break
    endswitch

    return 0
End

Window ExSwTest() : Graph
    PauseUpdate; Silent 1               // building window...
    Display /W=(803,377,1158,591)
    Button bNewSW,pos={35,21},size={181,30},proc=bpNewExSw,title="Exterior Subwindow"
    SetVariable svLeft,pos={118,82},size={96,15},title="left"
    SetVariable svLeft,limits={0,100,1},value= epsizes[0],bodyWidth= 76
    SetVariable svTop,pos={120,97},size={94,15},title="top"
    SetVariable svTop,limits={0,100,1},value= epsizes[1],bodyWidth= 76
    SetVariable svRight,pos={112,113},size={102,15},title="right"
    SetVariable svRight,limits={0,100,1},value= epsizes[2],bodyWidth= 76
    SetVariable svBottom,pos={103,129},size={111,15},title="bottom"
    SetVariable svBottom,limits={0,100,1},value= epsizes[3],bodyWidth= 76
    CheckBox ckUseRect,pos={70,62},size={61,14},title="Use Rect:",value= 0
    PopupMenu popSide,pos={73,149},size={78,20},title="Side"
    PopupMenu popSide,mode=1,popvalue="Right",value= #"\"Right;Left;Bottom;Top\""
    CheckBox ckResizeable,pos={76,176},size={65,14},title="Resizeable",value= 0
EndMacro

Function test()
    Make/O/N=4 epsizes=0
    Execute "ExSwTest()"
End
```

After compiling the procedures, execute `test()` on the command line. You can now experiment with different sides and size values.

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Interpretation of NewPanel Coordinates on page III-444 for a discussion of the units used with NewPanel /W.

The **ModifyPanel** operation.

NewPath

NewPath [***flags***] ***pathName*** [, ***pathToFolderStr***]

The NewPath operation creates a new symbolic path name that can be used as a shortcut to refer to a folder on disk.

Parameters

pathToFolderStr is a string containing the path to the folder for which you want to make a symbolic path. *pathToFolderStr* can also point to an alias (*Macintosh*) or shortcut (*Windows*) for a folder.

If you use a full path for *pathToFolderStr,* see **Path Separators** on page III-451 for details on forming the path. If you use a partial path or just a simple name for *pathToFolderStr,* and you use the /C flag, a new folder is created relative to the Igor Pro folder. No dialog is presented.

If you omit *pathToFolderStr*, you get a chance to select a folder or create a new folder from a dialog.

Flags

/C	Create the folder specified by *pathToFolderStr* if it does not already exist.
/M=*messageStr*	Specifies the prompt message in the dialog.
/O	Overwrites the symbolic path if it exists.
/Q	Suppresses printing path information in the history.
/Z	Doesn’t generate an error if the folder does not exist.

Details

Symbolic paths help to isolate your experiments from specific file system paths that contain files created or used by Igor. By using a symbolic path, if the actual location or name of the folder changes, you won’t need to change all of your commands. Instead, you need only to change the symbolic path so that it points to the changed folder location.

NewPath sets the variable V_flag to zero if the operation succeeded or to nonzero if it failed. The main use for this is to determine if the user clicked Cancel when you use NewPath to display a choose-folder dialog.

On the Macintosh, pressing Command-Option as the Choose Folder dialog comes up will allow you to choose package folders and folders inside packages.

Examples

```
NewPath Path1, "hd:IgorStuff:Test 1"                 // Macintosh
NewPath Path1, "C:IgorStuff:Test 1"                  // Windows
```

creates the symbolic path named Path1 which refers to the specified folder (the path's "value"). You can then refer to this folder in many Igor operations and dialogs by using the symbolic path name Path1.

Windows Note:

You can use either the colon or the backslash character to separate folders. However, the backslash character is Igor's escape character in strings. This means that you have to double each backslash to get one backslash like so:

```
NewPath stuff, "C:\\IgorStuff\\Test 1"
```

Because of this complication, it is recommended that you use Macintosh path syntax even on Windows. See **Path Separators** on page III-451 for details.

See Also

The **PathInfo** operation; especially if you need to preset a starting path for the dialog.

KillPath

NewWaterfall

`NewWaterfall` [***`flags`***] ***`matrixWave`*** [**`vs {`*`xWave`*`,`*`yWave`*`}`**]

The NewWaterfall operation creates a new waterfall plot window or subwindow using each column in the 2D matrix wave as a waterfall trace.

You can manually set x and z scaling by specifying *xWave* and *yWave* to override the default scalings. Either *xWave* or *yWave* may be omitted by using a "*".

Flags

/FG=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the frame guide to which the outer frame of the subwindow is attached inside the host window.

The standard frame guide names are FL, FR, FT, and FB, for the left, right, top, and bottom frame guides, respectively, or user-defined guide names as defined by the host. Use * to specify a default guide name.

Guides may override the numeric positioning set by /W.

/HIDE=*h* Hides (h = 1) or shows (h = 0, default) the window.

/HOST=*hcSpec* Embeds the new waterfall plot in the host window or subwindow specified by *hcSpec*.

When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/I Sets window coordinates to inches.

/K=*k* Specifies window behavior when the user attempts to close it.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.
k=3:	Hides the window.

If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.

/M Sets window coordinates to centimeters.

/N=*name* Requests that the created waterfall plot window have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen name.

/PG=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the inner plot rectangle of the waterfall plot subwindow inside its host window.

The standard plot rectangle guide names are PL, PR, PT, and PB, for the left, right, top, and bottom plot rectangle guides, respectively, or user-defined guide names as defined by the host. Use * to specify a default guide name.

Guides may override the numeric positioning set by /W.

/W=(*left*,*top*,*right*,*bottom*)

Specifies window size. Coordinates are in points unless /I or /M is specified before /W.

When used with the /HOST flag, the specified location coordinates of the sides can have one of two possible meanings:

When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.

When any value is greater than 1, coordinates are taken to be fixed locations measured in points, or **Control Panel Units** for control panel hosts, relative to the top left corner of the host frame.

When the subwindow position is fully specified using guides (using the /HOST, /FG, or /PG flags), the /W flag may still be used although it is not needed.

Details

The X and Z axes are always at the bottom and left, whereas the Y axis runs at a default 45 degrees along the right-hand side. The angle and length of the Y axis can be changed using the ModifyWaterfall operation. Other features of the graph can be changed using normal graph operations.

Each column from *matrixWave* is plotted in, and clipped by, a rectangle defined by the X and Z axes with the rectangle displaced along the angled Y axis as a function of the Y value.

Except when hidden lines are active, the traces are drawn from back to front.

To modify certain properties of a waterfall plot, you need to use the ModifyWaterfall operation. For other properties, use the usual axis and trace dialogs.

See Also

Waterfall Plots on page II-326.

The **ModifyWaterfall** and **ModifyGraph** operations.

norm

norm(*srcWave*)

The norm function evaluate the norm of *srcWave*. It returns:

$$\sqrt{\sum abs(w[i])^2}$$

This function does not support text waves.

See Also
MatrixOp

NormalizeUnicode

NormalizeUnicode(*sourceTextStr*, *normalizationForm*[, *options*])

The NormalizeUnicode function normalizes the UTF-8-encoded text in sourceTextStr using the specified normalization form. The output text encoding is UTF-8.

NormalizeUnicode was added in Igor Pro 7.00. Most users will have no need for this function and can ignore it.

As explained under Details, in Unicode there are sometimes multiple ways to spell what appears visually to be the same word. This can cause problems when comparing text. Two strings that appear to represent the same word and which you consider equivalent may be spelled differently, causing a comparison operation to indicate that they are unequal. The NormalizeUnicode function converts *sourceTextStr* to a normalized form, which aides comparison.

Parameters

sourceTextStr is the text that you want to normalize. It must be encoded as UTF-8.

normalizationForm specifies the normalization form to use. These forms are described at http://unicode.org/reports/tr15/#Norm_Forms. The allowed values are:

0:	NFD (Canonical Decomposition)
1:	NFC (Canonical Decomposition, followed by Canonical Composition)
2:	NFKD (Compatibility Decomposition)
3:	NFKC (Compatibility Decomposition, followed by Canonical Composition)

options is a bitwise parameter, with the bits defined as follows:

Bit 0:	If cleared, in the event of an error, a null string is returned and an error is generated. Use this if you want to abort procedure execution if an error occurs. If set, in the event of an error, a null string is returned but no error is generated. Use this if you want to detect and handle an error yourself. You can test for null using strlen as shown in **String Variable Text Encoding Error Example** on page III-479.

All other bits are reserved and must be cleared.

Details

The Unicode standard specifies that some sequences of code points represent essentially the same character. There are two types of equivalence: canonical equivalence and compatibility.

Sequences of code points defined as canonically equivalent are assumed to have the same appearance and meaning when printed or displayed. For example, the code point U+006E (LATIN SMALL LETTER N) followed by U+0303 (COMBINING TILDE) is defined by Unicode to be canonically equivalent to the single code point U+00F1 (LATIN SMALL LETTER N WITH TILDE). The former is called "decomposed" while the later is called "precomposed".

Sequences that are defined as compatible are assumed to have possibly distinct appearances, but the same meaning in some contexts. Thus, for example, the code point U+FB00 (LATIN SMALL LIGATURE FF) is defined to be compatible, but not canonically equivalent, to the sequence U+0066 U+0066 (two Latin "f" letters). Sequences that are canonically equivalent are also compatible, but the opposite is not necessarily true.

Text searching and sorting routines in Igor do not do any form of Unicode normalization. As a consequence, searching for the precomposed form of small letter n with tilde (U+00F1) in a string that contains the decomposed form (U+006E U+0303) will not result in a match. To get the desired result, you would need to first pass both the target string and the string to be searched through NormalizeUnicode using the same value for the *normalizationForm* parameter.

Example

```
Function TestNormalizeUnicode()
   String precomposed = "Ni" + "\u00F1" + "o"
   String decomposed = "Ni" + "\u006E\u0303" + "o"
   String precomposedTarget = "\u00F1"
   String decomposedTarget = "\u006E\u0303"
   Variable foundPos

   // SUCCESSFUL TESTS
   // Searching the precomposed string for the precomposed target is successful.
   foundPos = strsearch(precomposed, precomposedTarget, 0)
   Print foundPos                // Prints 2

   // Likewise, searching the decomposed string for the decomposed target is successful.
   foundPos = strsearch(decomposed, decomposedTarget, 0)
   Print foundPos                // Prints 2

   // UNSUCCESSFUL TESTS
   // Searching the precomposed string for the decomposed target fails.
   foundPos = strsearch(precomposed, decomposedTarget, 0)
   Print foundPos                // Prints -1

   // Likewise, searching the decomposed string for the precomposed target fails.
   foundPos = strsearch(decomposed, precomposedTarget, 0)
   Print foundPos                // Prints -1

   // USING NormalizeUnicode() FUNCTION
   Variable normForm = 2         // Could use 0-3 and the results would be the same.

   String precomposedNorm = NormalizeUnicode(precomposed, normForm)
   String decomposedNorm = NormalizeUnicode(decomposed, normForm)
   String precomposedTargetNorm = NormalizeUnicode(precomposedTarget, normForm)
   String decomposedTargetNorm = NormalizeUnicode(decomposedTarget, normForm)

   // Now, searching either precomposedNorm or decomposedNorm for either
   // precomposedTargetNorm or decomposedTargetNorm will give a match.
   Print strsearch(precomposedNorm, precomposedTargetNorm, 0)  // Prints 2
   Print strsearch(decomposedNorm, precomposedTargetNorm, 0)   // Prints 2
   Print strsearch(precomposedNorm, decomposedTargetNorm, 0)   // Prints 2
   Print strsearch(decomposedNorm, decomposedTargetNorm, 0)    // Prints 2
End
```

See Also

Text Encodings on page III-459, **String Variable Text Encoding Error Example** on page III-479

http://en.wikipedia.org/wiki/Unicode_equivalence

http://unicode.org/reports/tr15/#Norm_Forms

note

`note(`*`waveName`*`)`

The note function returns a string containing the note associated with the specified wave.

See Also

To create a wave note, use the **Note** *operation*.

Note

`Note [/K/NOCR]` ***`waveName`*** `[,` ***`str`***`]`

The Note operation appends *str* to the wave note for the named wave.

Parameters

str is a string expression.

Flags

/K	Kills existing note for specified wave.
/NOCR	Appends note without a preceding carriage return (\r character). No effect when used with /K.

Examples

```
Note/K wave0        // remove existing note
Note wave0, "This is the first line of the note"
Note wave0, "This is the second line of the note"
Note/K wave0, "This is now the only line of the note"
```

See Also

To get the contents of a wave note, use the **note** *function*.

Notebook

`Notebook` ***`winName`*****`,`** ***`keyword=value`*** `[,` ***`keyword=value`***`]`…

The Notebook operation sets various properties of the named notebook window. Notebook also inserts text and graphics. See Chapter III-1, **Notebooks**, for general information on notebooks.

Notebook returns an error if the notebook is open for read-only. Keywords that don't materially change the notebook, including findText, findPicture, selection, visible, magnification, userKillMode, showRuler and rulerUnits, are still permitted. See **Notebook Read/Write Properties** on page III-10 for further information.

Parameters

winName is either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-92 for details on host-child specifications.

If *winName* is an hcSpec, the host window or subwindow must be a control panel. Graphs and page layouts are not supported as hosts for notebook subwindows.

The parameters to the Notebook operation are of the form *keyword*=*value* where *keyword* says what to do and *value* is a parameter or list of parameters. Igor limits the parameters that you specify to legal values before applying them to the notebook.

The parameters are classified into related groups of *keyword*s.

See Also

To create or modify a notebook action special character, see **NotebookAction**.

To create a notebook subwindow in a control panel, see **Notebooks as Subwindows in Control Panels** on page III-91.

Notebook (*Document Properties*)

Notebook document property parameters

This section of Notebook relates to setting the document properties of the notebook.

adopt=*a* Adopts a notebook if it is a file saved to disk. Adopting a notebook makes it part of the packed experiment file, which becomes more self-contained; if you send the experiment to a colleague you will not need to send a notebook file.

a=0: Checks only whether the notebook is adoptable. Sets V_flag to 0 if the notebook is already adopted or to 1 if it is adoptable.

a=1: Checks only whether the notebook is adoptable. Sets V_flag to 0 if the notebook is already adopted or to 1 if it is adoptable.

backRGB=(*r*,*g*,*b*[,*a*]) Sets background color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. Alpha (*a*) is accepted but ignored.

changeableByCommandOnly=*c*

This changeableByCommandOnly property is used to prevent manual modifications to the notebook but allow modifications using commands.

c=0: Turn changeableByCommandOnly off.

c=1: Turn changeableByCommandOnly on.

See **Notebook Read/Write Properties** on page III-10 for details.

defaultTab=*dtwp* *dtwp* is the default tab width in points.

The defaultTab keyword sets the default tab mode to 1 meaning that the default tab width for all paragraphs is specified in units of points. See the defaultTab2 keyword for further discussion.

defaultTab2={*mode*,*dtwp*,*dtws*}

Controls the width of default tab stops.

The defaultTab2 keyword was added in Igor Pro 9.00.

mode is defined as follows:

mode=1: Points mode: The default tab width for all paragraphs is specified in units of points by dtwp.

mode=2: Spaces mode: The default tab width for all paragraphs is specified in units of spaces by *dtws*.

mode=3: Mixed mode: The default tab width for paragraphs controlled by proportional fonts is specified in units of points by *dtwp*. The default tab width for paragraphs controlled by monospace fonts is specified in units of spaces by *dtws*.

Specify -1 for mode to leave the mode unchanged.

The space character unit used in mode 2 and in mode 3 for monospace fonts is the width of a space character in the ruler font for a given paragraph. Plain text notebooks have only one ruler so the space character width is the same for all paragraphs. Formatted text notebooks can have many rulers and each has an associated space character width.

dtwp is the default tab width in points. It is used in mode 1 and in mode 3 for proportional fonts. Specify -1 for *dtwp* to leave the default tab width in points unchanged.

dtws is the default tab width in spaces. It is used in mode 2 and in mode 3 for monospace fonts. Specify -1 for *dtws* to leave the default tab width in spaces unchanged.

See **Notebook Default Tabs** on page III-6 for further discussion.

magnification=*m* Specifies the desired magnification in percent (between 25 and 500). Otherwise, *m* can be one of these special values:

m=1: Default magnification.

m=2: Default magnification.

In Igor Pro 6 this specified the no-longer-supported Fit Width mode.

m=3: Default magnification.

In Igor Pro 6 this specified the no-longer-supported Fit Page mode.

pageMargins={*left*, *top*, *right, bottom*}

Sets page margins in points. *left*, *top*, *right,* and *bottom* are distances from the respective edges of the physical page.

This setting overrides the margins set via the the page setup dialog and the PrintSettings operation margins keyword.

rulerUnits=*r* Sets the units for the ruler:

r=0: Points.

r=1: Inches.

r=2: Centimeters.

showRuler=*s* Hides (*s*=0) or shows (*s*=1) the ruler.

startPage=*sp* Sets the starting page number for printing.

statusWidth=*sw* As of Igor7, because of changes to the layout of notebook windows, this keyword does nothing.

In Igor6 it set the width in points of the status area on the left of the horizontal scroll bar.

userKillMode=*k* Specifies window behavior when the user attempts to close it.

k=0: Normal with dialog (default).

k=1: Clicking the close button kills the notebook with no dialog.

k=2: Clicking the close button does nothing.

k=3: Clicking the close button hides the notebook with no dialog.

writeBOM=*w* Sets the document's writeBOM property which determines if Igor writes a byte order mark when saving the notebook. This applies to plain text notebooks only and is ignored for formatted text notebooks.

w=-1: Does not change writeBOM flag.

w=0: Sets writeBOM to false.

w=1: Sets writeBOM to true.

See **Byte Order Marks** on page III-471 for details.

This keyword was added in Igor Pro 7.00.

writeProtect=*wp* The write-protect property is used to prevent inadvertent manual changes to the notebook.

wp=0: Turn write-protect off.

wp=1: Turn write-protect on.

See **Notebook Read/Write Properties** on page III-10 for details.

Notebook (*Headers and Footers*)

Notebook headers and footers

You can turn headers and footers on and off and position headers and footers using the keywords in this section.

There is currently no way to set the content of headers and footers except manually through the Document Settings dialog. You may be able to use stationery files to create files with specific headers and footers.

footerControl={*defaultFooter*, *firstFooter*, *evenOddFooter*}

defaultFooter is 1 to turn the default footer on, 0 to turn it off.

firstFooter is 1 to turn the first page footer on, 0 to turn it off.

evenOddFooter is 1 to turn different footers for even and odd pages on, 0 to use the same footer for even and odd pages.

footerPos=*pos* *pos* is the position of the footer relative to the bottom of the page in points.

headerControl={*defaultHeader* , *firstHeader* , *evenOddHeader*}

defaultHeader is 1 to turn the default header on, 0 to turn it off.

firstHeader is 1 to turn the first page header on, 0 to turn it off.

evenOddHeader is 1 to turn different headers for even and odd pages on, 0 to use the same header for even and odd pages.

headerPos=*pos* *pos* is the position of the header relative to the top of the page in points.

Notebook (*Miscellaneous*)

Notebook miscellaneous parameters

This section of Notebook relates to setting miscellaneous properties of the notebook.

autoSave=*v* Controls auto-save mode.

v=0:	Notebook subwindow contents will not be saved in recreation macros.
v=1:	Notebook subwindow contents will be saved in recreation macros (default).

This affects notebook subwindows in control panels only. Use autoSave=0 if you do not want the notebook's contents to be saved and restored when the control panel is recreated. Otherwise the notebook subwindow’s contents will be restored when recreated.

frameInset= *i* Specifies the number of pixels by which to inset the frame of a notebook subwindow. Does not affect a normal notebook window.

This keyword was added in Igor Pro 7.00.

frameStyle=*f* Specifies the frame style for a notebook subwindow. Does not affect a normal notebook window.

f=0:	None.
f=1:	Single.
f=2:	Double.
f=3:	Triple.
f=4:	Shadow.
f=5:	Indented.
f=6:	Raised.
f=7:	Text well.

The last three styles are fake 3D and will look best if the background color behind the subwindow is a light shade of gray.

This keyword was added in Igor Pro 7.00.

status={*messageStr*, *flags*}

Sets the message in the status area at the bottom left of the notebook window.

flags is interpreted bitwise. Message is erased when:

Bit 0:	Selection changes.
Bit 1:	Window is activated.
Bit 2:	Window is deactivated.
Bit 3:	Document is modified.

If all bits are zero, the message stays until a new message comes along. All other bits are reserved for future use and should be zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

updating={*flags*, *r*} Sets parameters related to the updating of special characters.

flags is interpreted bitwise:

Bit 0:	Suppress automatic periodic updating of date and time special characters. By default this bit is set so date and time special characters are updated only when the user explicitly requests it or during printing when they appear in headers and footers.
Bit 1:	Allow manual updating of special characters via the specialUpdate keyword or via the Special menu. By default this is cleared so manual updating is not allowed.

All other bits are reserved for future use and should be zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

r is the update rate in seconds for updating date and time special characters.

These settings have no effect on the updating of special characters in headers or footers. These characters are always automatically updated when the document is printed.

We recommend that you leave automatic updating off (set bit 0 of the *flags* parameter to 1) so that updating occurs only via the specialUpdate keyword or via the Special menu.

visible=*v* Sets notebook visibility.

v=0:	Hides notebook.
v=1:	Shows notebook but does not make it top window.
v=2:	Shows notebook and makes it top window.

Notebook (*Paragraph Properties*)

Notebook paragraph property parameters

This section of Notebook relates to setting the paragraph properties of the current selection in the notebook.

The margins, spacing, justification, tabs and rulerDefaults keywords provide control over paragraph properties which are governed by rulers. These keywords, in conjunction with the ruler and newRuler keywords, allow you to set paragraph properties. They are allowed for formatted text notebooks only, not for plain text notebooks.

The ruler keywords are described in detail below. Before we get to the detail, you should understand the different things you can do with rulers.

There are four things you can do with a ruler:

modify it	(analogous to manually adjusting a ruler).
redefine it	(analogous to the Redefine Ruler dialog).
create it	(analogous to the Define New Ruler dialog).
apply it	(analogous to selecting a ruler name from Ruler pop-up menu).

Igor's behavior in response to ruler keywords depends on the order in which the keywords appear.

To modify the ruler(s) for the selected paragraph(s), use the margins, spacing, justification, tabs and rulerDefaults keywords *without* using the newRuler or ruler keywords. For example:

```
Notebook Notebook0 tabs={36,144,288},justification=1
```

To redefine an existing ruler, invoke the ruler=*rulerName* keyword *before* any other keywords. For example:

```
Notebook Notebook0 ruler=Ruler1,tabs={36,144,288},justification=1
```

Unlike redefining the ruler manually, when you redefine an existing ruler using ruler=*rulerName*, it does not apply the ruler to the selected text. However, it does update any text governed by the redefined ruler.

To create a new ruler, invoke the newRuler=*rulerName* keyword *before* any other keywords. For example:

```
Notebook Notebook0 newRuler=Ruler1,tabs={36,144,288},justification=1
```

Unlike creating it manually, when you create a new ruler using newRuler=*rulerName*, it does not apply the new ruler to the selected text. If you do not set a particular ruler property when creating a new ruler, the property will be the same as for the Normal ruler. If the specified ruler already exists, newRuler=*rulerName* overwrites the existing ruler.

To apply an existing ruler to the selected text, invoke the ruler=*rulerName* keyword without any other keywords. For example:

```
Notebook Notebook0 ruler=ruler1
```

You and Igor will get confused if you mix ruler keywords with other types of keywords in the same command. It is alright, however to put a selection keyword at the start of the command. Mixing will not cause a crash or any drastic problem but it will likely produce results that you don't understand.

To keep things clear, follow these rules:

- If you use ruler=*rulerName* or newRuler=*rulerName*, put them before any other ruler keywords.
- Do not mix ruler keywords with other kinds, except that it is alright to use the selection keyword at the start of the command.

justification=*j* Sets text justification:

j=0:	Left aligned.
j=1:	Center aligned.
j=2:	Right aligned.
j=3:	Fully justified.

margins={*indent*,*left*,*right*}

indent sets the indentation of first line from left page margin.

left sets the paragraph's left margin in points measured from the left page margin.

right sets the paragraph's right margin in points measured from the left page margin.

newRuler=*rulerName*

Creates a new ruler with the specified name. If a ruler with this name already exists, it is overwritten.

ruler=*rulerName* Applies the named ruler to the selected text or to redefine the named ruler, as explained above.

rulerDefaults={*"fontName"*, *fSize*, *fStyle*, (*r*,*g*,*b*[,*a*])

}

"fontName" sets the ruler's text font, e.g., `"Helvetica"`.

fSize sets the ruler's text size.

fStyle sets the ruler's text style.

(*r*,*g*,*b*[,*a*]) sets the ruler's text color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque white.

You can use rulerDefaults only if you are redefining an existing ruler using ruler=*rulerName* or you are creating a new ruler using newRuler=*rulerName*.

spacing={*spaceBefore*,*spaceAfter*,*lineSpace*}

spaceBefore sets the extra space before paragraph in points.

spaceAfter sets the extra space after paragraph in points.

lineSpace sets the extra space between lines of a paragraph in points.

tabs={*tabSpec*} *tabSpec* is list of tab stops in points added to special values that change the tab stop type.

Tab stops have two parts: the tab stop position and the tab type. Each integer in the list of tabs encodes both of these parts as follows:

The low 11 bits contains the tab stop position in points.

The next two bits are reserved for future use and must be zero.

The high three bits are used to contain the tab type as follows:

left tab	0	
center tab	1	add 1*8192 to tab stop position.
right tab	2	add 2*8192 to tab stop position.
decimal tab	3	add 3*8192 to tab stop position.
comma tab	4	add 4*8192 to tab stop position.

Tabs Example

The following puts a left tab at 1 inch, a center tab at 3 inches and a decimal tab at 5 inches:

```
Notebook Notebook1 tabs={1*72, 3*72 + 8192, 5*72 + 3*8192}
```

Notebook (*Selection*)

Notebook selection parameters

This section of Notebook relates to selecting a range of the content of the notebook.

findPicture={*graphicNameStr*, *flags*}

Searches for the picture containing the named graphic (*Macintosh only*) or the next picture if you pass `""`. Sets V_flag to 1 if the picture was found or to 0 if not found.

flags is a bitwise parameter interpreted as follows:

Bit 0: Show selection after the find.

All other bits are reserved for future use. Set bit 0 by setting *flags* = 1.

The search is always forward from the end of the current selection to the end of the document.

findSpecialCharacter={*specialCharacterNameStr*, *flags*}

Searches for the special character with the specified name or the next special character if you pass `""`. Selects the special character if it is found.

Sets V_flag to 1 if the special character was found or to 0 if not. Sets S_name to the name of the found special character or to `""` if it was not found.

flags is a bitwise parameter interpreted as follows:

Bit 0: Show selection after the find.

All other bits are reserved for future use. Set bit 0 by setting *flags* = 1.

If *specialCharacterNameStr* is empty (`""`), the search proceeds from the end of the current selection to the end of the document. Otherwise the search always covers the entire document.

findText={*textToFindStr*, *flags*}

Searches for the specified text. Sets V_flag to 1 if the text was found or to 0 if not found.

textToFindStr is a string expression for the text you want to find. If the text contains a carriage return, Igor considers only the part of the text before the carriage return.

flags is a bitwise parameter interpreted as follows:

Bit 0:	Show selection after the find.
Bit 1:	Do case-sensitive search.
Bit 2:	Search for whole words.
Bit 3:	Wrap around.
Bit 4:	Search backward.

All other bits are reserved and must be set to zero.

To set bit 0 and bit 3, use $2^0+2^3 = 9$ for *flags*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

If you are searching forward, the search starts from the end of the current selection. If you are searching backward, the search starts from the start of the current selection.

If you specify `""` as the text to search for, it “finds” the current selection. This displays the current selection using `findText={"", 1}`.

selection={*selStart*, *selEnd*}

selStart and *selEnd* are locations within the document. You can specify these document locations by using the following expressions:

(*paragraph*, *pos*)	*paragraph* and *pos* are numeric expressions. *paragraph* is a paragraph number from 0 to *n*-1 where *n* is the number of paragraphs in the document. *pos* is a byte position from 0 to *n* where *n* is the number of bytes in the paragraph. Position 0 is to the left of the first character in the paragraph. Position *n* is to the right of the last character in the paragraph.
startOfFile	Start of the document.
endOfFile	End of the document.
startOfParagraph	Start of current *selStart* paragraph.
endOfParagraph	End of current *selStart* paragraph.
startOfNextParagraph	Start of paragraph after current *selStart* paragraph.
endOfNextParagraph	End of paragraph after current *selStart* paragraph.
startOfPrevParagraph	Start of paragraph before current *selStart* paragraph.
endOfPrevParagraph	End of paragraph before current *selStart* paragraph.
endOfChars	Just before the carriage return of current *selStart* paragraph.
startOfPrevChar	Start of the character before the character at the current selection start or selection end. This moves the selection start or selection end like pressing the left arrow key. Added in Igor Pro 7.00.
startOfNextChar	Start of the character after the character at the current selection start or selection end. This moves the selection like pressing the right arrow key. Added in Igor Pro 7.00.

Igor clips the specified locations to legal values. It also sets the V_flag variable to 0 if the *selStart* location that you specified was valid, to 1 if the start paragraph was out of bounds and to 2 if the start position was out of bounds. You can use the startOfNextParagraph keyword to step through the document one paragraph at a time. When V_flag is nonzero, you are at the end of the document.

The terms next and prev are relative to the paragraph containing the start of the selected text before the selection keyword was invoked.

The selection keyword just sets the selection. If you also want to scroll the selected text into view you must also use the findText keyword as shown in the examples.

Selection Examples

Following are some examples of setting the selection:

```
// select all text in notebook
Notebook Notebook1 selection={startOfFile, endOfFile}

// move selection to the start of the notebook and display the selection
Notebook Notebook1 selection={startOfFile,startOfFile}, findText={"",1}

// move selection to the end of the notebook and display the selection
Notebook Notebook1 selection={endOfFile,endOfFile}, findText={"",1}

// select all of paragraph 3
Notebook Notebook1 selection={(3,0), (4,0)}

// select all of paragraph 3 and display the selection
Notebook Notebook1 selection={(3,0), (4,0)}, findText={"",1}

// select all of current paragraph except for trailing CR, if any
Notebook Notebook1 selection={startOfParagraph, endOfChars}
```

```
// select the first occurrence of "Hello" in the document and display the selection
Notebook Notebook1 selection={startOfFile,startOfFile}, findText={"Hello",1}

// select the first picture in the document
Notebook Notebook1 selection={startOfFile,startOfFile}, findPicture={"",1}
```

See Also

The **GetSelection** operation to “copy” the selection.

Notebook (*Text Properties*)

Notebook text property parameters

This section of Notebook relates to setting the text properties of the current selection in the notebook.

font="*fontName*" — *"fontName"* is the name of the font. Use `"default"` to specify the paragraph's ruler font.

If you specify an unavailable font, it does nothing. This is so that, when you share procedures with a colleague, using a font that the colleague does not have will not cause your procedures to fail. The downside of this behavior is that if you misspell a font name you will get no error message.

fSize=*fontSize* — Text size from 3 to 32000 points.

Use -1 to specify the paragraph's ruler size.

fStyle=*fontStyle* — A binary coded integer with each bit controlling one aspect of the text style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

Use -1 to specify the paragraph's ruler style. To set bit 0 and bit 1 (bold italic), use $2^0+2^1 = 3$ for *fontStyle*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

syntaxColorSelection=*n*

Use *n*=1 to syntax-color the selected text in the notebook. This is the equivalent of selecting Notebook→Syntax Color Selection. Other values of *n* are reserved for future use.

To remove syntax coloring, use the textRGB keyword to set the selected text to a specified color.

The syntaxColorSelection keyword was added in Igor Pro 9.00.

textRGB=(*r*,*g*,*b*[,*a*]) — Specifies text color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

vOffset=*v* — Sets the vertical offset in points (positive offset is down, negative is up). Use this to create subscripts and superscripts. vOffset is allowed for formatted text files only, not for plain text files.

Notebook (*Writing Graphics*)

Writing notebook graphics parameters

This section of Notebook relates to inserting graphics at the current selection in the notebook.

These graphics keywords are allowed for formatted text files only, not for plain text files.

convertToPNG=*x* — Converts all pictures in the current selection to cross-platform PNG format. If the picture is already PNG, it does nothing.

x is the resolution expansion factor, an integer from 1 to 16 times screen resolution. *x* is clipped to legal limits.

frame=*f* Sets the frame used for the picture and insertPicture keywords.

f=0:	No frame (default).
f=1:	Single frame.
f=2:	Double frame.
f=3:	Triple frame.
f=4:	Shadow frame.

insertPicture={*pictureName*, *pathName*, *filePath*, *options*}

Inserts a picture from a file specified by *pathName* and *filePath*. The supported graphics file formats are listed under **Inserting Pictures** on page III-13.

pictureName is the special character name (see **Special Character Names** on page III-14) to use for the inserted notebook picture or $"" to automatically assign a name.

pathName is the name of an Igor symbolic path created via **NewPath** or $"" to use no path.

filePath is a full path to the file to be loaded or a partial path or simple file name relative to the specified symbolic path.

If *pathName* and *filePath* do not fully specify a file, an Open File dialog is displayed from which the user can choose the file to be inserted.

options is a bitwise parameter interpreted as follows:

Bit 0:	If set, an Open File dialog is displayed even if the file is fully specified by *pathName* and *filePath*.
Bit 1:	Determines what to do in the event of a name conflict. If set, the existing special character with the conflicting name is overwritten. If cleared, a unique name is created and used as the special character name for the inserted picture.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The variable V_flag is set to 1 if the picture was inserted or to 0 otherwise, for example, if the user canceled from the Open File dialog.

The string variable S_name is set to the special character name of the picture that was inserted or to "" if no picture was inserted.

The string variable S_fileName is set to the full path of the file that was inserted or to "" if no picture was inserted.

picture={*objectSpec*, *mode*, *flags* [, *expansion*]}

Inserts a picture based on the specified object.

objectSpec is usually just an object name, which is the name of a graph, table, page layout, Gizmo plot, or picture from Igor's picture gallery (Misc→Pictures). See further discussion below.

mode controls what happens when you insert a picture of a graph, table or page layout window. It does not affect insertions of pictures from the picture gallery.

mode specifies the format of the picture as follows:

mode	**Macintosh**	**Windows**
-9	SVG	SVG
-8	Igor PDF	Igor PDF
-7	TIFF	TIFF
-6	JPEG	JPEG
-5	PNG	PNG
-4	4X PNG	Device-independent bitmap
-2	8X PDF	8X Enhanced metafile
-1	8X PDF	8X Enhanced metafile
0	8X PDF	8X Enhanced metafile
1	1X PDF	8X Enhanced metafile
2	2X PDF	8X Enhanced metafile
4	4X PDF	8X Enhanced metafile
8	8X PDF	8X Enhanced metafile

Modes -6, -7, -8, and -9 require Igor Pro 7.00 or later.

In Igor 7 and 8, mode=-8 produced PDF on Macintosh and EMF on Windows. As of Igor Pro 9.00, it produces PDF on both Macintosh and Windows. If you were using -8 for EMF on Windows, change your code to use -2.

Mode -2 (PDF on Macintosh, EMF on Windows) is recommended for platform-specific graphics. Mode -5 (PNG) is recommended for platform-independent bitmap graphics. Mode -8 (PDF) is recommended for platform-independent vector graphics but see the note above about -8 on Windows.

If *objectSpec* names a Gizmo window, only modes -5, -6, or -7 are allowed.

Modes -2 through 8 are supported for backward compatibility. In previous versions of Igor, they selected other formats that are now obsolete.

See Chapter III-5, **Exporting Graphics (Macintosh)**, Chapter III-6, **Exporting Graphics (Windows)**, and **Metafile Formats** on page III-102 for further discussion of these formats.

flags is a bitwise parameter interpreted as follows:

Bit 0: 0 for black and white, 1 for color.

All other bits are reserved and must be set to zero.

For color, set *flags* = 2^0 = 1.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

expansion is optional and requires Igor Pro 7.00 or later. It affects only modes -5, -6, and -7.

expansion sets the expansion factor over screen resolution. *expansion* must be an integer between 1 and 8 and is usually 1, 2, 4 or 8. The default value is 1.

scaling={*h*, *v*} Sets the horizontal(*h*) and vertical (*v*) scaling for the selected picture or the picture and insertPicture keywords. *h* and *v* are in percent.

When using the picture keyword, you may include a coordinate specification after the object name in *objectSpec*. For example:

```
Notebook Notebook1 picture={Layout0(100, 50, 500, 700), 1, 1}
```

The coordinates are in points. A coordinate specification of (0, 0, 0, 0) behaves the same as no coordinate specification at all.

If the object is a graph, the coordinate specification determines the width and height of the graph. If you omit the coordinate specification, Igor takes the width and height from the graph window.

If the object is a layout, the coordinate specification identifies a section of the layout. If you omit the coordinate specification, Igor selects a section of the layout that includes all objects in the layout plus a small margin.

For any other kind of object, Igor ignores the coordinate specification if it is present.

The scaling and frame keywords affect the selected picture, if any. If no picture is selected, they affect the insertion of a picture using the picture or insertPicture keywords. For example, this command inserts a picture of Graph0 with 50% scaling and a double frame:

```
Notebook Test1 scaling={50, 50}, frame=2, picture={Graph0, 1, 1}
```

If no picture is selected and no picture is inserted, scaling and frame have no effect.

InsertPicture Example

```
Function InsertPictureFromFile(nb)
	String nb			// Notebook name or "" for top notebook

	if (strlen(nb) == 0)
		nb = WinName(0, 16, 1)
	endif

	if (strlen(nb) == 0)
		Abort "There are no notebooks"
	endif

	// Display Open File dialog to get the file to be inserted
	Variable refNum		// Required for Open but not really used
	String fileFilter = "Graphics Files:.eps,.jpg,.png;All Files:.*;"
	Open /D /R /F=fileFilter refNum
	String filePath = S_fileName
	if (strlen(filePath) == 0)
		Print "You cancelled"
		return -1
	endif

	Notebook $nb, insertPicture={$"", $"", filePath, 0}
	if (V_flag)
		Print "Picture inserted"
	else
		Print "No picture inserted"
	endif

	return 0
End
```

Save notebook pictures to files

The savePicture keyword is allowed for formatted text files only, not for plain text files.

savePicture={*pictureName*, *pathName*, *filePath*, *options*}

Saves a picture from a formatted text notebook to a file specified by *pathName* and *filePath*.

pictureName is the special character name (see **Special Character Names** on page III-14) of the picture to be saved or $"" to save the selected picture in which case one picture and one picture only must be selected in the notebook.

pathName is the name of an Igor symbolic path created via **NewPath** or $"" to use no path.

filePath is a full path to the file to be written or a partial path or simple file name relative to the specified symbolic path.

If *pathName* and *filePath* do not fully specify a file, a Save File dialog is displayed in which the user can specify the file to be written.

options is a bitwise parameter interpreted as follows:

Bit 0: If set, a Save File dialog is displayed even if the file is fully specified by *pathName* and filePath.

Bit 1: If set, a file with the same name is overwritten if it exists. If cleared, a Save File dialog is displayed if the specified file already exists.

Bit 2: If set then the leaf name specified by *filePath* is ignored and a name is automatically generated based on the picture name.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The variable V_flag is set to 1 if the picture was written or to 0 otherwise, for example, if the user canceled from the Save File dialog.

The string variable S_name is set to the special character name of the picture that was saved or to "" if no picture was saved.

The string variable S_fileName is set to the full path of the file that was written or to "" if no picture was written.

Notebook (*Writing Special Characters*)

Writing special character parameters

This section of Notebook relates to inserting special characters at the current selection in the notebook. To insert a notebook action, see **NotebookAction**.

The special characters are page break, short date, long date, abbreviated date and time. They act in some respects like a single character but have special properties. You can insert the special characters using the specialChar keyword.

The specialChar keyword is allowed for formatted text files only, not for plain text files.

Other special characters are allowed in headers and footers only and you can not insert them in a document using the specialChar keyword. These are window title, page number and total pages.

The special characters other than page break character are dynamic and update periodically.

specialChar={*type*, *flags*, *optionsStr*}

type is the special character type as follows:

1:	Page break.
2:	Short date.
3:	Long date.
4:	Abbreviated date.
5:	Time.

flags is reserved for future use. You should pass 0 for *flags*.

optionsStr is reserved for future use. You should pass `""` for *optionsStr*.

specialUpdate=*flags*

Updates special characters in the notebook.

flags is interpreted bitwise:

Bit 0: 0 to update all special characters.

Bit 1: 1 to update special characters in the selected text.

If 1, updates regardless of whether updating is enabled or not.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The specialUpdate keyword can update pictures of graphs, tables, and page layouts that were created from windows in the current experiment.

See Also

Chapter III-1, **Notebooks**. The **NewNotebook**, **NotebookAction**, and **OpenNotebook** operations; the **SpecialCharacterInfo** and **SpecialCharacterList** functions.

Notebook (*Accessing Contents*)

Accessing Notebook Contents

getData=*mode* Causes Igor to return the contents of the notebook in the S_value variable. The contents are binary data in a private Igor format encoded as text. The only use for this keyword is to transfer data from one notebook to another by calling getData followed by setData.

Causes Igor to return the contents of the notebook in the S_value variable. The contents are binary data in a private Igor format encoded as text. The only use for this keyword is to transfer data from one notebook to another by calling getData followed by setData.

mode=1: Stores in S_value plain text or formatted text data, depending on the type of the notebook, from the entire notebook.

mode=2: Stores in S_value plain text data, regardless of the type of the notebook, from the entire notebook.

mode=3: Stores in S_value plain text or formatted text data, depending on the type of the notebook, from the notebook selection only.

mode=4: Stores in S_value plain text data, regardless of the type of the notebook, from the notebook selection only.

See the Notebook In Panel example experiment for examples using getData and setData.

Notebook (*Writing Text*)

Writing notebook text parameters

This section of Notebook relates to inserting text at the current selection in the notebook.

text=*textStr* Inserts the text at the current selection.

Before the text is inserted, Igor converts escape sequences in *textStr* as described in **Escape Sequences in Strings** on page IV-14.

Then, it checks for illegal characters. The only character code that is illegal is zero (ASCII NUL character). If it finds an illegal character, Igor generates an error and does not insert the text.

setData=*dataStr* Inserts the data at the current selection.

dataStr is either a regular string expression or the result returned by Notebook getData.

zData=*dataStr* This keyword is used by Igor during the recreation of a notebook subwindow in a control panel. *dataStr* is encoded binary data created by Igor when the recreation macro was generated. It represents the contents of the notebook subwindow in a format private to Igor.

zDataEnd=1 This keyword is used by Igor during the recreation of a notebook subwindow in a control panel. It marks the end of encoded binary data created by Igor when the recreation macro was generated.

NotebookAction

NotebookAction [**/W=*winName***] ***keyword = value*** [, ***keyword = value*** …]

The NotebookAction operation creates or modifies an “action” in a notebook. A notebook action is an object that executes commands when clicked.

See Chapter III-1, **Notebooks**, for general information about notebooks.

NotebookAction returns an error if the notebook is open for read-only. See **Notebook Read/Write Properties** on page III-10 for further information.

Parameters

The parameters are in *keyword =value* format. Parameters are automatically limited to legal values before being applied to the notebook.

bgRGB=(*r*, *g*, *b*) Specifies the action background color. *r*, *g*, and *b* specify the amount of red, green, and blue as integers from 0 to 65535.

commands=*str* Specifies the command string to be executed when clicking the action. For multiline commands, add a carriage return (`\r`) between lines.

enableBGRGB=*enable*

Uses the background color specified by bgRGB (*enable*=1). Background color is ignored for *enable*=0.

frame=*f* Specifies the frame enclosing the action.

f=0: No frame.

f=1: Single frame (default).

f=2: Double frame.

f=3: Triple frame.

f=4: Shadow frame.

helpText=*helpTextStr*

Specifies the help string for the action. The text is limited to 255 bytes. On Macintosh, help appears when the cursor is over the action after choosing Help→Show Igor Tips. On Windows, help appears in the status line when the cursor is over the action.

ignoreErrors=*ignore*

Controls whether an error dialog will appear (*ignore*=0) or not (*ignore* is nonzero) if an error occurs while executing the action commands.

linkStyle=*linkStyle* Controls the action title text style. If *linkStyle*=1, the style is the same as a help link (blue underlined). If *linkStyle*=0, the style properties are the same as the preceding text.

name=*name* Specifies the name of the new or modified notebook action. This is a standard Igor name. See **Standard Object Names** on page III-501 for details.

padding={*leftPadding*, *rightPadding*, *topPadding*, *bottomPadding*, *internalPadding*}

Sets the padding in points. *internalPadding* sets the padding between the title and the picture when both elements are present.

picture=*name* Specifies a picture for the action icon. *name* is the name of a picture in the picture gallery (see **Pictures** on page III-509).

If *name* is null (`$""`), it clears the picture parameter.

procPICTName=*name*

Specifies a Proc Picture for the action icon (see **Proc Pictures** on page IV-56). *name* is the name of a Proc Picture or null (`$""`) to clear it. This will be a name like `ProcGlobal#myPictName` or `MyModuleName#myPictName`. If you use a module name, the Proc Picture must be declared static.

If you specify both picture and procPICTName, picture will be used.

quiet=*quiet* Displays action commands in the history area (*quiet*=0), otherwise (*quiet*=1) no commands will be recorded.

scaling={*h*, *v*} Scales the picture in percent horizontally, *h*, and vertically, *v*.

showMode=*mode* Determines if the title or picture are displayed.

mode=1:	Title only.
mode=2:	Picture only.
mode=3:	Picture below title.
mode=4:	Picture above title.
mode=5:	Picture to left of title.
mode=6:	Picture to right of title.

Without a picture specification, the action will use title mode regardless of what you specify.

title=*titleStr* Sets the action title to *titleStr*, which is limited to 255 bytes.

Flags

/W= *winName* Specifies the notebook window of interest.

winName is either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-92 for details on host-child specifications.

If /W is omitted, NotebookAction acts on the top notebook window.

Examples

```
String nb = WinName(0, 16, 1)          // Top visible notebook
```

```
NotebookAction name=Action0, title="Beep", commands="Beep"// Create action
NotebookAction name=Action0, enableBGRGB=1, padding={4,4,4,4,4}
```

```
Notebook $nb, findSpecialCharacter={"Action0",1}         // Select action
Notebook $nb, frame=1                                    // Set frame
```

See Also

Chapter III-1, **Notebooks**.

The **Notebook**, **NewNotebook**, and **OpenNotebook** operations; the **SpecialCharacterInfo** and **SpecialCharacterList** functions.

num2char

`num2char(`*`num`*` [, `*`options`*`)`

The num2char function returns a string containing a character.

The *options* parameter was added in Igor Pro 7.00 and defaults to 0.

As of Igor7, Igor represents text internally as UTF-8, a form of Unicode. Previously it represented text as system text encoding. Because of this change, the behavior of num2char is complicated.

Recommended use of num2char in Igor7 or later

If *num* is a Unicode code point, pass 0 for *options* and num2char will return a UTF-8 string containing the character for the Unicode code point represented by *num*.

If you want a string containing a single byte, even though it may not be a valid UTF-8 string, pass 1 for *options* and num2char will return a string containing the single byte whose value is *num*, provided that *num* is between 0 and 255.

Detailed description of num2char in Igor7 or later

If *num* is between 0 and 127, num2char returns a string containing a single byte whose value is *num*. This represents an ASCII character.

If *num* is between 128 and 255 and *options* is 1, num2char returns a string containing a single byte whose value is *num*. This is not valid UTF-8 text, but it is consistent with the behavior of num2char in Igor6.

If *num* is between 128 and 255 and *options* is 0 or omitted, num2char returns the UTF-8 representation of the character for the Unicode code point represented by *num*.

If *num* is greater than 255, num2char returns the UTF-8 representation of the character for the Unicode code point represented by *num* regardless of the value of *options* .

If you provide the *options* parameter, it must be either 0 or 1. Other values may be used for other purposes in the future.

Examples

```
Print num2char(65)            // Prints A
Print num2char(97)            // Prints a
Print num2char(0xF7)          // Prints division sign
Print num2char(0xF7,0)        // Prints division sign
Print num2char(0xF7,1)        // Prints missing character symbol
Print num2char(0x0127)        // Prints small letter h with stroke (h-bar)
Print num2char(0x0127,0)      // Prints small letter h with stroke (h-bar)
Print num2char(0x0127,1)      // Prints small letter h with stroke (h-bar)

// In the case of num2char(0xF7,1),num2char returns a string containing
// a single byte whose value is 0xF7. This is not a valid UTF-8 string.
```

See Also

The **char2num**, **str2num** and **num2str** functions.

Text Encodings on page III-459.

num2istr

`num2istr(`*`num`*`)`

The num2istr function returns a string representing *num* after rounding to the nearest integer.

num2str

`num2str(`*`num`*` [, `*`formatStr`*`])`

The num2str function returns a string representing the number *num*.

The optional *formatStr* parameter was added in Igor Pro 9.00.

If you omit *formatStr*, the precision of the output string is limited to only five decimal places. This can cause unexpected and confusing results. For this reason, we recommend that you specify a larger precision with *formatStr* or use **num2istr** or **sprintf** for more control of the format and precision of the number conversion.

Parameters

num The value to be converted to string representation.

formatStr Controls the format of the output string. See **printf** for details on format strings.

formatStr is optional and requires Igor Pro 9.00 or later.

If you omit formatStr, the format used is "%.5g". For more precision use something like "%.15g".

See Also

printf, **sprintf**, **str2num**, **char2num**, **num2char**

NumberByKey

`NumberByKey(`*`keyStr, kwListStr`*` [, `*`keySepStr`*` [, `*`listSepStr`*` [, `*`matchCase`*`]]])`

The NumberByKey function returns a numeric value extracted from *kwListStr* based on the specified key contained in *keyStr*. *kwListStr* should contain keyword-value pairs such as "`KEY=value1,KEY2=value2`" or "`Key:value1;KEY2:value2`", depending on the values for *keySepStr* and *listSepStr*.

Use NumberByKey to extract a numeric value from a strings containing "`key1=value1;key2=value2;`" style lists such as those returned by functions like **AxisInfo** or **TraceInfo**.

If the key is not found or if any of the arguments is **""** or if the conversion to a number fails then it returns NaN.

keySepStr, *listSepStr,* and *matchCase* are optional; their defaults are ":", ";", and 0 respectively.

Details

kwListStr is searched for an instance of the key string bound by *listSepStr* on the left and a *keySepStr* on the right. The text up to the next *listSepStr* is converted to the returned number.

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are usually case-insensitive. Setting the optional *matchCase* parameter to 1 makes the comparisons case sensitive.

In Igor6, only the first byte of *keySepStr* and *listSepStr* was used. In Igor7 and later, all bytes are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *matchCase* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print NumberByKey("AKEY", "AKEY:123;")                    // prints 123
Print NumberByKey("BKEY", "AKEY=123;Bkey=456;", "=")      // prints 456
Print NumberByKey("KEY2", "KEY1=123,KEY2=999,", "=", ",")// prints 999
Print NumberByKey("ckey", "CKEY=123;ckey=456;", "=")      // prints 123
Print NumberByKey("ckey", "CKEY=123;ckey=456;", "=", ";", 1)// prints 456
```

See Also

The **StringByKey**, **RemoveByKey**, **ReplaceNumberByKey**, **ReplaceStringByKey**, **ItemsInList**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

numpnts

`numpnts(`*`waveName`*`)`

The numpnts function returns the total number of data points in the named wave. To find the number of elements in a dimension of a multidimensional wave, use the **DimSize** function.

Do not use numpnts to test if a wave reference is null as this causes a runtime error. Use **WaveExists**.

numtype

numtype(*num*)

The numtype function returns a number which indicates what kind of value *num* contains.

Details

If *num* is a real number, numtype returns a real number whose value is:

0:	If *num* contains a normal number.
1:	If *num* contains +/-INF.
2:	If *num* contains NaN.

If *num* is a complex number, numtype returns a complex number in which the real part is the number type of the real part of *num* and the imaginary part is the number type of the imaginary part of *num*.

NumVarOrDefault

NumVarOrDefault(*pathStr*, *defVal*)

The NumVarOrDefault function checks to see if the *pathStr* points to a numeric variable. If the numeric variable exists, NumVarOrDefault returns its value. If the numeric variable does not exist, it returns *defVal* instead.

Details

NumVarOrDefault initializes input values of macros so they can remember their state without needing global variables to be defined first. String variables use the corresponding numeric function, **StrVarOrDefault**.

Examples

```
Function DemoNumVarOrDefault()
    Variable nVal = NumVarOrDefault("root:Packages:MyPackage:gNVal",2)
    String sVal = StrVarOrDefault("root:Packages:MyPackage:gSVal","Hello")

    Print nval, sval

    // Store values in package data folder for next time

    // Create package data folder if it does not yet exist
    NewDataFolder/O root:Packages
    NewDataFolder/O root:Packages:MyPackage

    DFREF dfr = root:Packages:MyPackage     // Get reference to package data folder

    // Create or overwrite globals in package data folder
    Variable/G dfr:gNVal = nVal
    String/G dfr:gSVal = sVal

    NVAR gNVal = dfr:gNVal
    gNVal += 1

    SVAR gSVal = dfr:gSVal
    gSVal += "!"
End
```

NVAR

NVAR [**/C**][**/Z**][**/SDFR**=*dfr*] ***localName*** [= ***pathToVar***][, ***localName1*** [= ***pathToVar1***]]…

NVAR is a declaration that creates a local reference to a global numeric variable accessed in a user-defined function.

The NVAR declaration is required when you access a global numeric variable in a function. At compile time, the NVAR statement specifies the local name referencing a global numeric variable. At runtime, it makes the connection between the local name and the actual global variable. For this connection to be made, the global numeric variable must exist when the NVAR statement is executed.

When *localName* is the same as the global numeric variable name and you want to reference a global variable in the current data folder, you can omit *pathToVar*.

pathToVar can be a full literal path (e.g., root:FolderA:var0), a partial literal path (e.g., :FolderA:var0) or $ followed by string variable containing a computed path (see **Converting a String into a Reference Using $** on page IV-62).

You can also use a data folder reference or the /SDFR flag to specify the location of the numeric variable if it is not in the current data folder. See **Data Folder References** on page IV-78 and **The /SDFR Flag** on page IV-80 for details.

If the global variable may not exist at runtime, use the /Z flag and call **NVAR_Exists** before accessing the variable. The /Z flag prevents Igor from flagging a missing global variable as an error and dropping into the Igor debugger. For example:

```
NVAR/Z nv=<pathToPossiblyMissingNumericVariable>
if( NVAR_Exists(nv) )
   <do something with nv>
endif
```

Note that to create a global numeric variable, you use the **Variable**/G operation.

Flags

/C	Variable is complex.
/SDFR=*dfr*	Specifies the source data folder. See **The /SDFR Flag** on page IV-80 for details.
/Z	Ignores variable reference checking failures.

See Also

NVAR_Exists function.

Accessing Global Variables and Waves on page IV-65.

Converting a String into a Reference Using $ on page IV-62.

NVAR_Exists

NVAR_Exists(*name*)

The NVAR_Exists function returns one if specified NVAR reference is valid or zero if not. It can be used only in user-defined functions.

For example, in a user function you can test if a global numeric variable exists like this:

```
NVAR /Z var1 = gVar1          // /Z prevents debugger from flagging bad NVAR
if (!NVAR_Exists(var1))       // No such global numeric variable?
   Variable/G gVar1 = 0       // Create and initialize it
endif
```

See Also

WaveExists, **SVAR_Exists**, and **Accessing Global Variables and Waves** on page IV-65.

Open

Open [***flags***] ***refNum*** [**as** ***fileNameStr***]

The Open operation can, depending on the flags passed to it:

- Open an existing file to read data from (/R flag without /D).
- Open a to append results to (/A flag without /D).
- Create a new file or overwrite an existing file to write results to (no /D, /R or /A flags).
- Display an Open File dialog (/D/R or /D/A flags with or without /MULT).
- Display a Save File dialog (/D flag without /R or /A).

Parameters

refNum is the name of a numeric variable to receive the file reference number. *refNum* is set by Open if Open actually opens a file for reading or writing (cases 1, 2 and 3). You use *refNum* with the **FReadLine**, **FStatus**, **FGetPos**, **FSetPos**, **FBinWrite**, **FBinRead**, **fprintf**, and **wfprintf** operations to read from or write to the file. When you're finished, use pass *refNum* to the **Close** operation to close the file.

Open does not set the file reference number when the /D flag is used (cases 4 and 5) but you must still supply a *refNum* parameter.

The following discussion of the *pathName* and *fileNameStr* parameters applies when you are attempting to open a file for reading or writing (cases 2, 3, and 5 above).

The targeted file is specified by a combination of the *pathName* parameter and the *fileNameStr* parameter. There are three ways to specify the targeted file:

Method	How To Use It
Symbolic path and simple file name	Use /P=*pathName* and *fileNameStr*, where *pathName* is the name of an Igor symbolic path (see **Symbolic Paths** on page II-22) that points to the folder containing the file and *fileNameStr* is the name of the file.
Symbolic path and partial path	Use /P=*pathName* and *fileNameStrs*, where *pathName* is the name of an Igor symbolic path that points to the folder containing the file and *fileNameStr* is a partial path starting from the folder and leading to the file.
Full path	Use just *fileNameStr*, where *fileNameStr* is a full path to the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

The targeted file is fully specified if *fileNameStr* is a full path or if both *pathName* and *fileNameStr* are present and not empty strings.

The targeted file is not fully specified in any of these cases:

- *as fileNameStr* is omitted
- *fileNameStr* is an empty string
- *fileNameStr* is not a full path and no symbolic path is specified

Opening an Existing File For Reading Only

This covers cases 1 (/R without /D).

If the file is fully-specified but does not exist, an error is generated. If you want to detect and handle the error yourself, use the /Z flag.

If the file is not fully-specified, Open displays an Open File dialog.

If a file is opened, *refNum* is set to the file reference number.

Opening an Existing File For Appending

This covers cases 1 (/R without /D) and 2 (/A without /D).

If the file is fully-specified and exists, it is opened for read/write and the current file position is moved to the end of the file.

If the file is fully-specified but does not exist, the file is created and opened for read/write.

If the file is not fully-specified, Open displays an Open File dialog.

If a file is opened, *refNum* is set to the file reference number.

Opening a File For Write

This covers case 3 (no /R, /A or /D).

If the targeted file exists, it is overwritten.

If the targeted file does not exist and it is fully-specified and targets a valid path, a new file is created.

If the file is fully-specified and targets an invalid path, an error is generated. If you want to detect and handle the error yourself, use the /Z flag.

If the file is not fully-specified, Open displays a Save File dialog.

If a file is opened, *refNum* is set to the file reference number.

Displaying an Open File Dialog To Select a Single File

This covers cases 4 (/D with /R or /A).

Open does not actually open the file but just displays the Open File dialog.

If the user chooses a file in the Open File dialog, the S_fileName output string variable is set to a full path to the file. You can use this in subsequent commands. If the user cancels, S_fileName is set to "".

See the documentation for the /D, /F and /M flags and then read **Displaying an Open File Dialog** on page IV-148 for details.

refNum is left unchanged.

Displaying an Open File Dialog To Select Multiple Files

This covers cases 4 (/D with /R or /A) with the /MULT=1 flag.

Open does not actually open the file but just displays the Open File dialog.

If the user chooses one or more files in the Open File dialog, the S_fileName output string variable is set to a carriage-return-delimited list of full paths to one or more files. You can use this in subsequent commands. If the user cancels, S_fileName is set to "".

See the documentation for the /D, /F, /M and /MULT flags and then read **Displaying a Multi-Selection Open File Dialog** on page IV-149 for details.

refNum is left unchanged.

Displaying a Save File Dialog

This covers cases 5 (/D without /R or /A).

Open does not actually open the file but just displays the Save File dialog.

If the user chooses a file in the Save File dialog, the S_fileName output string variable is set to a full path to the file. You can use this in subsequent commands. If the user cancels, S_fileName is set to "".

See the documentation for the /D, /F and /M flags and then read **Displaying a Save File Dialog** on page IV-150 for details.

refNum is left unchanged.

Flags

/A — Opens an existing file for appending or, if the file does not exist, creates a new file and opens it for appending.

/C=*creatorStr* — Specifies the file creator code. This is meaningful on Macintosh only and is ignored on Windows. For opening an existing file, creator defaults to "????" which means "any creator". For creating a new file, *creatorStr* defaults to "IGR0" which is Igor's creator code.

/D[=*mode*] — Specifies dialog-only mode.

/D:	A dialog is always displayed.
/D=1:	Same as /D.
/D=2:	A dialog is displayed only if *pathName* and *fileNameStr* do not specify a valid file.

Use this mode to allow the user to choose a file to be opened by a subsequent operation, such as **LoadWave**.

With /D or /D=1, open presents a dialog from which the user can select a file but does not actually open the file. Instead, Open puts the full path to the file into the string variable S_fileName.

/D=2 does the same thing except that it skips the dialog if *pathName* and *fileNameStr* specify a valid file. In this case, if pathName and fileNameStr refer to an alias (Macintosh) or shortcut (Windows), the target of the alias or shortcut is returned.

If the user clicks the Cancel button, S_fileName is set to an empty string.

Use Open/D/R to bring up an Open File dialog. See **Displaying an Open File Dialog** on page IV-148 for details.

Use Open/D/R/MULT=1 to bring up an Open File dialog to select multiple files. See **Displaying a Multi-Selection Open File Dialog** on page IV-149 for details.

Use Open/D to bring up a Save File dialog. See **Displaying a Save File Dialog** on page IV-150 for details.

See **Using Open in a Utility Routine** on page IV-151 for an example using /D=2.

Do not use /Z with /D.

/F=*fileFilterStr* — /F provides control over the file filter menu in the Open File dialog. See **Open File Dialog File Filters** on page IV-149 and **Save File Dialog File Filters** on page IV-151 for details.

/M=*messageStr* — Prompt message text in the dialog used to select the file, if any.

/MULT=m — Use /D/R/MULT=1 to display a multi-selection Open File dialog.

/D/R/MULT=0 or just /D/R displays a single-selection Open File dialog.

/MULT=1 is allowed only if /D or /D=1 and /R are specified.

See **Displaying a Multi-Selection Open File Dialog** on page IV-149 for details.

/P=*pathName* — Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/R — The file is opened read only.

/T=typeStr — When creating a new file on Macintosh (/A and /R flag omitted), /T sets the Macintosh file type property for the file if it does not already exist. For example, /T="BINA" sets the Macintosh file type to 'BINA'. If /T is omitted the Macintosh file type will be 'TEXT'. Apple has deemphasized Macintosh file types in favor of file name extensions.

For new code, /F is recommended instead of /T.

When opening an existing file (/A or /R flag used), /T provides control over the file filter menu in the Open File dialog. See **Open File Dialog File Filters** on page IV-149 for details.

When creating a new file (/A and /R flag omitted), /T provides control over the file filter menu in the Save File dialog. See **Save File Dialog File Filters** on page IV-151 for details.

/Z[=*z*] — Prevents aborting of procedure execution if an error occurs, for example if the procedure tries to open a file that does not exist for reading. Use /Z if you want to handle this case in your procedures rather than having execution abort.

When using /Z, /Z=1, or /Z=2, V_flag is set to 0 if no error occurred or to a nonzero value if an error did occur.

Do not use /Z with /D.

/Z=0: Same as no /Z.

/Z=1: Suppresses normal error reporting. When used with /R, it opens the file if it exists. /Z alone has the same effect as /Z=1.

/Z=2: Suppresses normal error reporting. When used with /R, it opens the file if it exists or displays a dialog if it does not exist.

Details

When Open returns, if a file was actually opened, the *refNum* parameter will contain a file reference number that you can pass to other operations to read or write data. If the file was not opened because of an error or because the user canceled or because /D was used, *refNum* will be unchanged.

If you use /R (open for read), Open opens an existing file for reading only.

If you use /A, Open opens an existing file for appending. If the file does not exist, it is created and then opened for appending.

If both /R and /A are omitted then Open creates and opens a file. If the specified file does not already exist, Open creates it and opens it for writing. If the file does already exist then Open opens it and sets the current file position to the start of the file. The current file position determines where in the file data will be written. Thus, you will be overwriting existing data in the file.

Warning: If you open an existing file for writing (you do not use /R) then you will overwrite or truncate existing data in the file. To avoid this, open for read (use /R) or open for append (use /A).

Output Variables

The Open operation returns information in the following variables:

`V_flag`	Set only when the /Z flag is used. V_flag is set to zero if the file was opened, to -1 if Open displayed a dialog (because the file was not fully-specified) and the user canceled, and to some nonzero value if an error occurred.
`S_fileName`	Stores the full path to the file that was opened. If /MULT=1 is used, S_fileName is a carriage-return-separated list of full paths to one or more files. If an error occurred or if the user canceled, S_fileName is set to an empty string.

When using /D, the value of V_flag is undefined. Do not use /Z with /D. Use S_fileName to determine if the user selected a file or canceled.

Examples

This example function illustrates using Open to open a text file from which data will be read. The function takes two parameters: an Igor symbolic path name and a file name. If either of these parameters is an empty string, the Open operation will display a dialog allowing the user to choose the file. Otherwise, the Open operation will open the file without displaying a dialog.

```
Function DemoOpen(pathName, fileName)
   String pathName      // Name of symbolic path or "" for dialog.
   String fileName      // File name, partial path, full path or "" for dialog.
   Variable refNum
   String str

   // Open file for read.
   Open/R/Z=2/P=$pathName refNum as fileName

   // Store results from Open in a safe place.
   Variable err = V_flag
   String fullPath = S_fileName

   if (err == -1)
      Print "DemoOpen canceled by user."
      return -1
   endif

   if (err != 0)
      DoAlert 0, "Error in DemoOpen"
      return err
   endif

   Printf "Reading from file \"%s\". First line is:\r", fullPath
   FReadLine refNum, str       // Read first line into string variable
   Print str
   Close refNum
   return 0
End
```

See Also

Symbolic Paths on page II-22.

Close, **FBinRead**, **FBinWrite**, **FReadLine**

FGetPos, **FSetPos**, **FStatus**

fprintf, **wfprintf**

Displaying an Open File Dialog on page IV-148, **Displaying a Multi-Selection Open File Dialog** on page IV-149, **Open File Dialog File Filters** on page IV-149

Displaying a Save File Dialog on page IV-150, **Save File Dialog File Filters** on page IV-151

Using Open in a Utility Routine on page IV-151

OpenHelp

OpenHelp [*flags*] *fileNameStr*

The OpenHelp operation opens the specified help file.

The OpenHelp operation was added in Igor Pro 7.00.

Parameters

The help file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If OpenHelp can not determine the location of the file from *fileNameStr* and *pathName*, it returns an error.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/INT[=*interactive*] Controls whether opening the help file is interactive or not.

/INT=1: If the help file being opened needs to be compiled, OpenHelp presents a dialog asking the user whether the file should be compiled. During the compile, a progress dialog is displayed. Any errors are presented to the user in an error dialog. This is the default behavior if /INT is omitted.

/INT=0: If the help file being opened needs to be compiled, OpenHelp compiles it without presenting a dialog. Compilation errors are not presented to the user but are reflected in the V_Flag output variable.

/P=*pathName* Specifies the folder to look in for the file. *pathName* is the name of an existing Igor symbolic path.

/PICT Scans the compiled help file for pictures and stores information about all pictures in a semicolon separated list into the S_pictureInfo output string. If the help file needs to be compiled but compilation fails, S_pictureInfo is set to "".

/V=*visible* Controls help window visibility.

visible=0: The help window will be initially hidden.

visible=1: The help window will be initially visible. This is the default if /V is omitted.

/W=(*left*,*top*,*right*,*bottom*)

Specifies window size and position. Coordinates are in points.

/Z[=z]	Controls error reporting. /Z=0: Report errors normally. /Z=0 is the same as omitting /Z altogether. This is the default behavior if /Z is omitted. /Z=1: Suppresses normal error reporting. /Z alone has the same effect as /Z=1. /Z=1 prevents aborting procedure execution if an error occurs, for example if the file does not exist or if there is a compilation error. Use /Z=1 if you want to handle errors in your procedures rather than having execution abort. When using /Z or /Z=1, check V_Flag to see if an error occurred.

Details

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "hd:Folder1:" or "C:\\Folder1\\". See **Symbolic Paths** on page II-22 for details.

If the specified file is already open but not as a help window (for example as a notebook), OpenHelp returns an error.

If the /W or /V flag is used, or both, the window size and position and visibility are set as specified even if the file itself is already open, so long as the file is already opened as a help window.

Output Variables

The OpenHelp operation returns information in the following variables:

`V_Flag`	Set to a non-zero value if an error occurred and to zero if no error occurred.
`V_alreadyOpen`	Set to 1 if the specified help file was already open as a help file or to zero otherwise.
`S_pictureInfo`	Scans the compiled help file for pictures and stores information about all pictures in a semicolon separated list into the S_pictureInfo output string. If the help file needs to be compiled but compilation fails, S_pictureInfo is set to "".

See Also

CloseHelp

OpenNotebook

`OpenNotebook` `[`***`flags`***`] [`***`fileNameStr`***`]`

The OpenNotebook operation opens a file for reading or writing as an Igor notebook.

Unlike the Open operation, OpenNotebook will not create a file if the specified file does not exist. To create a new notebook, use the **NewNotebook** operation.

Parameters

The file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/A	Moves the notebook's selection to the end of the notebook.
/ENCG=*textEncoding*	Specifies the text encoding of the plain text file to be opened as a notebook. This flag was added in Igor Pro 7.00. This is relevant for plain text notebooks only and is ignored for formatted notebooks because they can contain multiple text encodings. See **Plain Text File Text Encodings** on page III-466 and **Formatted Text Notebook File Text Encodings** on page III-472 for details. OpenNotebook uses the text encoding specified by /ENCG and the rules described under **Determining the Text Encoding for a Plain Text File** on page III-467 to determine the source text encoding for conversion to UTF-8. Passing 0 for *textEncoding* acts as if /ENCG were omitted. See **Text Encoding Names and Codes** on page III-490 for a list of accepted values for *textEncoding*.
/K=*k*	Specifies window behavior when the user attempts to close it. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. *k*=3: Hides the window. If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.
/M=*messageStr*	Prompt message text in the dialog used to find the file, if any.
/N=*winName*	Specifies the window name to be assigned to the new notebook. If omitted, it assigns a name like "Notebook0".
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/R	Opens the file as read only.
/T=*typeStr*	Specifies the type or types of files that can be opened.
/V=*visible*	Hides (*visible*= 0) or shows (*visible*= 1; default) the notebook.
/W=(*left*,*top*,*right*,*bottom*)	Specifies window size and position. Coordinates are in points.
/Z	Suppresses error generation. Use this to check if a file exists. If you use /Z, OpenNotebook sets the variable V_flag to 0 if the notebook was opened or to nonzero if there was an error, usually because the specified file does not exist.

Details

The /A (append) flag has no effect other than to move the selection to the end of the notebook after it is opened.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-22 for details.

The /T=*typeStr* flag affects only the dialog that OpenNotebook presents if you do not specify a path and filename. The dialog presents only those files whose type is specified by /T=*typeStr*. There are two file types that are allowed for notebooks: `'TEXT'` which is a plain text file and `'WMT0'` which is a WaveMetrics formatted text file. Therefore, the file type, if you use it, should be either "`TEXT`" or "`WMT0`". If /T=*typeStr* is missing, it defaults to "`TEXTWMT0`". This opens either type of notebook file. On Windows, Igor considers files with ".txt" extensions to be of type TEXT and considers files with ".ifn" to be of type WMT0. See **File Types and Extensions** on page III-455 for details.

See Also

The **Notebook** and **NewNotebook** operations, and Chapter III-1, **Notebooks**.

OpenProc

OpenProc [*flags*] [*fileNameStr*]

The OpenProc operation opens a file as an Igor procedure file.

Note: This operation is used automatically to open procedure files when you open an Igor experiment. You can invoke OpenProc only from the command line. Do not invoke it from a procedure. To open procedure files from a procedure or from a menu definition, use the Execute/P operation.

Parameters

The file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/A Moves the procedure window's selection to the end of the window.

/ENCG=*textEncoding*

Specifies the text encoding of the plain text file to be opened as a procedure file.

This flag was added in Igor Pro 7.00.

OpenProc uses the text encoding specified by /ENCG and the rules described under **Determining the Text Encoding for a Plain Text File** on page III-467 to determine the source text encoding for conversion to UTF-8.

Passing 0 for *textEncoding* acts as if /ENCG were omitted.

See **Text Encoding Names and Codes** on page III-490 for a list of accepted values for *textEncoding*.

/M=*messageStr* Prompt message text in the dialog used to find the file, if any.

/P=*pathName* Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/R The file is opened read only.

/T=*typeStr* Specifies the type or types of files that can be opened.

/V=*visible* Hides (*visible*= 0) or shows (*visible*= 1; default) the procedure window.

/Z Suppresses error generation if the specified file does not exist.

Details

The /A (append) flag has no effect other than to move the selection to the end of the procedure file after it is opened.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-22 for details.

OpenProc automatically opens procedure files when you open an Igor experiment. Normally, you will have no use for it. You can not open a procedure file while procedures are executing. Thus, you can't invoke OpenProc from within a procedure. You can only invoke it from the command line or from a user menu definition (actually, you may get away with it in a macro, but it's not recommend).

See Also

Chapter III-13, **Procedure Windows**, **Execute**, **CloseProc**

The **Execute** operation.

OperationList

`OperationList(`*`matchStr`*`, `*`separatorStr`*`, `*`optionsStr`*`)`

The OperationList function returns a string containing a list of internal (built-in) or external operation names corresponding to *matchstr*.

Parameters

Only operation names that match *matchStr* string are listed. Use `"*"` to match all names. See **WaveList** for examples.

separatorStr is appended to each operation name as the output string is generated. *separatorStr* is usually ";" for list processing (See **Processing Lists of Waves** on page IV-198 for details).

Use *optionsStr* to further qualify the list of operations. *optionsStr* is a (case-insensitive) string containing one of these values:

`"internal"`	Restricts the list to built-in operations.
`"external"`	Restricts the list to external operations (see **Igor Extensions** on page III-511).

Any other value for *optionsStr* (`"all"` is recommended) will return both internal and external operations.

See Also

The **DisplayProcedure** operation and the **FunctionList**, **MacroList**, **StringFromList**, and **WinList** functions.

Optimize

`Optimize` `[`***`flags`***`]` ***`funcSpec`***`,` ***`pWave`***

The Optimize operation determines extrema (minima or maxima) of a specified nonlinear function. The function must be defined in the form of an Igor user function.

Use the first form for univariate functions (one dimensional functions; functions taking just one variable). Use the second form with multivariate functions (functions in more than one dimension; functions of more than one variable).

Optimize uses Brent's method for univariate functions. For multivariate functions you can choose several variations of quasi-Newton methods or simulated annealing.

Flags

/A [=*findMax*]	Finds a maximum (/A=1 or /A) or minimum (/A=0 or no flag).
/D=*nDigits*	Specifies the number of good digits returned (default is 15) by the function being optimized. If you use /X=*xWave* with a single-precision wave, the default is seven. Ignored with simulated annealing (/M={3,0}).
/DSA=*destWave*	Sets a wave to track the current best model only found with simulated annealing (/M={3,0}). *destWave* must have the same number of points as the X vector.
/F=*trustRegion*	Sets the initial trust region when /M={1 or 2, …} with multivariate functions. The value is a scaled step size (see **Multivariate Details**). After the first iteration the trust region is adjusted according to conditions. Ignored with simulated annealing (/M={3,0}).
/H= *highBracket* /L= *lowBracket*	Find the extrema of a univariate function. *lowBracket* and *highBracket* are X values on either side of the extreme point. An extreme point is found between the bracketing values. If *lowBracket* and *highBracket* are equal, Optimize adds 1.0 to *highBracket* before looking for an extreme point. Default values for *lowBracket* and *highBracket* are zero. Thus, if neither *lowBracket* nor *highBracket* is present, this is the same as /L=0/H=1. Ignored with simulated annealing (/M={3,0}).

/I=*maxIters* Sets the maximum number of iterations in searching for an extreme point to *maxIters*. Default is 100 for *stepMethod* (/M flag) 0-2, 10000 for *stepMethod* = 3 (simulated annealing).

If you use this form of the /I flag with simulated annealing, *maxItersAtT* is set to *maxIters*/2 and *maxAcceptances* is set to *maxIters*/10.

/I={*maxIters*, *maxItersAtT*, *maxAcceptances*}

Specifies the number of iterations for simulated annealing. The maximum number of iterations is set by *maxIters*, *maxItersAtT* sets the maximum number of iterations at a given temperature in the cooling schedule, and *maxAcceptances* sets the total number of accepted changes in the X vector (whether they increase or decrease the function) at a given temperature.

If you use this form of the flag with any *stepMethod* (/M flag) other than 3, *maxItersAtT* and *maxAcceptances* are ignored.

Defaults for *stepMethod* = 3 are {10000, 5000, 500}.

/M={*stepMethod*, *hessMethod*}

Specifies the method used for selecting the next step (*stepMethod*) and the method for calculating the Hessian (matrix of second derivatives) with multivariate functions.

stepMethod	**Method**
0	Line Search
1	Dogleg
2	More-Hebdon
3	Simulated Annealing

hessMethod	**Method**
0	secant (BFGS)
1	finite differences

Default values are {0,0}. The *hessMethod* variable is ignored if you select *stepMethod* = 3.

/Q Suppresses printout of results in the history area. Ordinarily, the results of root searches are printed in the history.

/R={*typX1*, *typX2*, …}

/R=*typXWave* Specifies the expected size of X values with multivariate functions. These values are used to scale X values. If the X values you expect are very different from one, you will get more accurate results if you can give a reasonable estimate. Optimize will use *typXi* to scale X_i to reduce floating-point truncation error.

You must provide the same number of values in either a wave or a list of values as you provide to the /X flag.

Ignored with simulated annealing (/M={3,0}).

/S=*stepMax* Limits the largest scaled step size allowed with multivariate functions. Optimize will stop if five consecutive steps exceed *stepMax*.

Ignored with simulated annealing (/M={3,0}).

/SSA=*stepWave* Name of a 3-column wave having number of rows equal to the length of the X vector only when used with simulated annealing (/M={3,0}). *stepWave* sets information about the step size used to generate new X vectors. The step sizes are in terms of normalized X values. The normalization is such that the X_i ranges from -1 to 1 based on the ranges set by the /XSA flag.

Column zero sets the step size used when creating new trial X vectors. Default is 1.0.

Column one sets the minimum step size. Default is 0.001.

Column two sets the maximum step size. Default is 1.0.

/T=*tol*

Sets the stopping criterion with univariate functions. Optimize will attempt to find a minimum within ± *tol*.

When this form is used with a multivariate function, *gradTol* is set to *tol* and *stepTol* is set to *gradTol*^{2}.

Ignored with simulated annealing (/M={3,0}).

/T={*gradtol*, *stepTol*}

Sets the stopping criteria for multivariate functions. Iterations stop if a point is found with estimated scaled gradient less than *gradTol*, or if an iteration takes a scaled step shorter than *stepTol*. Default values are {8.53618x10^{-6}, 7.28664x10^{-11}}. These values are (6.022x10^{-16})$^{1/3}$ and (6.022x10^{-16})$^{2/3}$ as suggested by Dennis and Schnabel. 6.022x10^{-16} is the smallest double precision floating point number that, when added to 1, is different from 1.

Ignored with simulated annealing (/M={3,0}).

/TSA={*InitialTemp*, *CoolingRate*}

Used only with simulated annealing (/M={3,0}).

InitialTemp sets the initial temperature. If *InitialTemp* is set to zero, Optimize calls your function 100 times to estimate the best initial temperature. This is the recommended setting unless your function is very expensive to evaluate (in which case, you may not want to use simulated annealing at all).

CoolingRate sets the factor by which the temperature is decreased.

/X=*xWave*

/X={*x1*, *x2*, …}

Sets the starting point for searching for an extreme point with multivariate functions or with simulated annealing (/M={3,0}). The starting point can be specified with a wave having as many points as the number of independent variables, or you can write out a list of X values in braces. If you are finding extreme points of a univariate function, use /L and /H instead unless you are using the simulated annealing method. If you specify a wave, this wave is also used to receive the result of the extreme point search.

/XSA=*XLimitWave*

Name of a 2-column wave having number of rows equal to the length of the X vector only when used with simulated annealing (/M={3,0}).

Column zero sets the minimum value allowed for each element of the X vector.

Column one sets the maximum value allowed for each element of the X vector.

Default is ±X_i*10 if X_i is nonzero, or ±1 if X_i is zero. While a default is provided, it is highly recommended that you provide an *XLimitWave*.

/Y=*funcSize*

Specifies expected sizes of function values with multivariate functions. If you expect your function will return values very different from one, you should set *funcSize* to the expected size. Optimize will use this value to scale the function results to reduce floating-point truncation error.

Parameters

func specifies the name of your user-defined function that will be optimized.

pwave gives the name of a parameter wave that will be passed to your function as the first parameter. It is not modified by Igor. It is intended for your private use to pass adjustable constants to your function.

Function Format

Finding extreme points of a nonlinear function requires that you realize the function as a Igor user function of a certain form. See **Finding Minima and Maxima of Functions** on page III-343 for detailed examples.

Your function must look like this:

```
Function myFunc(w,x1, x2, …)
	Wave w
	Variable x1, x2
```

```
   return f(x1, x2, …)           // an expression …
End
```

A univariate function would have only one X variable.

A multivariate function can use a wave to pass in the X values:

```
Function myFunc(w,xw)
   Wave w
   Wave xw

   return f(xw)                  // an expression …
End
```

Replace "f(…)" with an appropriate numerical expression.

Univariate Details

The method used by Optimize to find extreme points of univariate functions requires that the point be bracketed before starting. If you don't use /L and /H to specify the bracketing X values, the defaults are zero and one. Optimize first attempts to find the requested extreme point using the bracketing values (or the default). If that is unsuccessful, it attempts to bracket an extreme point by expanding the bracketing interval. If a suitable interval is found (the search is by no means perfectly reliable), then the search for an extreme point is made again.

Optimize uses Brent's method for univariate functions, which requires no derivatives. This combines a quadratic extrapolation with checking for wild results. In the case of wild results (points beyond the best current bracketing values) the method reverts to a golden section bisection algorithm. For well-behaved functions, the quadratic extrapolation converges superlinearly. The golden section bisection algorithm converges more slowly but features global convergence, that is, if an extremum is there, it will be found.

The stopping criterion is

$$\left| x + \frac{a+b}{2} \right| + \frac{a-b}{2} \le \frac{2}{3} tol.$$

In this expression, *a* and *b* are the current bracketing values, and *x* is the best estimate of the extreme point within the bracketing interval.

The left side of this expression works out to being simply the distance from the current solution to the boundary of the bracketing interval.

Note: Optimizing a univariate function with the simulated annealing method (/M={3,0}) works like a multivariate function, and this section does not apply. See the sections devoted to simulated annealing.

Multivariate Details

With multivariate functions, Optimize scales certain quantities to reduce floating point truncation error. You enter scaling factors using the /R and /Y flags. The /R flag specifies the expected magnitude of X values; Optimize then uses X_i/typX_i in all calculations. Likewise, /Y specifies the expected magnitude of function values.

This scaling can be important for maintaining accuracy if your X's or Y's are very different from one, and especially if your X's have values spanning orders of magnitude.

The Optimize operation uses a quasi-Newton method with derivatives estimated numerically. The function gradient is calculated using finite differences. For estimation of the Hessian (second derivative matrix) you can use either a secant method (*hessMethod* = 0) or finite differences (*hessMethod* = 1). The finite difference method gives a more accurate estimate and may succeed with difficult functions but requires more function evaluations per iteration. The finite difference method's greater accuracy may reduce the total number of iterations required, so the overall number of function evaluations depends on details of the problem being solved. Usually the secant method requires fewer function evaluations and is preferred for functions that are expensive to evaluate.

Once a Newton step is calculated, there are three choices for the method used to find the best next value-line search along the Newton direction (*stepMethod* = 0), double dogleg (*stepMethod* = 1), or More-Hebdon (*stepMethod* = 2). The best method can be found only by experimentation. See Dennis and Schnabel (cited in **References**) for details.

The /F=*trustRegion*, /S=*stepMax* and /T={…, *stepTol*} all refer to scaled step sizes. That is,

$$stepX_i = \frac{\left|\Delta x_i\right|}{\max\left(\left|x_i\right|, typX_i\right)}.$$

The Optimize operation presumes that an extreme point has been found when either the gradient at the latest point is less than *gradTol* or when the last step taken was smaller than *stepTol*. These criteria both refer to scaled quantities:

$$\max_{1\le i\le n}\left\{\left|g_i\right|\frac{\max\left(\left|x_i\right|, typX_i\right)}{\max\left(\left|f\right|, funcSize\right)}\right\} \le gradTol,$$

or

$$\max_{1\le i\le n}\left\{\left|g_i\right|\frac{\left|\Delta x_i\right|}{\max\left(\left|x_i\right|, typX_i\right)}\right\} \le stepTol.$$

Simulated Annealing Introduction

The simulated annealing or Metropolis algorithm optimizes a function using a random search of the X vector space. It does not use derivatives to guide the search, making it a good choice if the function to be optimized is in some way poorly behaved. For instance, it is a good method for functions with discontinuities in the function value or in the derivatives.

Simulated annealing also has a good chance of finding a global minimum or maximum of a function having multiple local minima or maxima.

Because simulated annealing uses a random search method, it may require a large number of function evaluations to find a minimum, and it is not guaranteed that it will stop at an actual minimum. For these reasons, it is best to use one of the other methods unless those methods have failed.

The simulated annealing method generates new trial solutions by adding a random vector to the current X vector. The elements of the random vector are set to $stepsize_i$*R_i, where R_i is a random number in the interval (-1, 1). As the solution progresses, the *stepsize* is gradually decreased.

Bad trials, that is, those that change the function value in the wrong direction are accepted with a probability that depends on the simulated temperature. It is this aspect that allows simulated annealing to find a global minimum.

Function values are generated and accepted or rejected for some number of iterations at a given temperature, then the temperature is reduced. The probability of a bad iteration being accepted decreases with decreasing temperature. A too-fast cooling rate can freeze in a bad solution.

Simulated Annealing Details

It is highly recommended that you use the XSA flag to specify *XLimitWave*. This wave sets bounds on the values of the elements of the X vector during the random search. The defaults may be adequate but are totally *ad hoc*. You are better off to specify bounds that make sense to the problem you are solving.

The values of *XLimitWave* in addition to bounding the search space also scale the X vector during computations of probabilities, temperatures, etc. Consequently, the X limits can affect the performance of the algorithm.

A large number of iterations is required to have a good probability of finding a reasonable solution.

It is recommended that you set the initial temperature to zero so that Optimize can estimate a good initial temperature. If you can’t afford the 100 function evaluations required, you probably shouldn’t be using simulated annealing.

Optimize uses an exponential cooling schedule in which T_{i+1} = *CoolingRate**T_i (see the /TSA flag). *CoolingRate* must be in the range 0 to 1. A fast cooling rate (small value of *CoolingRate*) can cause simulated quenching; that is, a bad solution can be frozen in. Very slow cooling will result in slow convergence.

When simulated annealing is selected, the optimization is treated as multivariate even if your function has only a single X input. That is, the output variables and waves are the ones listed under multivariate functions.

Variables and Waves for Output

The Optimize operation reports success or failure via the V_flag variable. A nonzero value is an error code.

Variables for a univariate function:

`V_flag`
0: Search for an extreme point was successful.
57: User abort.
785: Function returned NaN.
786: Unable to find bracketing values for an extreme point.

If you searched for a minimum:

`V_minloc` X value at the minimum.

`V_min` Function value (Y) at the minimum.

If you searched for a maximum:

`V_maxloc` X value at the maximum.

`V_max` Function value (Y) at the maximum.

For simulated annealing only:

`V_SANumIncreases` Number of “bad” iterations accepted.

`V_SANumReductions` Number of iterations resulting in a better solution.

Variables for a multivariate function:

`V_flag`
0: Search for an extreme point was successful.
57: User abort.
788: Iteration limit was exceeded.
789: Maximum step size was exceeded in five consecutive iterations.
790: The number of points in the typical X size wave specified by /R does not match the number of X values specified by the /X flag
791: Gradient nearly zero and no iterations taken. This means the starting point is very nearly a critical point. It could be a solution, or it could be so close to a saddle point or a maximum (when searching for a minimum) that the gradient has no useful information. Try a slightly different starting point.

`V_OptTermCode` Indicates why Optimize stopped. This may be useful information even if V_flag is zero. Values are:
1: Gradient tolerance was satisfied.
2: Step size tolerance was satisfied.
3: No step was found that was better than the last iteration. This could be because the current step is a solution, or your function may be too nonlinear for Optimize to solve, or your tolerances may be too large (or too small), or finite difference gradients are not sufficiently accurate for this problem.
4: Iteration limit was exceeded.

5:	Maximum step size was exceeded in five consecutive iterations. This may mean that the maximum step size is too small, or that the function is unbounded in the search direction (that is, goes to -inf if you are searching for a minimum), or that the function approaches the solution asymptotically (function is bounded but doesn't have a well-defined extreme point).
6:	Same as V_flag = 791.

If you searched for a minimum:

`V_min`	Function value (Y) at the minimum.

If you searched for a maximum:

`V_max`	Function value (Y) at the maximum.

Variables for all functions:

`V_OptNumIters`	Number of iterations taken before Optimize terminated.
`V_OptNumFunctionCalls`	Number of times your function was called before Optimize terminated.

Waves for a multivariate function:

`W_extremum`	Solution if you didn't use /X=<*xWave*>. Otherwise the solution is returned in your X wave.
`W_OptGradient`	Estimated gradient of your function at the solution.

See Also

Finding Minima and Maxima of Functions on page III-343 for further details and examples.

References

The Optimize operation uses Brent's method for univariate functions. *Numerical Recipes* has an excellent discussion (see section 10.2) of this method (but we didn't use their code):

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

For multivariate functions Optimize uses code based on Dennis and Schnabel. To truly understand what Optimize does, read their book:

Dennis, J. E., Jr., and Robert B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Methods*, 378 pp., Society for Industrial and Applied Mathematics, Philadelphia, 1996.

Override

`Override constant` ***`objectName = newVal`***

`Override strconstant` ***`objectName = newVal`***

`Override Function` ***`funcName`*****`()`**

The Override keyword redefines a constant, strconstant, or user function. The *objectName* or *funcName* must be the same as the name of the original object or function that is being redefined. The override must be defined before the target object appears in the compile sequence.

See Also

Function Overrides on page IV-106 and **Constants** on page IV-51 for further details.

p

`p`

The p function returns the row number of the current row of the destination wave when used in a wave assignment statement. The row number is the same as the point number for a 1D wave.

Details

Outside of a wave assignment statement p acts like a normal variable. That is, you can assign a value to it and use it in an expression.

See Also

Waveform Arithmetic and Assignments on page II-74.

For other dimensions, the **q**, **r**, and **s** functions.

For scaled dimension indices, the **x**, **y**, **z**, and **t** functions.

p2rect

p2rect(*z*)

The p2rect function returns a complex value in rectangular coordinates derived from the complex value *z* which is assumed to be in polar coordinates (magnitude is stored in the real part and the angle, in radians, in the imaginary part of *z*).

Examples

Assume waveIn and waveOut are complex, then:

```
waveOut = p2rect(waveIn)
```

sets each point of waveOut to the rectangular coordinates based on the magnitude in the real part and the angle (in radians) in the imaginary part of the points in waveIn.

You may get unexpected results if the number of points in waveIn differs from the number of points in waveOut.

See Also

The functions **cmplx**, **conj**, **imag**, **r2polar**, and **real**.

PadString

PadString(*str*, *finalLength*, *padValue*)

The PadString function returns a string identical to *str* except that it has been extended to a total length of *finalLength* using bytes of *padValue*. Use zero to create a C-language style string or use 0x20 to pad with spaces (FORTRAN style). This is useful when reading or writing binary files using **FBinRead** and **FBinWrite**.

See Also

UnPadString, **ReplaceString**, **ReplicateString**

Panel

Panel

Panel is a procedure subtype keyword that identifies a macro as being a control panel recreation macro. It is automatically used when Igor creates a window recreation macro for a control panel. See **Procedure Subtypes** on page IV-204 and **Saving a Window as a Recreation Macro** on page II-47 for details.

PanelResolution

PanelResolution(*wName*)

The PanelResolution function returns the current resolution of the specified window in pixels per inch.

If *wName* is empty, it returns the current global setting for panel resolution in pixels per inch which is controlled by **SetIgorOption PanelResolution** (see page III-456).

If *wName* is the name of a graph window, it returns the resolution for the **ControlBar** area in pixels per inch.

wName can be a subwindow specification.

The PanelResolution function was added in Igor Pro 7.00.

In general, PanelResolution and **ScreenResolution** return the same thing. However, on Windows when the screen resolution is 96 DPI, which is typical for normal-resolution screens, panels can use 72 DPI for compatibility with Igor Pro 6 and earlier.

See Also

Control Panel Resolution on Windows on page III-456, **ScreenResolution**, **Control Panel Expansion** on page III-443

ParamIsDefault

`ParamIsDefault(`*`pName`*`)`

The ParamIsDefault function determines if an optional user function parameter *pName* was specified during the function call. It returns 1 when *pName* is default (not specified) or it returns 0 when it was specified.

Details

ParamIsDefault works only in the body of a user function and only with optional parameters. The variable *pName* must be valid at compile time; you can not defer lookup to runtime with $.

See Also

Optional Parameters on page IV-33 and **Using Optional Parameters** on page IV-60.

ParseFilePath

`ParseFilePath(`*`mode`*`, `*`pathInStr`*`, `*`separatorStr`*`, `*`whichEnd`*`, `*`whichElement`*`)`

The ParseFilePath function provides the ability to manipulate file paths and to extract sections of file paths.

Parameters

The meaning of the parameters depends on *mode*.

mode	**Information Returned**
0	Returns the element specified by *whichEnd* and *whichElement*. *whichEnd* is 0 to select an element relative to the beginning of *pathInStr*, 1 to select an element relative to the end. *whichElement* is zero-based. Pass ":" if *pathInStr* is a Macintosh HFS path, "\\" if it is a Windows path. See **Path Separators** on page III-451 for details about Macintosh versus Windows paths.
1	Returns the entire *pathInStr*, up to but not including the element specified by *whichEnd* and *whichElement*. *whichEnd* is 0 to select an element relative to the beginning of *pathInStr*, 1 to select an element relative to the end. *whichElement* is zero-based. Pass ":" if *pathInStr* is a Macintosh HFS path, "\\" if it is a Windows path. See **Path Separators** on page III-451 for details about Macintosh versus Windows paths.
2	Returns the entire *pathInStr* with a trailing separator added if it is not already there. This is useful when you have a path to a folder and want to tack on a file name. Pass ":" if *pathInStr* is a Macintosh HFS path, "\\" if it is a Windows path. See **Path Separators** on page III-451 for details about Macintosh versus Windows paths. *whichEnd* and *whichElement* are ignored. Pass 0 for them.
3	Returns the last element of *pathInStr* with the extension, if any, removed. The extension is anything after the last dot in *pathInStr*. *whichEnd* and *whichElement* are ignored. Pass 0 for them.
4	Returns the extension in *pathInStr* or `""` if there is no extension. The extension is anything after the last dot in *pathInStr*. Pass ":" if *pathInStr* is a Macintosh HFS path, "\\" if it is a Windows path. See **Path Separators** on page III-451 for details about Macintosh versus Windows paths. *whichEnd* and *whichElement* are ignored. Pass 0 for them.
5	Returns the entire *pathInStr* but converts it to a format determined by *separatorStr*.

mode	**Information Returned**
	separatorStr = ":" Converts the path to Macintosh HFS style if it is Windows style. Does nothing to a Macintosh HFS path. *separatorStr* = "\\" Converts the path to Windows style if it is Macintosh style. Does nothing to a Windows path. *separatorStr* = "*" Converts the path to the native style of the operating system Igor is running on. Does nothing to a native path. For historical reasons, on Macintosh “native” means colon-separated HFS path, not UNIX path. *separatorStr* = "/" Macintosh-only: Converts the Macintosh-style *pathInStr* input to a Posix (UNIX) path. Unlike the other conversions, the directory or file to which *pathInStr* refers must exist, otherwise "" is returned. To generate a Posix path for a non-existent file, generate the path for the existing folder and append the file name. This always returns "" on Windows. *whichEnd* and *whichElement* are ignored. Pass 0 for them.
6	UNC volume name ("\\Server\Share") if *pathIn* starts with a UNC volume name or "" if not. Pass "*" for *separatorStr*. *whichEnd* and *whichElement* are ignored. Pass 0 for them.
7	UNC server name ("Server" from "\\Server\Share") if *pathIn* starts with a UNC volume name or "" if not. Pass "*" for *separatorStr*. *whichEnd* and *whichElement* are ignored. Pass 0 for them.

mode	**Information Returned**
8	UNC share name ("Share" from "\\Server\Share") if *pathIn* starts with a UNC volume name or "" if not. Pass "*" for *separatorStr*. *whichEnd* and *whichElement* are ignored. Pass 0 for them.
9	Macintosh only. On Windows this mode returns an error. Returns a Posix version of *pathInStr* which must be a full HFS path pointing to an existing volume, directory or file. This is the same as mode 5 except that *separatorStr* must be "*". *whichEnd* and *whichElement* are ignored. Pass 0 for them. You would typically use this mode when you are about to execute a Unix command, which requires Posix paths, via ExecuteScriptText. This mode was created in Igor Pro 7.00 to provide an alternative to the obsolete HFSToPosix function provided by the HSFAndPosix XOP. With HFSToPosix, if the input path referred to a directory, the output always ended with a slash. With ParseFilePath(9), the output will end with a slash only if the input path ends with a colon.
10	Macintosh only. On Windows this mode returns an error. Returns the HFS path corresponding to the Posix path in pathInStr . *pathInStr* must be a full Posix path starting with a slash character. It does not need to point to an existing directory or file. The returned path may or may not refer to an existing volume, folder or file, depending on pathInStr . Pass "*" for *separatorStr*. *whichEnd* and *whichElement* are ignored. Pass 0 for them. You would typically use this mode when you receive a Posix path from a Unix command executed via ExecuteScriptText and you want to use that path in Igor. This mode was created in Igor Pro 7.00 to provide an alternative to the obsolete PosixToHFS function provided by the HSFAndPosix XOP.

Details

When dealing with Windows paths, you need to be aware that Igor treats the backslash character as an escape character. When you want to put a backslash in a literal string, you need to use two backslashes. See **Escape Sequences in Strings** on page IV-14 and **Path Separators** on page III-451 for details.

On Windows two types of file paths are used: drive-letter paths and UNC (“Universal Naming Convention”) paths. For example:

```
// This is a drive-letter path.
C:\Program Files\WaveMetrics\Igor Pro Folder\WaveMetrics Procedures

// This is a UNC path.
\\BigServer\SharedApps\WaveMetrics\Igor Pro Folder\WaveMetrics Procedures
```

In this example, ParseFilePath considers the volume name to be `C:` in the first case and `\\BigServer\SharedApps` in the second. The volume name is treated as one element by ParseFilePath, except for modes 7 and 8 which permit you to extract the components of the UNC volume name.

Except for the leading backslashes in a UNC path, ParseFilePath modes 0 and 1 internally strip any leading or trailing separator (as defined by the *separatorStr* parameter) from `pathInStr` before it starts parsing. So if you pass `":Igor Pro Folder:WaveMetrics Procedures:"`, it is the same as if you had passed `"Igor Pro Folder:WaveMetrics Procedures"`.

If there is no element corresponding to *whichElement* and *mode* is 0, ParseFilePath returns `""`.

If there is no element corresponding to *whichElement* and *mode* is 1, ParseFilePath returns the entire *pathInStr*.

Examples

```
String pathIn, pathOut
// Full path
pathIn= "hd:Igor Pro Folder:WaveMetrics Procedures:Waves:Wave Lists.ipf"
// Extract first element.
Print ParseFilePath(0, pathIn, ":", 0, 0)      // Prints "hd"
// Extract second element.
Print ParseFilePath(0, pathIn, ":", 0, 1)      // Prints "Igor Pro Folder"
// Extract last element.
Print ParseFilePath(0, pathIn, ":", 1, 0)      // Prints "Wave Lists.ipf"
// Extract next to last element.
Print ParseFilePath(0, pathIn, ":", 1, 1)      // Prints "Waves"
// Get path to folder containing the file.
// Prints "hd:Igor Pro Folder:WaveMetrics Procedures:Waves:"
Print ParseFilePath(1, pathIn, ":", 1, 0)
// Extract the file name without extension.
Print ParseFilePath(3, pathIn, ":", 0, 0)      // Prints "Wave Lists"
// Extract the extension.
Print ParseFilePath(4, pathIn, ":", 0, 0)      // Prints "ipf"
// Make sure the given path ends with a colon and concatenate file name.
String path = <routine that returns a Macintosh-style path to a folder>
path = ParseFilePath(2, path, ":", 0, 0)
path += "AFile.txt"
```

See Also

Escape Sequences in Strings on page IV-14, **UNC Paths** on page III-451, and **Path Separators** on page III-451 for details. The **RemoveEnding** function.

ParseOperationTemplate

`ParseOperationTemplate` `[`***`flags`***`]` ***`cmdTemplate`***

The ParseOperationTemplate operation helps XOP programmers and WaveMetrics programmers write code to implement Igor operations. If you are not an XOP programmer nor a WaveMetrics programmer, it will be of no interest.

ParseOperationTemplate generates starter code for programmers who are creating Igor operations. The starter code is copied to the clipboard, overwriting any previous clipboard contents.

Flags

/C=*c* If *c* is nonzero, ParseOperationTemplate stores code for your ExecuteOperation and RegisterOperation functions in the clipboard.

c=0:	Do not generate code
c=1:	Generate simplified C code - not recommended
c=2:	Generate C code
c=6:	Generate C++ code

The only difference between /C=6 and /C=2 is that the ExecuteOperation function is declared as extern "C" instead of static. C++ files that use static work fine although extern "C" is correct.

/S=*s*	Stores a definition of your runtime parameter structure in the clipboard if *s* is nonzero. *s*=0: Do not generate the runtime parameter structure *s*=1: Use your mnemonic names - recommended *s*=2: Automatically generate mnemonic names - not recommended We recommend that you use /S=1 and provide unique mnemonic parameter names in your template. ParseOperationTemplate then uses your parameter names as structure field names. If you use /S=2, ParseOperationTemplate creates unique field names by concatenating flag or keyword text and your mnemonic names. This is left over from the early days of Operation Handler and is not recommended.
/T	Stores a comment listing your command template in the clipboard.
/TS	Identifies a ThreadSafe operation by adding an extra field to the runtime parameter structure. This is only of use to WaveMetrics programmers.

Parameters

cmdTemplate is the template that describes the syntax for your operation. See the *Igor XOP Toolkit Reference Manual* for details.

Details

ParseOperationTemplate parses your command template, generating structures that embody the syntax of your operation. It then uses these structures to generate code that can serve as a starting point for implementing your operation. The starter code is stored in the clipboard.

For most uses, the recommended flags are:

```
/T/S=1/C=2       // For non-threadsafe operations
/T/S=1/C=2/TS    // For threadsafe operations
```

ParseOperationTemplate sets the following output variable, but only when called from a function or macro:

`V_flag`	0:	*cmdTemplate* was successfully parsed.
	-1:	*cmdTemplate* was not successfully parsed.

If V_flag is nonzero, this indicates that your *cmdTemplate* syntax is incorrect. See the *Igor XOP Toolkit Reference Manual* for details.

Examples

```
Function Test()
    String cmdTemplate
    cmdTemplate = "MyTest"
    cmdTemplate += " /A={number:aNum1,string:aStrH}"
    cmdTemplate += " /B=wave:bWaveH"
    cmdTemplate += " key1={name:k1N1[,wave:k1WaveH,name:k1N2,string[2]:k1StrHArray]}"

    // If your XOP is C instead of C++, use /C=2 instead of /C=6
    TestOperationParser/T/S=1/C=6 cmdTemplate
    Print V_flag, S_value
End
```

See Also

Igor Extensions on page III-511.

PathInfo

`PathInfo` `[`***`/S /SHOW`***`]` ***`pathName`***

The PathInfo operation stores information about the named symbolic path in the following variables:

`V_flag`:	0 if the symbolic path does not exist, 1 if it does exist.
`S_path`:	The full path (e.g., "hd:This:That:").

The path returned is a colon-separated path which can be used on Macintosh or Windows. See **Path Separators** on page III-451 for details.

Flags

/S	Presets the next otherwise undirected open or save file dialog to the given disk folder. This flag is ignored when PathInfo is called from a preemptive thread.
/SHOW	Shows the folder, if it exists, in the Finder (Mac OS X) or Windows Explorer (Windows). This flag is ignored when PathInfo is called from a preemptive thread.

Details

The use of PathInfo in a preemptive thread requires Igor Pro 8.00 or later.

Examples

```
// The following lines perform equivalent actions:
PathInfo/S myPath;Open refNum
Open/P=myPath refNum
```

```
// Show Igor's Preferences folder in the Finder/Windows Explorer.
String fullpath= SpecialDirPath("Preferences",0,0,0)
NewPath/O/Q tempPathName, fullpath
PathInfo/SHOW tempPathName
```

See Also

Symbolic Paths on page II-22.

The **NewPath**, **GetFileFolderInfo**, **ParseFilePath** and **SpecialDirPath** operations.

PathList

PathList(*matchStr*, *separatorStr*, *optionsStr*)

The PathList function returns a string containing a list of symbolic paths selected based on the *matchStr* parameter.

Details

For a path name to appear in the output string, it must match *matchStr*. *separatorStr* is appended to each path name as the output string is generated.

PathList works like the WaveList function, except that the *optionsStr* parameter is reserved for future use. Pass "" for it.

Examples

When a new experiment is created there is only one path:

```
Print PathList("*",";","")
```

Prints the following in the history area:

```
  Igor;
```

See Also

The **WaveList** function for an explanation of the *matchStr* and *separatorStr* parameters and for examples. See also **Symbolic Paths** on page II-22 for an explanation of symbolic paths.

PauseForUser

PauseForUser [**/C**] ***mainWindowName*** [, ***targetWindowName***]

The PauseForUser operation pauses function execution to allow the user to manually interact with a window. For example, you can call PauseForUser from a loop to allow the user to move the cursors on a graph. In this scenario, *targetWindowName* would be the name of the graph and *mainWindowName* would be the name of a control panel containing a message telling the user to adjust the cursors and then click, for example, the Continue button.

If *targetWindowName* is omitted then *mainWindowName* plays the role of target window.

PauseForUser works with graph, table, and panel windows only.

Flags

/C	Tells PauseForUser to return immediately after handling any pending events. See Details.

Details

During execution of PauseForUser, only mouse and keyboard activity directed toward either *mainWindowName* or *targetWindowName* is allowed.

While waiting for user action, PauseForUser disables double-clicks and any contextual menus that can lead to dialogs in order to prevent changes on the command line. It also disables killing windows by clicking the close icon in the title bar unless the window was originally created with the /K=1 flag (kill with no dialog).

If /C is omitted, PauseForUser returns only when the main window has been killed.

If /C is present, PauseForUser handles any pending events, sets V_Flag to the truth the target window still exists, and then returns control to the calling user-defined function. Use PauseForUser/C in a loop if you need to do something while waiting for the user to finish interacting with the target window, such as updating a progress indicator.

As of Igor Pro 8.00, the PauseForUser operation is allowed only in user-defined functions. It returns an error if called from a macro or from the command line.

See Also

Pause For User on page IV-152 for examples and further discussion.

PauseUpdate

```
PauseUpdate
```

The PauseUpdate operation delays the updating of graphs and tables until you invoke a corresponding ResumeUpdate command.

Details

PauseUpdate is useful in a macro that changes a number of things relating to the appearance of a graph. It prevents the graph from being updated after each change. Its effect ends when the macro in which it occurs ends. It also affects updating of tables.

This operation is not allowed from the command line. It is allowed but has no effect in user-defined functions. During execution of a user-defined function, windows update only when you explicitly call the DoUpdate operation.

See Also

The **DelayUpdate**, **DoUpdate**, **ResumeUpdate,** and **Silent** operations.

PCA

```
PCA [flags][wave0, wave1,… wave99]
```

The PCA operation performs principal component analysis. Input data can be in the form of a list of 1D waves, a single 2D wave, or a string containing a list of 1D waves. The operation can produce multiple output waves depending on the specified flags.

Flags

/ALL	Shortcut for the combination of commonly used flags: /CVAR, /SL, /NF, /IND, /IE, and /RMS.
/COV	Calculates the input wave(s) covariance matrix, which as the input for the remainder of the analysis. The covariance matrix is computed by first creating a matrix copying each input 1D wave into sequential columns and then multiplying that matrix by its transpose.
/CVAR	Computes the cumulative percent variance defined as 100 * sum of first *m* eigenvalues divided by the sum of all eigenvalues. The results are stored in the wave W_CumulativeVAR in the current data folder. See also /VAR.

/IE

Computes the imbedded error. Returns errors in the wave W_IE in the current data folder. The wave is scaled using `SetScale/P x 1,1,"", W_IE`. The imbedded error is a function of the number of factors, the number of rows and columns and the sum of the eigenvectors not included in the significant factors. The behavior of IE determines the number of significant factors.

/IND

Computes the factor indicator function. Note that if you specify /IND the residual standard deviation will also be calculated. Returns results in the wave W_IND in the current data folder. The wave is scaled using `SetScale/P x 1,1,"", W_IND`.

/LEIV

Limits eigenvalues so that the SVD calculation does not require too much memory. The limit is set to the minimum of the number of rows or columns of the input.

/NF

Finds the number of significant factors and stores it in the variable V_npnts. You must use /IND in order to compute the significant factors.

/O

Overwrites input waves.

/Q

Suppresses printing of factors in the history area.

/RSD[=*rsdMode*]

Computes the Residual Standard Deviation (RSD) and returns the RSD in the wave W_RSD in the current data folder. The first element in W_RSD is NaN and all remaining wave elements correspond to the number of significant factors.

rsdMode =0: Covariance about the origin.

rsdMode =1: Correlation about the origin.

/RMS

Computes the RMS error. Returns results in the wave W_RMS in the current data folder. The wave is scaled using `SetScale/P x 1,1,"", W_RMS`.

/SCMT

Saves C matrix after the singular value decomposition (SVD) in the wave M_C in the current data folder.

/SCR

Converts the individual wave input into standard scores. Does not work when the input is a single 2D wave. It is an error to convert to standard scores when one or more entries in the waves are NaN or INF. If you use this feature make sure to use the appropriate form of the RSD calculation.

/SDM

Saves a copy of the data matrix at the end of the calculation. This is useful if your input consists of individual waves or if you want to save the computed standard scores. If the input is a 2D matrix, you will get a copy of the input matrix in the wave M_D.

/SEVC

Saves the eigenvalue vector in the wave W_Eigen, which are the raw eigenvalues generated by the SVD, in the current data folder. Normally, if the SVD was applied to a raw data matrix, i.e., not covariance or correlation matrix, you must square each element of the wave to obtain the PCA eigenvalues. Note that this wave has default wave scaling.

/SL

Computes percent significance level and stores it in the wave W_PSL in the current data folder.

/SQEV

Does not square SVD eigenvalues. If you specify /COV there is no need to use this flag.

Use only if your input is already a covariance matrix. In this case the results of the SVD are the eigenvalues not their square roots.

/SRMT

Saves R matrix after the SVD in the wave M_R in the current data folder.

/U

Leaves the input waves unchanged only when the input is a 2D wave. Note that covariance calculations will not be made even if the appropriate flag is used.

/VAR

Computes the variance associated with each eigenvalue. The variance is defined as the ratio of the eigenvalue to the sum of all eigenvalues. The results are stored in the wave W_VAR in the current data folder. See also /CVAR above.

/WSTR=*waveListStr*

Semicolon-separated string list of the names of all input waves.

/Z No error reporting.

Details

The input is either via /WSTR=*waveListStr* or a list of up to 100 1D waves or a single 2D wave following the last flag.

waveListStr is string containing a semicolon-separated list of 1D waves to be used for the data matrix. *waveListStr* can include any legal path to a wave. Liberal names can be quoted or not quoted. It is assumed that all waves are of the same numerical type (either single or double precision) and that all waves have the same number of points.

Regardless of the inputs, the operation expects that the number of rows in the resulting matrix is greater than or equal to the number of columns.

The operation starts by creating the data matrix from the input wave(s). If you provide a list of 1D waves they become the columns of the data matrix. You can choose to use the covariance matrix (/COV) as the data matrix and you can also choose to normalize each column of the data matrix to convert it into standard scores. This involves computing the average and standard deviation of each column and then setting the new values to be:

$$newValue = \frac{oldValue - colAverage}{colStdv}.$$

You can pre-process the input data using **MatrixOp** with the SubtractMean, NormalizeRows, and NormalizeCols functions.

After creating the data matrix the operation computes the singular value decomposition (SVD) of the data matrix. Results of the SVD can be saved or processed further. Save the C and R matrices using /SCMT and /SRMT. These are related to the input data matrix through: $D = R \cdot C$.

The remainder of the operation lets you compute various statistical quantities defined by Malinowski (see References). Use the flags to determine which ones are computed.

The operation generates a number of output waves. All waves are stored in the current data folder.

You can save the input matrix D in the wave M_D, the optional SVD results are stored in the waves M_C that contains the column matrix C, M_R that contains the row matrix R, and W_Eigen that contains the eigenvalues of the data matrix. Note that these can be the eigenvalues or the square of the eigenvalues depending on the input matrix being a covariance matrix or not (see /SQEV).

The optional 1D output waves (W_RSD, W_RMS, W_IE, W_IND, W_PSL) are saved with wave scaling to make it easier to display the wave as a function of the number of factors.

References

Kaiser, H., Computer Program for Varimax Rotation in Factor Analysis, *Educational and Psychological Measurement*, *XIX*, 413-420, 1959.

Malinowski, E.R., *Factor Analysis in Chemistry*, 3rd ed., John Wiley, 2002.

See Also

ICA

pcsr

`pcsr(`*`cursorName`* `[,` *`graphNameStr`*`])`

The pcsr function returns the point number of the point which the specified cursor (A through J) is on in the top (or named) graph. When used with cursors on images or waterfall plots, pcsr returns the row number, and when used with a free cursor, it returns the relative X coordinate.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The pcsr result is not affected by any X axis.

See Also

The **hcsr**, **qcsr**, **vcsr**, **xcsr**, and **zcsr** functions.

Programming With Cursors on page II-321.

Pi

`Pi`

The Pi function returns π (3.141592…).

PICTInfo

`PICTInfo(`*`pictNameStr`*`)`

The PICTInfo function returns a string containing a semicolon-separated list of information about the named picture. If the named picture does not exist, then `""` is returned. Valid picture names can be found in the Pictures dialog.

Details

The string contains six pieces of information, each prefaced by a keyword and colon and terminated with a semicolon.

Keyword	Information Following Keyword
TYPE	One of: “PICT”, “PNG”, “JPEG”, “Enhanced metafile”, “Windows metafile”, “DIB”, “Windows bitmap”, or “Unknown type”.
BYTES	Amount of memory used by the picture.
WIDTH	Width of the picture in pixels.
HEIGHT	Height of the picture in pixels.
PHYSWIDTH	Physical width of the picture in points.
PHYSHEIGHT	Physical height of the picture in points.

Examples

```
Print PICTInfo("PICT_0")
```

will print the following in the history area:

```
TYPE:PICT;BYTES:55734;WIDTH:468;HEIGHT:340;PHYSWIDTH:468;PHYSHEIGHT:340;
```

See Also

The **ImageLoad** operation for loading PICT and other image file types into waves, and the **PICTList** function. The **StringFromList** operation for parsing the information string.

See **Pictures** on page III-509 and **Pictures Dialog** on page III-510 for general information on picture handling.

PICTList

`PICTList(`*`matchStr`*`, `*`separatorStr`*`, `*`optionsStr`*`)`

The PICTList function returns a string containing a list of pictures based on *matchStr* and *optionsStr* parameters. See **Details** for information on listing pictures in graphs, panels, layouts, and the picture gallery.

Details

For a picture name to appear in the output string, it must match *matchStr* and also must fit the requirements of *optionsStr*. *separatorStr* is appended to each picture name as the output string is generated.

The name of each picture is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all picture names.
"xyz"	Matches picture name xyz only.
"*xyz"	Matches picture names which end with xyz.
"xyz*"	Matches picture names which begin with xyz.
"*xyz*"	Matches picture names which contain xyz.
"abc*xyz"	Matches picture names which begin with abc and end with xyz.

matchStr may begin with the ! character to return items that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches picture names which do not end with xyz.

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

optionsStr is used to further qualify the picture.

Use `""` accept all pictures in the Pictures Dialog that are permitted by *matchStr*.

Use the WIN: keyword to limit the pictures to the named or target window:

`"WIN:"`	Match all pictures displayed in the top graph, panel, or layout.
`"WIN:`*`windowName`*`"`	Match all pictures displayed in the named graph, panel, or layout window.

Examples

`PICTList("*",";","")`	Returns a list of all pictures in the Pictures Dialog.
`PICTList("*", ";","WIN:")`	Returns a list of all pictures displayed in the top panel, graph, or layout.
`PICTList("*_bkg", ";", "WIN:Layout0")`	
	Returns a list of pictures whose names end in “_bkg” and which are displayed in Layout0.

See Also

The **ImageLoad** operation for loading PICT and other image file types into waves, and the **PICTInfo** function. Also the **StringFromList** function for retrieving items from lists.

See **Pictures** on page III-509 and **Pictures Dialog** on page III-510 for general information on picture handling.

Picture

`Picture` ***`pictureName`***

The Picture keyword introduces an ASCII code picture definition of binary image data.

See Also

Proc Pictures on page IV-56 for further information.

PixelFromAxisVal

`PixelFromAxisVal(`*`graphNameStr`*`, `*`axNameStr`*`, `*`val`*`)`

The PixelFromAxisVal function returns the local graph pixel coordinate corresponding to the axis value in the graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

If the specified axis is not found and if the name is "left" or "bottom" then the first vertical or horizontal axis will be used.

If *graphNameStr* references a subwindow, the returned pixel value is relative to top left corner of base window, not the subwindow.

Axis ranges and other graph properties are computed when the graph is redrawn. Since automatic screen updates are suppressed while a user-defined function is running, if the graph was recently created or modified, you must call DoUpdate to redraw the graph so you get accurate axis information.

See Also

The **AxisValFromPixel** and **TraceFromPixel** functions.

PlayMovie

`PlayMovie [`*`flags`*`] [as `*`fileNameStr`*`]`

The PlayMovie operation opens a movie file in a window and plays it.

Parameters

The file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

The file is passed to the operating system to be opened with the default program for the given filename extension and the /W flag is ignored.

On Macintosh, prior to Mac OS 10.15 (Catalina,) QuickTime could be used to play a movie in an Igor window and could be controlled using **PlayMovieAction**. Since QuickTime is no longer available, movies no longer open in Igor windows on any operating system and PlayMovieAction after PlayMovie is no longer of use.

Flags

/I	This flag is obsolete and is ignored.
/M	This flag is obsolete and is ignored.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/W=(*left*,*top*,*right*,*bottom*)	This flag is obsolete and is ignored.
/Z	No error reporting; an error is indicated by nonzero value of the variable V_flag. If the user clicks the cancel button in the Open File dialog, V_flag is set to -1.

Details

If the movie file to be played is not fully specified by /P and *fileNameStr*, PlayMovie displays an Open File dialog to let you choose a movie file. See **Symbolic Paths** on page II-22 and **Path Separators** on page III-451 for details.

See Also

Movies on page IV-245.

The **PlayMovieAction** operation.

PlayMovieAction

PlayMovieAction [**/A/Z**] ***keyword*** [***=value***][**,** ***keyword*** [***=value***]]

The PlayMovieAction operation is used to extract frames from a movie file.

Flags

/A — Macintosh: /A is ignored.

Windows: Uses alternate deprecated technology, AVI instead of MMF.

/Z — Errors do not stop procedure execution. Use V_Flag to see if an error occurred.

Keywords

extract — Extracts current frame into an 8-bit RGB image wave named M_MovieFrame. (Can be combined with frame=*f*.)

extract=*e* — Extracts *e* frames into a single multiframe wave, M_MovieChunk. This wave will have 3 planes for RGB and will have *e* chunks.

e=1 is the same as plain extract.

For *e*>1, the current time is automatically updated.

frame=*f* — Moves to specified movie frame.

getID — Returns top movie ID number in V_Value. Do not use in same call with getTime.

getTime — Reads current movie time into variable V_value (in seconds).

gotoBeginning — Goes to beginning of movie.

gotoEnd — Closes the movie opened using the open keyword movie file.

kill — Closes open movie.

loop=*mode* — On Macintosh only, *mode* chooses between two extraction methods. The default, equivalent to loop=0, is fast but can not back up; an attempt to extract a frame previous to the last one extracted results in an error. The alternate method, loop=1, provides random access but can be very slow when reading sequential frames.

The loop keyword was added in Igor Pro 8.00. It is ignored on Windows.

open=*fullPath* — Opens the specifed movie file to enable frame extraction. V_Flag is set to zero if no error occurred and V_Value is set to the file reference number.

ref=*refNum* — The ref keyword is used with all PlayMovieAction commands after using the open keyword to access a movie file. *refNum* must be the file reference number returned in V_Value in the open step.

The ref keyword is needed only if multiple files or windows are open. You can also use setFrontMovie to set the active movie.

setFrontMovie= *id* — Sets the movie with given *id* as the active movie file.

Do not use setFrontMovie and getID in same call to PlayMovieAction.

start — Obsolete. Movie windows are no longer supported in Igor itself.

step=*s* — Moves by *s* frames into movie (0 is same as 1, negative values move backwards).

stop — Obsolete. Movie windows are no longer supported in Igor itself.

Details

Operations are performed in the following order: kill, gotoBeginning, gotoEnd, frame, step, getTime, extract. kill overrides all other parameters.

If you want to extract a grayscale image, you can convert the RGB image into grayscale using the ImageTransform command as follows:

```
PlayMovieAction extract
ImageTransform rgb2gray M_MovieFrame
NewImage M_RGB2Gray
```

When you are finished extracting frames, use the kill keyword to close the file.

To get a full path for use with the open keyword, use the **PathInfo** or **Open** /D/R commands.

Examples

These commands show to determine the number of frames in a simple movie:

```
PlayMovieAction open = <full path to movie file>
PlayMovieAction gotoEnd,getTime
Variable tend= V_value
PlayMovieAction step=-1,getTime
Print "frames= ",tend/(tend-V_value)
PlayMovieAction kill
```

See Also

Movies on page IV-245.

The **PlayMovie** operation.

PlaySnd

PlaySnd [*flags*] ***fileNameStr***

Note: PlaySnd is obsolete. Use **PlaySound** instead.

Available only on the Macintosh.

The PlaySnd operation plays a sound from the file's data fork, or from an `'snd '` resource.

Parameters

The file containing the sound is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/I=*resourceIndex*	Specifies the `'snd '` resource to load by resource index, starting from 1.
/M=*promptStr*	Specifies a prompt if PlaySnd needs to put up a dialog to find the file.
/N=*resNameStr*	Specifies the resource to load by resource name.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/Q	Quiet: suppresses the insertion of `'snd '` info into the history area.
/R=*resourceID*	Specifies the `'snd '` resource to load by resource ID.
/Z	Does not play the sound, just checks for its existence.

Details

If none of /I, /N or /R are specified, PlaySnd tries to play a sound stored in the data fork of the file. If the file dialog is used, only files of type `'sfil'` are shown.

If any of /I, /N or /R are specified, PlaySnd tries to play a sound from an `'snd '` resource. Most programs store sounds in `'snd '` resources. If the file dialog is used, files of all types are shown.

If /P=*pathName* is omitted, then *fileNameStr* can take on three special values:

"Clipboard"	Loads data from Clipboard.
"System"	Loads data from System file.

"Igor"	Loads data from Igor Pro application.

If you specify /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-22 for details.

There are no sounds in the Igor Pro application file.

If the file is not fully specified and *fileNameStr* is not one of these special values, then PlaySnd presents a dialog from which you can select a file. "Fully specified" means that Igor can determine the name of the file (from the *fileNameStr* parameter) and the folder containing the file (from the /P=*pathName* flag or from the *fileNameStr* parameter).

PlaySnd sets the variable V_flag to 1 if the sound exists and fits in available memory or to 0 otherwise.

If the sound exists, PlaySnd also sets the string variable S_Info to:

`"SOURCE:`*`sourceName`*`;RESOURCENAME:`*`resourceName`*`;RESOURCEID:`*`resourceID`*`"`

If the sound is not a resource then *resourceName* is `""` and *resourceID* is 0. *sourceName* will be the name of the file that was loaded or "Clipboard", "System" or "Igor".

Examples

```
PlaySnd/I=1/P=mySnds/Z "Wild Eep"
If (V_flag)            // Any 'snd ' in the "Wild Eep" file?
   Print S_info        // Yes, print resource number, etc.
Endif
```

This prints the following into the history area:

```
SOURCE:resource fork;RESOURCENAME:Wild Eep;RESOURCEID:8;
```

PlaySound

`PlaySound` `[`**`/A`**`[`**`=`*`a`***`]` **`/BITS=`*`bits`*** **`/C`**`]` ***`soundWave`***

`PlaySound /A``[`**`=`*`a`***`]` **`/BITS=`*`bits`*** **`{`*`soundWave1`*`,`** ***`soundWave2`*** `[`**`,`** ***`soundWaveN`****…*`]`**`}`**

The PlaySound operation plays the audio samples in the named wave. The various sound output parameters — number of samples, sample rate, number of channels, and number of bits of resolution — are determined by the corresponding parameters of the wave.

Flags

/A[=*a*] Plays sounds asynchronously so that sounds will continue to play after the command itself has executed.

/A=0:	Same as no /A flag.
/A=1:	Plays sounds asynchronously; same as /A.
/A=2:	Stop playing any current sound before starting this one.
/A=3:	Return with user abort error if output buffers are full (rather than waiting.) Use `GetRTError(1)` to detect and clear the error condition.

/BITS=*bits* Controls the number of bits used for each sound sample sent to the sound output hardware.

Use /BITS=24 with a 32-bit integer wave for 24-bit sound data capable of representing values from -8,388,608 to +8,388,607.

If you omit /BITS or use /BITS=0, PlaySound uses the wave's data type and size to determine how many bits are used for each sound sample.

The /BITS flag was added in Igor Pro 9.00.

/C Obsolete - do not use.

On Windows /C causes sound wave data greater than 16-bits to be converted to 16-bit integer. Such data should range from -32768 to +32767.

On Macintosh /C is ignored.

Details

The wave's time per point, as determined by its X scaling, must be a valid sampling rate. A value of 1/44100 (CD standard) is typical.

Sound waves can be 8, 16, or 32-bit signed integer waves, or single-precision floating point waves:

- 8-bit integer waves support a sample range of -128 to +127.
- 16-bit inter waves support a sample range of -32768 to +32767.
- 32-bit integer waves are used to hold both 24-bit and 32-bit audio. Use /BITS=24 to tell PlaySound that the wave contains 24-bit instead of 32-bit values.
- 32-bit floating point waves support a range of -1 to +1.

Do not use complex waves with PlaySound.

To play a stereo sound, provide a 2 column wave with the left channel in column 0. Actually, the software will attempt to play as many channels as there are columns in the wave. You can also use multiple1D waves with the /A flag. To use this method, enclose the list of 1D waves in braces

With the /A flag, the sound plays asynchronously (i.e., the command returns before the sound is finished). If another command is issued before the sound is finished then the new command will wait until the last sound finishes. A PlaySound without the /A flag can play on top of the current sound. The transition between sounds should be seamless on Macintosh but may be slightly delayed on Windows.

It is OK to kill the sound wave immediately after PlaySound returns even if the /A flag is used.

Examples

Under Windows, support for sound is somewhat idiosyncratic so these sound examples may not work correctly with your particular hardware configuration.

```
Make/B/O/N=1000 sineSound                 // 8 bit samples
SetScale/P x,0,1e-4,sineSound             // Set sample rate to 10Khz
sineSound= 100*sin(2*Pi*1000*x)           // Create 1Khz sinewave tone
PlaySound sineSound
```

The following example will create a rising pitch in the left channel and a falling pitch in the right channel:

```
Make/W/O/N=(20000,2) stereoSineSound      // 16 bit data
SetScale/P x,0,1e-4,stereoSineSound       // Set sample rate to 10Khz
stereoSineSound= 20000*sin(2*Pi*(1000 + (1-2*q)*150*x)*x)
PlaySound/A stereoSineSound               // 16 bit, asynchronous
```

Multichannel sounds as in the previous example but from multiple 1D waves:

```
Make/W/O/N=20000 stereoSineSoundL,stereoSineSoundR    // 16 bit data
SetScale/P x,0,1e-4,stereoSineSoundL,stereoSineSoundR// Set sample rate to 10Khz
stereoSineSoundL= 20000*sin(2*Pi*(1000 + 150*x)*x)// rising pitch in left
stereoSineSoundR= 20000*sin(2*Pi*(1000 - 150*x)*x)// falling in right
PlaySound/A {stereoSineSoundL,stereoSineSoundR}   // two 1D waves
```

See Also

SoundLoadWave, **SoundSaveWave**

pnt2x

`pnt2x(`*`waveName`*`, `*`pointNum`*`)`

The pnt2x function returns the X value of the named wave at the point *pointNum*. The point number is truncated to an integer before use.

For higher dimensions, use **IndexToScale**.

Details

The result is derived from the wave’s X scaling, not any X axis of a graph it may be displayed in.

If you would like to convert a fractional point number to an X value you can use:
`leftx(`*`waveName`*`)+deltax(`*`waveName`*`)*`*`pointNum`*.

See Also

DimDelta, **DimOffset**, **x2pnt**, **IndexToScale**

Waveform Model of Data on page II-62 and **Changing Dimension and Data Scaling** on page II-68 for an explanation of waves and dimension scaling.

Point

The Point structure is used as a substructure usually to store the location of the mouse on the screen.

```
Structure Point
   Int16 v
   Int16 h
EndStructure
```

PointF

The PointF structure is the same as Point but with floating point fields.

```
Structure Point
   float v
   float h
EndStructure
```

poissonNoise

poissonNoise(*num*)

The poissonNoise function returns a pseudo-random value from the Poisson distribution whose probability distribution function is

$$f(x;\lambda) = \frac{e^{-\lambda}\lambda^{x}}{x!}, \qquad \begin{array}{c} \lambda > 0 \\ x = 0,1,2... \end{array}$$

with mean and variance equal to *num* (= λ).

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable “random” numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-390.

Chapter III-12, **Statistics** for a function and operation overview.

poly

poly(*coefsWaveName*, *x1*)

The poly function returns the value of a polynomial function at x = *x1*.

coefsWaveName is a wave that contains the polynomial coefficients. The number of points in the wave determines the number of terms in the polynomial.

If you use poly in a real expression, x1 must be real and poly returns a real value. The wave containing the coefficients can be real or complex. Complex coefficients are interpreted as real(coef).

If you use poly in a complex expression, x1 must complex and poly returns a complex value. The wave containing the coefficients can be real or complex. Real coefficients are interpreted as cmplx(coef,0). Support for complex expressions was added in Igor Pro 9.00.

Examples

```
// Fill wave0 with 100 points containing the polynomial 1 + 2*x + 3*x^2 + 4*x^3
// evaluated over the range from x = -1 to x= 1
Make coefs = {1, 2, 3, 4}        // f(x) = 1 + 2*x + 3*x^2 + 4*x^3
Make/N=100/O wave0; SetScale/I x, -1, 1, wave0
wave0 = poly(coefs, x)
Display wave0
```

poly2D

poly2D(*coefsWaveName*, *x1*, *y1*)

The poly2D function returns the value of a 2D polynomial function at x = *x1*, y = *y1*.

coefsWaveName is a wave that contains the polynomial coefficients. The number of points in the wave determines the number of terms in the polynomial and therefore the polynomial degree.

In a complex expression, poly2D requires x1 and y1 to be complex numbers, and returns a complex value. The wave containing the coefficients may be real or complex. Real coefficients are interpreted as cmplx(coef, 0). Passing complex coefficients in a real expression will use only the real part of the coefficients.

Details

The coefficients wave contains polynomial coefficients for low degree terms first. All coefficients for terms of a given degree must be present, even if they are zero. Among coefficients for a given degree, those for terms having higher powers of X are first. Thus, poly2D returns, for a coefficient wave cw:

f(x,y) = cw[0] + cw[1]*x + cw[2]*y + cw[3]*x^2 + cw[4]*x*y + cw[5]*y^2 + …

A 2D polynomial of degree N has (N+1)(N+2)/2 terms.

Poly2D Example 1

Fill mat1 with the polynomial 1 + 2*x + 2.5*y + 3*x^2 + 3.5*xy + 4*y^2 evaluated over the range x = (-1, 1) and y = (-1, 1):

```
Function Poly2DExample1()
   Make/O coefs1 = {1, 2, 2.5, 3, 3.5, 4}
   Make/N=(20,20)/O mat1
   SetScale/I x, -1, 1, mat1
   SetScale/I y, -1, 1, mat1
   mat1 = poly2D(coefs1, x, y)
   Display; AppendMatrixContour mat1
End
```

The polynomial is second degree, so the first command above made the wave coefs with six elements because (2+1)(2+2)/2 = 6.

Poly2D Example 2

Fill mat2 with the polynomial 1 + 2*x + 3*y+ 4*x^2 + 4*y^2 + 5*x^3 + 6*y^3 evaluated over the range x = (-1, 1) and y = (-1, 1). The first zero eliminates the second-order cross term x*y and the second and third zeros eliminate the third-order cross terms x^2*y and x*y^2:

```
Function Poly2DExample2()
   Make/O coefs2 = {1, 2, 3, 4, 0, 4, 5, 0, 0, 6}
   Make/N=(20,20)/O mat2
   SetScale/I x, -1, 1, mat2
   SetScale/I y, -1, 1, mat2
   mat2 = poly2D(coefs2, x, y)
   Display; AppendMatrixContour mat2
End
```

Poly2D Example 3

This example illustrates using poly2D in a complex expression:

```
Function Poly2DExample3()
   Make/N=(200,200)/C/O cMat3       // Complex-valued matrix
   SetScale/I x -pi,pi,cMat3
   SetScale/I y -pi,pi,cMat3
   Make/D/O coefs3={1,1.5,2,2.5,3,3.5}
   cMat3 = poly2d(coefs3, cmplx(sin(x),cos(y)), cmplx(cos(x),sin(y)))
   Display; Appendimage cMat3
   ModifyImage cMat3 ctab= {*,*,Rainbow256,0}
   ModifyImage cMat3 imCmplxMode=3  // Display the complex phase
End
```

PolygonArea

PolygonArea(*xWave*, *yWave*)

The PolygonArea function returns the area of a simple, closed, convex or nonconvex planar polygon described by consecutive vertices in *xWave* and *yWave*.

A simple polygon has no internal “holes” and its boundary curve does not intersect itself. Both *xWave* and `yWave` must be 1D, real, numerical waves of the same dimensions. The minimum number of vertices is 3. The function uses the shoelace algorithm to compute the area (see theorem 1.3.3 in the reference below). If there is any error in the input, the function returns NaN.

Example

```
Function estimatePi(num)
   Variable num

   Make/O/N=(num+1) xxx,yyy
   xxx=sin(2*pi*x/num)
   yyy=cos(2*pi*x/num)

   Printf "Relative Error=%g\r",(pi-PolygonArea(xxx,yyy))/pi
End
```

See also

areaXY, **faverageXY**, **PolygonOp**

References

O'Rourke, Joseph, *Computational Geometry in C*, 2nd ed., Cambridge University Press, New York, 1998.

PolygonOp

`PolygonOp [`*`flags`*`] [keyword=`*`value`*`]`

The PolygonOp operation performs operations on planar polygons. Polygons and polygon paths are represented by XY pairs of waves with NaNs separating closed polygons.

The operation performed is specified using the operation keyword. The resulting polygon is returned as an XY pair of waves as specified by the /DSTX and /DSTY flags.

PolygonOp uses the clipper library by Angus Johnson. It was added in Igor Pro 9.00.

Flags

/ADDR=*addingRule*

addingRule determines how the primary and secondary polygons are added to the primary and secondary paths. The interior of the polygon is defined as the region that would be "filled" using one of the standard fill rules. The even-odd rule applies to scan lines while the other rules are based on winding counts.

addingRule is one of the following:

0:	Even-odd (default). All odd numbered sub-regions are filled, while even numbered sub-regions are not filled.
1:	Non-zero. All non-zero winding subregions are filled.
2:	Positive. All sub-regions with winding counts>0 are filled.
3:	Negative. All sub-regions with winding counts<0 are filled.

/NCFP Disables closing the resulting polygons by connecting the last vertex to the first vertex of each output path.

/DIST=*dist* Specifies the distance in scaled units (see the /SM flag) below which vertices are stripped. The default value is approximately sqrt(2).

/DSTX=*destX* Specifies the X destination wave. The output is corrected for any scaling (see /SM below).

In a user-defined function, if *destX* is a simple wave name, PolygonOp automatically creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-72 for details.

If you omit /DSTX, the X destination wave defaults to polyOpWaveX in the current data folder.

If *destX* is a simple name and you include /FREE, the X destination is created as a free wave.

/DSTY=*destY* Specifies the Y destination wave. The output is corrected for any scaling (see /SM below).

In a user-defined function, if *destY* is a simple wave name, PolygonOp automatically creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-72 for details.

If you omit /DSTY, the Y destination wave defaults to polyOpWaveX in the current data folder.

If *destY* is a simple name and you include /FREE, the Y destination is created as a free wave.

/FA Computes the final area of the resulting polygon and stores it in V_area. This is an algebraic (signed) area in contrast to the area returned by PolygonArea which is always non-negative.

/FREE Creates the output waves specified by /DSTX and /DSTY as free waves.

/JTYP=*jointType* Specifies the joint type used when offsetting a polygon. The following joint types are supported:

0:	Square joint
1:	Round joint
2:	Miter joint

Square and round joints may give rise to additional vertices along the path.

/MLMT=*miterLimit* When the path consists of two edges connected at an acute angle, the offset could give rise to very long spikes. The miter limit sets the maximum distance in multiples of the offset that vertices can be offset from their original positions before a square joint (see /JTYP) is used. The default value for *miterLimit* is 2. The *miterLimit* parameter is scaled by /SM scaleFactor.

/OFST=*offset* Applies an offset to the final polygon. For example, if the polygon represents a rectangle, a positive offset creates an inscribing rectangle while a negative offset creates an inscribed rectangle. If the command includes a binary operation between primary and secondary polygon paths then the offset applies to the result of the binary operation. The *offset* parameter is scaled by /SM *scaleFactor*.

/Q Quiet mode. Do not print any information in the history area of the command window.

/RPRC=*rPrecision* The rounding precision *rPrecision* specifies the maximum distance in scaled units that a line segment can deviate from the exact arc. This applies only when /JTYP=1 and when offseting a polygon. The *rPrecision* parameter is scaled by /SM *scaleFactor*.

/SM={*scaleMethod* [,*scaleFactor*]}

Specifies a scaling method and a scaling factor used when converting the input data into the internal integer representation employed by the algorithm.

scaleMethod is one of the following:

0: No scaling. Floating point vertex locations are converted to integers by simple truncation:

```
Vc = trunc(Vin)
```

1: Multiplication by scaleFactor followed by truncation to integer:

```
Vc = trunc(Vin*scaleFactor)
```

2: Scaling followed by rounding to integer:

```
Vc = round(Vin*scaleFactor)
```

scaleFactor must be a finite positive number and defaults to 1.0. All scaling is reversed on output using multiplication by the reciprocal of scaleFactor. Depending on the range of your data you may be able to use a scale factor which improves the resolution of the calculation. For example, if your data are O(1) and you use a scale factor of 10 or 100 the resulting conversion into integers will be more accurate. On the other hand, if your data are O(2^30) a scale factor greater than 1 is not ideal.

/SPT={*testPointX*, *testPointY*}

Specifies a single point for testing if the point is inside the primary path polygon. The coordinates of point are scaled using the same scaling as the primary waves. The result of the test is stored in V_value as 0 if that the point is outside the polygon, 1 if that the point is inside the polygon, and 2 if the point is on the boundary of the polygon.

/Z Suppresses error reporting. If you use /Z, check the V_Flag output variable to see if the operation succeeded.

Keywords

operation=*op*

You can omit the operation=op pair when you are applying an offset to the primary polygon or when testing for pointsInPoly. Otherwise, *op* is one of the following:

opN is one of the following:

Intersection Computes the intersection of the region defined by the primary path or paths and the region defined by the secondary path or paths.

Union Computes the union between the region defined by the primary path or paths and the region defined by the secondary path or paths.

Difference Computes the difference between the region defined by the primary path or paths and the region defined by the secondary path or paths.

XOR Computes the exclusive OR between the regions defined by the primary path or paths and the regions defined by the secondary path or paths. This includes all regions where either the primary or secondary polygons are filled but not where both are filled.

PointInPoly Tests if the points specified by *pointsWaves* are inside the target polygon. The results are stored in the unsigned byte wave W_inPoly in the current data folder: 0 for a point outside, 1 for a point inside, and 2 for a point on the boundary.

The target polygon is defined by *primaryWaves*. The tested points are internally scaled with the same scaling method (/SM) that applies to primary waves. You can perform this test on a target polygon that is a result of any binary polygon operation if you only specify the *pointsWaves*, for example:

```
PolygonOp operation=union,
primaryWaves={wx,wy}, secondarywaves={sx,sy},
pointsWaves={px,py}
```

Area Computes the polygon area and stores it in V_area when

CleanPolygon Removes vertices from the primary paths that satisfy one of the following conditions:

1. The vertices lie on a co-linear edge. For every three vertices the middle vertex is removed if it lies close enough (see /DIST above) to the edge connecting the vertices on both of its sides.

2. The vertices are situated below a critical distance from an adjacent vertex (see /DIST above).

See also the Simplify operation.

MinkowskiDiff If the primary and secondary paths represent sets of vectors, the Minkowski difference of the two sets is defined as the set formed by subtracting each vector in the primary set from each vector in the secondary set.

MinkowskiSum If the primary and secondary paths represent sets of vectors, the Minkowski sum of the two sets is defined as the set formed by adding each vector in the primary set to each vector in the secondary set.

Simplify Removes self-intersections from the input polygons. In case of touching vertices the resulting polygon is split into two polygons.

Reverse Reverses the polygon paths.

pointsWaves={*pointsXWave*, *pointsYWave*}

Specifies a pair of 1D waves that define points for testing pointInPoly. The waves must have the same number type and the same number of points. The polygon that is used in the test is either the primary polygon, when the operation is PointInPoly, or the polygon resulting from the combination of the primary and secondary polygons together with the operation specified by the operation keyword. The waves must contain at least one point.

primaryWaves={*srcXWave*, *srcYWave*}

Specifies the primary path or paths. In case of more than one path, the paths are separated by a NaN in both waves. Both waves must have the same number type and the same number of points. The waves must contain at least 3 points.

secondaryWaves={*secondaryXWave*, *secondaryYWave*}

Specifies a the secondary or clipping path or paths. In case of more than one path, the paths are separated by a NaN in both waves. Both waves must have the same number type and the same number of points. The waves must contain at least 3 points.

Details

PolygonOp performs various operations on polygon paths that are defined by pairs of 1D real numeric waves. A path is defined by an XY pair of waves that may describe one or more closed polygons. Consecutive polygons are separated by a single NaN value in both path waves.

You must always specify the primary path waves. You must specify the secondary waves if you need to perform a binary polygon operation.

To simplify the computation the input waves are internally mapped into 32-bit signed integer arrays so that vertices correspond to discrete integer pairs (pixels).

To support a broad range of inputs, the operation allows you to specify how the input waves are scaled into integers. Scaling is isotropic; there is currently no support for anamorphic scaling. The scaling factor determines the number of significant figures that remain after conversion to integers. In extreme cases it is advisable to preprocess the polygon waves, e.g., by removing a mean offset and/or applying anamorphic scaling. On output, an inverse scaling is applied with no compensation for the initial truncation or rounding. The inverse scaling is also applied to area calculations.

Output Variables

PolygonOp sets the following output variables:

V_flag	Zero if the operation succeeds or to a non-zero error code.
V_value	The pointInPolyResult result for a single point specified using /SPT.
V_area	The area of the resulting polygon corrected for scaling.
S_waveNames	Semicolon separated list of the names of the waves created by the operation.

See Also

DrawPoly, **FindPointsInPoly**, **PolygonArea**

Demos

Choose Files→Example Experiments→Feature Demos 2→PolygonOp Demo.

popup

`popup menuList`

The popup keyword is used with Prompt statements in Functions and Macros. It indicates that you want a pop-up menu instead of the normal text entry item in a DoPrompt simple input dialog (or a Macro's missing parameter dialog (archaic)). *menuList* is a string expression containing a list of items, separated by semicolons, that are to appear in the pop-up menu.

Pop-up menus accept both numeric and string parameters. For numeric parameters, the number of the item selected is placed in the variable. Numbering starts from one. For string parameters, the selected item's text is placed in the string variable.

Pop-up items support all of the special characters available for user-defined menu definitions (see **Special Characters in Menu Item Strings** on page IV-133) with the exception that items in pop-up menus are limited to 50 bytes, keyboard shortcuts are not supported, and special characters must be enabled.

See Also

Prompt, **DoPrompt**, and **Pop-Up Menus in Simple Dialogs** on page IV-145.

See **WaveList**, **TraceNameList**, **ContourNameList**, **ImageNameList**, **FontList**, **MacroList**, **FunctionList**, **StringList**, and **VariableList** for functions useful in generating lists of Igor objects.

Chapter III-14, **Controls and Control Panels** for details about control panels and controls.

PopupContextualMenu

```
PopupContextualMenu [ /C=(xpix, ypix) /N /ASYN[=func] ] popupStr
```

The PopupContextualMenu operation displays a pop-up menu.

The menu appears at the current mouse position or at the location specified by the /C flag.

The content of the menu is specified by *popupStr* as a semicolon-separated list of items or, if you include the /N flag, by a user-defined menu definition referred to by the name contained in *popupStr*.

If you omit the /ASYN flag, the menu is tracked and the operation does not return until the user makes a selection or cancels the menu by clicking outside of its window.

If you include /ASYN, the menu is displayed and the operation returns immediately. When the user makes a selection, then the result is sent to the specified function or to the user-defined menu's execution text. You can use /ASYN to allow a background task to continue while a contextual menu is popped up. /ASYN requires Igor Pro 7.00 or later.

Parameters

If *popupStr* specifies the pop-up menu's items (/N is not specified), then *popupStr* is a semicolon-separated list of items such as "yes;no;maybe;", or a string expression that returns such a list, such as **TraceNameList**.

The menu items can be formatted and checkmarked, like user-defined menus can. See **Special Characters in Menu Item Strings** on page IV-133.

If /N is specified, *popupStr* must be the name of a user-defined menu that also has the popupcontextualmenu keyword. See Example 3.

Flags

/ASYN When used with /N: The user-defined menu is displayed and operation returns immediately. The result of menu selection is handled by the user-defined menu's execution text. See **User-Defined Menus** on page IV-125.

/ASYN=*func* When used without /N: The user-defined menu is displayed and operation returns immediately. The result of menu selection or cancellation is handled by calling the named function, which must have the following format:

```
Function func(popupStr, selectedText, menuItemNum)
   String popupStr
   String selectedText
   Variable menuItemNum
```

In Igor Pro 8.05 and later, if the user cancels the menu selection, selectedText will be "" and menuItemNum will be 0.

/C=(*xpix*, *ypix*) Sets the coordinates of the menu's top left corner.

Units are in pixels relative to the top-most window or the window specified by /W, like the MOUSEX and MOUSEY values passed to a window hook. See the window hook example, below and **SetWindow**.

If /C is not specified, the menu's top left corner appears at the current mouse position.

/N Indicates that *popupStr* contains the name of a menu definition instead of containing a list of menu items.

/W=*winName* The /C coordinates are relative to the top/left corner of the named window or subwindow. If you omit /W, /C uses the top-most window having focus.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/W was added in Igor Pro 7.00.

Details

If you omit /N and /ASYN, PopupContextualMenu sets the following variables:

`V_flag=0` User cancelled the menu without selecting an item, or there was an error such as an empty *popupStr*.

`V_flag=`>= 1 1 if the first menu item was selected, 2 for the second, etc.

`S_selection` `""` if the user cancelled or error, else the text of the selected menu item.

If you include /N and omit /ASYN, PopupContextualMenu sets the following variables in a manner similar to **GetLastUserMenuInfo**:

V_kind The kind of menu that was selected:

V_kind	Menu Kind
0	Normal text menu item, including **Optional Menu Items** (see page IV-130) and **Multiple Menu Items** (see page IV-131).
3	"*FONT*"
6	"*LINESTYLEPOP*"
7	"*PATTERNPOP*"
8	"*MARKERPOP*"
9	"*CHARACTER*"
10	"*COLORPOP*"
13	"*COLORTABLEPOP*"

See **Specialized Menu Item Definitions** on page IV-132 for details about these special user-defined menus.

V_flag -1 if the user didn't select any item, otherwise V_flag returns a value which depends on the kind of menu the item was selected from:

V_kind	V_flag Meaning
0	Text menu item number (the first menu item is number 1).
3	Font menu item number (use S_selection, instead).
6	Line style number (0 is solid line)
7	Pattern number (1 is the first selection, a SW-NE light diagonal).
8	Marker number (1 is the first selection, the X marker).
9	Character as an integer, = char2num(S_selection). Use S_selection instead.
10	Color menu item (use V_Red, V_Green, V_Blue, and V_Alpha instead).
13	Color table list menu item (use S_selection instead).

S_selection The menu item text, depending on the kind of menu it was selected from:

V_kind	S_selection Meaning
0	Text menu item text.
3	Font name or "default".
6	Name of the line style menu or submenu.
7	Name of the pattern menu or submenu.
8	Name of the marker menu or submenu.
9	Character as string.
10	Name of the color menu or submenu.
13	Color table name.

In the case of **Specialized Menu Item Definitions** (see page IV-132), S_selection will be the title of the menu or submenu, etc.

V_Red, V_Green, V_Blue, V_Alpha

If a user-defined color menu ("*COLORPOP*" menu item) was selected then these values hold the red, green, and blue values of the chosen color. The values range from 0 to 65535 - see **RGBA Values**.

Will be 0 if the last user-defined menu selection was not a color menu selection.

If you include /N and /ASYN, PopupContextualMenu sets the following variables:

`V_flag=0` There was an error such as an empty *popupStr* or *popupStr* did not name a compiled user-defined menu.

`V_flag=-1` No error. The named user menu was valid and no item was selected yet.

`S_selection` ""

If you include /N and omit /ASYN, PopupContextualMenu sets the following variables:

`V_flag=0` There was an error such as an empty *popupStr*.

`V_flag=-1` No error. *popupStr* was valid and no item was selected yet.

`S_selection` ""

Examples

Example 1 - *popupStr* contains a list of menu items

```
// Menu formatting example

String checked= "\\M0:!" + num2char(18) + ":"  // checkmark code
String items= "first;\M1-;"+checked+"third;"   // 2nd is divider, 3rd is checked
PopupContextualMenu items
switch( V_Flag )
    case 1:
        // do something because first item was chosen
        break;
    case 3:
        // do something because first item was chosen
        break;
endswitch
```

Example 2 - *popupStr* contains a list of menu items

```
// Window hook example

SetWindow kwTopWin hook=TableHook, hookevents=1      // mouse down events
Function TableHook(infoStr)
    String infoStr

    Variable handledEvent=0
    String event= StringByKey("EVENT",infoStr)
    strswitch(event)
        case "mousedown":
            Variable isContextualMenu= NumberByKey("MODIFIERS",infoStr) & 0x10
            if( isContextualMenu )
                Variable xpix= NumberByKey("MOUSEX",infoStr)
                Variable ypix= NumberByKey("MOUSEY",infoStr)
                PopupContextualMenu/C=(xpix,ypix) "yes;no;maybe;"
                strswitch(S_selection)
                    case "yes":
                        // do something because "yes" was chosen
                        break
                    case "no":
                        break
                    case "maybe":
                        // do something because "maybe" was chosen
                        break
            endswitch
            handledEvent=1
        endif
    endswitch
    return handledEvent
End
```

Example 3 - popupStr contains the name of a user-defined menu

```
// User-defined contextual menu example

// dynamic menu (to keep WaveList items updated), otherwise not required.
// contextualmenu keyword is required, and implies /Q for all menu items.
//
// NOTE: Actions here are accomplished by the menu definition's
// execution text, such as DoSomethingWithColor.
// See Example 4 for another approach.
//
Menu "ForContext", contextualmenu, dynamic
    "Hello", Beep
    Submenu "Color"
        "*COLORPOP*", DoSomethingWithColor()
    End
    Submenu "Waves"
        WaveList("*",";",""), /Q, DoSomethingWithWave()
    End
End

Function DoSomethingWithColor()
    GetLastUserMenuInfo
    Print V_Red, V_Green, V_Blue, V_Alpha
End

Function DoSomethingWithWave()
    GetLastUserMenuInfo
    WAVE w = $S_value
    Print "User selected "+GetWavesDataFolder(w,2)
End

// Use this code in a function or macro:
PopupContextualMenu/N "ForContext"
if( V_flag < 0 )
    Print "User did not select anything"
endif
```

Example 4 - popupStr contains the name of a user-defined menu

```
// User-defined contextual menu example

Menu "JustColorPop", contextualmenu
    "*COLORPOP*(65535,0,0)", ;// initially red, empty execution text
End

// Use this code in a function or macro
PopupContextualMenu/C=(xpix, ypix)/N "JustColorPop"
if( V_flag < 0 )
    Print "User did not select anything"
else
    Print V_Red, V_Green, V_Blue, V_Alpha
endif
```

Example 5 - popupStr contains a list of menu items, asynchronous popup result

```
Function YourFunction()
    // Use this code in a function or macro:
    PopupContextualMenu/ASYN=Callback "first;second;third;"
    (YourFunction continues...)
End

// Routine called when/if popup menu item is selected. selectedItem=1 is the first item.
Function Callback(String list, String selectedText, Variable selectedItem)
```

```
    Print "Callback: ", list, selectedText, selectedItem
End
```

Example 6 - popupStr contains the name of a user-defined menu, asynchronous popup result

```
Function YourFunction()
    PopupContextualMenu/ASYN/N "ForContext"
    (YourFunction continues...)
End
// Selection result is handled in "ForContext" menu's execution texts, as in Example 4
```

See Also

Creating a Contextual Menu on page IV-162, **User-Defined Menus** on page IV-125.

Special Characters in Menu Item Strings on page IV-133.

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

The **SetWindow** and **PopupMenu** operations.

PopupMenu

PopupMenu [**/Z**] ***ctrlName*** [***keyword = value*** [, ***keyword = value*** …]]

The PopupMenu operation creates or modifies a pop-up menu control in the target or named window.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the PopupMenu control to be created or changed.

The following keyword=value parameters are supported:

align=*alignment*
Sets the alignment mode of the control. The alignment mode controls the interpretation of the *leftOrRight* parameter to the pos keyword. The align keyword was added in Igor Pro 8.00.

If *alignment*=0 (default), *leftOrRight* specifies the position of the left end of the control and the left end position remains fixed if the control size is changed.

If *alignment*=1, *leftOrRight* specifies the position of the right end of the control and the right end position remains fixed if the control size is changed.

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **Button** and **DefaultGUIControls** for more appearance details.

bodyWidth=*width*
Specifies an explicit size for the body (nontitle) portion of a PopupMenu control. By default (bodyWidth=0), the body portion autosizes depending on the current text. If you supply a bodyWidth>0, then the body is fixed at the size you specify regardless of the body text. This makes it easier to keep a set of controls right aligned when experiments are transferred between Macintosh and Windows, or when the default font is changed.

disable=*d*
Sets user editability of the control.

d=0:	Normal.
d=1:	Hide.
d=2:	Draw in gray state; disable control action.

fColor=(*r*,*g*,*b*[,*a*]) Sets the initial color of the title. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

To further change the color of the title text, use escape sequences as described for title=*titleStr*.

focusRing=*fr* Enables or disables the drawing of a rectangle indicating keyboard focus:

fr=0:	Focus rectangle will not be drawn.
fr=1:	Focus rectangle will be drawn (default).

On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.

font="*fontName*" Sets the font used for the pop-up title, e.g., `font="Helvetica"`.

fsize=*s* Sets the font size for the pop-up title.

fstyle=*fs* *fs* is a bitwise parameter with each bit controlling one aspect of the font style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*} Sets the help for the control.

helpStr is limited to 1970 bytes (255 in Igor Pro 8 and before).

You can insert a line break by putting "\r" in a quoted string.

mode=*m* Specifies the pop-up title location.

m=0:	Title is in pop-up menu.
m=1:	Title is to the left of pop-up menu, the chosen menu item appears in the pop-up menu, and menu item number *m* is initially selected.

noproc Specifies that no procedure is to execute when choosing in the pop-up menu.

popColor=(*r*,*g*,*b*[,*a*]) Specifies the color initially chosen in the color pop-up palette. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. See the **Colors, Color Tables, Line Styles, Markers, and Patterns** section.

popmatch=*matchStr*

Sets mode to the enabled menu item that matches *matchStr*. *matchStr* may be a "wildcard" expression. See **StringMatch**. If no item is matched, mode is unchanged.

popvalue=*valueStr* Sets the string displayed by the menu when first created, if mode is not zero. See **Popvalue Keyword** section.

pos={*leftOrRight*,*top*} Sets the position in **Control Panel Units** of the top/left corner of the control if its alignment mode is 0 or the top/right corner of the control if its alignment mode is 1. See the align keyword above for details.

pos+={*dx*,*dy*} Offsets the position of the pop-up in **Control Panel Units**.

proc=*procName* Specifies the procedure to execute when the pop-up menu is clicked. See ***Pop-up Menu Action Procedure*** below.

rename=*newName* Gives pop-up menu a new name.

size={*width*,*height*} Sets pop-up menu size in **Control Panel Units**.

title=*titleStr* Sets title of pop-up menu to the specified string expression. Defaults to "" (no title).

Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.

userdata(*UDName*)=*UDStr*

Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create. You can retrieve the data using **GetUserData**.

userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

value=*itemListSpec* Specifies the pop-up menu's items. *itemListSpec* can take several forms as described below under ***Setting The Popup Menu Items***.

win=*winName* Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

The target window, or the window named with the win=*winName* keyword, must be a graph or panel.

Pop-up Menu Action Procedure

The action procedure for a pop-up menu control takes a predefined WMPopupAction structure as a parameter to the function:

```
Function PopupMenuAction(PU_Struct) : PopupMenuControl
   STRUCT WMPopupAction &PU_Struct
   ...
   return 0
End
```

The ": PopupMenuControl" designation tells Igor to include this procedure in the list of available popup menu action procedures in the PopupMenu Control dialog used to create a popup menu.

See **WMPopupAction** for details on the WMPopupAction structure.

Although the return value is not currently used, action procedures should always return zero.

You may see an old format pop-up menu action procedure in old code:

```
Function PopupMenuAction (ctrlName,popNum,popStr) : PopupMenuControl
   String ctrlName
   Variable popNum     // which item is currently selected (1-based)
   String popStr       // contents of current popup item as string
   ...
   return 0
End
```

This old format should not be used in new code.

Setting The Popup Menu Items

This section discusses popup menus containing lists of text items. The next section discusses popup menus for choosing colors, line styles, markers and patterns.

The items in the popup menu are determined by the *itemListSpec* parameter used with the value keyword. *itemListSpec* can take several different forms from simple to complex.

No matter what the form, Igor winds up storing an expression that returns a string in the popup menu's internal structure. This expression may be a literal string ("Red;Green;Blue;"), a call to a built-in or user-defined function that returns a string, or the path to a global string variable. Igor evaluates this expression when the popup menu is first created and again each time the user clicks on the menu. You can see the string expression for a given popup menu using the PopupMenu Control dialog.

The right form for *itemListSpec* depends on your application. Here is a guide to choosing the right form with the simpler forms first.

A literal string expression

Use this if you know the items you want in your popup menu when you write the PopupMenu call. For example:

```
Function PopupDemo1()   // Literal string
   NewPanel
   PopupMenu popup0, value="Red;Green;Blue;"
End
```

This method is limited to 2500 bytes of menu item text.

A function call

Use this if you need to compute the popup menu item list when the user clicks the popup menu. The function must return a string containing a semicolon-separated list of menu items. This example creates a popup menu which displays the name of each wave in the current data folder at the time the menu is clicked:

```
Function PopupDemo2()   // Built-in function
   NewPanel
   PopupMenu popup0, value=WaveList("*", ";", "")
End
```

You can also use a user-defined function. This example shows how to list waves from other than the current data folder:

```
Function/S MyPopupWaveList()
   DFREF saveDF

   // Create some waves for demo purposes
   saveDF = GetDataFolderDFR()
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S PopupMenuDemo
   Make/O demo0, demo1, demo2
   SetDataFolder saveDF

   saveDF = GetDataFolderDFR()
   SetDataFolder root:Packages:PopupMenuDemo
   String list = WaveList("*", ";", "")
   SetDataFolder saveDF

   return list
End

Function PopupDemo3()   // User-defined function
   NewPanel
   PopupMenu popup0, value=MyPopupWaveList()
End
```

# followed by a local string variable specifying items

Use this when the popup menu item list is not known when you write the code but you can compute it at runtime. For example:

```
Function PopupDemo4()   // Local string variable specifying items
   NewPanel
   String quote = "\""
   String list
   if (CmpStr(IgorInfo(2),"Windows") == 0)
      list = quote + "Windows XP; Windows VISTA;" + quote
   else
      list = quote + "Mac OS X 10.4;Mac OS X 10.5;" + quote
   endif
   PopupMenu popup0, value=#list
End
```

The strange-looking use of the quote string variable is necessary because the parameter passed to the value=# keyword is evaluated once when the PopupMenu command executes and the result of that evaluation is evaluated again when the PopupMenu is created or clicked. The result of the first evaluation must be a legal string expression.

This method is limited to 2500 bytes of menu item text.

# followed by a local string variable specifying a function

Use this when you need to compute the popup menu item list at click time and you need to select the function which computes the list when the popup menu is created. For example:

```
Function/S WindowsItemList()
    String list
    list = "Windows XP; Windows VISTA;"
    return list
End

Function/S MacItemList()
    String list
    list = "Mac OS X 10.4;Mac OS X 10.5;"
    return list
End

Function PopupDemo5()   // Local string variable specifying function
    String listFunc
    if (CmpStr(IgorInfo(2),"Windows") == 0)
        listFunc = "WindowsItemList()"
    else
        listFunc = "MacItemList()"
    endif
    NewPanel
    PopupMenu popup0, value=#listFunc
End
```

This form is useful when you create a control panel in an independent module. Since the control panel runs in the global name space, you must specify the independent module name in the invocation of the function that provides the popup menu items. For example:

```
// Calling a non-static function in an independent module from #included code
#pragma IndependentModuleName=IM
. . .
String listFunc= GetIndependentModuleName()+"#PublicFunctionInIndepMod()"
PopupMenu popup0, value=#listFunc

// Calling a static function in an independent module from #included code
#pragma IndependentModuleName=IM
#pragma ModuleName=ModName
. . .
String listFunc= GetIndependentModuleName()+"#ModName#StaticFunctionInIndepMod()"
PopupMenu popup0, value=#listFunc
```

We use GetIndependentModuleName rather than hard-coding the name of the independent module so that the code will continue to work if the name of the independent module is changed. Also, because this code does not depend on the specific name of the independent module, it can be added to an independent module via a #included procedure file.

Also see **GetIndependentModuleName** and **Independent Modules and Popup Menus** on page IV-241.

# followed by a quoted literal path to a global string variable

Use this if you want to compute the popup menu item list before it is clicked, not each time it is clicked. This would be advantageous if it takes a long time to compute the item list, and the list changes only at well-defined times when you can set the global string variable.

The global string variable must exist when the PopupMenu command executes and when the menu is clicked. In this example, the gPopupMenuItems global string variable is created and initialized when the popup menu is created but can be changed to a different value later before the menu is clicked:

```
Function PopupDemo6()   // Global string variable containing list
    NewDataFolder/O root:Packages
    NewDataFolder/O root:Packages:PopupMenuDemo
    String/G root:Packages:PopupMenuDemo:gPopupMenuItems = "Red;Green;Blue;"

    NewPanel
    PopupMenu popup0 ,value=#"root:Packages:PopupMenuDemo:gPopupMenuItems"
End
```

# followed by a local string variable containing a path to a global string variable

Use this when the popup menu item list contents will be stored in a global string variable whose location is not known until the popup menu is created. For example:

```
Function PopupDemo7()   // Local string containing path to global string
    String graphName = WinName(0, 1, 1)// Name of top graph
    if (strlen(graphName) == 0)
        Print "There are no graphs."
        return -1
    endif

    NewDataFolder/O root:Packages
    NewDataFolder/O root:Packages:PopupMenuDemo

    // Create data folder for graph
    NewDataFolder/O root:Packages:PopupMenuDemo:$(graphName)

    String list = "Red;Green;Blue;"
    String/G root:Packages:PopupMenuDemo:$(graphName):gPopupMenuItems = list

    NewPanel

    String path         // Local string containing path to global string
    path = "root:Packages:PopupMenuDemo:" + graphName + ":gPopupMenuItems"
    PopupMenu popup0, value=#path

    return 0
End
```

Colors, Color Tables, Line Styles, Markers, and Patterns

You can create PopupMenu controls for color, color tables, line style (dash modes), markers, and patterns. To do so, simply specify the *itemListSpec* parameter to the value keyword as one of `"*COLORPOP*"`, `"*COLORTABLEPOP*"`, `"*COLORTABLEPOPNONAMES*"`, `"*LINESTYLEPOP*"`, `"*MARKERPOP*"`, or `"*PATTERNPOP*"`. In these modes the body of the control will contain a color box, a color table (gradient), a line style sample, a marker, or a pattern sample.

For these special pop-up menus, `mode=0` (“Title in Box” checked) is not used.

For a line style pop-up menu, the mode value is the line style number plus one. Thus line style 0 (a solid line) is `mode=1`.

For a marker pop-up, the mode value is the marker number plus one, and marker 0 (the + marker) is `mode=1`.

For a pattern pop-up, the mode value is the **SetDrawEnv** fillPat number minus 4, so mode=1 corresponds to `fillpat=5`, the SW-NE lines fill pattern shown above.

For a color table pop-up, the mode value is the CTabList() index plus 1, so mode=1 corresponds to the first item in the list returned by **CTabList**, which is “Grays”:

```
ControlInfo $ctrlName                           // Sets V_Value
Print StringFromList(V_Value-1,CTabList())      // Prints "Grays"
```

ControlInfo also returns the color table name in S_Value.

To set the pop-up to a given color table name, you can use code like this:

```
Variable m = 1 + WhichListItem(ctabName, CTabList())
PopupMenu $ctrlName mode=m
```

For color pop-up menus, you set the current value using the `popColor=(`*r*`,` *g*`,` *b*[,*a*]`)` keyword. On output (via the popStr parameter of your action procedure or via the S_value output from **ControlInfo**) the color is encoded as "(*r*,*g*,*b*)" or "(r,g,b,a)". To get these numerical values, you can extract them from the string using the MyRGBStrToRGB function below or use ControlInfo which sets V_Red, V_Green, V_Blue and V_Alpha.

The following example demonstrates the line style and color pop-up menus. To run the example, copy the following code to the procedure window of a new experiment and then run the panel macro.

```
Window Panel0() : Panel
    PauseUpdate; Silent 1           // building window …
    NewPanel /W=(150,50,400,182)
    PopupMenu popup0,pos={74,31},size={96,20},proc=ColorPopMenuProc,title="colors"
    PopupMenu popup0,mode=1,popColor= (0,65535,65535),value= "*COLORPOP*"
    PopupMenu popup1,pos={9,68},size={221,20},proc=LStylePopMenuProc
```

```
	PopupMenu popup1,title="line styles",mode=1,value= "*LINESTYLEPOP*"
EndMacro

Function ColorPopMenuProc(ctrlName,popNum,popStr) : PopupMenuControl
	String ctrlName
	Variable popNum
	String popStr

	Variable r,g,b,a
	MyRGBStrToRGB(popStr,r,g,b,a)	// One way to get r, g, b
	print popStr," gives: ",r,g,b,a

	ControlInfo $ctrlName		// Another way: Sets V_Red,V_Green,V_Blue,V_Alpha
	Printf "ControlInfo returned (%d,%d,%d,%d)\r", V_Red, V_Green, V_Blue, V_Alpha

	return 0
End

// Take (r,g,b, and possibly a) as a string and extract numeric r,g,b,a values
Function MyRGBstrToRGB(rgbStr,r,g,b,a)
	String rgbStr
	Variable &r, &g, &b, &a

	if (CmpStr(rgbStr[0], "(") == 0)
		rgbStr = rgbStr[1,inf]
	endif
	r = str2num(StringFromList(0, rgbStr, ","))
	g = str2num(StringFromList(1, rgbStr, ","))
	b = str2num(StringFromList(2, rgbStr, ","))
	a = 65535
	if (itemsinList(rgbStr) > 3)
		a = str2num(StringFromList(3, rgbStr, ","))
	endif
End

Function LStylePopMenuProc(ctrlName,popNum,popStr) : PopupMenuControl
	String ctrlName
	Variable popNum
	String popStr

	Print "style:",popNum-1

	return 0
End
```

Popvalue Keyword

There are times when the displayed value cannot be determined and saved such that it can be displayed when the pop-up menu is recreated. For instance, because window recreation macros are evaluated in the root folder, a pop-up menu of waves may not contain the correct list when a panel is recreated. That is, the intention may be to have the menu show a particular wave from a data folder other than root. When the panel recreation macro runs, the function that lists waves will list waves in the root data folder. The desired selection may be wrong or nonexistent.

Similarly, a pop-up menu of fonts may need to display a particular font upon recreation on a different computer having a different list of fonts. The mode=*m* keyword probably won't pick the correct font from the new list.

The solution to these problems is to save the correct selection with the popvalue=*valueStr* keyword. The list function will not be executed when the menu is first created. If the menu is popped, the list function will be evaluated, and the correct list will be displayed then.

It is a good idea to set the mode=*m* keyword to the correct number, if it is known. That way, when the menu is popped the correct item is chosen.

Normally you can let Igor redraw the pop-up menu when it redraws the graph or control panel containing it. However, there are situations in which you may want to force the pop-up menu to be redrawn. This can be done using the **ControlUpdate** operation.

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

The **GetUserData** function for retrieving named user data.

The **ControlInfo** operation for information about the control.

The **ControlUpdate**, **WaveList**, and **TraceNameList** operations.

Special Characters in Menu Item Strings on page IV-133.

PopupMenuControl

`PopupMenuControl`

PopupMenuControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined pop-up menu control. See **Procedure Subtypes** on page IV-204 for details. See **PopupMenu** for details on creating a popup menu control.

PossiblyQuoteName

`PossiblyQuoteName(`*`nameStr`*`)`

The PossiblyQuoteName function returns the input name string if it conforms to the rules of standard wave or Data Folder names. If it does not, then the name is returned in single quotes. This is used when generating a command string that you will pass to the Execute command. You might get the input name string from a function such as **NameOfWave** or **CsrXWave**.

Examples

```
Print PossiblyQuoteName("wave0")         // prints wave0
Print PossiblyQuoteName("wave 0")        // prints 'wave 0'
```

Details

See **Programming with Liberal Names** on page IV-168 for an example.

Preferences

`Preferences [/Q] [`*`newPrefsState`*`]`

The Preferences operation sets or displays the state of user preferences.

User preferences affect the creation of *new* graphs, panels, tables, layouts, notebooks, procedure windows, and the command window. They also affect the appearance of waves appended to graphs and tables, and objects appended to layouts.

Parameters

If *newPrefsState* is present, it sets the state of user preferences as follows:

newPrefsState=0:	Preferences off (use factory defaults).
newPrefsState=1:	Preferences on.

If *newPrefsState* is omitted, the state of user preferences is printed in the history area.

Flags

/Q	Disables printing to the history.

Details

The Preferences operation sets the variable V_flag to the state of user preferences that were in effect *before* the Preferences command executed: 1 for on, 0 for off.

You can also set the state of Preferences with the Misc menu.

Under most circumstances we want procedures to be independent of preferences so that a particular procedure will do the same thing regardless of the state of preferences. To achieve this, preferences are automatically off when you initiate procedure execution. When execution is complete, the state of preferences is restored to what it was before.

If you want preferences to be in effect during procedure execution, you must turn it on with the Preferences operation.

If the preferences setting is changed by a procedure, the effect of the call is propagated down the calling chain. If a macro changes the preferences setting, that change is undone when the macro returns. If a function changes the preferences setting, the change persists after the function returns. However, even with a function, the changed preferences state does not persist when Igor regains control.

Examples

```
Function Test()
   Variable oldPrefState
   Preferences 1; oldPrefState=V_flag      // remember prefs setting
   Make wave0=x
   Display wave0                   // Display uses preferences
   Preferences oldPrefState        // put prefs back, like a macro would
End
```

See Also

Chapter III-18, **Preferences**.

PrimeFactors

`PrimeFactors [/Q]` ***`inNumber`***

PrimeFactors calculates the prime factors of *inNumber*. By default factors are printed in the history and are also stored in the wave W_PrimeFactors in the current data folder.

Flags

/Q	Suppresses printing of factors in the history area.

Details

The largest number that this operation can handle is 2^{32}-1.

See Also

gcd, **RatioFromNumber**

Print

`Print` `[`***`flags`***`]` ***`expression`*** `[,` ***`expression`***`]…`

The Print operation prints the evaluated expressions in the history area.

Parameters

An expression can be a wave, a numeric expression (e.g., `3*π/4`), a string expression (e.g., `"Today is " + date()`), or a individual structure element or an entire structure variable.

Flags

/C	Evaluates all numeric expressions as complex.
/D	Prints a greater number of digits.
/F	Prints numeric wave data (1D and 2D waves only) using “nice,” easily readable formatting.
/LEN=*len*	Sets the string break length to *len* number of bytes. The default is 200 and *len* is clipped to between 200 and 2500.
/S	Obsolete. Numeric results are printed with a moderate number of digits whether you use /S or not. To print more digits, use /D.
/SR	Prints a wave subrange for expressions that start as “*`waveName`*`[`“. Without /SR, such an expression is taken as the start of a numeric expression such as `wave[3]-wave[2]`. (You can still use *wave*[*pnt*] but only if it does not start the numeric expression.) Wave subrange printing is not done with /F. You can specify a single row or column using [*r*] syntax. For example, to print column 4 of a matrix, use: `Print mymat[][4]`

Details

Numeric expressions are always evaluated in double precision. The /D flag just controls the number of digits displayed.

Print determines if an expression is real, complex, or string from the first symbol in the expression. Usually this works fine, but occasionally Print guesses wrong and you may have to rearrange your expression. For example:

```
Print 1+cmplx(1,2)
```

will give an error because the first symbol, “1”, is real but the expression should be complex. Changing this to

```
Print cmplx(1,2)+1
```

will work.

Printing numeric or string expressions involving structure elements must not start with the structure element. Instead an appropriate numeric or string literal must appear first so that Igor can determine what kind of expression to compile. For example rather than

```
Print astruct.astring + "hello"
```

use

```
Print "" + astruct.astring + "hello"
```

Print breaks long strings into multiple lines. If there are no natural breaks (carriage returns or semicolons) within a default length, then it breaks the string arbitrarily.

The default line length is 200 bytes. You can override this using the /LEN flag. The maximum number of bytes that can be printed on a line in the history area is 2500.

When printing waves, you can use either formatted (specified by /F) or unformatted (default) methods. Unformatted output is in an executable syntax for each printed line: `wave={}`.

Note: Executing lines printed from floating point waves will not exactly reproduce the source data due to roundoff or insufficient digits in the printed output.

Printing formatted wave data gives easily (human) readable output, and works best for small 1D and 2D waves. If the data are too large or in an unsupported format (3D or greater, or the wave is text), then the output will be unformatted. Formatting is done using spaces, so the output will look best in a fixed-width font.

Printed wave data, both formatted and unformatted, are limited to no more than 100 lines of output. When the line limit is exceeded a warning message will be printed at the end of the truncated output. For text waves, output is limited to 50 bytes of each string element, and there is no warning when a string is truncated.

See Also

The **printf** operation.

The **PrintGraphs**, **PrintTable**, **PrintLayout** and**PrintNotebook** operations.

printf

printf *formatStr* [, ***parameter*** [, ***parameter***]…]

The printf operation prints formatted output to the history area.

Parameters

formatStr is a string which specifies the formatting of the output.

The type of the parameter, string or numeric, must agree with the corresponding conversion specification in *formatStr*, or else the results will be indeterminate.

The printf parameters can be numeric or string expressions. Numeric and string structure fields are allowed except that complex structure fields and non-numeric (e.g., WAVE, FUNCREF) structure fields are not allowed.

Details

The *formatStr* contains literal text and conversion specifications.

A conversion specification starts with the % character and ends with a conversion character (for example, g, e, f, d, or s as illustrated below). In between the % and the conversion character you may include one or more flag characters, a field width specifier, and a precision specifier. The first % corresponds to the first parameter, the second % corresponds to the second parameter, etc. If *formatStr* contains no % characters, no parameters are expected.

Here are some simple examples. `numVar` is a numeric variable and `strVar` is a string variable.

```
printf "The answer is: %g\r", numVar
printf "Created wave %s\r", strVar
printf "Created wave %s, %d points\r", strVar, numVar
```

%g is a general-purpose format (floating point or scientific notation) that represents the value of numVar. %d is an integer format that represents the value of `numVar`. %s specifies that the corresponding parameter (strVar) is a string.

The "\r" in these examples appends a carriage return to the end of the printed text.

Here is a complex example using all of these elements of a conversion specification:

```
printf "%+015.4f\r", 1e6*PI
```

This prints:

```
The answer is: +003141592.6536
```

"+" is a flag character that tells printf to put a + or - sign in front of the number.

"015" is a field width specifier that tells printf to print the number in a field of at least 15 bytes, padded with leading zeros. Using "15" instead of "015" would cause printf to pad with spaces before the + sign instead of zeros after it.

".4" is a precision specifier that tells printf to print four digits after the decimal point.

"f" tells printf to use a floating point format.

The most common conversions characters are “f” for floating point, “g” for general, “d” for decimal, and “s” for string. They are interpreted as for the printf() function in the C programming language.

The escape codes \t and \r represent the tab and return characters respectively. See **Escape Sequences in Strings** on page IV-14 for more information.

Printf Flag Characters

The supported flag characters and their meanings are as follows:

-	Left align the result in the field.
+	Put a plus or minus sign before the number.
<space>	Put a space before a positive number.
#	Specifies alternate form for e, f, g, and x formats.

The meaning of the precision specifier depends on the numeric format (%g, %e, %f, %d, etc.) being used:

e, E, f	Precision specifies number of digits after decimal point.
g, G	Precision specifies maximum number of significant digits.
d, o, u, x, X	Precision specifies minimum number of digits.

You can replace both the field width and precision specifiers with an asterisk. This gets the field width or precision specifier from a parameter. For example:

```
printf "%*.*f\r" 4, 3, 1e6*PI
```

means that the field width is 4 and the precision is 3. You could use numeric expressions instead of the literal numbers to control the field width and precision algorithmically.

Printf Conversion Characters

Here is a complete list of the conversion characters supported by printf:

To include a literal percent character, use two consecutive % characters, like this:

```
Variable percentage = 37.3
Printf "%.1f%% of participants had prior medical conditions", percentage
```

Igor also supports a non-C, WaveMetrics extension to the conversion characters recognized by printf. This conversion specification starts with “%W”. It is followed by a flag digit and a format character. For example,

```
printf "%W0Ps", 12.345E-6
```

f — Converts a numeric parameter as [-]ddd.ddd, where the number of digits after the decimal point is determined by the precision specifier and defaults to 6. If the # flag is present, a decimal point will be used even if there are no digits to the right of it.

This conversion character uses the “round-to-half-even” rule, also known as “banker’s rounding”. When the truncated digits are exactly 0.5000..., the quantity is rounded to an even number. For example:

```
Printf “%.0f\r”, 15.5         // Prints 16 (rounded up to even)
Printf “%.0f\r”, 16.5         // Prints 16 (rounded down to even)
```

e, E — Converts a numeric parameter as [-]d.ddde+/-dd, where the number of digits after the decimal point is determined by the precision specifier and defaults to 6. If you use “E” instead of “e” then printf uses a capital “E” in the number. If the # flag is present, a decimal point will be used even if there are no digits to the right of it.

g, G — Converts a numeric parameter using “f” or “e” style conversion depending on the magnitude of the number. “e” is used if the exponent is less than -4 or greater than the precision. “G” uses “f” or “E” style conversion. If the # flag is present, a decimal point will be used even if there are no digits to the right of it and trailing zeros will not be removed.

d, o, u — Converts a numeric parameter as a signed decimal integer, unsigned octal integer or unsigned decimal integer. The precision defaults to one and specifies the minimum number of digits to print.

These conversion characters use the “round-away-from-zero” rule, like Igor’s **round** function. For example:

```
Printf “%d\r”, 15.5           // Prints 16 (rounded away from zero)
Printf “%d\r”, 16.5           // Prints 17 (rounded away from zero)
```

x, X — Converts a numeric parameter as an unsigned hexadecimal integer, rounding floating point values. Also supports integer data up to 64 bits.

The “x” style uses lower case for the hexadecimal numerals “abcdef” where the “X” style uses upper case.

The precision defaults to one and specifies the minimum number of digits to print.

If the # flag is present, the string “0x” or “0X” is prepended to the number if it is not zero.

s — Converts a string parameter. If a precision is specified, it sets the maximum number of bytes from the string parameter to be printed.

As of Igor Pro 9.00, there is no limit to the length of the string parameter. It was limited to 2400 bytes in Igor Pro 8 and to 1000 bytes before that.

b — WaveMetrics extension. Converts a numeric parameter to binary.

c — Converts a numeric parameter to a single ASCII character.

% — Prints a % sign. No parameter is used.

%W — WaveMetrics extension. See description below.

prints 12.345000µs. In this example, the “%W0” introduces the WaveMetrics conversion specification. The “0” (zero) following the “W” is the flag digit. The “P” that follows is the format specifier character, which prints the number using a prefix, in this case, “µ”.

There is only one WaveMetrics format specifier character, “P”, which prints using a prefix such as µ, m, k, or M. It recognizes two flag-digits, “0” or “1”. Option “0” prints with no space between the numeric part and the prefix character while flag “1” prints with 1 space. Numbers greater than tera or less than femto print using a power of ten notation. Here are a few examples:

```
printf "%.2W0PHz", 12.342E6       // prints 12.34MHz
printf "%.2W1PHz", 12.342E6       // prints 12.34 MHz
printf "%.0W0Ps", 12.342E-6       // prints 12µs
printf "%.0W1Ps", 12.342E-9       // prints 12 ns
```

See Also

The **sprintf**, **fprintf,** and **wfprintf** operations; **Creating Formatted Text** on page IV-259 and **Escape Sequences in Strings** on page IV-14.

PrintGraphs

`PrintGraphs [`*`flags`*`]`*`graphSpec`*` [, `*`graphSpec`*`]`...

The PrintGraphs operation prints one or more graphs.

PrintGraphs prints one or more graphs on a single page from the command line or from a procedure. The graphs can be overlaid or positioned any way you want.

Parameters

The *graphSpec* specifies the name of a graph to print, the position of the graph on the page and some other options.

Flags

/C=*num*	Renders graphs in black and white (*num*=0) or in color (*num*=1; default).
/D	Disables high resolution printing. This flag is of use only on Macintosh. It has no effect on Windows.
/G=*grout*	Specifies grout, the spacing between objects, for tiling in prevailing units.
/I	Coordinates are in inches.
/M	Coordinates are in centimeters.
/R	Coordinates are in percent of page size (see **Examples**).
/PD[=*d*]	Displays print dialog. This allows the user to use Print Preview or to print to a file. If present the /PD flag must be the first flag. *d*=0: Default. Prints without displaying the Print dialog. *d*=1: Displays the Print dialog. /PD is equivalent to /PD=1. *d*=2: Displays the Print Preview dialog. Requires Igor Pro 7.00 or later.
/S	Stacks graphs.
/T	Tiles graphs.

Details

Graph coordinates are in inches (/I) or centimeters (/M) relative to the top left corner of the physical page. If none of these options is present, coordinates are assumed to be in points.

The form of a *graphSpec* is:

`graphName` `[`**`(`*`left`*`, `*`top`*`, `*`right`*`, `*`bottom`*`)`**`] [`**`/F=`*`f`***`] [`**`/T`**`]`

Here are some examples:

```
// Take size and position from window size and position.
PrintGraphs Graph0, Graph1

// Specify size and position explicitly.
PrintGraphs/I Graph0(1, 1, 6, 5)/F=1, Graph1(1, 6, 6, 10)/F=1
```

If the coordinates are missing and the /T or /S flags are present before *graphSpec* then the graphs are tiled or stacked. If the coordinates are missing but no /T or /S flags are present then the graph is sized and positioned based on its position on the desktop.

Finally there are these *graphSpec* options, which appear after the graph name:

/F=*f* Specifies a frame around the graph.

f=0:	No frame (default).
f=1:	Single frame.
f=2:	Double frame.
f=3:	Triple frame.
f=4:	Shadow frame.

/T Graph is transparent. This allows special effects when graphs are overlaid.

For this to be effective, the graph and its contents must also be transparent. Graphs are transparent only if their backgrounds are white. Annotations have their own transparent/opaque settings. PICTs may have been created transparent or opaque; an opaque PICT cannot be made transparent.

Examples

You can put an entire *graphSpec* into a string variable and use the string variable in its place. In this case the name of the string variable must be preceded by the $ character. This is handy for printing from a procedure and also keeps the PrintGraphs command down to a reasonable number of characters. For example:

```
String spec0, spec1, spec2
spec0 = "Graph0(1, 1, 6, 5)/F=1"
spec1 = "Graph1(1, 6, 6, 10)/F=1"
spec2 = ""                       // PrintGraphs will ignore spec2.
PrintGraphs/I $spec0, $spec1, $spec2
```

If you use a string for a *graphSpec* and that string contains no characters then PrintGraphs will ignore that *graphSpec*.

See Also

The **PrintSettings**, **PrintTable**, **PrintLayout** and **PrintNotebook** operations.

PrintLayout

PrintLayout [**/C=*num* /D**] ***winName***

The PrintLayout operation prints the named page layout window.

Parameters

winName is the window name of the page layout to print.

Flags

/C=*num* Renders graphs, tables, and annotations in black-and-white (*num*=0) or in color (*num*=1; default). It has no effect on pictures, which are colored independently.

/D Prints the layout at the default resolution of the output device. Otherwise it is printed at the highest resolution. This flag is of use only on Macintosh. It has no effect on Windows.

Details

Normally page layouts are printed at the highest available resolution of the output device (printer, plotter, or whatever). On Macintosh, it may not work properly at high resolution with some unusual output devices. If this happens, you can try using the /D flag to see if it works properly at the default resolution.

See Also

The **PrintSettings**, **PrintGraphs**, **PrintTable** and **PrintNotebook** operations.

PrintNotebook

`PrintNotebook` `[`***`flags`***`]` ***`notebookName`***

The PrintNotebook operation prints the named notebook window.

Parameters

notebookName is either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-92 for details on host-child specifications.

Flags

/B=*hiResMethod*	*Macintosh only*; this flag has no effect on Windows.	
	hiResMethod=1:	Print HiRes PICTs using high resolution bitmaps.
	hiResMethod=0:	Don't print HiRes PICTs using high resolution bitmaps.
	hiResMethod=-1:	Print using the default method. Prints HiRes PICTs using high resolution bitmaps and is the same as method 1.
/P=(*startPage*,*endPage*)	Specifies a page range to print. 1 is the first page.	
/S=*selection*	Controls what is printed.	
	selection=0:	Print entire notebook (default).
	selection=1:	Print selection only.

Details

If no /B flag is given, the default method of handling HiRes PICTs is used (/B=1). Printing of HiRes PICTs is not well supported on the Macintosh, so by default it prints them using temporary high resolution bitmaps. If a future version of the Mac OS improves in this respect, we will change the default method to print directly.

See Also

Chapter III-1, **Notebooks**.

The **PrintSettings**, **PrintGraphs**, **PrintTable** and **PrintLayout** operations.

PrintSettings

```
PrintSettings [/I /M /W=winName] [copySource=source, orientation=o,
   margins={left,top,right,bottom}, scale=s, colorMode=m, getPrinterList,
   getPrinter, setPrinter=printerNameStr, getPageSettings, getPageDimensions]
```

The PrintSettings operation gets or sets parameters associated with printing, such as a list of available printers or page setup information for a particular window.

An exception is the graphMode and graphSize keyword pair which affect printing of all graphs. This pair was added in Igor Pro 7.00.

Prior to Igor Pro 7.00, PrintSettings applied to a page layout affected the size and orientation of the layout page. In Igor Pro 7.00 and later, the size and orientation of the layout page are independent of print settings. See **Page Layout Page Sizes** on page II-478 for details.

When getting or setting page setup information, PrintSettings acts on a particular window called the destination window. The destination window is the top graph, table, page layout, or notebook window or the window specified by the /W flag.

PrintSettings can not act on page setup records associated with the command window, procedure windows, help windows, control panel, XOP windows, or any type of window other than graphs, tables, page layouts, and notebooks.

The PrintSettings operation services the keywords in the order shown above, not in the order in which they appear in the command. Thus, for example, the getPageSettings and getPageDimensions keywords report the settings after all other keywords are executed.

Flags

/I — Measurements are in inches. If both /I and /M are omitted, measurements are in points.

/M — Measurements are in centimeters. If both /I and /M are omitted, measurements are in points.

/W=*winName* — Acts on the page setup record of the graph, table, page layout, or notebook window identified by *winName*. If *winName* is omitted or if *winName* is `""`, then it used the page setup for the top window.

Keywords

colorMode=*m* — Sets the color mode for the page setup to monochrome (*m*=0) or to color (*m*=1).

This keyword does nothing on Macintosh because it is not supported by Mac OS X.

copySource=*source* — Copies page setup settings from the specified source to the destination window. *source* can be the name of a graph, table, page layout, or notebook window or it can be one of the following special keywords:

Default_Settings: Sets the page setup record to the default for the associated printer as specified by the printer driver.

Factory_Settings: Sets the page setup record to the WaveMetrics factory default. This is the page setup you get when creating a new window with user preferences turned off.

Preferred_Settings: Sets the page setup record to the user preferred page setup. This is the page setup you get when creating a new window with user preferences turned on. Because there is only one page setup for all graphs and one page setup for all tables, this has no effect when the destination window is a graph or table. It does work for layouts and notebooks.

getPageDimensions — Returns page dimensions via the string variable S_value, which contains keyword-value pairs that can be extracted using **NumberByKey** and **StringByKey**. See **Details** for keyword-value pair descriptions.

getPageSettings — Returns page setup settings in the string variable S_value, which contains keyword-value pairs that can be extracted using **NumberByKey** and **StringByKey**. See **Details** for keyword-value pair descriptions.

getPrinter — Returns the name of the selected printer for the destination window in the string variable S_value. On Macintosh the returned value will be `""` if the setPrinter keyword was never used on the destination window. This means that the window will use the operating system's "current printer".

getPrinterList — Returns a semicolon-separated list of printer names in the string variable S_value.

Mac OS X: Returns a list of printers added through Print Center.

Windows: Returns the names of any local printers and names of network printers to which the user has made previous connections.

graphMode=*g* — Sets the printing mode for graphs:

1: Fill page

2: Same size

3: Same aspect ratio

4: Custom size as set by graphSize keyword

5: Same size or shrink to fit page (default)

The graphMode keyword was added in Igor Pro 7.00.

graphSize={*left*, *top*, *width*, *height*}

Sets the custom graph size used when graphMode is 4. Parameters are in points unless /I or /M is used.

Invoking the graphSize keyword automatically sets the graphMode to 4.

left and *top* are clipped so that they are no smaller than the minimum allowed by the printer driver. *width* and *height* are not clipped.

This setting is not saved and is set to a default value when Igor starts.

The graphSize keyword was added in Igor Pro 7.00.

margins={*left*, *top*, *right*, *bottom*}

Sets the page margins. Dimensions are in points unless /I or /M is used.

This setting is ignored for notebook windows. Use the Notebook operation pageMargins keyword instead.

The margins are clipped so that they are no smaller than the minimum allowed by the printer driver and no larger than one-half the size of the paper.

The terms *left*, *top*, *right*, and *bottom* refer to the sides of the page after possible rotation for landscape orientation.

Passing zero for all four margins sets the margins to the minimum margin allowed by the printer.

On Macintosh only, passing -1 for all four margins sets the margins to whatever minimum margin is allowed by the printer, even if the printer is changed later. This is how Igor Pro behaved on Macintosh prior to the creation of the PrintSettings operation, when the minimum printer margins were always used.

orientation=*o* Sets the paper orientation to portrait (*o*=0) or to landscape (*o*=nonzero).

scale=*s* In Igor Pro 7 and later, the scale keyword returns an “unimplemented” error, unless *s*=100, because it is currently not supported. Let us know if this feature is important to you. Though *s*=100 does not generate an error, it does nothing. You can still set the scaling manually using the Page Setup dialog.

setPrinter=*printerNameStr*

Sets the selected printer for the destination window.

SetPrinter attempts to preserve orientation, margins, scale, and color mode but other settings may revert to the default state.

printerNameStr is a name as returned by the getPrinterList keyword and may not be identical to the name displayed in various dialogs. For example, on Mac OS X, the printer name “DESKJET 840C” is returned by getPrinterList as “DESKJET_840C”. The latter is the “Queue Name” displayed by the Mac OS X Print Center or Printer Setup Utility programs.

If *printerNameStr* is `""`, the printer for the destination window is set to the default state. This means different things depending on the operating system:

Mac OS X:	The destination window will use the operating system’s “current printer”, as if the setPrinter keyword had never been used.
Windows:	The destination window will use the system default printer.

If you receive an error when using setPrinter, use the getPrinterList keyword to verify that the printer name you are using is correct. Verify that the printer is connected and turned on.

Windows printer names are sometimes UNC names of the form “\\Server\Printer”. You must double-up backslashes when using a UNC name in a literal string. See **UNC Paths** on page III-451 for details.

Details

All graphs in the current experiment share a single page setup record so if you change the page setup for one graph, you change it for all graphs.

All tables in the current experiment share a single page setup record.

Each page layout window has its own page setup record.

Each notebook window has its own page setup record.

The keyword-value pairs for the getPageSettings keyword are as follows:

Keyword	Information Following Keyword
ORIENTATION:	0 if the page is in portrait orientation, 1 if it is in landscape orientation.
MARGINS:	The left, top, right, and bottom margins in points, separated by commas. These margins are ignored for notebook windows. Use the Notebook operation pageMargins keyword instead.
SCALE:	The page scaling expressed in percent. 50 means that the graphics are drawn at 50% of their normal size.
COLORMODE:	0 for black&white, 1 for color. This is not supported on Macintosh and always returns 1.

The keyword-value pairs for the getPageDimensions keyword are as follows:

Keyword	Information Following Keyword
PAPER:	The left, top, right, and bottom coordinates of the paper in points, separated by commas. The top and left are negative numbers so that the page can start at (0,0).
PAGE:	The left, top, right, and bottom coordinates of the page in points, separated by commas. The term page refers to the part of the paper inside the margins. The top/left corner of the page is always at (0, 0).
PRINTAREA:	The left, top, right, and bottom coordinates of the page in points, separated by commas. The print area is the part of the paper on which printing can occur, as determined by the printer. This is equal to the paper inset by the minimum supported margins. The top and left are negative numbers so that the page can start at (0,0).

Examples

For an example using the PrintSettings operation, see the PrintSettings Tests example experiment file in the “Igor Pro Folder:Examples:Testing” folder.

Here are some simple examples showing how you can use the PrintSettings operation.

```
Function GetOrientation(name)     // Returns 0 (portrait) or 1 (landscape)
   String name            // Name of graph, table, layout or notebook

   PrintSettings/W=$name getPageSettings
   Variable orientation = NumberByKey("ORIENTATION", S_value)
   return orientation
End

Function SetOrientationToLandscape(name)
   String name            // Name of graph, table, layout or notebook

   PrintSettings/W=$name orientation=1
End

Function/S GetPrinterList()
   PrintSettings getPrinterList
   return S_value
End

Function SetPrinter(destWinName, printerName)
   String destWinName, printerName

   PrintSettings/W=$destWinName setPrinter=printerName
   return 0
End
```

See Also

The **PrintGraphs**, **PrintTable**, **PrintLayout** and **PrintNotebook** operations.

PrintTable

PrintTable [**/P=(*startPage*,*endPage*) /S=*selection***] ***winName***

The PrintTable operation prints the named table window.

Parameters

winName is the window name of the table to print.

Flags

/P=(*startPage*,*endPage*)	Specifies a page range to print. 1 is the first page. If /P is omitted all pages are printed unless /S is used.
/S=*selection*	Controls what is printed. *selection*=0: Print entire table (default). *selection*=1: Print selection only.

See Also

Chapter II-12, **Tables**.

The **PrintSettings**, **PrintGraphs**, **PrintLayout** and **PrintNotebook** operations.

Proc

Proc *macroName*([***parameters***]**)** [**:*macro type***]

The Proc keyword introduces a macro that does not appear in any menu. Otherwise, it works the same as **Macro**. See **Macro Syntax** on page IV-118 for further information.

ProcedureText

ProcedureText(*macroOrFunctionNameStr* [**, *linesOfContext*** [**,**
   procedureWinTitleStr]]**)**

The ProcedureText function returns a string containing the text of the named macro or function as it exists in some procedure file, optionally with additional lines that are before and after to provide context or to collect documenting comments.

Alternatively, all of the text in the specified procedure window can be returned.

Parameters

macroOrFunctionNameStr identifies the macro or function. It may be just the name of a global (nonstatic) procedure, or it may include a module name, such as `"myModule#myFunction"` to specify the static function myFunction in a procedure window that contains a `#pragma ModuleName=myModule` statement.

If *macroOrFunctionNameStr* is set to "", and *procedureWinTitleStr* specifies the title of a single procedure window, then all of the text in the procedure window is returned.

linesOfContext optionally specifies the number of lines around the function to include in the returned string. The default is 0 (no additional contextual lines of text are returned). This parameter is ignored if *macroOrFunctionNameStr* is "" and *procedureWinTitleStr* specifies the title of a single procedure window.

Setting *linesOfContext* to a positive number returns that many lines before the procedure and after the procedure. Blank lines are not omitted.

Setting *linesOfContext* to -1 returns lines before the procedure that are not part of the preceding macro or function. Usually these lines are comment lines describing the named procedure. Blank lines are omitted.

Setting *linesOfContext* to -n, where n>1, returns at most n lines before the procedure that are not part of the preceding macro or function. Blank lines are not omitted in this case. n can be -inf, which acts the same as -1 but includes blank lines.

The optional *procedureWinTitleStr* can be the title of a procedure window (such as `"Procedure"` or `"File Name Utilities.ipf"`). The text of the named macro or function in the specified procedure window is returned.

You can use *procedureWinTitleStr* to select one of several static functions with identical names among different procedure windows, even if they do not use a `#pragma moduleName=myModule` statement.

Advanced Parameters

If `SetIgorOption IndependentModuleDev=1`, *procedureWinTitleStr* can also be a title followed by a space and, in brackets, an independent module name. In such cases ProcedureText retrieves function text from the specified procedure window and independent module. (See **Independent Modules** on page IV-238 for independent module details.)

For example, in a procedure file containing:

```
#pragma IndependentModule=myIM
#include <Axis Utilities>
```

A call to ProcedureText like this:

```
String text=ProcedureText("HVAxisList",0,"Axis Utilities.ipf [myIM]")
```

will return the text of the `HVAxisList` function located in the Axis Utilities.ipf procedure window, which is normally a hidden part of the `myIM` independent module.

You can see procedure window titles in this format in the Windows→Procedure Windows menu when `SetIgorOption IndependentModuleDev=1` and when an experiment contains procedure windows that comprise an independent module, as does `#include <New Polar Graphs>`.

procedureWinTitleStr can also be just an independent module name in brackets to retrieve function text from *any* procedure window that belongs to the named independent module:

```
String text=ProcedureText("HVAxisList",0,"[myIM]")
```

See Also

Regular Modules on page IV-236 and **Independent Modules** on page IV-238.

The **WinRecreation** and **FunctionList** functions.

ProcedureVersion

ProcedureVersion(*macroOrFunctionNameStr* [, *procedureWinTitleStr*])

The ProcedureVersion function returns the version number as specified by the first `#pragma version=versionNum` in the procedure file containing the named macro, function. Alternatively it can returh the version number of a specified procedure window.

The ProcedureVersion function was added in Igor Pro 9.01.

Parameters

macroOrFunctionNameStr identifies the macro or function. It may be just the name of a global (nonstatic) procedure, or it may include a module name, such as `"MyModule#MyFunction"` to specify the static function MyFunction in a procedure window that contains a *#pragma ModuleName=MyModule* statement.

If *macroOrFunctionNameStr* is "" and *procedureWinTitleStr* is not specified then the value of the first #pragma version statement in the procedure window containing the currently running macro or function is returned.

If *macroOrFunctionNameStr* is "" and *procedureWinTitleStr* is specified then the value of the first #pragma version statement in that procedure window is returned.

Details

The return value is rounded to a multiple of 0.001.

The returned value is 0 if the #pragma version statement is absent or the procedure window is "invisible" because the function is an independent module but `SetIgorOption IndependentModule=0`.

Example

```
#pragma version=1.2345        // 1 more digit than recommended
...
Function Test()
   Variable version = ProcedureVersion("") // Version of procedure file containing test
   Print version                           // 1.234
End
```

See Also

ProcedureText, **FunctionList**, **MacroList**

Procedure File Version Information on page IV-166, **The ModuleName Pragma** on page IV-54

SetIgorOption IndependentModuleDev=1 on page IV-239, **Invisible Procedure Windows Using Independent Modules** on page III-402

ProcGlobal

`ProcGlobal#`*`procPictureName`*

The ProcGlobal keyword is used with Proc Pictures to avoid possible naming conflicts with any other global pictures in the experiment. When you add a picture to an experiment using the Pictures dialog, such a picture is global in scope and may potentially have the same name as a Proc Picture. When a Proc Picture is global (and only then), you should use the ProcGlobal keyword to make sure that the Proc Picture is used with your code and to avoid confusion with pictures in the Pictures dialog.

See Also

See **Proc Pictures** on page IV-56 for details. **Pictures Dialog** on page III-510.

Project

`Project` `[`**`/C={`*`long,lat`*`}/M=`*`method`*` /P={`*`p1,p2`*`,…}`**`]` ***`longitudeWave, latitudeWave`***

The Project operation calculates projections of XY data, which most often are longitude and latitude waves of geographic coordinates. The output waves are W_XProjection and W_YProjection. Longitude and Latitude are in degrees.

Parameters

longitudeWave is the name of the wave supplying the longitude or equivalent coordinates. *latitudeWave* is the name of the wave supplying the latitude or equivalent coordinates.

Flags

/C={*long*,*lat*} Specifies longitude and latitude center of projection. By default *long*=0 and *lat*=90.

/M=*method* Indicates the type of projection. *method* can be one of the following:

0: Orthographic (default).
1: Stereographic.
2: Gnomonic.
3: General perspective.
4: Lambert equal area.
5: Equidistant.
6: Mercator.
7: Transverse Mercator.
8: Albers Equal Area conic.
9: Eckert IV (Igor Pro 9.00 or later)
10: Winkel III (Igor Pro 9.00 or later)

/P={*p1*,*p2*,…} One or more parameters required by a particular projection. See the following sections for parameters required by the various projections.

Gnomonic

Here there is one extra parameter that defines the boundaries based on the angle. The specific expression for the limit is that cos(c) in Eq. (5-3) of Snyder is greater than the specified parameter:

```
/P={cos(c)}
```

The actual transformation uses Eqs. (22-4) and (22-5) of Snyder with k' given by (22-3).

General Perspective

Here there is one extra parameter that defines the boundaries based on the angle. The specific expression for the limit is that cos(c) in Eq. (5-3) of Snyder is greater than the specified parameter.

The actual transformation uses Eqs. (22-4) and (22-5) with k' given by (22-3). Here we specify the height H is units of sphere radius. The tilt of the plane is specified by omega and gamma following the notation of Snyder page 175.

The parameters actually specified by the command are:

`/P={`*H*`,`*omega*`,`*gamma*`,`*deltax*`,`*deltay* `}`

H is the height (in radii) above the surface of the earth, *gamma* is the azimuth east of north of the Y axis, and *omega* is the tilt angle or the angle between the projection plane and the tangent plane. The x output will be limited to ± *deltax* and the y output will be limited to the range ± *deltay*.

Mercator

This projection requires the following parameters:

`/P={`*minLong*`,`*maxLong*`,`*minLat*`,`*maxLat*`}`

If /P is not specified, the default is {0,360,-90,90}

Note that this projection flips the sign of y when cos(longitude-long_0) changes sign. If you are plotting a continuous path in which consecutive points exhibit the sign change, you should add a NaN entry in the wave so that the path does not wrap.

Albers Equal Area Conic

This projection requires:

/P={*minLong*, *maxLong*, *minLat*, *maxLat*, *Phi1*, *Phi2*}

Phi1 and *Phi2* are the specification of the two standard parallels, the other four parameters determine the boundary of the map area for display.

References

Snyder, John P., *Map Projections—A Working Manual*, U.S.G.S. Professional Paper 1395, U.S. Government Printing Office, Washington D.C., 1987, reprinted 1989, 1994, 1997 with corrections.

See Also

“Transforming Data into a Common Spatial Reference” in the “IgorGIS Help” file.

Prompt

`Prompt` ***`variableName`*****`,`** ***`titleStr`*** `[`**`, popup,`** ***`menuListStr`***`]`

The Prompt command is used in functions for the simple input dialog and in macros for the missing parameter dialog. Prompt supplies text to describe *variableName* to the user, and optionally provides a pop-up menu of choices for the value of *variableName*.

Parameters

variableName is the name of a macro input parameter or function variable.

titleStr is a string or string expression containing the text to present in the dialog to describe what *variableName* is. *titleStr* is limited to 255 bytes.

The optional keyword `popup` is used to provide a pop-up list of choices for the values of *variableName*. If `popup` is used, then *menuListStr* is required.

menuListStr is a string or string expression that contains a semicolon-separated list of choices for the value of *variableName*. If *variableName* is a string, choosing from this list will set the string to the selection. If it is a numeric variable, then it is set to the item number of the selection (if the first item is selected, the numeric variable is set to 1, etc.).

Details

In macros, there must be a blank line after the set of input parameter declarations and prompt statements and there must not be any blank lines within the set.

In user-defined functions, Prompt may be used anywhere within the body of the function, but must precede any DoPrompt that uses the Prompt variable.

menuListStr may be continued on succeeding lines only in macros, as long as no comment is appended to the Prompt line. The additional lines should start with a semicolon, and are appended to the *menuListStr*s on preceding lines.

See Also

For use in user-defined functions, see **The Simple Input Dialog** on page IV-144.

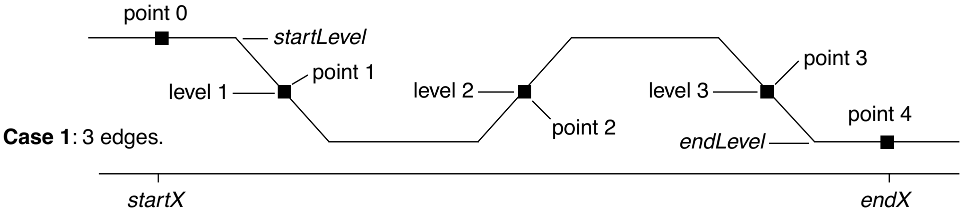
For use in macros, see **The Missing Parameter Dialog** on page IV-121.

For use in functions and macros, see the **DoPrompt** and **popup** keywords.

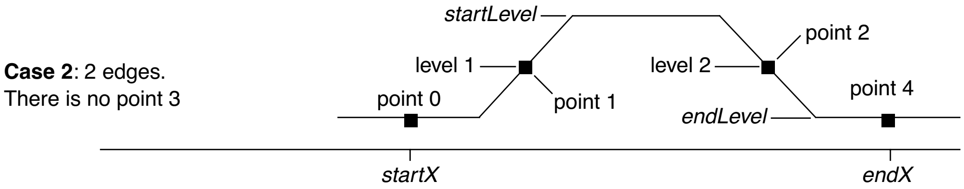
PulseStats

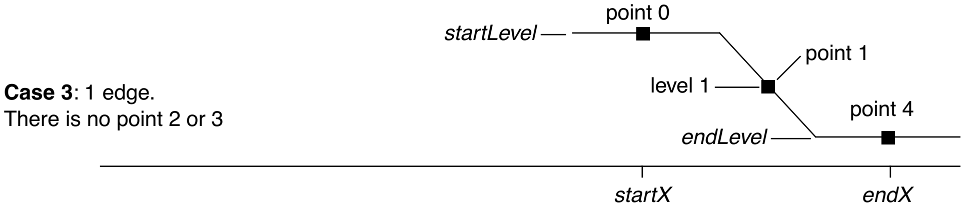
`PulseStats [`*`flags`*`]` ***`waveName`***

The PulseStats operation produces simple statistics on a region of the named wave that is expected to contain three edges as shown below. If more than three edges exist, PulseStats works on the first three edges it finds.



PulseStats handles other cases in which there are only one or two edges.





Flags

/A=*n* Determines *startLevel* and *endLevel* automatically by averaging *n* points centered at *startX* and *endX*. This does not work in case 2, which requires that you use the /L flag. Default is /A=1.

/B=*box* Sets box size for sliding average. This should be an odd number. If /B=*box* is omitted or *box* equals 1, no averaging is done.

/F=*f* Specifies levels 1, 2, and 3 as a fraction of (*endLevel-startLevel*):

```
level1 = level2 = level3 = f*(endLevel-startLevel) + startLevel
```

f must be between 0 and 1. The default value is 0.5 which sets the levels to midway between the base levels.

/L=(*startLevel*, *endLevel*)

Sets *startLevel* and *endLevel* explicitly.

/M=*dx* Sets minimum edge width. Once an edge is found, the search for the next edge starts *dx* units beyond the found edge. Default *dx* is 0.

/P Output edge locations (see **Details**) are set in terms of point number. If /P is omitted, edge locations are set in terms of X values.

/Q Prevents results from being printed in history and prevents error if edge is not found.

/R=(*startX*,*endX*)	Specifies an X range of the wave to search. You may exchange *startX* and *endX* to reverse the search direction.
/R=[*startP*,*endP*]	Specifies a point range of the wave to search. You may exchange *startP* and *endP* to reverse the search direction. If you specify the range as /R=[*startP*] then the end of the range is taken as the end of the wave. If /R is omitted, the entire wave is searched.
/T=*dx*	Forces search in two directions for a possibly more accurate result. *dx* controls where the second search starts.

Details

The /B=box, /T=dx, /P and /Q flags behave the same as for the **FindLevel** operation.

PulseStats considers a region of the input wave between two X locations, called *startX* and *endX*. *startX* and *endX* are set by the /R=(*startX*,*endX*) flag. If this flag is missing, *startX* and *endX* default to the start and end of the entire wave.

The *startLevel* and *endLevel* values define the base levels of the pulse. You can explicitly set these levels with the /L=(*startLevel*, *endLevel*) flag or you can let PulseStats find the base levels for you by using the /A=*n* flag. With this flag, PulseStats determines *startLevel* and *endLevel* by averaging *n* points centered at *startX* and at *endX*. In case 2, you must use /L=(*startLevel*, *endLevel*) since *startLevel* is not at point 0.

Given *startLevel* and *endLevel* and an *f* value (which you can set with the /F=*f* flag), PulseStats computes level1, level2 and level3 which are always equal. With the default *f* value of 0.5, level1 is midway between *startLevel* and *endLevel*.

With these levels defined, PulseStats searches the wave from *startX* to *endX* looking for one, two or three level crossings. PulseStats sets the following variables:

`V_flag`	0: All three level crossings were found. 1: One or two level crossings were found. 2: No level crossings were found.
`V_PulseLoc1`	X location where level1 was found.
`V_PulseLoc2`	X location where level2 was found.
`V_PulseLoc3`	X location where level3 was found.
`V_PulseLvl0`	*startLevel* value.
`V_PulseLvl123`	Level1 value that is the same as level2 and level3.
`V_PulseLvl4`	*endLevel* value.
`V_PulseAmp4_0`	Pulse amplitude (*endLevel* - *startLevel*).
`V_PulseWidth2_1`	Left pulse width (x distance between point 2 and point 1).
`V_PulseWidth3_2`	Right pulse width (x distance between point 3 and point 2).
`V_PulseWidth3_1`	Pulse period (x distance between point 3 and point 1).
`V_PulsePolarity`	Trend of the edge at point 1 (-1 if decreasing, +1 if increasing).

X locations and distances are in terms of the X scaling of the source wave, unless you use the /P flag in which case they are in terms of point number.

If any level crossings are missing then PulseStats sets the associated variables to NaN (Not a Number). If one crossing is missing, variables depending on point 3 are set to NaN. If two crossings are missing, variables depending on points 2 and 3 are set to NaN. If all crossings are missing, variables depending on points 1, 2, and 3 are set to NaN. You can use the numtype function to test a variable to see if it is NaN.

The PulseStats operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

See Also

The **FindLevel** operation about the /B=*box*, /T=*dx*, /P and /Q flags, **EdgeStats** and the **numtype** function.

PutScrapText

`PutScrapText textStr`

The PutScrapText operation places *textStr* on the Clipboard (aka “scrap”). This text will be used when the user subsequently chooses Paste from the Edit menu.

Details

All contents of the Clipboard (including pictures) are cleared before the text is placed there.

Examples

Put two lines of text into the Clipboard:

```
String text = "This is the first line.\rAnd this is the second."
PutScrapText text
```

Empty the Clipboard:

```
PutScrapText ""
```

See Also

The **GetScrapText** function and the **SavePICT** operation.

pwd

`pwd`

The pwd operation prints the full path of the current data folder to the history area. It is equivalent to Print GetDataFolder(1).

pwd is named after the UNIX "print working directory" command.

See Also

GetDataFolder, **cd**, **Dir**, **Data Folders** on page II-107

q

`q`

The q function returns the current column index of the destination wave when used in a multidimensional wave assignment statement. The corresponding scaled column index is available as the **y** function.

Details

Unlike **p**, outside of a wave assignment statement, q does not act like a normal variable.

See Also

Waveform Arithmetic and Assignments on page II-74.

For other dimensions, the **p**, **r**, and **s** functions.

For scaled dimension indices, the **x**, **y**, **z** and **t** functions.

qcsr

`qcsr(cursorName [, graphNameStr])`

The qcsr function can be used with cursors on images or waterfall plots to return the column number. It can also be used with free cursors to return the relative Y coordinate.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

See Also

The **hcsr**, **pcsr**, **vcsr**, **xcsr**, and **zcsr** functions.

Programming With Cursors on page II-321.

Quit

`Quit [/N/Y]`

The Quit operation quits Igor Pro.

Flags

/N	Quits without saving changes and without dialog.
/Y	Saves current experiment before quitting without putting up dialog unless current experiment is “Untitled”.

r

`r`

The r function returns the current layer index of the destination wave when used in a multidimensional wave assignment statement. The corresponding scaled layer index is available as the **z** function.

Details

Unlike **p**, outside of a wave assignment statement, r does not act like a normal variable.

See Also

Waveform Arithmetic and Assignments on page II-74. For other dimensions, the **p**, **q**, **s**, and **t** functions. For scaled dimension indices, the **x**, **y**, **z**, and **t** functions.

r2polar

`r2polar(z)`

The r2polar function returns a complex value in polar coordinates derived from the complex value *z*, which is assumed to be in rectangular coordinates. The magnitude is stored in the real part and the angle (in radians) is stored in the imaginary part of the returned complex value.

Examples

Assume waveIn and waveOut are complex.

```
waveOut= r2polar(waveIn)
```

sets each point of waveOut to the polar coordinates derived from the real and imaginary parts of waveIn.

You may get unexpected results if the number of points in waveIn differs from the number of points in waveOut.

See Also

The functions **cmplx**, **conj**, **imag**, **p2rect**, and **real**.

RatioFromNumber

`RatioFromNumber [flags] num`

The RatioFromNumber operation computes two integers whose ratio is equal to *num* ± *maxError* (/MERR flag). The ratio is returned in V_numerator and V_denominator.

Parameters

num is the number to approximate by V_numerator/V_denominator.

Flags

/MERR=*maxError*	Specifies the maximum tolerable error. The computed ratio differs from *num* by no more than *maxError* (default value is *num**1e-6). *maxError* must be a value between 0 and *num*. See Details about setting *maxError* to 0.

/MITS = *maxIterations* Keeps returned values small by specifying a small number for *maxIterations*.

maxIterations must be a value between 1 and 32767 (default is 100).

/V[=*v*] Prints output variables to history.

v=1: Prints variables (same as /V).
v=0: Nothing printed (same as no /V).

Details

The ratio is computed by continued fraction expansion and recurrence relations for the convergents and checking num - (V_numerator/V_denominator) against *maxError*.

Setting *maxError* = 0 computes a maximally accurate ratio. The returned values can be surprisingly large:

```
RatioFromNumber/V/MERR=0 (1/1666)
  V_numerator= 4398046511104; V_denominator= 7.3271454874993e+15;
  ratio= 0.00060024009603842; V_difference= 0;
```

Using the default /MERR returns the expected 1 and 1666. The difference is attributable to floating-point roundoff errors.

The ratio is computed by continued fraction expansion and recurrence relations for the convergents and checking *num* - (V_numerator/V_denominator) against /MERR.

Output Variables

RatioFromNumber sets the following output variables:

`V_difference` V_numerator/V_denominator - *num* (positive if the approximation is too big).

`V_flag` 0: V_difference less than or equal to /MERR.
1: V_difference greater than /MERR.

`V_numerator, V_denominator`

Values for the numerator and denominator. The ratio of V_numerator/V_denominator approximates *num*.

`V_iterations` The number of iterations actually used.

RatioFromNumber prints the output variables if you specify /V or /V=1 but only when running in the main thread.

Examples

```
RatioFromNumber/V pi
  V_numerator= 355; V_denominator= 113; ratio= 3.141592920354;
  V_difference= 2.6676418940497e-07; V_iterations= 3;
```

```
RatioFromNumber/V/MITS=2 pi
  V_numerator= 22; V_denominator= 7; ratio= 3.1428571428571;
  V_difference= 0.0012644892673497; V_iterations= 1;
```

See Also

gcd, **trunc**, **PrimeFactors**

Rect

The Rect structure is used as a substructure usually to store the coordinates of a window or control.

```
Structure Rect
   Int16 top
   Int16 left
   Int16 bottom
   Int16 right
EndStructure
```

RectF

The RectF structure is the same as Rect but with floating point fields.

```
Structure RectF
   float top
   float left
   float bottom
   float right
EndStructure
```

ReadVariables

`ReadVariables`

The ReadVariables operation reads variables into an experiment.

ReadVariables is used automatically when you open an experiment. You need not invoke it.

real

`real(`*`z`*`)`

The real function returns the real component of the complex value *z*.

See Also

The functions **cmplx**, **conj**, **imag**, **p2rect**, and **r2polar**.

Redimension

`Redimension` `[`***`flags`***`]` ***`waveName`*** `[,` ***`waveName`***`]`...

The Redimension operation remakes the named waves, preserving their contents as much as possible.

Flags

/B Converts waves to 8-bit signed integer or unsigned integer if /U is present.

/C Converts real waves to complex.

/D Converts single precision waves to double precision.

/E=*e* Controls the redimension mode:

e=0: No special action (default).

e=1: Force reshape without converting or moving data.

e=2: Perform endian swap. See **FBinRead** for a discussion of endian byte ordering.

/I Converts waves to 32-bit signed integer or unsigned integer if /U is present.

/L Converts waves to 64-bit signed integer or unsigned integer if /U is present. Requires Igor Pro 7.00 or later.

/N=*n* *n* is the new number of points each wave will have. Multidimensional waves are converted to 1 dimension. If n =-1, the wave is converted to a 1-dimensional wave with the original number of rows.

/N=(*n1*, *n2*, *n3*, *n4*)

n1, *n2*, *n3*, *n4* specify the number of rows, columns, layers, and chunks each wave will have. Trailing zeros can be omitted (e.g., /N=(*n1*, *n2*, 0, 0) can be abbreviated as /N=(*n1*, *n2*)). If any dimension size is to remain unchanged, pass -1 for that dimension.

/R Converts complex waves to real by discarding the imaginary part.

/S Converts double precision waves to single precision.

/U Converts integer waves to unsigned.

/W Converts waves to 16-bit integer (unsigned integer if /U is present).

/Y=*type* Specifies wave data type. See details below.

Wave Data Types

As a replacement for the above number type flags you can use /Y=*numType* to set the number type as an integer code. See the **WaveType** function for code values. Do not use /Y in combination with other type flags. This technique cannot be used to change the number type without changing the real/complex setting.

Details

The waves must already exist. New points in waves that are extended are zeroed.

In general, Redimension does not move data from one dimension to another. For instance, if you have a 6x6 matrix wave, and you would like it to be 3x12, the rows have been shortened and the data for the last three rows is lost.

As a special case, if converting to or from a 1D wave, Redimension will leave the data in place while changing the dimensionality of the wave. For example, you can use Redimension to convert a 36-element 1D wave into a 6x6 matrix in which the elements in the first column (column 0) are the first 6 elements of the 1D wave, the elements of the second column are the next 6, etc. When redimensioning from a 1D wave, columns are filled first, then layers, followed by chunks.

Examples

Reshaping a 1D wave having 4 elements to make a 2x2 matrix:

```
Make/N=4 vector=x
Redimension/N=(2,2) vector
```

See Also

Make, **DeletePoints**, **InsertPoints**, **Concatenate**, **SplitWave**

Remez

Remez [/N=*num* /Q[=*iter*]] *frWave*, *wtWave*, *gridWave*, *coefsWave*

The Remez operation calculates the coefficients for digital filters given a desired frequency response as input.

Remez is primarily used for the MPR filter feature of the Igor Filter Design Laboratory (IFDL) package.

Parameters

frWave contains the desired response.

wtWave contains the weight function array. For a differentiator, the weight function is inversely proportional to frequency.

gridWave contains the frequencies corresponding to each point in *frWave* and *wtWave*. Its values range from 0 to 0.5 with gaps where the band edges occur.

coefsWave receives the resulting coefficients. Its length defines the number of coefficients (nfilt in the IEEE program referenced below).

Flags

/N=*mode*	*mode*=0: Selects multiple passband/stopband filter (default). *mode*=1: Selects differentiator or Hilbert transform filter.
/Q[=*iter*]	Determines if execution stops if the filter doesn't converge. If you omit /Q, execution stops if the filter doesn't converge. If you specify /Q or /Q=0, execution continues if the filter doesn't converge, regardless of the number of iterations. For *iter*>=1, execution stops if the filter fails to converge in *iter* iterations or less. If the filter does converge after *iter* iterations, execution does stop. Use /Q=3 to stop execution for serious errors (after only 1, 2, or 3 iterations) but not for minor errors (after 4 or more iterations).

Details

Remez returns symmetrical coefficients suitable for use with **FilterFIR** in *coefsWave*.

The algorithm is based upon the McClellan-Parks-Rabiner Fortran program as found in the IEEE and Elliot references cited below.

If the filter converged, Remez sets V_Flag to 0. Otherwise it sets it to the number of iterations before it failed.

Example

This example specifies a length 41 lowpass filter with passband 0 to 0.14 x fs and stopband 0.18 x fs to 0.5 x fs. The passband weight is equal to the stopband weight.

```
Make/O/N=41 coefs = NaN
Make/O/N=(41*16) fr, wt, grid
grid = 0.5*p/numpnts(grid)       // Frequencies where fr and wt define desire response
wt = 1
fr[0,0.14*numpnts(fr)] = 1       // Low pass from 0 to 0.14 x fs

// Remove transition frequencies
DeletePoints 0.14*numpnts(fr), 0.04*numpnts(fr), fr, wt, grid

// Compute filter coefs
Remez fr, wt, grid, coefs

// Analyze the filter's frequency response
FFT/OUT=3/PAD={256}/DEST=coefs_FFT coefs

// Display filter response for 1Hz sample rate
Display coefs_FFT
```

References

J. H. McClellan, T.W. Parks, and L. R. Rabiner, *A computer program for designing optimum FIR linear phase digital filters*. IEEE Transactions on Audio and Electroacoustics, AU-21, 506-526 (December 1973).

L. R. Rabiner, J. H. McClellan, and T.W. Parks, *FIR digital filter design techniques using weighted Chebyschev approximation*, Proc. IEEE 63, 595-610 (April 1975)

Elliot, Douglas F.,contributing editor, *Handbook of Digital Signal Processing Engineering Applications*, Academic Press, San Diego, CA, 1987.

IEEE Digital Signal Processing Committee, Editor, *Programs for Digital Signal Processing*, IEEE Press, New York, 1979 .

See Also

FMaxFlat, **FilterFIR**

Remove

`Remove`

When interpreting a command, Igor treats the Remove operation as **RemoveFromGraph**, **RemoveFromTable**, or **See Also**, depending on the target window. This does not work when executing a user-defined function. Therefore, we recommend that you use **RemoveFromGraph**, **RemoveFromTable**, or **RemoveLayoutObjects** rather than Remove.

RemoveByKey

`RemoveByKey(`*`keyStr, kwListStr`*`[, `*`keySepStr`*` [, `*`listSepStr`*` [, `*`matchCase`*`]]])`

The RemoveByKey function returns *kwListStr* after removing the keyword-value pair specified by *keyStr*. *kwListStr* should contain keyword-value pairs such as `"KEY=value1,KEY2=value2"` or `"Key:value1;KEY2:value2"`, depending on the values for *keySepStr* and *listSepStr*.

Use RemoveByKey to remove information from a string containing a `"key1:value1;key2:value2;"` or `"key1=value1,key2=value2,"` style list such as those returned by functions like **AxisInfo** or **TraceInfo**.

If *keyStr* is not found then *kwListStr* is returned unchanged.

keySepStr, *listSepStr,* and *matchCase* are optional; their defaults are ":", ";", and 0 respectively.

Details

kwListStr is searched for an instance of the key string bound by *listSepStr* on the left and a *keySepStr* on the right. The key, the *keySepStr*, and the text up to and including the next *listSepStr* (if any) are removed from the returned string.

If the resulting string contains only *listSepStr* characters, then an empty string (`""`) is returned.

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are usually case-insensitive. Setting the optional *matchCase* parameter to 1 makes the comparisons case sensitive.

In Igor6, only the first byte of *keySepStr* and *listSepStr* was used. In Igor7 and later, all bytes are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *matchCase* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print RemoveByKey("AKEY", "AKEY:123;BKEY:val")        // prints "BKEY:val"
Print RemoveByKey("AKEY", "akey=1;BK=b;", "=")        // prints "BK=b;"
Print RemoveByKey("AKEY", "AKEY=1,BK=b,", "=", ",")   // prints "BK=b,"
Print RemoveByKey("ckey","CKEY:1;BKEY:2")             // prints "BKEY:2"
Print RemoveByKey("ckey","CKEY:1;BKEY:2",":",";",1)   // prints "CKEY:1;BKEY:2"
```

See Also

The **NumberByKey**, **StringByKey**, **ReplaceNumberByKey**, **ReplaceStringByKey**, **ItemsInList**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

RemoveContour

`RemoveContour` `[`***`/W=winName`***`]` ***`contourInstanceName`*** `[,` ***`contourInstanceName`***`]`...

The RemoveContour operation removes the traces, and releases memory associated with the contour plot of *contourInstanceName* in the target or named graph.

Parameters

contourInstanceName is usually simply the name of a wave. More precisely, *contourInstanceName* is a wave name, optionally followed by the # character and an instance number to identify which contour plot of a given wave is to be removed.

Flags

/ALL	Removes all contour plots from the graph. Any contour name parameters listed are ignored. /ALL was added in Igor Pro 9.00.
/W=*winName*	Removes contours from the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

If the axes used by the contour plot are no longer in use, they will also be removed.

A contour instance name in a string can be used with the $ operator to specify *contourInstance*.

Examples

```
Display;AppendMatrixContour zw      //new graph, contour of zw matrix
AppendMatrixContour zw              //two contours of zw
RemoveContour zw#1                  //remove the second contour
```

See Also

The **AppendMatrixContour** and **AppendXYZContour** operations.

RemoveEnding

RemoveEnding(*str* [, *endingStr*])

The RemoveEnding function removes one character from the end of *str*, or it removes the *endingStr* from the end of *str*.

The RemoveEnding function returns *str* with *endingStr* removed from the end. If you omit *endingStr*, it returns *str* with one grapheme removed from the end.

Details

If you specify *endingStr*, RemoveEnding compares it to the end of *str* using case-insensitive comparison. If there is a match, RemoveEnding returns the contents of *str* up to *endingStr*. If there is no match, RemoveEnding returns the entirety of *str*.

If you omit *endingStr*, RemoveEnding returns *str* with the last grapheme removed. A grapheme is whatever visually appears to be one character even if it consists of more than one character. In "ABC", the last grapheme is "C" which is also the last character. In "ABÇ", "Ç" consists of two characters: a C character and a "combining cedilla" character; RemoveEnding removes "Ç" which is the last grapheme.

Examples

```
Print RemoveEnding("123")                      // Prints "12"
Print RemoveEnding("ABÇ")                      // Prints "AB"
Print RemoveEnding("no semi" , ";")            // Prints "no semi"
Print RemoveEnding("trailing semi;" , ";")     // Prints "trailing semi"
Print RemoveEnding("file.txt" , ".TXT")        // Prints "file"
```

See Also

The **CmpStr** and **ParseFilePath** functions.

RemoveFromGizmo

RemoveFromGizmo [***flags***]

The RemoveFromGizmo operation removes the specified object from the specified list and optionally performs an update.

Documentation for the RemoveFromGizmo operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "RemoveFromGizmo"
```

RemoveFromGraph

RemoveFromGraph [**/W=*winName*/Z**] ***traceName*** [, ***traceName***]…

The RemoveFromGraph operation removes the specified wave traces from the target or named graph. A trace is a representation of the data in a wave, usually connected line segments.

Parameters

traceName is usually just the name of a wave.

More generally, *traceName* is a wave name, optionally followed by the # character and an instance number - for example, wave0#1. See **Instance Notation** on page IV-20 for details.

Flags

/ALL	Removes all non-contour traces from the graph. Any trace name parameters listed are ignored. /ALL was added in Igor Pro 9.00.
/W=*winName*	Removes traces from the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.
/Z	Suppresses errors if specified trace or image is not on the graph.

Details

Up to 100 *traceName*s may be specified, subject to the 2500 byte command length limit.

If the axes used by the given trace are not in use after removing the trace, they will also be removed.

A string containing a trace name can be used with the $ operator to specify *traceName*.

Specifying $"#0" for *traceName* removes the first trace in the graph. $"#1" removes the second trace in the graph, and so on. $"" is equivalent to $"#0".

Note that removing all the contour traces from a contour plot is not the same as removing the contour plot itself. Use the **RemoveContour** operation.

Examples

The command:

```
Display myWave,myWave;Modify mode(myWave#1)=6
```

appends two instances of myWave to the graph.The first/backmost instance of myWave is instance 0, and its trace name is just myWave as a synonym for myWave#0. The second or frontmost instance of myWave is myWave#1 and it is displayed with the cityscape mode.

To remove the second instance from the graph requires the command:

```
RemoveFromGraph myWave#1
```

or

```
String MyTraceName="myWave#1"
RemoveFromGraph $MyTraceName
```

See Also

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

RemoveFromLayout

RemoveFromLayout *objectSpec* [, ***objectSpec***]...

Deprecated — use **RemoveLayoutObjects**.

The RemoveFromLayout operation removes the specified objects from the top layout.

Parameters

objectSpec is either an object name (e.g., Graph0) or an *objectName* with an instance (e.g., Graph0#1). An instance is needed only if the same object appears in the layout more than one time. Graph0 is equivalent to Graph0#0 and Graph0#1 refers to the second instance of Graph0 in the layout.

See Also

The **RemoveLayoutObjects** operation.

RemoveFromList

RemoveFromList(*itemOrListStr, listStr* [, ***listSepStr*** [, ***matchCase***]]**)**

The RemoveFromList function returns *listStr* after removing the item or items specified by *itemOrListStr*. *listStr* should contain items separated by *listSepStr* which typically is ";".

If *itemOrListStr* contains multiple items, they should be separated by the *listSepStr* character, too.

Use RemoveFromList to remove item(s) from a string containing a list of items separated by a string (usually a single ASCII character), such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

If all items in *itemOrListStr* are not found or if any of the arguments is `""` then *listStr* is returned unchanged (unless *listStr* contains only list separators, in which case an empty string is returned).

listSepStr and *matchCase* are optional; their defaults are ";" and 1 respectively.

Details

itemStr may have any length.

listStr is searched for an instance of the item string(s) bound by *listSepStr* on the left and right. All instances of the item(s) and any trailing *listSepStr* (if any) are removed from the returned string.

If the resulting string contains only *listSepStr* characters, then an empty string (`""`) is returned.

listStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *listSepStr* are case-sensitive. Searches for *items* in *itemOrListStr* are usually case-sensitive. Setting the optional *matchCase* parameter to 0 makes the comparisons case insensitive.

In Igor6, only the first byte of *listSepStr* was used. In Igor7 and later, all bytes are used.

If *matchCase* is specified, then *listSepStr* must also be specified.

Examples

```
Print RemoveFromList("wave1", "wave0;wave1;")         // prints "wave0;"
Print RemoveFromList("wave1", ";wave1;;;;")           // prints ""
Print RemoveFromList("KEY=joy", "AX=3,KEY=joy", ",")  // prints "AX=3,"
Print RemoveFromList("fred", "fred\twilma", "\t")     // prints "wilma"
Print RemoveFromList("fred;barney","fred;wilma;barney")// prints "wilma;"
Print "X"+RemoveFromList("",";;;;")+"Y"               // prints "XY"
Print RemoveFromList("FRED", "fred;wilma")        // prints "fred;wilma"
Print RemoveFromList("FRED", "fred;wilma", ";", 0)    // prints "wilma"
```

See Also

The **FindListItem**, **FunctionList**, **ItemsInList**, **RemoveByKey**, **RemoveListItem**, **StringFromList**, **StringList**, **TraceNameList**, **UpperStr**, **VariableList**, and **WaveList** functions.

RemoveFromTable

RemoveFromTable [**/W=*winName***] ***columnSpec*** [, ***columnSpec***]…

The RemoveFromTable operation removes the specified columns from the top table.

Parameters

*columnSpec*s are the same as for the **Edit** operation; usually they are just the names of waves.

Flags

/W=*winName*

Removes columns from the named table window or subwindow. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

See Also

Edit about *columnSpec*s, and **AppendToTable**.

RemoveImage

RemoveImage [**/W=*winName*/Z**] ***imageInstance*** [, ***imageInstance***]…

The RemoveImage operation removes the given image from the target or named graph.

Parameters

imageInstance is usually simply the name of a wave. More precisely, *imageInstance* is a wave name, optionally followed by the # character and an instance number to identify which image of a given wave is to be removed.

Flags

/ALL — Removes all image plots from the graph. Any image name parameters listed are ignored. /ALL was added in Igor Pro 9.00.

/W=*winName* — Removes an image from the named graph window or subwindow. When omitted, action will affect the active window or subwindow. Must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/Z — Suppresses errors if specified image is not on the graph.

Details

If the axes used by the given image are not in use after removing the image, they will also be removed.

An image name in a string can be used with the $ operator to specify *imageInstance*.

See Also

The **AppendImage** operation.

RemoveLayoutObjects

`RemoveLayoutObjects` `[/PAGE=`*`page`***`/W=`*`winName`*`/Z`**`]` ***`objectSpec`*** `[,` ***`objectSpec`***`]`

The RemoveLayoutObjects operation removes the specified object or objects from the top page layout, or from the layout specified by the /W flag. It targets the active page or the page specified by the /PAGE flag.

Unlike the RemoveFromLayout operation, RemoveLayoutObjects can be used in user-defined functions. Therefore, RemoveLayoutObjects should be used in new programming.

Parameters

objectSpec is either an object name (e.g., Graph0) or an *objectName* with an instance (e.g., Graph0#1). An instance is needed only if the same object appears in the layout more than one time. Graph0 is equivalent to Graph0#0 and Graph0#1 refers to the second instance of Graph0 in the layout.

Flags

/PAGE=*page* — Removes the object from the specified page.

Page numbers start from 1. To target the active page, omit /PAGE or use *page*=0.

The /PAGE flag was added in Igor Pro 7.00.

/W=*winName* — *winName* is the name of the page layout window from which the object is to be removed. If /W is omitted or if *winName* is `$""`, the top page layout is used.

/Z — Does not report errors if the specified layout object does not exist.

See Also

NewLayout, **AppendLayoutObject**, **ModifyLayout**, **LayoutPageAction**

RemoveListItem

`RemoveListItem(`*`index`*`,` ***`listStr`*** `[,` ***`listSepStr`*** `[,` ***`offset`***`]]`**`)`**

The RemoveListItem function returns *listStr* after removing the item specified by the list index *index*.

RemoveListItem removes an item from a string containing a list of items separated by a separator, such as strings returned by functions like **TraceNameList** and **AnnotationList**.

Parameters

index is the zero-based index of the list item that you want to remove.

listStr contains a series of text items separated by *listSepStr*. The trailing separator is optional though recommended.

listSepStr is optional. If omitted it defaults to ";". Prior to Igor Pro 7.00, only the first byte of *listSepStr* was used. Now all bytes are used.

offset is optional and requires Igor Pro 7.00 or later. If omitted it defaults to 0. The search begins offset bytes into *listStr*. When iterating through lists containing large numbers of items, using the *offset* parameter provides dramatically faster execution. For an example using the offset parameter, see **StringFromList**.

Details

RemoveListItem differs from **RemoveFromList** in that it specifies the item to be removed by index and removes only that item, while RemoveFromList specifies the item to be removed by value, and removes all matching items.

If *index* less than 0 or greater than ItemsInList(*listStr*) - 1, or if *listSepStr* is `""` then *listStr* is returned unchanged (unless *listStr* contains only list separators, in which case an empty string is returned).

If the resulting string contains only *listSepStr* characters, then an empty string (`""`) is returned.

Examples

```
Print RemoveListItem(1, "wave0;wave1;w2;")        // Prints "wave0;w2;"
```

See Also

The **AddListItem**, **FindListItem**, **FunctionList**, **ItemsInList**, **RemoveByKey**, **RemoveFromList**, **StringFromList**, **StringList**, **TraceNameList**, **VariableList**, **WaveList**, and **WhichListItem** functions.

RemovePath

RemovePath [**/A/Z**] ***pathName***

The RemovePath operation removes a path from the list of symbolic paths. RemovePath is an old name for the new **KillPath** operation, which we recommend you use instead.

Rename

Rename ***oldName*, *newName***

The Rename operation renames waves, strings, or numeric variables from *oldName* to *newName*.

Parameters

oldName may be a simple object name or a data folder path and name. *newName* must be a simple object name.

Details

You can not rename an object using a name that already exists. The following will result in an error:

```
Make wave0, wave1
// Rename wave0 and overwrite wave1.
Rename wave0, wave1           // This will not work.
```

However, you can achieve the desired effect as follows:

```
Make wave0, wave1
Duplicate/O wave0, wave1; KillWaves wave0
```

See Also

The **Duplicate** operation.

RenameDataFolder

RenameDataFolder ***sourceDataFolderSpec*, *newName***

The RenameDataFolder operation changes the name of the source data folder to the new name.

sourceDataFolderSpec can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

newName is just the new name for the data folder, without any path.

Details

RenameDataFolder generates an error if the new name is already in use as a data folder contained within the source data folder.

Examples

```
RenameDataFolder root:foo,foo2       // Change name of foo to foo2
```

See Also

Chapter II-8, **Data Folders**.

RenamePath

`RenamePath` ***`oldName`*****`,`** ***`newName`***

The RenamePath operation renames an existing symbolic path from *oldName* to *newName*.

See Also

Symbolic Paths on page II-22

RenamePICT

`RenamePICT` ***`oldName`*****`,`** ***`newName`***

The RenamePICT operation renames an existing picture to from *oldName* to *newName*.

See Also

Pictures on page III-509.

RenameWindow

`RenameWindow` ***`oldName`*****`,`** ***`newName`***

The RenameWindow operation renames an existing window or subwindow from *oldName* to *newName*.

Parameters

oldName is the name of an existing window or subwindow.

When identifying a subwindow with *oldName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

See Also

The **DoWindow** operation.

ReorderImages

`ReorderImages` `[`**`/W=`*****`winName`***`]` ***`anchorImage`*****`, {`*****`imageA`*****`,`** ***`imageB`*****`, …}`**

The ReorderImages operation changes the ordering of graph images to that specified in the braces.

Flags

/W=*winName*	Reorders images in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

Igor keeps a list of images in a graph and draws the images in the listed order. The first image drawn is consequently at the bottom. All other images are drawn on top of it. The last image is the top one; no other image obscures it.

ReorderImages works by removing the images in the braces from the list and then reinserting them at the location specified by *anchorImage*. If *anchorImage* is not in the braces, the images in braces are placed before *anchorImage*.

If the list of images is A, B, C, D, E, F, G and you execute the command

```
ReorderImages F, {B,C}
```

images B and C are placed just before F: A, D, E, **B**, **C**, **F**, G.

The result of

```
ReorderImages E, {D,E,C}
```

is to reorder C, D and E and put them where E was. Starting from the initial ordering this gives A, B, **D**, **E**, **C**, F, G.

ReorderImages generates an error if the same trace is in the list twice.

In Igor7 or later, *anchorImage* can be _front_ or _back_. To move A to the front, you can write:

```
ReorderImage _front_, {A}
```

See Also

The **ReorderTraces** operation.

ReorderTraces

ReorderTraces [***/W=winName /L[=axisName] /R[=axisName]***] ***anchorTrace***, **{*traceA*, *traceB*, …}**

The ReorderTraces operation changes the ordering of graph traces to that specified in the braces.

Flags

/W=*winName*	Reorders traces in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.
/L[=*axisName*]	Moves traces to a newor existing axis. This feature was added in Igor Pro 8.00.
/R[=*axisName*]	To reference the built-in left or right axes, you can use /L or /R without specifying the axis name. To reference a free axis, you must specify the axis name. If the specified axis does not already exist, a new axis is created by cloning the axis controlling *traceA*. If the move results in no traces assigned to an axis, then that axis is deleted. Reordering is optional. Specify `_none_` for *anchorTrace* for no reordering. You can move a trace manually by right-clicking a trace and choosing Move to Opposite Axis.

Details

Igor keeps a list of traces in a graph and draws the traces in the listed order. The first trace drawn is consequently at the bottom. All other traces are drawn on top of it. The last trace is the top one; no other trace obscures it.

ReorderTraces works by removing the traces in the braces from the list and then reinserting them at the location specified by *anchorTrace*. If *anchorTrace* is not in the braces, the traces in braces are placed before *anchorTrace*.

If the list of traces is A, B, C, D, E, F, G and you execute the command

```
ReorderTraces F, {B,C}
```

traces B and C are placed just before F: A, D, E, **B**, **C**, **F**, G.

The result of

```
ReorderTraces E, {D,E,C}
```

is to reorder C, D and E and put them where E was. Starting from the initial ordering results in A, B, **D**, **E**, **C**, F, G.

ReorderTraces generates an error if the same trace is in the list twice.

In Igor7 or later, *anchorImage* can be _front_ or _back_. To move A to the front, you can write:

```
ReorderTraces _front_, {A}
```

See Also

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

The **ReorderImages** operation.

ReplaceNumberByKey

ReplaceNumberByKey(*keyStr*, *kwListStr*, *newNum* [, *keySepStr* [, *listSepStr* [, *case*]]])

The ReplaceNumberByKey function returns *kwListStr* after replacing the numeric value of the keyword-value pair specified by *keyStr*. *kwListStr* should contain keyword-value pairs such as `"KEY=value1,KEY2=value2"` or `"Key:value1;KEY2:value2"`, depending on the values for *keySepStr* and *listSepStr*.

Use ReplaceNumberByKey to add or modify numeric information in a string containing a `"key1:value1;key2:value2;"` style list such as those returned by functions like **AxisInfo** or **TraceInfo**.

If *keyStr* is not found in *kwListStr*, then the key and the value are appended to the end of the returned string.

keySepStr, *listSepStr*, and *case* are optional; their defaults are ":", ";", and 0 respectively.

Details

The actual string appended is:

[*listSepStr*] *keyStr keySepStr newNum listSepStr*

The optional leading list separator *listSepStr* is added only if *kwListStr* does not already end with a list separator.

kwListStr is searched for an instance of the key string bound by *listSepStr* on the left and a *keySepStr* on the right. The text up to the next ";" is replaced by *newNum* after conversion to text using the %.15g format (see **printf** for format conversion specifications).

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are usually case-insensitive. Setting the optional *case* parameter to 0 makes the comparisons case sensitive.

In Igor6, only the first byte of *keySepStr* and *listSepStr* was used. In Igor7 and later, all bytes are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *case* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print ReplaceNumberByKey("K1", "K1:7;", 4)                // prints "K1:4;"
Print ReplaceNumberByKey("k2", "K2=8;", 5, "=")           // prints "K2=5;"
Print ReplaceNumberByKey("K3", "K3:9,", 6, ":", ",")      // prints "K3:6,"
Print ReplaceNumberByKey("k3", "K0:9", 6, ":", ",")       // prints "K0:9,k3:6,"
Print ReplaceNumberByKey("k3", "K3:9,", 6, ":", ",")      // prints "K3:6,"
Print ReplaceNumberByKey("k3", "K3:9,", 6, ":", ",", 1)  // prints "K3:9,k3:6,"
```

See Also

The **ReplaceStringByKey**, **NumberByKey**, **StringByKey**, **RemoveByKey**, **ItemsInList**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

ReplaceString

ReplaceString(*replaceThisStr*, *inStr*, *withThisStr* [, *caseSense* [, *maxReplace*]])

The ReplaceString function returns *inStr* after replacing any instance of *replaceThisStr* with *withThisStr*.

The comparison of *replaceThisStr* to the contents of *inStr* is case-insensitive. Setting the optional *caseSense* parameter to nonzero makes the comparison case-sensitive.

Usually all instances of *replaceThisStr* are replaced. Setting the optional *maxReplace* parameter limits the replacements to that number.

Details

If *replaceThisStr* is not found, *inStr* is returned unchanged.

If *maxReplace* is less than 1, then no replacements are made. Setting `maxReplace = Inf` is the same as omitting it.

Examples

```
Print ReplaceString("hello", "say hello", "goodbye")// prints "say goodbye"
Print ReplaceString("\r\n", "line1\r\nline2", "") // prints "line1line2"
Print ReplaceString("A", "an Ack-Ack", "a", 1)    // prints "an ack-ack"
Print ReplaceString("A", "an Ack-Ack", "a", 1, 1) // prints "an ack-Ack"
Print ReplaceString("", "input", "whatever")   // prints "input" (no change)
```

See Also

ReplaceStringByKey, **CmpStr**, **StringMatch**, **strsearch**, **TrimString**, **ReplaceString**

ReplaceStringByKey

ReplaceStringByKey(*keyStr*, *kwListStr*, *newTextStr* [, ***keySepStr*** [, ***listSepStr*** [, ***matchCase***]]]**)**

The ReplaceStringByKey function returns *kwListStr* after replacing the text value of the keyword-value pair specified by *keyStr*. *kwListStr* should contain keyword-value pairs such as "KEY=value1,KEY2=value2" or "Key:value1;KEY2:value2", depending on the values for *keySepStr* and *listSepStr*.

Use ReplaceStringByKey to add or modify text information in a string containing a "key1:value1;key2:value2;" style list such as those returned by functions like **AxisInfo** or **TraceInfo**.

If *keyStr* is not found in *kwListStr*, then the key and the value are appended to the end of the returned string.

keySepStr, *listSepStr*, and *matchCase* are optional; their defaults are ":", ";", and 0 respectively.

Details

The actual string appended is:

[*listSepStr*] *keyStr keySepStr newTextStr listSepStr*

The optional leading list separator *listSepStr* is added only if *kwListStr* does not already end with a list separator.

kwListStr is searched for an instance of the key string bound by a ";" on the left and a ":" on the right. The text up to the next ";" is replaced by *newTextStr*.

If *newTextStr* is "", any existing value is deleted, but the key, the key separator, and the list separator are retained. To remove a keyword-value pair, use the **RemoveByKey** function.

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are case-insensitive. Setting the optional *matchCase* parameter to 1 makes the comparisons case-sensitive.

In Igor6, only the first byte of *keySepStr* and *listSepStr* was used. In Igor7 and later, all bytes are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *matchCase* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print ReplaceStringByKey("KY", "KY:a;KZ:c", "b")      // prints "KY:b;KZ:c"
Print ReplaceStringByKey("KY", "ky=a;", "b", "=")     // prints "ky=b;"
Print ReplaceStringByKey("KY", "KY:a,", "b", ":", ",")// prints "KY:b,"
Print ReplaceStringByKey("ky", "ZZ:a,", "b", ":", ",")// prints "ZZ:a,ky:b,"
Print ReplaceStringByKey("kz", "KZ:a,", "b", ":", ",")// prints "KZ:b,"
Print ReplaceStringByKey("kz", "KZ:a,", "b", ":", ",", 1)// prints "KZ:a,kz:b,"
```

See Also

The **ReplaceString**, **ReplaceNumberByKey**, **NumberByKey**, **StringByKey**, **ItemsInList**, **RemoveByKey**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

ReplaceText

`ReplaceText [/W=winName/N=name] textStr`

The ReplaceText operation replaces the text in the most recently created or changed annotation or in the annotation specified by /W=*winName* and/N=*name*.

Parameters

textStr can contain escape codes to set the font, size, style, color and other properties. See **Annotation Escape Codes** on page III-53 for details.

If the annotation is a color scale, this command replaces the text of the color scale's main axis label.

Flags

/N=*name*	Replaces the text of the named tag or textbox.
/W=*winName*	Replaces text in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

See Also

Tag, **TextBox**, **ColorScale**, **Legend**, **AppendText**, **Annotation Escape Codes** on page III-53.

ReplaceWave

```
ReplaceWave [/W=winName] allinCDF
ReplaceWave [/X/W=winName] trace=traceName, waveName
ReplaceWave [/X/Y/W=winName] image=imageName, waveName
ReplaceWave [/X/Y/W=winName] contour=contourName, waveName
```

The ReplaceWave operation replaces waves displayed in a graph with other waves. The waves to be replaced, and the replacement waves are chosen by the flags, the keyword and the wave names on the command line.

Flags

/W=*winName*	Replaces the wave in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.
/X	Replaces the wave supplying X coordinates. If the trace represents a category plot, the wave must be either a text wave or the special keyword '_labels_' to use dimension labels from the Y wave controlling the axis.
/Y	Replaces the wave supplying Y data.

Keywords

allinCDF	Searches the current data folder for waves with the same names as waves used in the graph. If found and if the waves are of the correct type, they replace the existing waves. Thus, if you have several data folders with identically-named waves containing data from different experimental runs, you can browse through the runs by moving from one data folder to another, using `ReplaceWave allinCDF` to update the graph.
contour=*contourName*	Replaces the wave supplying the Z data for *contourName*. If /X or /Y is used, replaces the wave used to set the X or Y data spacing (if the Z data are in a matrix) or the wave used to supply the X or Y positions if XYZ triplets were specified with three separate waves.

image=*imageName* Replaces the wave supplying the Z data for *imageName*. If /X or /Y is used, replaces the wave used to set the X or Y data spacing.

trace=*traceName* Replaces the wave associated with *traceName*. With the /X flag, *waveName* will replace the X wave associated with *traceName*, otherwise it will replace the Y wave. Note that *traceName* is derived from the Y wave name; if you created a graph using `Display jack vs sam`, you would use `ReplaceWave/X trace=jack,newsam` to replace the X wave.

For traces, the ReplaceWave/Y flag is equivalent to ReplaceWave with no flags.

Details

Waves are replaced in the graph specified by /W=*winName* otherwise waves are replaced in the top graph.

Updating a contour plot in response to replacing a wave can be time-consuming. If you must replace more than one wave, put all the commands separated by semicolons on a single line. In a macro, use **DelayUpdate** to prevent updates between command lines.

When using the allinCDF keyword, ReplaceWave cannot find waves buried in dynamic annotation text (for instance, using the \{} syntax in an annotation). ReplaceWave will not replace waves used for error bars, either.

Subsets of data, including individual rows or columns from a matrix, may be specified using **Subrange Display Syntax** on page II-321.

Examples

Make XY plot, then replace the waves:

```
Make fred=x, sam=log(x)
Display fred vs sam
Make fred2=2*x, sam2=ln(x)
ReplaceWave/X trace=fred, sam2
ReplaceWave trace=fred, fred2           // trace is now named fred2
```

Make contour plot with XYZ triplet waves, then replace the waves. Note the DelayUpdate commands after the first two ReplaceWave commands:

```
Make/N=100 junkx, junky, junkz          // Waves for XYZ triplets
junkx=trunc(x/10)                       // X wave for XYZ triplets
junky=mod(x,10)                         // Y wave for XYZ triplets
junkz=sin(junkx[p])*cos(junky[p])       // Z wave for XYZ triplets
Display; AppendXYZContour junkz vs {junkx, junky}  // Make contour plot
Make/O/N=150 junkx2, junky2, junkz2     // Make replacement waves
junkx2=trunc(x/15)
junky2=mod(x,15)
junkz2=sin(junkx2[p])*cos(junky2[p])
ReplaceWave/X contour=junkz,junkx2; DelayUpdate
ReplaceWave/Y contour=junkz,junky2; DelayUpdate
ReplaceWave contour=junkz,junkz2
```

This example is suitable for copying all the lines and pasting into the command line, or for use in a macro. If you are typing on the command line, you would want to put the ReplaceWave commands all on one line:

```
ReplaceWave/X contour=junkz,junkx2; ReplaceWave/Y contour=…
```

See Also

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

ReplicateString

ReplicateString(*str*, *totalNumCopies*)

The ReplicateString function returns a string containing *str* repeated *totalNumCopies* times.

The ReplicateString function was added in Igor Pro 9.00.

Example

```
String in = "αßγ"
String out = ReplicateString(in, 3)  // Returns "αßγαßγαßγ"
```

See Also

PadString, **ReplaceString**

Resample

Resample [***flags***] ***waveName*** [, ***waveName***]…

The Resample operation resamples *waveName* by interpolating or up-sampling (set by /UP=*upSample*), lowpass filtering, and decimating or down-sampling (set by /DOWN=*downSample*).

Lowpass filtering is specified with /N and /WINF or with /COEF=*coefsWaveName*.

The sampling frequency (1/DimDelta) of a resampled output wave *waveName* is changed by the ratio of *upSample*/*downSample*. For example, if *upSample*=4 and *downSample*=3, then the final sampling rate is 4/3 of the original value.

Straight interpolation can be accomplished by setting *upSample* to the interpolation factor and *downSample*=1, in which case the sample rate is multiplied by *upSample*. Deltax(*waveName*) will be proportionally smaller. Linear interpolation is accomplished by setting *numReconstructionSamples*=3 and *windowKind*=None.

For decimation only, set *upSample*=1 and *downSample* to the decimation factor. The sample rate is divided by *downSample*, and deltax(*waveName*) will be proportionally larger.

Use **RatioFromNumber** to choose appropriate values for *upSample* and *downSample*, or use /SAME=*sWaveName* or /RATE=*sampRate*. See **Resampling Rates Example** for details.

When using /COEF=*coefsWaveName*, the filter coefficients should implement a low-pass filter appropriate for the *upSample* and *downSample* values or aliasing (filtering errors) will result. See **Advanced Externally-Supplied Low Pass Filter Example** for details.

Resampling Rates Flags

The *upSample* and *downSample* values define how much interpolation and decimation to perform. They can be set directly with /UP and /DOWN or indirectly with /SAME or /RATE

/DOWN=*downSample*
Down-samples or decimates the filtered result by this integer factor after up-sampling and lowpass filtering. The default is 1 (no down-sampling).

For example, /DOWN=3 places only every third value in the output wave.

Down-sampling divides the sampling rate of the filtered data by a factor of *downSample*. The DimDelta(*waveName*, *dim*) value is multiplied by the same factor.

/RATE=*sampRate*
Converts the output *waveName* to the specified sampling rate frequency (normally Hz).

The necessary *upSample* and *downSample* values for each *waveName* are computed internally as if you had executed:

```
RatioFromNumber (deltax(waveName)*sampRate)
upSample = V_numerator
downSample = V_denominator
```

/RATE returns V_numerator and V_denominator set to these automatically-determined values for the last *waveName*.

/SAME=*sWaveName*

Converts the output *waveName* to the same sampling rate as *sWaveName*, 1/DimDelta(*sWaveName*, *dim*). The necessary *upSample* and *downSample* values are computed internally as if you had executed:

```
Variable dd = DimDelta(waveName,dim)
RatioFromNumber dd/DimDelta(sWaveName,dim)
upSample = V_numerator
downSample = V_denominator
```

/SAME returns V_numerator and V_denominator set to these automatically determined values for the last *waveName*.

/UP=*upSample* Up-samples or interpolates the input by this integer factor. The default is 1 (no up-sampling).

For example, /UP=4 inserts three extra points between each input point (producing 4 times as many values) before the lowpass filtering and down-sampling occurs.

Up-sampling multiplies the sampling rate of the input data by a factor of *upSample*, though no additional signal information is created. The DimDelta(*waveName*, *dim*) value is divided by the same factor.

Internal Sinc Reconstruction Filter Flags

If /COEF=*coefsWaveName* is not specified, Resample computes a windowed sinc filter from /N, /DOWN, /UP, and /WINF flag values.

If /COEF=*coefsWaveName* is specified, then *coefsWaveName* supplies the filter, and /N and /WINF are ignored. See **Externally-Supplied Low Pass Filter Flags**.

/COEF Replaces the first *waveName* with coefficients generated by *downSample*, *upSample*, *numReconstructionSamples*, and *windowKind*, a windowed sinc impulse response.

When resampling multiple *waveName*s with different filters (because /RATE or /SAME were specified and the multiple *waveName*s had different sampling rates), the filter used to resample the last *waveName* is returned.

/N=*numReconstructionSamples*

Specifies the number of input values used to created the up-sampled values (default is 21).

The value of *numReconstructionSamples* must be odd.

The size of the computed filter is (*numReconstructionSamples*-1) * *upSample* + 1.

Bigger is better: 15 is usually on the low side for yielding reasonably accurate results, and although 101 will nearly always give very good results, it will be slow.

Use /COEF to output the impulse response, and the FFT to display the frequency response of the interpolator:

```
Make/O coefs
Variable numReconstructionSamples= 51, upSample= 5
Resample/COEF/N=(numReconstructionSamples)/UP=(upSample) coefs
Variable evenNum= 2*floor((numpnts(coefs)+1)/2)
FFT/OUT=3/PAD={evenNum}/DEST=coefs_FFT coefs
Display coefs_FFT
```

Bigger is also slower: the filtering is computed in the time-domain, and execution time is linearly related to

upSample/*downSample* * *numReconstructionSamples*.

/WINF=*windowKind*

Applies the window, *windowKind*, to the computed filter coefficients. If /WINF is omitted, the Hanning window is used. For no coefficient windowing, use /WINF=None, though this is discouraged unless you want linear interpolation, in which case it should be paired with /N=3.

Windows alter the frequency response of the filter in obvious and subtle ways, enhancing the stop-band rejection or steepening the transition region between passed and rejected frequencies. They matter less when *numReconstructionSamples* is large.

Choices for *windowKind* are:

Bartlett, Blackman367, Blackman361, Blackman492, Blackman474, Cos1, Cos2, Cos3, Cos4, Hamming, Hanning, KaiserBessel20, KaiserBessel25, KaiserBessel30, Parzen, Poisson2, Poisson3, Poisson4, Riemann and None.

See **FFT** for window equations and details.

Externally-Supplied Low Pass Filter Flags

/COEF =*coefsWaveName*

Identifies the wave, *coefsWaveName*, containing filter coefficients that implement a low-pass filter with a cutoff frequency of the lesser of 0.5/*upSample* and 0.5/*downSample*, where 0.5 corresponds to the Nyquist frequency of the up-sampled data.

For example, if *upSample*=2, then the filter must contain the classic “half-band” filter, which stops the higher half of the frequencies and passes the lower half. If *upSample*=10, then the filter must pass only the lowest 1/10th of the frequencies.

For *downSample* > *upSample*, the low-pass filter’s cutoff frequency must be 0.5/*downSample*. This prevents the decimation from introducing aliasing to the resampled data.

To avoid shifting the output with respect to the input, *coefsWaveName* must have an odd length with the “center” coefficient in the middle of the wave.

The length of *coefsWaveName* must be 1+*upSample***n*, where *n* is any even integer.

Note: Instead of using /N=*numReconstructionSamples* with /COEF=*coefsWaveName*, *numReconstructionSamples* is computed from *upSample* and the number of points in *coefsWaveName*:

numReconstructionSamples=1+(numpnts(*coefsWaveName*)-1)/*upSample*.

Coefficients are usually symmetrical about the middle point, but this is not enforced.

coefsWaveName must not be a destination *waveName*.

coefsWaveName must be single- or double-precision numeric and one-dimensional.

Data Range Flags

/DIM=*d*

Specifies the wave dimension to resample.

For *d* =0, 1, …, resampling is along rows, columns, etc.

The default is /DIM=0, which resamples each individual column (each one a channel, say left and right) in a multidimensional *waveName* where each row comprises all sound samples at a particular time.

To resample in multiple dimensions, execute the command once for each dimension. For example, use /DIM=0 followed by another command with /DIM=1 to resample a two-dimensional wave in each direction.

E=*endEffect*

Determines how to handle the ends of the resampled wave(s) (w) when fabricating missing neighbor values.

endEffect=0:	Bounce method. Uses w[*i*] in place of the missing w[-*i*] and w[*n*-*i*] in place of the missing w[*n*+*i*].
endEffect=1:	Wrap method. Uses w[*n*-*i*] in place of the missing w[-*i*] and vice versa.
endEffect=2:	Zero method (default). Uses 0 for any missing value.
endEffect=3:	Repeat method. Uses w[0] in place of the missing w[-*i*] and w[*n*] in place of the missing w[*n*+*i*].

Parameters

waveName can be a wave with any number of dimensions. Only one dimension is resampled. Use multiple Resample calls to resample across multiple dimensions.

Without /DIM, resampling is done along the row (first) dimension for each column. That is, each column is resampled as if it were a separate one-dimensional wave. This allows multichannel audio to be resampled to another frequency.

If /DIM=1, then resampling proceeds across all the columns of each row.

If /COEF is specified without *coefsWaveName*, then the first *waveName* is overwritten by the filter coefficients instead of being resampled.

Details

The filtering convolution is performed in the time-domain. That is, the FFT is not employed to filter the data. For this reason the coefficients length (/N or the length of *coefsWaveName*) should be small in comparison to the resampled waves.

Resample should not be used on data that contains NaNs; use **Smooth**/M=NaN to replace NaN values with a local median, or use **MatrixOp** zapNaNs to remove them.

Resample assumes that the middle point of *coefsWaveName* corresponds to the delay=0 point. The “middle” point number = trunc((numpnts(*coefsWaveName*)-1)/2). *coefsWaveName* usually contains the two-sided impulse response of a filter, an odd number of points, and implements a low-pass filter whose cutoff frequency is the lesser of 0.5/*upSample* and 0.5/*downSample* (0.5 corresponds to the Nyquist frequency = 1/2 sampling frequency).

When /COEF creates a coefficients wave it sets the wave’s X scaling deltax property to the deltax of the input wave multiplied by the /DOWN factor and divided by the /UP factor. It also sets the x0 property so that the zero-phase (center) coefficient is located at x=0.

Simple Examples

```
// Interpolation by factor of 4, default filter
Resample/UP=4 data

// Interpolation by factor of 7, linear interpolation
Resample/UP=7/N=3/WINF=None data

// Decimation by factor of 3, default filter
Resample/DOWN=3 data

// Match sampling rates, default filter
Resample/SAME=dataAtDesiredRate dataAtWrongRate1, dataAtWrongRate2,...

// Resample waves to 10 KHz sampling rate
Resample/RATE=10e3 dataAtWrongRate1, dataAtWrongRate2,...

// Interpolate an image by a factor of 2
Resample/UP=2 image             // default is /DIM=0, resample rows
Resample/UP=2/DIM=1 image       // resample across columns
```

Note: Interpolating by a factor of two does not produce an image with twice as many rows and columns. The new number of rows = (original rows-1)**upSample* +1, and a similar computation applies to columns.

Resampling Rates Example

Suppose we have an audio wave sampled at 44,100 Hz and we wish to resample it to a higher 192,000 Hz frequency.

We can use /RATE= 192000 and let Resample determine the correct values (provided *waveName* has its X scaling set properly to reflect sampling at 44100 Hz), but let’s compute *upSample* and *downSample* ourselves.

Because the sampling rate = 1/deltax(*wave*), we can recast the /SAME formula to RatioFromNumber (*desiredSamplingRate*/*currentSamplingRate*):

```
•RatioFromNumber/V (192000 / 44100)
  V_numerator= 640; V_denominator= 147;
  ratio= 4.3537414965986;
  V_difference= 0;
```

Then *upSample*=640 and *downSample*=147.

The 44100 Hz input data will be interpolated by 640 to 28,224,000 Hz.

The result is low-pass filtered with a “cutoff frequency” of 1/640th of the interpolated Nyquist frequency = (28224000/2)/640 = 22,050 Hz, the same as the input signal’s original Nyquist frequency.

The result will be decimated by 147 to 192,000 Hz, which is the desired output sampling frequency.

Note: If *downSample* had been greater than *upSample*, then the low-pass filter's cutoff frequency would have been 1/*downSample*th of the interpolated Nyquist frequency = (28224000/2)/*downSample*. This prevents the decimation from introducing aliasing to the resampled data.

```
Resample/UP=640/DOWN=147 sound          // convert 44.1 KHz to 192 KHz
```

Advanced Externally-Supplied Low Pass Filter Example

You can generate an appropriate filter by executing commands like these:

```
// Compute a filter for after the input is upsampled
// to restore the frequency content to the original range.
Variable fc = min(0.5 / upSample, 0.5 * upSample / downSample)
// Transition width, small widths need big n
Variable tw= fc/10
// Set end of pass band
Variable f1= fc-tw/2
// Set start of stop band
Variable f2= fc+tw/2
// Use bigger values of n to make the filter smoother
Variable nReconstruct= 31
Variable n= (nReconstruct-1)*upSample+1        // odd = no phase shift
// Create a wave to hold the coefficients; it gets resized to n
Make/O/N=0 coefsWaveName
FilterFIR/COEF/LO={f1,f2,n} coefsWaveName
```

However, FilterFIR does not create windowed sinc lowpass filters that have the endearing property that the original input values are unaltered in the filtered output, though only if *upSample* > *downSample*. This is called a "Nyquist filter" or "Kth-band filter" in the literature.

If *upSample* > *downSample*, you can enforce the Nyquist criterion by "zeroizing" the designed filter by setting every *upSample*th value to 0 except the center one.

```
// coefsWaveName length must be 1+upSample*n, where n is any even integer
Function Zeroize(w, upSample)
   Wave w                          // coefsWaveName
   Variable upSample               // upSample value

   Variable n= DimSize(w,0)
   Variable centerP= floor((n-1)/2)        // if n=101, centerP= 50
   Variable i
   for (i=0; i<n; i+=upSample)
      if( i != centerP )
         w[i] = 0
      endif
   endfor
End
```

Resample zeroizes the internally-generated low pass filter when *upSample* > *downSample*.

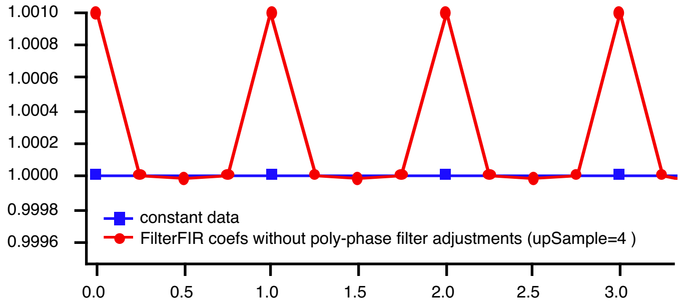
Additionally, the FilterFIR command generates a low-pass filter whose gain needs to be multiplied by *upsample*:

*coefsWaveName *= upSample*

When designing an externally supplied filter, you should also consider the filter's "polyphase" nature; *coefsWaveName* is actually a set of *upSample* interleaved filters, each with its own response. It makes sense to adjust these filters to produce consistent responses. If you don't, the results will contain ringing with a period of *upSample*/*downSample*. This is most apparent when *downSample* is 1.

Using the filter we've designed so far with *upSample*=4, here's the output of a constant-input wave:

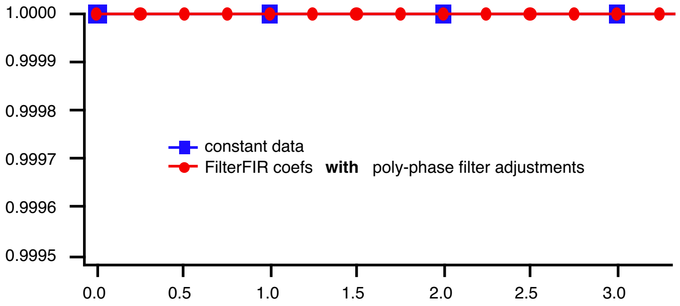
```
Make/O constantData= 1
Resample/COEF=coefsWaveName/UP=4 constantData
```



The graph shows that the filter response at 0 Hz for the first of 4 filters is 1.0010, the second and fourth filter's responses are very close to 1.0, and the third filter's response at 0 Hz is a little less than 1.0.

These variations can be eliminated by normalizing the sum of each polyphase filter to 1.0:

```
Function PolyphaseNormalize(w, upSample)
    Wave w                          // coefsWaveName
    Variable upSample               // upSample value

    Variable n= DimSize(w,0)
    Variable filt
    // for each filter (0..upSample-1)
    for (filt=0; filt<upSample; filt+=1)
        Variable total=0
        Variable pt
        // compute total for this filter
        for (pt=filt; pt<n; pt+=upSample)
            total += w[pt]
        endfor
        // divide by total to normalize total to 1
        for (pt=filt; pt<n; pt+=upSample)
            w[pt] /= total
        endfor
    endfor
End
```



Now the filter is ready to be used to filter data:

```
Resample/COEF=coefsWaveName/UP=(upSample)/DOWN=(downSample) dataWave
```

You can see that designing an externally-supplied lowpass filter is much more complicated than using the internal sinc reconstruction filter, which does all this zeroizing, scaling, and polyphase normalization for you.

References

Mintzer, F., On half-band, third-band, and Nth band FIR filters and their design, *IEEE Trans. on Acoust., Speech, Signal Process., ASSP-30*, 734-738, 1982.

See Also

RatioFromNumber, **FilterFIR**, **interp**, **Interp2D**, **Interpolate2**, **ImageInterpolate**, **Loess**, **Smooth**,

Multidimensional Decimation

ResumeUpdate

```
ResumeUpdate
```

The ResumeUpdate operation cancels the corresponding **PauseUpdate**.

This operation is of use in macros. It is not allowed from the command line. It is allowed but has no effect in user-defined functions. During execution of a user-defined function, windows update only when you explicitly call the DoUpdate operation.

See Also

The **DelayUpdate**, **DoUpdate**, and **PauseUpdate** operations.

return

```
return [expression]
```

The return flow control keyword immediately stops execution of the current procedure. If called by another procedure, it returns *expression* and control to the calling procedure.

Functions can return only a single value directly to the calling procedure with a return statement. The return value must be compatible with the function type. A function may contain any number of return statements; only the first one encountered during procedure execution is evaluated.

A macro has no return value, so return simply quits the macro.

See Also

The Return Statement on page IV-35.

Reverse

```
Reverse [type flags][/DIM=d /P] waveA [/D = destWaveA][, waveB [/D = destWaveB][, …]]
```

The Reverse operation reverses data in a wave in a specified dimension. It does not accept text waves.

Flags

/DIM = *d*	Specifies the wave dimension to reverse. *d*=-1: Treats entire wave as 1D (default). For *d*=0, 1, …, operates along rows, columns, etc.
/P	Suppresses adjustment of dimension scaling. Without /P the scaled dimension value of reversed points remains the same.

Type Flags *(used only in functions)*

Reverse also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when it is needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-73 and **WAVE Reference Type Flags** on page IV-74 for a complete list of type flags and further details.

Wave Parameters

Note: ***All*** wave parameters must follow *wave* in the command. All wave parameter flags and type flags must appear immediately after the operation name (Reverse).

/D=*destWave*	Specifies the name of the wave to hold the reversed data. It creates *destWave* if it does not already exist or overwrites it if it exists.

Details

If the optional /D = *destWave* flag is omitted, then the wave is reversed in place.

See Also

Sorting on page III-132, **Sort**, **SortColumns**

RGBColor

The RGBColor structure is used as a substructure usually to store various color settings.

```
Structure RGBColor
   UInt16 red
   UInt16 green
   UInt16 blue
EndStructure
```

RGBAColor

The RGBAColor structure is the same as RGBColor but with an alpha field to represent translucency.

```
Structure RGBAColor
   UInt16 red
   UInt16 green
   UInt16 blue
   UInt16 alpha
EndStructure
```

rightx

rightx(*waveName*)

The rightx function returns the X value corresponding to point N of the named 1D wave of length N.

Details

Note that the point numbers in a wave run from 0 to N-1 so there is no point with this X value. To get the X value of the last point in a wave (point N-1), use the following:

```
pnt2x(waveName,numpnts(waveName)-1)        // N = numpnts(waveName)
```

which is more accurate than:

```
rightx(waveName) - deltax(waveName)
```

The rightx function is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details. The equivalent information for any dimension can be calculated this way:

IndexN = DimSize(*wave*, *dim*)*DimDelta(*wave*, *dim*) + DimOffset(*wave*, *dim*)

Here *IndexN* is the value of the scaled dimension index corresponding to element N of the dimension *dim* in a wave named *wave* that has N elements in that dimension.

See Also

The **deltax** and **leftx** functions, also the **pnt2x** and **numpnts** functions.

For an explanation of waves and dimension scaling, see **Changing Dimension and Data Scaling** on page II-68.

For multidimensional waves, see **DimDelta**, **DimOffset**, and **DimSize**.

root

root[**:*dataFolderName***[**:*dataFolderName***[**:**…]]][**:*objectName***]

Igor's data folder hierarchy starts with the root folder as its basis. The root data folder always exists and it contains all other objects (waves, variables, strings, and data folders). By default, the root data folder is the current data folder in a new experiment. In commands, root is used as part of a path specifying the location of a data object in the folder hierarchy.

See Also

Chapter II-8, **Data Folders**.

Rotate

Rotate *rotPoints*, *waveName* [**,** ***waveName***]…

The Rotate operation rotates the Y values of waves in wavelist by *rotPoints* points.

Parameters

If *rotPoints* is positive then values are rotated from the start of the wave toward the end and *rotPoints* values from the end of a wave wrap around to the start of the wave.

If *rotPoints* is negative then values are rotated from the end of the wave toward the start and *rotPoints* values from the start of a wave wrap around to the end of the wave.

Details

The X scaling of the named waves is changed so that the X values for the Y values remains the same except for the points that wrap around.

The Rotate operation is not multidimensional aware. To rotate rows or columns of 2D waves, see the rotateRows, rotateCols, rotateLayers and rotateChunks keywords for **MatrixOp** and the rotateRows and rotateCols keywords for **ImageTransform**.

For general information about multidimensional analysis, see **Analysis on Multidimensional Waves** on page II-95.

See Also

The shift parameter of the **WaveTransform** operation.

round

`round(`*`num`*`)`

The round function returns the integer value closest to *num*.

The rounding method is “away from zero”.

The result for INF and NAN is undefined.

See Also

The **ceil**, **floor**, and **trunc** functions.

rtGlobals

`#pragma rtGlobals = 0, 1, 2,` `or` **`3`**

`#pragma rtglobals=<n>` is a compiler directive that controls compiler and runtime behaviors for the procedure file in which it appears.

This statement must be flush against the left edge of the procedure file with no indentation. It is usually placed at the top of the file.

`#pragma rtglobals=0` turns off runtime creation of globals. This is obsolete.

`#pragma rtglobals=1` is a directive that turns on runtime lookup of globals. This is the default behavior if #pragma rtGlobals is omitted from a given procedure file.

`#pragma rtGlobals=2` turns off compatibility mode. This is mostly obsolete. See **Legacy Code Issues** on page IV-113 for details.

`#pragma rtglobals=3` turns on runtime lookup of globals, strict wave reference mode and wave index bounds checking.

rtGlobals=3 is recommended.

See **The rtGlobals Pragma** on page IV-52 for a detailed explanation of rtGlobals.

s

`s`

The s function returns the current chunk index of the destination wave when used in a multidimensional wave assignment statement. The corresponding scaled chunk index is available as the **t** function.

Details

Unlike **p**, outside of a wave assignment statement, s does not act like a normal variable.

See Also

Waveform Arithmetic and Assignments on page II-74.

For other dimensions, the **p**, **r**, and **q** functions.

For scaled dimension indices, the **x**, **y**, **z** and **t** functions.

Save

`Save [`*`flags`*`]` ***`waveList`*** **`[as`** ***`fileNameStr`*****`]`**

The Save operation saves the named waves to disk as text (/F, /G or /J) or as Igor binary.

Parameters

waveList is either a list of wave names or, if the /B flag is present, a string list of references to waves. For example, the following commands are equivalent, assuming that the waves in question are in the root data folder and root is the current data folder:

```
Save/J wave0,wave1 as "Test.dat"
Save/J root:wave0,root:wave1 as "Test.dat"
Save/J/B "wave0;wave1;" as "Test.dat"
Save/J/B "root:wave0;root:wave1;" as "Test.dat"
String list="root:wave0;root:wave1;"; Save/J/B list as "Test.dat"
```

The form using the /B flag and a string containing a list of references to waves saves a very large number of waves using one command. This is not possible using a list of wave names because of the 2500 byte command line length limit. When using this form, the string must contain semicolon-separated wave names or data folder paths leading to waves. Liberal names in the string may be quoted or unquoted.

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If it cannot determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/A[=*a*] Appends to the file rather than overwriting it (with /T, /G or /J).

a=0: Does not append.

a=1: Appends to the file with a blank line before the appended data (same as /A only).

a=2: Appends to the file with no blank line before the appended data.

/B The *waveList* parameter is a string containing a list of references to waves instead of a literal list of waves.

/C Saves a copy of the wave when saving as Igor binary.

/DLIM=*delimStr* Specifies the string to use as a column delimiter. This flag affects general text saves (/G) and delimited text saves (/J) only.

delimStr defaults to tab. It can consist of multiple characters.

If you choose a delimiter that also appears in the data you are saving, for example if you choose to save text waves containing commas using comma as the separator, the resulting file is likely to be misinterpreted by any software loading it.

/DLIM was added in Igor Pro 7.00.

/DSYM=*dsStr* Specifies a string containing the character to use as the decimal symbol for all numbers (default is a period). If *dsStr* is empty (`""`), then the decimal symbol is as defined in system preferences as of when Igor was launched.

/E=*useEscapeCodes* Determines whether to use escape sequences for special characters.

/E=1: Converts carriage-return, linefeed, tab, and backslash characters to escape sequences when writing general or delimited text files (default; same as no /E).

/E=0: No escape sequences used in general or delimited text files. When saving text waves containing backslashes (such as Windows paths) in a file intended for another program, you probably should use /E=0.

/F — Writes delimited and general text files with numeric formatting as it appears in the top table. Has no effect if there is no top table or if the wave being saved does not appear in the top table.

Note: The text written to the file is exactly as displayed in the table. Set the table to display as many digits of precision as you want in the file.

Note: Fractional and out-of-range floating point wave data can not be formatted as octal or hex. See **Octal Numeric Formats** on page II-257 and **Hexadecimal Numeric Formats** on page II-257 for details.

Note: When saving a multi-column wave (1D complex wave or multidimensional wave), all columns of the wave are saved using the table format for the first table column from the wave.

/G — Saves waves in general text format.

/H — "Adopts" the waves specified by *waveList*.

"Adopt" means that any connection between the waves and external files is severed. The waves become part of the current experiment. When the experiment is next saved, the waves are saved in the experiment file (for an packed experiment) or in the experiment folder (for an unpacked experiment).

When you use the /H flag, all other flags and the *fileNameStr* parameter are ignored. The wave is not actually saved but rather is marked for saving as part of the current experiment.

You would normally do this to make an experiment more self-contained which makes it easier to send to other people. See **Sharing Versus Copying Igor Binary Wave Files** on page II-156 and the **LoadWave** /H flag.

/I — Presents a dialog from which you can specify file name and folder.

/J — Saves waves in delimited text format.

The delimiter defaults to tab unless you specify another delimiter using /DLIM.

/M=*termStr* — Specifies the terminator character or characters to use at the end of each line of text. The default is `/M="\r"`, which uses a carriage return character. This is the Macintosh convention. To use the Windows convention, carriage return plus linefeed, specify `/M="\r\n"`. To use the Unix convention, just a linefeed, specify `/M="\n"`.

/O — Overwrites file if it exists already.

/P=*pathName* — Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.

/T — Saves waves in Igor Text format.

/U={*writeRowLabels*, *rowPositionAction*, *writeColLabels*, *colPositionAction*}

These parameters affect the saving of a matrix (2D wave) to a delimited text (/J) or general text (/G) file. They are accepted no matter what the save type is but are ignored when they don’t apply.

If *writeRowLabels* is nonzero, Save writes the row labels of the matrix as the first column of data in the file.

rowPositionAction has one of the following values:

0: Don’t write a row position column.

1: Writes a row position column based on the row scaling of the matrix wave.

2: Writes a row position column based on the contents of the row position wave for the matrix. The row position wave is an optional 1D wave whose name is the same as the matrix wave but with the prefix “RP_”.

writeColLabels and *colPositionAction* have analogous meanings. The prefix used for the column position wave is “CP_”.

See Chapter II-9, **Importing and Exporting Data**, for further details.

/W Saves wave names (with /G or /J).

Details

The Save operation saves only the named waves; it does not save the entire experiment.

Waves saved in Igor binary format are saved one wave per file. If you are saving more than one wave, you must not specify a *fileNameStr*. Save will give each file a name which consists of the wave name concatenated with “.ibw”.

When you save a wave as Igor binary, unless you use the /C flag to save a copy, the current experiment subsequently references the file to which the wave was saved. See **References to Files and Folders** on page II-24 for details.

In a general text file (/G), waves with different numbers of points are saved in different groups. Waves with different precisions and number types are saved in same group if they have the same number of points.

In a delimited text file (/J), all waves are saved in one group whether or not they have the same number of points.

If you save multiple 2D waves, the blocks of data are written one after the other.

If you save 3D waves, the data for each wave is written as a contiguous block having as many columns as there are columns in the wave, and R*L rows, where R is the number of rows in the multidimensional wave and L is the number of layers. All rows for layer 0 are saved followed by all rows for layer 1, and so on.

If you save 4D waves, the data for each wave is written as a contiguous block having R*L*C rows, where R is the number of rows, L is the number of layers and C is the number of chunks. Igor writes all data for chunk 0 followed by all data for chunk 1, and so on.

The Save operation will always present a save dialog if you try to save to an existing file without using the overwrite flag.

Here are some details about saving an Igor binary wave file.

If you omit the path or the file name, the Save operation will normally present a save dialog. However, if the wave has already been saved to a stand-alone file and if you use the overwrite flag, it will save the wave to the same file without a dialog. Also, if the wave has never been saved and the current experiment is an unpacked experiment, it will save to the home folder without a dialog.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like “`hd:Folder1:`” or “`C:\\Folder1\\`”. See **Symbolic Paths** on page II-22 for details.

Examples

This function uses the string list of references to waves to save some or all of the waves in the current data folder:

```
Function SaveWavesFromCurrentDF(matchStr)
   String matchStr                  // As for the WaveList function.

   String list
   list = WaveList(matchStr, ";", "")
   Save/O/J/W/I/B list
End
```

For example, to save all of the waves in the current data folder, execute:

```
SaveWavesFromCurrentDF("*")
```

To save those waves in the current data folder whose name starts with “wave”, execute:

```
SaveWavesFromCurrentDF("wave*")
```

This function saves all of the waves used in a particular graph:

```
Function SaveWavesFromGraph(graphName)      // Saves all waves in graph.
   String graphName                         // "" for top graph.

   String list, traceList, traceName
   Variable index = 0
   list = ""
   traceList = TraceNameList(graphName, ";", 1)
```

```
		do
			traceName = StringFromList(index, traceList, ";")
			if (strlen(traceName) == 0)
				break
			endif
			Wave w = TraceNameToWaveRef(graphName, traceName)
			list += GetWavesDataFolder(w,2) + ";"
			index += 1
		while(1)

		if (strlen(list) > 0)
			Save/O/J/W/I/B list
		endif
	End
```

See Also

Exporting Data on page II-177.

SaveData

`SaveData [`*`flags`*`]`*`fileOrFolderNameStr`*

The SaveData operation writes data from the current data folder of the current experiment to a packed experiment file on disk or to a file system folder. “Data” means Igor waves, numeric and string variables, and data folders containing them. The data is written as a packed experiment file or as unpacked Igor binary wave files in a file-system folder.

Warning: If you make a mistake using SaveData, it is possible to overwrite critical data, even entire folders containing critical data. It is your responsibility to make sure that any file or folder that you can not afford to lose is backed up. If you provide procedures for use by other people, you should warn them as well.

SaveData provides a way to save data for archival storage or unload data from memory during a lengthy process like data acquisition. The file or files that SaveData writes are disassociated from the current experiment.

Use SaveData to save experiment data using Igor procedures. To save experiment data interactively, use the Save Copy button in the Data Browser (Data menu).

Parameters

fileOrFolderNameStr specifies the packed experiment file (if /D is omitted) or the file system folder (if /D is present) in which the data is to be saved. The documentation below refers to this file or folder as the “target”.

If you use a full or partial path for *fileOrFolderNameStr*, see **Path Separators** on page III-451 for details on forming the path.

If *fileOrFolderNameStr* is omitted or is empty (`""`), SaveData displays a dialog from which you can select the target. You also get a dialog if the target is not fully specified by *fileOrFolderNameStr* or the /P=*pathName* flag.

If you are saving to a packed experiment file (/D omitted or /D=0), SaveData writes an HDF5 packed experiment file if the file name extension is ".h5xp". Otherwise it writes a standard packed experiment file (.pxp format). The .h5xp format requires Igor Pro 9 or later.

Flags

/COMP={*minWaveElements*, *gzipLevel*, *shuffle*}

Specifies that compression is to be applied to numeric waves saved when saving as an HDF5 packed experiment file. The /COMP flag was added in Igor Pro 9.00.

minWaveElements is the minimum number of elements that a numeric wave must have to be eligible for compression. Waves with fewer than this many total elements are not compressed.

gzipLevel is a value from 0 to 9. 0 means no GZIP compression.

shuffle is 0 to turn shuffle off or 1 to turn shuffle on.

When compression is applied by SaveData, the entire wave is saved in one chunk. See **HDF5 Layout Chunk Size** on page II-214 for background information and **SaveExperiment Compression** on page II-214 for details.

/D [=*d*] Writes to a file-system folder (a directory). If omitted, SaveData writes to an Igor packed experiment file.

d=1: If the target folder already exists, the new data is "mixed-in" with the data already there (same as /D).

d=2: If the target folder already exists, **it is completely deleted before the writing of data starts**.

If in doubt, use /D=1. See **Details** below.

/I Presents a dialog in which you can interactively choose the target.

/J=*objectNamesStr* Saves only the objects named in the semicolon-separated list of object names.

When saving as an HDF5 packed experiment file (.h5xp), /J is not supported and returns an error if you use it and provide a non-empty string as *objectNamesStr*.

See **Saving Specific Objects** below for further discussion of /J.

/L=*saveFlags* Controls what kind of data objects are saved with a bit for each data type:

saveFlags	Bit Number	Saves this Type of Object
1	0	Waves
2	1	Numeric variables
4	2	String variables

To save multiple data types, sum the values shown in the *saveFlags* column. For example, /L=1 saves waves only, /L=2 saves numeric variables only and /L=3 saves both waves and numeric variables.

If /L is not specified, all of these object types are saved. This is equivalent to /L=7. All other bits are reserved and must be set to zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/M=*modDateTime* Saves waves modified on or after the specified modification date/time. Waves modified before *modDateTime* will not be saved. Applies to waves only (not variables or strings).

modDateTime is in standard Igor time format — seconds since 1/1/1904. If *modDateTime* is zero, all waves will be saved, as if there were no /M flag at all.

/O Overwrites existing files or folders on disk.

Warning: If you use the /O flag and if the target already exists, it will be overwritten without any warning. If you use /O with /D=2, **you will completely overwrite the target folder and all of its contents, including subfolders**. Do not use /O with /D unless you are absolutely sure you know what your doing.

/P=*pathName* Specifies the folder in which to save the specified file or folder.

pathName is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\`". See **Symbolic Paths** on page II-22 for details.

When used with the /D flag, if /P=*pathName* is present and *fileOrFolderNameStr* is ":", the target is the directory specified by /P=*pathName*.

/Q Suppresses normal messages in the history area.

/R Recursively saves subdata folders.

/T [=*topLevelName*] Creates an enclosing data folder in the target with the specified name, *topLevelName*, and writes the data to the new data folder.

If just /T is specified, it creates an enclosing data folder in the target using the name of the data folder being saved. However, if the data folder being saved is the root data folder, the name Data is used instead of root. In packed experiment files and unpacked experiment folders, the root data folder is implicit.

If /T is omitted, the contents of the current data folder are saved with no enclosing data folder.

Saving Specific Objects

If /J=*objectNamesStr* is used, then only the objects named in *objectNamesStr* are saved. For example, specifying `/J="wave0;wave1;"` will save only the two named waves, ignoring any other data in all data folders.

The list of object names used with /J must be semicolon-separated. A semicolon after the last object name in the list is optional. The object names must not be quoted even if they are liberal. The list is limited to 1000 characters.

Using `/J=""` acts like no /J at all.

When saving as an HDF5 packed experiment file (.h5xp), /J is not supported and returns an error if you use it and provide a non-empty string as *objectNamesStr*.

Details

The /M=*modDateTime* flag can be used in data acquisition projects to save only those waves modified since the previous save. For example, assume that we have a global variable in the root data folder named gLastWaveSaveDateTime. Then this function will write out only those waves modified since the previous save:

```
Function SaveModifiedWaves(savePath)
   String savePath     // Symbolic path pointing to output directory
   NVAR lastSave = root:gLastWaveSaveDateTime
   SaveData/O/P=$savePath/D=1/L=1/M=(lastSave) ":"
   lastSave = datetime
End
```

Because the datetime function and the wave modification date have a coarse resolution (one second), this function may sometimes save the same wave twice.

The /M flag makes sense only in conjunction with the /D=1 flag because /D=1 is the only way to mix-in new data with existing data.

Writing to a Packed Experiment File

When writing to a packed file, SaveData creates a standard packed Igor experiment file which you can open as an experiment, browse using the Data Browser, or access using the **LoadData** operation.

If you do not use the /O (overwrite) flag and the packed file already exists on disk, SaveData will present a dialog to confirm which file you want to write to. If you use the /O flag, SaveData will overwrite without presenting a dialog. When writing a packed file, SaveData always completely overwrites the preexisting packed file.

Appending to a packed experiment file is not supported because dealing with the possibility of name conflicts (e.g., two waves with the same name in the same data folder in the packed experiment file) would be technically difficult, very slow and errors would result in corrupted files.

Writing to a File-System Folder

When saving to a folder on disk, SaveData writes wave files, variables files, and subfolders. This resembles the experiment folder of an unpacked experiment, but it does not contain other unpacked experiment files, such as history or procedures. You can browse the folder using the Data Browser or access it using the **LoadData** operation.

If the target directory does not exist, SaveData creates it.

If you do not use the /O (overwrite) flag and the target folder already exists on disk, SaveData will present a dialog to confirm that you want to write to it. SaveData checks for the existence of the top file system folder only. For example, if you write data to hd:Data:Run1, SaveData will display a dialog if hd:Data:Run1 exists. But SaveData will not display a dialog for any folders inside hd:Data:Run1.

If you use the /O flag, SaveData will write without presenting a dialog.

When writing to a directory, SaveData can operate in one of two modes. If you use /D=1 or just /D, SaveData operates in “mix-in” mode. If you use /D=2, SaveData operates in “delete” mode.

Warning: If the target directory exists and delete mode is used, SaveData deletes the target directory and all of its contents. Then SaveData creates the target directory and writes the data to it. This is a complete overwrite operation.

If the target directory exists and mix-in mode is used, SaveData does not do any explicit deletion. It writes data to the target directory and any subdirectories. Conflicting files in any directory are overwritten but other files are left intact.

To prevent you from inadvertently deleting an entire volume, SaveData will not permit you to target the root directory of any volume. You must target a subdirectory.

The /J flag will not work as expected when writing numeric and string variables in mix-in mode. Instead of mixing-in the specified variables, SaveData will overwrite all variables already in the target. This is because all numeric and string variables in a particular data folder are stored in a single file-system folder (named “variables”), so it is not possible to mix-in. Since waves are written one-to-a-file, /J will work as expected for waves.

When SaveData writes a wave to a file-system folder, the file name for the wave is the same as the wave name, with the extension “.ibw” added. This is true even if the wave in the experiment was loaded from a file with a different name.

Limitations of SaveData

SaveData does not save free waves, free data folders, wave waves or DFREF waves all of which are ignored by SaveData.

When saving as HDF5 packed (.h5xp), the entire experiment must use UTF-8 text encoding. If the experiment uses non-UTF8 text encoding, SaveData displays an alert directing you to Misc→Text Encoding→Convert to UTF-8 Text Encoding and returns an error.

When saving as HDF5 packed (.h5xp), the /J flag is not supported and returns an error if you use it and provide a non-empty string as objectNamesStr.

Outputs

SaveData sets the variable V_flag to zero if the operation succeeded or to nonzero if it failed. The main use for this is to determine if the user clicked Cancel during an interactive save. This would occur if you use the /I flag or if you omit /O and the target already exists. V_flag will also be nonzero if an error occurs during the save.

SaveData sets the string variable S_path to the full file system path to the file or folder that was written. S_path uses Macintosh path syntax (e.g., "hd:FolderA:FolderB:"), even on Windows. When saving unpacked, S_path includes a trailing colon.

Examples

Write the contents of the current data folder and all subdata folders to a packed experiment file:

```
Function SaveDataInPackedFile(pathName, fileName)
   String pathName               // Name of symbolic path
   String fileName               // Name of packed file to be written

   SaveData/R/P=$pathName fileName
End
```

Write the contents of the current data folder and all subdata folders to an unpacked file-system folder:

```
Function SaveDataInUnpackedFolder(pathName, folderName)
   String pathName               // Name of symbolic path
   String folderName             // Name of file-system folder

   SaveData/D=1/R/P=$pathName folderName
End
```

Copy the contents of an unpacked file-system folder to a packed experiment file:

```
Function TransferUnpackedToPacked(path1, folderName, path2, fileName)
   String path1           // Points to parent of unpacked folder
   String folderName      // Name of folder containing unpacked data
   String path2           // Points to folder where file is to be written
   String fileName        // Name of packed file to be written
```

```
    DFREF savedDF = GetDataFolderDFR()

    NewDataFolder/O/S :TempTransfer

    // Load all data from the unpacked folder.
    LoadData/D/Q/R/P=$path1 folderName

    // Save all data to the packed file.
    SaveData/R/P=$path2 fileName

    KillDataFolder :             // Kill TempTransfer

    SetDataFolder savedDF
End
```

See Also

The **LoadData** and **SaveGraphCopy** operations; the **SpecialDirPath** function. **Saving Package Preferences** on page IV-251; **Exporting Data** on page II-177; **The Data Browser** on page II-114.

SaveExperiment

SaveExperiment [**flags**] [**as** ***fileName***]

The SaveExperiment operation saves the current experiment.

Warning: SaveExperiment overwrites any previously-existing file named fileName.

Parameters

The optional *fileName* string contains the name of the experiment to be saved. *fileName* can be the currently open experiment, in which case it overwrites the experiment file.

If *fileName* and *pathName* are omitted and the experiment is Untitled, you will need to locate where the experiment file will be saved interactively via a dialog.

If you use a full or partial path for *pathName*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/C — Saves an experiment copy (valid only when *fileName* or *pathName* is provided or both if experiment is Untitled).

/COMP={*minWaveElements*, *gzipLevel*, *shuffle*}

Specifies that compression is to be applied to numeric waves saved when saving as an HDF5 packed experiment file. The /COMP flag was added in Igor Pro 9.00.

minWaveElements is the minimum number of elements that a numeric wave must have to be eligible for compression. Waves with fewer than this many total elements are not compressed.

gzipLevel is a value from 0 to 9. 0 means no GZIP compression.

shuffle is 0 to turn shuffle off or 1 to turn shuffle on.

When compression is applied by SaveExperiment, the entire wave is saved in one chunk. See **HDF5 Layout Chunk Size** on page II-214 for background information and **SaveExperiment Compression** on page II-214 for details.

/F={*format*, *unpackedExpFolderNameStr*, *unpackedExpFolderMode*}

Specifies the experiment file format.

See ***Experiment File Format*** below for details.

/P=*pathName* — Specifies folder in which to save the experiment. *pathName* is the name of an existing symbolic path.

Details

SaveExperiment acts like the Save menu command in the File menu. If the experiment is associated with an already saved file, then SaveExperiment with no parameters will simply save the current experiment. If the experiment resides only in memory and has not yet been saved, then a dialog will be presented unless the path and file name are specified.

If you use a full path in the name you will not need the /P flag. If instead you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-22 for details.

Experiment File Format

For background information on experiment file formats, see **Experiments** on page II-16.

The /F flag provides control of the file format of a previously-unsaved experiment independent of the user's preferences as set in the Experiment Settings section of the Miscellaneous Settings dialog. It also allows you to save a previously-saved experiment using a different experiment file format.

If you just want to save the current experiment in its current format, you don't need to use /F.

If you use /F, you must fully-specify the location of the experiment file through the /P flag and the *fileName* parameter or through *fileName* alone if it contains a full path.

The format parameter controls the experiment file format used by SaveExperiment:

format =-1:	Default format
format =0:	Unpacked experiment file
format =1:	Packed experiment file
format =2:	HDF5 packed experiment file

If /F is omitted or if *format* is -1 then the experiment is saved in its current format or, if it was never saved to disk, as a packed experiment file (.pxp).

If *format* = 0, the experiment is saved in unpacked experiment file format. *fileName* must end with ".uxp" or ".uxt".

If *format* = 1, the experiment is saved in packed experiment file format. *fileName* must end with ".pxp" or ".pxt".

If *format* = 2, the experiment is saved in HDF5 packed experiment file format. *fileName* must end with ".h5xp" or ".h5xt". This format requires Igor Pro 9.00 or later.

Unpacked Experiment Folder

The unpacked experiment folder is the folder in which wave files, the history file, the variables file, and other experiment files are stored for an unpacked experiment. See **Saving as an Unpacked Experiment File** on page II-17 for details.

The /F *unpackedExpFolderNameStr* parameter specifies the name of the experiment folder for an unpacked experiment. It contains a folder name, not a full or partial path. It is ignored unless saving in unpacked experiment format.

The unpacked experiment folder is created in the same directory as the experiment file.

If /F=0 is used and *unpackedExpFolderNameStr* is "" then the experiment folder name is the same as the experiment file name with the extension removed and a space and "Folder" added.

If the specified unpacked experiment folder already exists and is the current experiment's unpacked experiment folder, it is reused. "Reuse" means that SaveExperiment saves files in the unpacked experiment folder, possibly overwriting files already in it, but does not delete any files or folders already in it.

The *unpackedExpFolderMode* parameter controls what happens if the folder to be used as the unpacked experiment folder already exists and is not the current experiment's unpacked experiment folder:

unpackedExpFolderMode=0 :	SaveExperiment returns an error.
unpackedExpFolderMode=1 :	SaveExperiment displays a dialog asking the user if it is OK to reuse the folder. If the user answers yes, the operation proceeds. Otherwise, it returns an error.
unpackedExpFolderMode=2 :	SaveExperiment reuses the folder without asking the user.

Warning: If you pass 2 for *unpackedExpFolderMode*, files and folders in the unpacked experiment folder may be overwritten without the user's express permission.

See Also

Saving Experiments on page II-16, **ExperimentInfo**

SaveGizmoCopy

`SaveGizmoCopy` `[`***`flags`***`][`**`as`** ***`fileNameStr`***`]`

The SaveGizmoCopy operation saves a Gizmo window and its waves in an Igor packed experiment file.

SaveGizmoCopy was added in Igor Pro 8.00.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/I	Presents a dialog from which you can specify file name and folder.
/O	Overwrites file if it exists already.
/P=*pathName*	Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.
/T=*saveType*	Specifies the file format of the saved file. *saveType*=0: Packed experiment file. *saveType*=1: HDF5 packed experiment file. If *fileNameStr* is specified the file name extension must be ".h5xp". The /T flag was added in Igor Pro 9.00.
/W= *winName*	*winName* is the name of the Gizmo window to be saved. If /W is omitted or if *winName* is `""`, the top Gizmo window is saved.
/Z	Errors are not fatal and error dialogs are suppressed. See Details.

Details

The main uses for saving as a packed experiment are to save an archival copy of data or to prepare to merge data from multiple experiments (see **Merging Experiments** on page II-19). The resulting experiment file preserves the data folder hierarchy of the waves displayed in the Gizmo window starting from the root data folder. Only the Gizmo window, its waves as well as the objects on Gizmo's object list and attribute list are saved in the packed experiment file. Associated procedures including hook functions are not saved.

SaveGizmoCopy does not know about dependencies. If a Gizmo window contains a wave, wave0, that is dependent on another wave, wave1 which is not used in the Gizmo window, SaveGizmoCopy will save wave0 but not wave1. When the saved experiment is open, there will be a broken dependency.

SaveGizmoCopy sets the variable V_flag to 0 if the operation completes normally, to -1 if the user cancels, or to another nonzero value that indicates that an error occurred. If you want to detect the user canceling an interactive save, use the /Z flag and check V_flag after calling SaveGizmoCopy.

See Also

SaveGraphCopy, **SaveTableCopy**, **SaveData**, **Merging Experiments** on page II-19

SaveGraphCopy

`SaveGraphCopy` `[`***`flags`***`][`**`as`** ***`fileNameStr`***`]`

The SaveGraphCopy operation saves a graph and its waves in an Igor packed experiment file.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/I Presents a dialog from which you can specify file name and folder.

/O Overwrites file if it exists already.

/P=*pathName* Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.

/T=*saveType* Specifies the file format of the saved file.

saveType=0: Packed experiment file.

saveType=1: HDF5 packed experiment file. If *fileNameStr* is specified the file name extension must be ".h5xp".

The /T flag was added in Igor Pro 9.00.

/W= *winName* *winName* is the name of the graph to be saved. If /W is omitted or if *winName* is "", the top graph is saved.

/Z Errors are not fatal and error dialogs are suppressed. See Details.

Details

The main uses for saving as a packed experiment are to save an archival copy of data or to prepare to merge data from multiple experiments (see **Merging Experiments** on page II-19). The resulting experiment file preserves the data folder hierarchy of the waves displayed in the graph starting from the “top” data folder, which is the data folder that encloses all waves displayed in the graph. The top data folder becomes the root data folder of the resulting experiment file. Only the graph, its waves, dashed line settings, and any pictures used in the graph are saved in the packed experiment file, not procedures, variables, strings or any other objects in the experiment.

SaveGraphCopy does not work well with graphs containing controls. First, the controls may depend on waves, variables or FIFOs (for chart controls) that SaveGraphCopy will not save. Second, controls typically rely on procedures which are not saved by SaveGraphCopy.

SaveGraphCopy does not know about dependencies. If a graph contains a wave, wave0, that is dependent on another wave, wave1 which is not in the graph, SaveGraphCopy will save wave0 but not wave1. When the saved experiment is open, there will be a broken dependency.

SaveGraphCopy sets the variable V_flag to 0 if the operation completes normally, to -1 if the user cancels, or to another nonzero value that indicates that an error occurred. If you want to detect the user canceling an interactive save, use the /Z flag and check V_flag after calling SaveGraphCopy.

The **SaveData** operation also has the ability to save data from a graph to a packed experiment file. SaveData is more complex but a bit more flexible than SaveGraphCopy.

Examples

This function saves all graphs in the experiment to individual packed experiment files.

```
Function SaveAllGraphsToPackedFiles(pathName)
    String pathName         // Name of an Igor symbolic path.

    String graphName
    Variable index

    index = 0
    do
        graphName = WinName(index, 1)
        if (strlen(graphName) == 0)
            break
        endif
        String fileName
        sprintf fileName, "%s.pxp", graphName

        SaveGraphCopy/P=$pathName/W=$graphName as fileName

        index += 1
    while(1)
End
```

See Also

SaveTableCopy, **SaveGizmoCopy**, **SaveData**, **Merging Experiments** on page II-19

SaveNotebook

`SaveNotebook [`*`flags`*`] `*`notebookName`*` [as `*`fileNameStr`*`]`

The SaveNotebook operation saves the named notebook.

Parameters

notebookName is either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-92 for details on host-child specifications.

If *notebookName* is an host-child specification, /S must be used and *saveType* must be 3 or higher.

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/ENCG=*textEncoding*

Specifies text encoding in which the notebook is to be saved.

This flag was added in Igor Pro 7.00.

This is relevant for plain text notebooks only and is ignored for formatted notebooks because they can contain multiple text encodings. See **Plain Text File Text Encodings** on page III-466 and **Formatted Text Notebook File Text Encodings** on page III-472 for details.

If omitted, the file is saved in its original text encoding. Normally you should omit /ENCG. Use it only if you have some reason to change the file's text encoding.

Passing 0 for *textEncoding* acts as if /ENCG were omitted.

See **Text Encoding Names and Codes** on page III-490 for a list of accepted values for *textEncoding*.

This flag does not affect HTML export. Use /H instead.

/H={*encodingName*, *writeParagraphProperties*, *writeCharacterProperties*, *PNGOrJPEG*, *quality*, *bitDepth*}

Controls the creation of an HTML file.

encodingName specifies the HTML file text encoding. The recommended value is "UTF-8".

writeParagraphProperties determines what paragraph properties SaveNotebook will write to the HTML file. This is a bitwise parameter with the bits defined as follows:

Bit 0: Write paragraph alignment.
Bit 1: Write first indent.
Bit 2: Write minimum line spacing.
Bit 3: Write space-before and space-after paragraph.

All other bits are reserved for future use and should be set to zero.

writeCharacterProperties determines what character properties SaveNotebook will write to the HTML file. This is a bitwise parameter with the bits defined as follows:

Bit 0: Write font families.
Bit 1: Write font sizes.
Bit 2: Write font styles.
Bit 3: Write text colors.
Bit 4: Write text vertical offsets.

All other bits are reserved for future use and should be set to zero.

If you set bit 2, SaveNotebook exports only the bold, underline, and italic styles because other character styles are not supported by HTML.

PNGOrJPEG determines whether SaveNotebook will write picture files as PNG or JPEG:

0: PNG (default).
1: JPEG.
2: JPEG.

In Igor7 and later, there is no difference between *PNGOrJPEG*=1 and *PNGOrJPEG*=2.

See **Details** for more on HTML picture files.

quality specifies the degree of compression or image quality when writing pictures as JPEG files. Legal values are in the range 0.0 to 1.0.

In Igor7 or later, the quality used is 0.9 regardless of what you pass for this parameter.

bitDepth specifies the color depth when writing pictures as JPEG files. Legal values are legal: 1, 8, 16, 24, and 32.

In Igor7 or later, the bit depth used is 32 regardless of what you pass for this parameter.

/I Saves interactively. A dialog is displayed.

/M=*messageStr* Specifies prompt message used in save dialog.

/O Overwrites existing file without asking permission.

/P=*pathName* Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.

/S=*saveType* Controls the type of save.

saveType=1:	Normal save (default).
saveType=2:	Save-as.
saveType=3:	Save-a-copy.
saveType=4:	Export as RTF (Rich Text Format).
saveType=5:	Export as HTML (Hypertext Markup Language).
saveType=6:	Export as plain text.
saveType=7:	Export as formatted notebook.
saveType=8:	Export as plain text with line breaks.

Details

Interactive (/I) means that Igor displays the Save, Save As, or Save a Copy dialog.

The save will be interactive under the following conditions:

- You include the /I flag and the *saveType* is 2, 3, 4, 5, 6, 7 or 8.
- *saveType* is 2, 3, 4, 5, 6, 7 or 8 and you do not specify the path or filename.

If the *saveType* is normal and the notebook has previously been saved to a file then the /I flag, the path and file name that you specify, if any, are ignored and the notebook is saved to its associated file without user intervention.

The full path to the saved file is stored in the string S_path. If the save was unsuccessful, S_path will be `""`.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like “`hd:Folder1:`” or “`C:\\Folder1\\`”. See **Symbolic Paths** on page II-22 for details.

saveType=8 applies to formatted notebooks only. It exports the notebook as plain text with line breaks where text wraps in the formatted notebook. This feature was added in Igor Pro 8.00. Special characters such as pictures are skipped. If the notebook is plain text then *saveType*=8 acts like *saveType*=6.

Exporting as RTF

For background information on writing RTF files, see **Import and Export Via Rich Text Format Files** on page III-20.

Exporting as HTML

For background information on writing HTML files, see **Exporting a Notebook as HTML** on page III-21.

You can pass “UTF-8” or “UTF-2” for the *encodingName* parameter. In virtually all cases, you should use “UTF-8”.

When creating an HTML file, SaveNotebook can write pictures using the PNG or JPEG graphics formats. PNG is recommended because it is lossless.

See Also

Chapter III-1, **Notebooks**.

Setting Bit Parameters on page IV-12 for further details about bit settings.

SavePackagePreferences

`SavePackagePreferences` `[`**`/FLSH=`*`flush`* `/KILL /P=`*`pathName`***`]` ***`packageName`*`,`** ***`prefsFileName`*`,`** ***`recordID`*`,`** ***`prefsStruct`***

The SavePackagePreferences operation saves preference data in the specified structure so that it can be accessed later via the **LoadPackagePreferences** operation.

Note: The package preferences structure must not use fields of type Variable, String, WAVE, NVAR, SVAR or FUNCREF because these fields refer to data that may not exist when LoadPackagePreferences is called.

The structure can use fields of type char, uchar, int16, uint16, int32, uint32, int64, uint64, float and double as well as fixed-size arrays of these types and substructures with fields of these types.

The data is stored in memory and by default flushed to disk when the current experiment is saved or closed and when Igor quits.

If the /P flag is present then the location on disk of the preference file is determined by *pathName* and *prefsFileName*. However in the usual case the /P flag will be omitted and the preference file is located in a file named *prefsFileName* in a directory named *packageName* in the Packages directory in Igor’s preferences directory.

Note: You must choose a very distinctive name for *packageName* as this is the only thing preventing collisions between your package and someone else’s package.

See **Saving Package Preferences** on page IV-251 for background information and examples.

Parameters

packageName is the name of your package of Igor procedures. It is limited to 31 bytes and must be a legal name for a directory on disk. This name must be very distinctive as this is the only thing preventing collisions between your package and someone else’s package.

prefsFileName is the name of a preference file to be saved by SavePackagePreferences. It should include an extension, typically ".bin".

prefsStruct is the structure containing the data to be saved in the preference file on disk.

recordID is a unique positive integer that you assign to each record that you store in the preferences file. If you store more than one structure in the file, you would use distinct *recordID*s to identify which structure you want to save. In the simple case you will store just one structure in the preference file and you can use 0 (or any positive integer of your choice) as the *recordID*.

Flags

/FLSH=*flush* Controls when the data is actually written to the preference file:

flush=0: The data will be flushed to disk when the current experiment is saved, reverted or closed or when Igor quits. This is the default behavior used when /FLSH is omitted and is recommended for most purposes.

flush=1: The data is flushed to disk immediately.

/KILL Instead of saving prefsStruct under the specified record ID, that record is deleted from the package's preference if it exists. If it does not exist, nothing is done and no error is returned.

/P=*pathName* Specifies the directory in which to save the file specified by *prefsFileName*.

pathName is the name of an existing symbolic path. See **Symbolic Paths** on page II-22 for details.

`/P=$<empty string variable>` acts as if the /P flag were omitted.

Details

SavePackagePreferences sets the following output variables:

`V_flag`	Set to 0 if preferences were successfully saved or to a nonzero error code if they were not saved. The latter case is unlikely and would indicate some kind of corruption such as if Igor's preferences directory were deleted.
`V_structSize`	Set to the size in bytes of prefsStruct. This may be useful in handling structure version changes.

Example

See the example under **Saving Package Preferences in a Special-Format Binary File** on page IV-252.

See Also

LoadPackagePreferences.

SavePICT

`SavePICT [`*`flags`*`] [as `*`fileNameStr`*`]`

The SavePICT operation creates a picture file representing the top graph, table or layout. The picture file can be opened by many word processing, drawing, and page layout programs.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

If you omit *fileNameStr* but include /P=*pathName*, SavePICT writes the file using a default file name. The default file name is the window name followed by an extension, such as ".png", ".emf" or ".svg", that depends on the graphic format being exported.

If you specify the file name as "`Clipboard`", and do *not* specify a /P=*pathName*, Igor copies the picture to the Clipboard, rather than to a file. EPS is a file-only format and can not be stored in the clipboard.

If you specify the file name as "`_string_`" the output will be saved into a string variable named S_Value, which is used with the **ListBox** binary bitmap display mode.

If you use the special name _PictGallery_ with the /P flag, then the picture will be stored in Igor's picture gallery (see **Pictures** on page III-509) with the name you provide via *fileNameStr*. This feature was added in support of making movies using the /PICT flag with **NewMovie**.

Flags

/B=*dpi* Controls image resolution in dots-per-inch (*dpi*). The legal values for *dpi* are *n**72 where *n* can be from 1 to 8. The actual image *dpi* is not used. Igor calculates *n* from your value of *dpi* and then multiplies *n* by your computer's screen resolution. This is because bitmap images that are not an integer multiple of the screen resolution look quite bad.

Also see the /RES flag.

/C=*c* Specifies color mode.

c=0: Black and white.

c=1: RGB color (default).

c=2: CMYK color (EPS and native TIFF only).

/D=*d* Obsolete in Igor Pro 7 or later.

/E=*e* Sets graphics format used when exporting a graphic. See **Details** for formats. See also Chapter III-5, **Exporting Graphics (Macintosh)**, or Chapter III-6, **Exporting Graphics (Windows)**, for a description of these modes and when to use them.

/EF= *e* Sets font embedding.

e=0: No font embedding. Not honored in Igor Pro 7 or later.

e=1: Embed nonstandard fonts.

e=2: Embed all fonts.

/I Specifies that /W coordinates are inches.

/M Specifies that /W coordinates are centimeters.

/N=*winSpec* /N is antiquated but still supported. Use /WIN instead.

/O Overwrites file if it exists.

/P=*pathName* Saves file into a folder specified by *pathName*, which is the name of an existing symbolic path.

/PGR=(*firstPage*, *lastPage*)

Controls which pages in a multi-page layout are saved.

firstPage and lastPage are one-based page numbers. All pages from *firstPage* to *lastPage* are saved if the file format supports it.

The special value 0 refers to the current page and -1 refers to the last page in the layout.

Currently only the PDF formats support saving multiple pages. Other file formats save only *firstPage* and ignore the value of *lastPage*.

/PGR was added in Igor Pro 7.00.

/PICT=*pict* Saves specified named picture rather than the target window. Native format of the picture is used and all format flags are ignored.

/PLL=*p* Specifies Postscript language level when used in conjunction with EPS export.

p=1: For very old Postscript printers.

p=2: For all other uses (default).

/Q=*q* Sets quality factor (0.0 is lowest, 1.0 is highest). Default is dependent on individual format. Used only by lossy formats such as JPEG.

/R=*resID* Obsolete in Igor Pro 7 or later.

/RES=*dpi* Controls the resolution of image formats in dots-per-inch. Unlike the similar /B flag, the value for /RES is the actual output resolution and is useful when your publisher demands a specific resolution.

/S Suppresses the preview that is normally included with an EPS file.

Obsolete in Igor Pro 7 or later.

/SNAP=*s* Saves a snapshot (screen dump) of a graph or panel window.

s=1: Include all controls in capture.

s=2: Capture only window data content.

Snapshot mode is available only for graphs and panels and only for bitmap export formats PNG, JPEG, and TIFF at screen resolution. When using /W to specify the size of a graph, the capture is sized to fit within the specified rectangle while maintaining the window aspect ratio. Coordinates used with /W are in pixels.

/T=*t* Obsolete QuickTime export type. Not supported in Igor Pro 7 or later.

/TRAN[=1 *or* 0] Makes white background areas transparent using an RGBA type PNG when used with native PNG export of graphs or page layouts.

/W=(*left*,*top*,*right*,*bottom*)

Specifies the size of the picture when exporting a graph. If /W is omitted, it uses the graph window size.

When exporting a page layout, specifies the part of the page to export. Only objects that fall completely within the specified area are exported. If /W is omitted, the area of the layout containing objects is exported.

When exporting a page layout in Igor Pro 7.00 or later, you can specify /W=(0,0,0,0) to use the full page size.

Coordinates for /W are in points unless /I or /M are specified before /W.

/WIN=*winSpec* Saves the named window or subwindow. *winSpec* can be just a window name, or a window name following by a “#” character and the name of the subwindow, as in `/WIN=Panel0#G0`.

/Z Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SavePICT sets the variable V_flag to 0 if the operation succeeds or to a nonzero error code if it fails.

If you specify a path using the /P=*pathName* flag, then Igor saves the file in the folder identified by the path. Note that *pathName* is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like “`hd:Folder1:`” or “`C:\\Folder1\\`”. See **Symbolic Paths** on page II-22 for details. Otherwise, with no path specified, Igor presents a standard save dialog to let you specify where the file is to be saved.

Graphics formats, specified via /E, are as follows:

/E Value	Macintosh File Format	Windows File Format
-9	SVG file.	SVG file.
-8	PDF file.	PDF file.

/E Value	Macintosh File Format	Windows File Format
-7	TIFF file. Lossless but larger file than PNG; best for text, graph traces, and simple images with sharp edges. The default resolution is 72 dpi. You can specify the resolution with the /B or /RES flag. Cross-platform compatible.	
-6	JPEG file. Lossy compression; best used for grayscale and color images with smooth tones. The /Q flag specifies compression quality and the /B or /RES flag sets the resolution. Cross-platform compatible. PNG is a better choice for scientific graphics.	
-5	PNG (Portable Network Graphics) file. Lossless compression; best for text, graph traces, and simple images with sharp edges. The default resolution is 72 dpi. Specify the resolution with /B or /RES. Cross-platform compatible.	
-4	High resolution bitmap PICT file. Default resolution is 288 dpi. Specify the resolution with /B or /RES.	Device-independent bitmap file (DIB). Default resolution is 4x screen resolution. Specify the resolution with /B or /RES.
-3	Encapsulated PostScript (EPS) file. Use /S to suppress the screen preview if exporting to Latex.	Encapsulated PostScript (EPS) file. Use /S to suppress the screen preview if exporting to Latex.
-2	Quartz PDF.	High-resolution Enhanced Metafile (EMF).
-1	Quartz PDF (was PostScript PICT).	Obsolete (was PostScript-enhanced metafile).
0	Quartz PDF (was PostScript PICT with QuickDraw text).	Obsolete (was PostScript-enhanced metafile).
1	Low resolution Quartz PDF at 1x normal size.	High-resolution Enhanced Metafile (EMF).
2	Low resolution Quartz PDF at 2x normal size.	High-resolution Enhanced Metafile (EMF).
4	Low resolution Quartz PDF at 4x normal size.	High-resolution Enhanced Metafile (EMF).
8	Low resolution Quartz PDF at 8x normal size.	High-resolution Enhanced Metafile (EMF).

The low resolution PDF formats on Macintosh are probably not useful and are just placeholders for compatibility with old procedures.

See Also

The **ImageSave** operation for saving waves as PICTs and other image file formats. The **LoadPICT** operation.

See Chapter III-5, **Exporting Graphics (Macintosh)**, or Chapter III-6, **Exporting Graphics (Windows)**, for a description of the /E modes.

SaveTableCopy

SaveTableCopy [***flags***][**as** ***fileNameStr***]

The SaveTableCopy operation saves a copy of the data displayed in a table on disk. The saved file can be an Igor packed experiment file, a tab-delimited text file, or a comma-separated values text file.

When saving as text, by default the data format matches the format shown in the table. This causes trunctation if the underlying data has more precision than shown in the table. If you specify /F=1, SaveTableCopy uses as many digits as needed to represent the data with full precision.

The point column is never saved.

To save data as text with full precision, use the **Save** operation.

When saving 3D and 4D waves as text, only the visible layer is saved. To save the entirety of a 3D or 4D wave, use the **Save** operation.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/A=*a* Appends data to the file rather than overwriting.

a=0: Does not append.

a=1: Appends to the file with a blank line before the appended data.

a=2: Appends to the file with no blank line before the appended data.

/A applies when saving text files and is ignored when saving packed experiment files.

If the file does not exist, a new file is created and /A has no effect.

/F=*f* Controls the precision of saved numeric data.

f=0: Numeric data is written exactly as shown in the table. This may cause truncation. This is the default behavior if /F is omitted.

f=1: Numeric data is written with as many digits as needed to represent the data with full precision.

The /F flag was added in Igor Pro 7.00

/I Presents a dialog from which you can specify file name and folder.

/M=*termStr* Specifies the terminator character or characters to use at the end of each line of text. The default is `/M="\r"` on Macintosh and `/M="\r\n"` on Windows; it is used when /M is omitted. To use the Unix convention, just a linefeed, specify `/M="\n"`.

/N=*n* Specifies whether to use column names, titles, or dimension labels.

n is a bitwise parameter with the bits defined as follows:

Bit 0: Include column names or titles. The column title is included if it is not empty. If it is empty, the column name is included.

Bit 1: Include horizontal dimension labels if they are showing in the table.

The default setting for *n* is 1. All other bits are reserved and must be zero.

/O Overwrites file if it exists already.

/P=*pathName* Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.

/S=*s* Saves all of the data in the table (*s*=0; default) or the selection only (*s*=1).

/S applies when saving text files and is ignored when saving packed experiment files.

/T=*saveType* Specifies the file format of the saved file.

saveType=0: Packed experiment file.

saveType=1: Tab-delimited text file.

saveType=2: Comma-separated values text file.

saveType=3: Space-delimited values text file.

saveType=4: HDF5 packed experiment file (requires Igor Pro 9 or later). If *fileNameStr* is specified the file name extension must be ".h5xp".

/W= *winName* *winName* is the name of the table to be saved. If /W is omitted or if *winName* is `""`, the top table is saved.

/Z Errors are not fatal and error dialogs are suppressed. See Details.

Details

The main uses for saving a table as a packed experiment are to save an archival copy of data or to prepare to merge data from multiple experiments (see **Merging Experiments** on page II-19). The resulting experiment file preserves the data folder hierarchy of the waves displayed in the table starting from the "top" data folder, which is the data folder that encloses all waves displayed in the table. The top data folder becomes the root data folder of the resulting experiment file. Only the table and its waves are saved in the packed experiment file, not variables or strings or any other objects in the experiment.

SaveTableCopy does not know about dependencies. If a table contains a wave, wave0, that is dependent on another wave, wave1 which is not in the table, SaveTableCopy will save wave0 but not wave1. When the saved experiment is open, there will be a broken dependency.

The main use for saving as a tab or comma-delimited text file is for exporting data to another program.

When calling SaveTableCopy from a procedure, you should call DoUpdate before calling SaveTable copy. This insures that the table is up-to-date if your procedure has redimensioned or otherwise changed the number of points in the waves in the table.

SaveTableCopy sets the variable V_flag to 0 if the operation completes normally, to -1 if the user cancels, or to another nonzero value that indicates that an error occurred. If you want to detect the user canceling an interactive save, use the /Z flag and check V_flag after calling SaveTableCopy.

The **SaveData** operation also has the ability to save a table to a packed experiment file. SaveData is more complex but a bit more flexible than SaveTableCopy.

Examples

This function saves all tables to a single tab-delimited text file.

```
Function SaveAllTablesToTextFile(pathName, fileName)
    String pathName         // Name of an Igor symbolic path.
    String fileName

    String tableName
    Variable index

    index = 0
    do
        tableName = WinName(index, 2)
        if (strlen(tableName) == 0)
            break
        endif

        SaveTableCopy/P=$pathName/W=$tableName/T=1/A=1 as fileName

        index += 1
    while(1)
End
```

See Also

SaveGraphCopy, **SaveGizmoCopy**, **SaveData**, **Merging Experiments** on page II-19

sawtooth

sawtooth(*num*)

The sawtooth function returns ((*num* +n2π) mod 2π)/2π where n is used to correct if *num* is negative. Sawtooth is used to create arbitrary periodic waveforms like sine and cosine.

Examples

```
wave1 = sawtooth(x)
```

creates a sawtooth in wave1 whose Y values range from 0 to 1 as its X values go through 2π units.

```
wave1 = exp(sawtooth(x))
```

creates a series of exponentials in wave1 of amplitude exp(1) and period 2π.

You can also use sawtooth to create periodic repetitions of a given part of a wave:

```
wave1 = wave2(sawtooth(x))
```

creates a periodic repetition of wave2 in wave1 given the correct X scaling for the waves.

ScaleToIndex

`ScaleToIndex(`*`wave`*`, `*`coordValue`*`, `*`dim`*`)`

The ScaleToIndex function returns the number of the element in the requested dimension whose scaled index value is closest to *coordValue*.

The ScaleToIndex function was added in Igor Pro 7.00.

Parameters

dim is a dimension number: 0 for rows, 1 for columns, 2 for layers, 3 for chunks.

coordValue is a scaled index in that dimension.

Details

The ScaleToIndex function returns the value of the expression:

```
round((coordValue - DimOffset(wave,dim)) / DimDelta(wave,dim))
```

With *dim*=0, ScaleToIndex is equivalent to **x2pnt**.

If *coordValue* is NaN or +/-INF, ScaleToIndex returns NaN. Otherwise, the result is computed based on the **DimOffset** and **DimDelta** of the specified dimension of the wave. The result is not clipped to a valid element number for the wave dimension.

See Also

IndexToScale, **x2pnt**, **DimDelta**, **DimOffset**

Waveform Model of Data on page II-62 for an explanation of wave scaling.

ScreenResolution

`ScreenResolution`

The ScreenResolution function returns the logical resolution of your video display screen in dots per inch (dpi). On Macintosh this is always 72. On Windows it is usually 96 (small fonts) or 120 (large fonts).

Examples

```
// 72 is the number of points in an inch which is constant.
Variable pixels = numPoints * (ScreenResolution/72)   // Convert points to pixels
Variable points = numPixels * (72/ScreenResolution)   // Convert pixels to points
```

See Also

PanelResolution

sec

`sec(`*`angle`*`)`

The sec function returns the secant of *angle* which is in radians:

$$\sec(x) = \frac{1}{\cos(x)}.$$

In complex expressions, *angle* is complex, and sec(*angle*) returns a complex value.

See Also

sin, **cos**, **tan**, **csc**, **cot**

sech

`sech(`*`x`*`)`

The sech function returns the hyperbolic secant of *x*.

$$\operatorname{csch}(x) = \frac{1}{\cosh(x)} = \frac{2}{e^x + e^{-x}}.$$

In complex expressions, *x* is complex, and sech(*x*) returns a complex value.

See Also

cosh, **tanh**, **coth**, **csch**

Secs2Date

Secs2Date(*seconds*, *format* [, *sep*])

The Secs2Date function returns a string containing a date.

With *format* values 0, 1, and 2, the formatting of dates depends on operating system settings entered in the Language & Region control panel (*Macintosh*) or the Region control panel (*Windows*). These date formats do not work with dates before 0001-01-01 in which case Date2Secs returns an empty string.

If *format* is -1, the format is independent of operating system settings. The fixed-length format is "*day* /*month* /*year* (*dayOfWeekNum*)", where *dayOfWeekNum* is 1 for Sunday, 2 for Monday… and 7 for Saturday.

If format is -2, the format is YYYY-MM-DD.

The optional sep parameter affects format -2 only. If sep is omitted, the separator character is "-". Otherwise, sep specifies the separator character.

Parameters

seconds is the number of seconds from 1/1/1904 to the date to be returned.

seconds is limited to the range -1094110934400 (-32768-01-01) to 973973807999 (32768-12-31). For *seconds* outside that range, Secs2Date returns an empty string.

format is a number between -2 and 2 which specifies how the date is to be constructed.

Examples

```
Print Secs2Date(DateTime,-2)      // 1993-03-14
Print Secs2Date(DateTime,-2,"/")  // 1993/03/14
Print Secs2Date(DateTime,-1)      // 15/03/1993 (2)
Print Secs2Date(DateTime,0)       // 3/15/93 (depends on system settings)
Print Secs2Date(DateTime,1)       // Monday, March 15, 1993 (depends on system settings)
Print Secs2Date(DateTime,2)       // Mon, Mar 15, 1993 (depends on system settings)
```

See Also

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-85.

The **date**, **date2secs** and **DateTime** functions.

Secs2Time

Secs2Time(*seconds*, *format*, [*fracDigits*])

The Secs2Time function returns a string containing a time.

Parameters

seconds is the number of seconds from 1/1/1904 to the time to be returned.

format is a number between 0 and 5 that specifies how the time is to be constructed. It is interpreted as follows:

0: Normal time, no seconds.

1: Normal time, with seconds.

2: Military time, no seconds.

3: Military time, with seconds and optional fractional seconds.

4: Elapsed time, no seconds.

5: Elapsed time, with seconds and optional fractional seconds.

"Normal" formats (0 and 1) follow the preferred formatting of the short time format as set in the International control panel (*Macintosh*) or in the Regional and Language Options control panel (*Windows*).

"Military" means that the hour is a number from 0 to 23. Hours greater than 23 are wrapped.

"Elapsed" means that the hour is a number from -9999 to 9999. The result for hours outside that range is undefined.

The *fracDigits* parameter is optional and specifies the number of digits of fractional seconds. The default value is 0. The *fracDigits* parameter is ignored for format=0, 1, 2,and 4.

Examples

```
Print Secs2Time(DateTime,0)                 // prints 1:07 PM
Print Secs2Time(DateTime,1)                 // prints 1:07:28 PM
Print Secs2Time(DateTime,2)                 // prints 13:07
Print Secs2Time(DateTime,3)                 // prints 13:07:29
Print Secs2Time(30*60*60+45*60+55,4)        // Prints 30:45
Print Secs2Time(30*60*60+45*60+55,5)        // Prints 30:45:55
```

See Also

For a discussion of how Igor represents dates, see **Date/Time Waves** on page II-85.

The **Secs2Date**, **date**, **date2secs** and **DateTime** functions. Also, **Operators** on page IV-6 for ?: details.

SelectNumber

SelectNumber(*whichOne*, *val1*, *val2* [, *val3*])

The SelectNumber function returns one of *val1*, *val2*, or (optionally) *val3* based on the value of *whichOne*.

SelectNumber(*whichOne*, *val1*, *val2*) returns *val1* if *whichOne* is zero, else it returns *val2*.

SelectNumber(*whichOne*, *val1*, *val2*, *val3*) returns *val1* if *whichOne* is negative, *val2* if *whichOne* is zero, or *val3* if *whichOne* is positive.

Details

SelectNumber works with complex (or real)*val1*, *val2*, and *val3* when the result is assigned to a complex wave or variable. (Print expects a real result, see the "causes error" example, below).

If *whichOne* is NaN, then NaN is returned.

whichOne must always be a real value.

Unlike the ? : conditional operator, SelectNumber always evaluates all of the numeric expression parameters *val1*, *val2*, …

SelectNumber works in a macro, whereas the conditional operator does not.

Examples

```
Print SelectNumber(0,1,2)                      // prints 1
Print SelectNumber(0,1,2,3)                    // prints 2
wv=SelectNumber(numtype(wv[p])==2,wv[p],0)     // replace NaNs with zeros

// chooses among complex values
Variable/C cx= SelectNumber(negZeroPos,cmplx(-1,-1),0,cmplx(1,1))

// causes error because Print expects a real value (not complex)
Print SelectNumber(negZeroPos,cmplx(-1,-1),0,cmplx(1,1))

// The real function expects a complex result
Print real(SelectNumber(negZeroPos,cmplx(-1,-1),0,cmplx(1,1)))
```

See Also

The **SelectString** and **limit** functions, and **Waveform Arithmetic and Assignments** on page II-74. Also, **Operators** on page IV-6 for details about the ?: operator.

SelectString

SelectString(*whichOne*, *str1*, *str2* [, *str3*])

The SelectString function returns one of *str1*, *str2*, or (optionally) *str3* based on the value of *whichOne*.

SelectString(*whichOne*, *str1*, *str2*) returns *str1* if *whichOne* is zero, else it returns *str2*.

SelectString(*whichOne*, *str1*, *str2*, *str3*) returns *str1* if *whichOne* is negative, *str2* if *whichOne* is zero, or *str3* if *whichOne* is positive.

Details

If *whichOne* is NaN, then `""` is returned.

whichOne must always be a real value.

Unlike the ? : conditional operator, SelectString always evaluates all of the string expression parameters *str1*, *str2*, …

SelectString works in a macro, whereas the conditional operator does not.

Examples

```
Print SelectString(0,"hello","there")                // prints "hello"
Print SelectString(1,"hello","there")                // prints "there"
Print SelectString(-3,"hello","there","jack")        // prints "hello"
Print SelectString(0,"hello","there","jack")         // prints "there"
Print SelectString(100,"hello","there","jack")       // prints "jack"
```

See Also

The **SelectNumber** function and **String Expressions** on page IV-13. Also, **Operators** on page IV-6 for details about the ?: operator.

SetActiveSubwindow

`SetActiveSubwindow` ***`subWinSpec`***

The SetActiveSubwindow operation specifies the subwindow that is to be activated. This operation is mainly for use by recreation macros.

Parameters

subWinSpec specifies an existing subwindow. See **Subwindow Syntax** on page III-92 for details on subwindow specifications.

Use `_endfloat_` for *subWinSpec* to make a newly-created floating panel not be the default target.

See Also

GetWindow with the activeSW keyword.

SetAxis

`SetAxis` `[`***`flags`***`]` ***`axisName`*** `[,` ***`num1`***`,` ***`num2`***`]`

The SetAxis operation sets the extent (or "range") of the named axis.

Parameters

axisName is usually "left", "right", "top" or "bottom", but it can also be the name of a free axis, such as "vertCrossing".

If *axisName* is a vertical axis such as "left" or "right" then *num1* sets the bottom end of the axis and *num2* sets the top end of the axis.

If *axisName* is a horizontal axis such as "top" or "bottom" then *num1* sets the left end of the axis and *num2* sets the right end of the axis.

You can flip the graph by reversing *num1* and *num2* (or by using /A/R). This is particularly useful for images, because Igor plots an image inverted.

If you pass * (asterisk) for *num1* and/or *num2* then the corresponding end of the axis will be autoscaled.

Flags

/A[=*a*]	Autoscale axis (when used, *num1*, *num2* should be omitted).	
	a=0:	No autoscale. Same as no /A flag.
	a=1:	Normal autoscale. Same as /A.
	a=2:	Autoscale Y axis to a subset of the data defined by the current X axis range.

/E=*z* Sets the treatment of zero when the axis is in autoscale mode.

z=0: Normal mode where zero is not treated special.

z=1: Forces the smaller end of the axis to be set to zero (autoscale from zero).

z=2: Axis is symmetric about zero.

z=3: If the data is unipolar (all positive or all negative), this behaves like /E=1 (autoscale from zero). If the data is bipolar, it behaves like /E=0 (normal autoscaling).

/N=*n* Sets the algorithm for axis autoscaling.

n=0: Normal mode; sets the axis limits equal to the data limits.

n=1: Picks nice values for the axis limits.

n=2: Picks nice values; also ensures that the data is inset from the axis ends.

/R Reverses the autoscaled axis (smaller values at the left for horizontal axes, at the top for vertical axes) when used with /A. Although it only has an effect for autoscale, it can be used with nonautoscale version of SetAxis so that the next time the Axis Range tab is used the “reverse axis” checkbox will already be set.

/W=*winName* Sets axes in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/Z No error reporting if named axis doesn’t exist in a style macro.

SetBackground

```
SetBackground numericExpression
```

The SetBackground operation sets *numericExpression* as the current unnamed background task.

SetBackground works only with the unnamed background task. New code should used named background tasks instead. See **Background Tasks** on page IV-319 for details.

The background task runs while Igor is not busy with other things. Normally, there won’t be a background task. The most common use for the background task is to monitor or drive a continuous data acquisition process.

Parameters

numericExpression is a single precision numeric expression that Igor executes when it isn’t doing anything else.

Details

numericExpression is expected to return one of three numeric values:

0: Background task executed normally.

1: Background task wants to stop.

2: Background task encountered error and wants to stop.

Usually the expression will be a call to a user-defined numeric function or external function to drive or monitor data acquisition. The expression should be designed to execute very quickly and it should not present a dialog to the user nor should it create or destroy windows. Generally, it should do nothing more than store data into waves or variables. You can use Igor’s dependency mechanism to perform more extensive tasks.

SetBackground designates the background task but you must use CtrlBackground to start it. You can also use KillBackground to stop it. You can not call SetBackground from the background function itself.

See Also

The **BackgroundInfo**, **CtrlBackground**, **CtrlNamedBackground**, **KillBackground**, and **SetProcessSleep** operations, and **Background Tasks** on page IV-319.

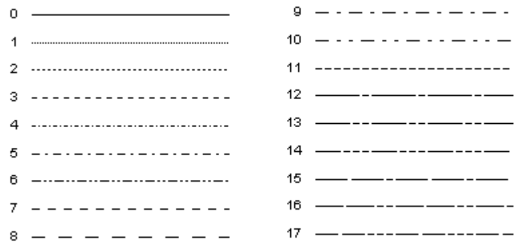
SetDashPattern

SetDashPattern *dashNumber*, {*d1*, *s1* [, *d2*, *s2*]…}

The SetDashPattern operation defines a dashed-line pattern for a user-defined dashed line. These dashed lines are used by the drawing tools and the Modify Waves Appearance dialog, and are elsewhere referred to as “line styles”.

Parameters

dashNumber specifies which dash pattern is to be set. It must be between 1 and 17. Dash pattern 0 is reserved for a solid line.



{*d1,s1* [,*d2,s2*]…} defines the dash pattern. The dash pattern consists of 1 to 8 “dash,skip” pairs. Each pair consists of the number of drawn points followed by the number of skipped points.

d1 specifies the number of drawn points and *s1* specifies the number of skipped points in the first “dash,skip” pair. *d2* and *s2* specify the number of drawn and skipped points in the second pair and so on. Each draw or skip value must be between 1 and 127.

Details

SetDashPattern updates *all* graphs, panels and layouts so that any dashed lines will be updated with the new pattern. If you repeatedly call SetDashPattern from within a macro, you should precede the commands with the PauseUpdate operation to prevent multiple updates (which would be slow).

Dashed lines may also be redefined by the Dashed Lines dialog which you can choose from the Misc menu.

The dashed line patterns are saved as part of the experiment. When a new experiment is opened, the preferred dash patterns are restored.

Some programs and printer drivers do not properly render dashed lines with many “dash,skip” pairs.

Examples

```
Make test; Display test
SetDashPattern 17, {20,3,15,8}       // sets last dashed line pattern
ModifyGraph lstyle(test)=17          // apply pattern to trace
```

See Also

PauseUpdate and **ResumeUpdate** operations, and **Dashed Lines** on page III-496.

SetDataFolder

SetDataFolder *dataFolderSpec*

The SetDataFolder operation sets the current data folder to the specified data folder.

Parameters

dataFolderSpec can be a simple name (MyDataFolder), a path (root:MyDataFolder) or a string expression containing a name or path. It can also be a data folder reference created by the **DFREF** keyword or returned by **GetDataFolderDFR**.

If *dataFolderSpec* is a path it can be a partial path relative to the current data folder (:MyDataFolder) or an absolute path starting from root (root:MyDataFolder).

Examples

```
SetDataFolder foo             // Sets CDF to foo in the current data folder
SetDataFolder :bar:foo        // Sets CDF to foo in bar current data folder
SetDataFolder root:foo        // Sets CDF to foo in the root data folder
DFREF savedDF= GetDataFolderDFR()    // Remember current data folder
NewDataFolder/O/S root:MyDataFolder  // Set CDF to a new data folder
```

```
Variable/G newVariable=1            // Do work in the new data folder
SetDataFolder savedDF               // Restore current data folder
```

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-78.

SetDimLabel

`SetDimLabel` ***`dimNumber`*****`,`** ***`dimIndex`*****`,`** ***`label`*****`,`** ***`wavelist`***

The SetDimLabel operation sets the dimension label or dimension element label to the specified label.

Parameters

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers and 3 for chunks.

If *dimIndex* is -1, it sets the label for the entire dimension. For dimIndex ≥ 0, it sets the dimension label for that element of the dimension.

label is a name (e.g., time), not a string (e.g., "time").

label is limited to 255 bytes. If you use more than 31 bytes, your wave will require Igor Pro 8.00 or later.

Details

Dimension labels can contain up to 255 bytes and may contain spaces and other normally-illegal characters allowed in liberal names if you surround the name in single quotes or if you use the $ operator to convert a string expression to a name.

Dimension labels have the same characteristics as object names. See **Object Names** on page III-501 for a discussion of object names in general.

Prior to Igor Pro 9.00, Igor allowed you to create dimension labels containing illegal characters (double-quotes, single-quotes, colon, semicolon, and control characters). This is now flagged as an error. Existing experiments containing dimension labels with illegal names can still be loaded without error.

See Also

GetDimLabel, **FindDimLabel**, **CopyDimLabels**

Dimension Labels on page II-93 and **Example: Wave Assignment and Indexing Using Labels** on page II-82 for further usage details and examples.

SetDrawEnv

`SetDrawEnv` `[`**`/W=`*****`winName`***`]` ***`keyword`*** `[`**`=`*****`value`***`][`**`,`** ***`keyword`*** `[`**`=`*****`value`***`]]`…

The SetDrawEnv operation sets properties of the drawing environment.

If one or more draw objects are selected in the top window then the SetDrawEnv command will apply only to those objects.

If no objects are selected *and* if the keyword save *is not* used *then* the command applies only to the next object drawn.

If no objects are selected *and* if the keyword “save” *is* used *then* the command sets the environment for all following objects.

Each draw layer has its own draw environment settings.

Parameters

SetDrawEnv can accept multiple *keyword*=*value* parameters on one line.

arrow=*arr*	Specifies the arrow head position on lines. *arr*=0: No arrowhead (default). *arr*=1: Arrowhead at end. *arr*=2: Arrowhead at start. *arr*=3: Arrowhead at start and end.
arrowfat=*afat*	Sets ratio of arrowhead width to length (default is 0.5).
arrowlen=*alen*	Sets length of arrowhead in points (default is 10).

arrowSharp=*s* Specifies the continuously variable barb sharpness between -1.0 and 1. 0.

s=1: No barb; lines only.

s=0: Blunt (default).

s=-1: Diamond.

arrowframe=*f* Specifies the stroke outline thickness of the arrow in points (default is *f*=0 for solid fill).

astyle=s Specifies which side of the line has barbs relative to a right-facing arrow.

s=0: None.

s=1: Top.

s=2: Bottom.

s=3: Both (default).

clipRect=(*left*, *top*, *right*, *bottom*)

Clips drawing to the specifed rectangle. The clipRect keyword was added in Igor Pro 8.00.

left, *top*, *right*, and *bottom* are specified in the current coordinate system of the drawing environment.

If a save is in effect, you can use values of all 0 to cancel the clipping or, better, you can a use push, clipRect, save, drawing, pop sequence. When a save is in effect, the clipping is set once and remains unchanged for the remaining objects even if the coordinate system is changed.

dash=*dsh* *dsh* is a dash pattern number between 0 and 17 (see **SetDashPattern** for patterns). 0 (solid line) is the default.

fillbgc=(*r*,*g*,*b*[,*a*]) Specifies fill background color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is the window's background color.

fillfgc=(*r*,*g*,*b*[,*a*]) Specifies fill foreground color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque white.

fillpat=*fpatt* Specifies fill pattern density.

fpatt=-1: Erase to background color.

fpatt=0: No fill.

fpatt=1: 100% (solid pattern, default).

fpatt=2: 75% gray.

fpatt=3: 50% gray.

fpatt=4: 25% gray.

fillRule=*fr* Determines how polygons and Bezier curves with intersecting edges are filled:

fr=0: Winding rule (default)

fr=1: Even-odd rule

The fillRule keyword applies only to polygons created with **DrawPoly** and **DrawBezier**, not to those created manually.

See **Polygon and Bezier Curve Fill Rules** on page III-72 for a discussion of the rules used for filling polygons and Bezier curves with intersecting edges.

The fillRule keyword was added in Igor Pro 9.00.

fname="*fontName*" Sets font name, default is the default font or the graph font.

fsize=*size* Sets text size, default is 12 points.

fstyle=*fs*

fs is a bitwise parameter with each bit controlling one aspect of the font style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

gedit= *flag*

Supplies optional edit flag for a group of objects. Use with gstart.

flag=0:	Select entire group, moveable (default).
flag=1:	Individual components editable as if not grouped. Allows objects to be grouped by name but still be editable.

gname= *name*

Supplies optional *name* for an object group. Use with gstart.

gstart

Marks the start of a group of objects.

gstop

Marks the end of a group of objects.

gradient

See **Gradient Fills** on page III-498 for details.

gradientExtra

See **Gradient Fills** on page III-498 for details.

linebgc=(*r*,*g*,*b*[,*a*])

Specifies line background color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is the window's background color.

linecap=*cap*

Sets the line end cap style:

cap=0:	Flat caps (default)
cap=1:	Round caps
cap=2:	Square caps

See **Line Join and End Cap Styles** on page III-496 for further information.

linecap was added in Igor Pro 8.00.

linefgc=(*r*,*g*,*b*[,*a*])

Specifies line foreground color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

linejoin=*j*

Sets the line join style:

j=0:	Miter joins
j=1:	Round joins
j=2:	Bevel joins (default)

For a miter join, you can also set the miter limit using the lineMiterLimit keyword.

See **Line Join and End Cap Styles** on page III-496 for further information.

linejoin was added in Igor Pro 8.00.

lineMiterLimit=*ml*

Applies only to a miter line join style. See the linejoin keyword.

ml >= 1:	Sets miter limit to *ml*
ml = INF:	Sets miter limit to unlimited
ml = 0:	Leaves miter limit unchanged
ml = -1:	Sets miter limit to default (10)

See **Line Join and End Cap Styles** on page III-496 for further information.

lineMiterLimit was added in Igor Pro 8.00.

linethick=*thick*

thick is a line thickness ≥ 0, default is 1 point.

origin= *x0,y0* Moves coordinate system origin to *x0,y0*. Unlike translate, rotate, and scale, this survives a change in coordinate system and is most useful that way. See **Coordinate Transformation**.

pop Pops a draw environment from the stack. Pops should always match pushes.

push Pushes the current draw environment onto a stack (limited to 10).

rotate= *deg* Rotates coordinate system by *deg* degrees. Only makes sense if X and Y coordinate systems are the same. See **Coordinate Transformation**.

rounding=*rnd* Radius for rounded rectangles in points, default is 10.

rsabout Redefines coordinate system rotation or scaling to occur at the translation point instead of the current origin. To use, combine rotate or scale with translate and rsabout parameters.

save Stores the current drawing environment as the default environment.

scale= *sx,sy* Scales coordinate system by *sx* and *sy*. Affects only coordinates — not line thickness or arrow head sizes. See **Coordinate Transformation**.

subpaths=*sp* Controls the way polygon and Bezier curve segments separated by NaN values are drawn.:

sp=0: Each segment is drawn as a separate polygon or Bezier curve, and any arrows are added to each segment as if they are separate polygons or Bezier curves. This is the default if you omit the subpaths keyword.

sp=1: The segments are treated as subpaths within a single polygon or Bezier curve, making it possible to define a shape with holes. Any arrows are added only to the first or last points in the entire shape.

The subpaths keyword applies only to polygons created with **DrawPoly** and **DrawBezier**, not to those created manually.

Also see the fillRule keyword.

The subpaths keyword was added in Igor Pro 9.00.

textrgb=(*r*,*g*,*b*[,*a*]) Specifies text color. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

textrot=*rot* Text rotation in degrees.

rot is a value from -360 to 360.

0 is normal (default) horizontal left-to-right text,

90 is vertical bottom-to-top text, etc.

textxjust=*xj* Sets horizontal text alignment.

xj=0: Left aligned text (default).

xj=1: Center aligned text.

xj=2: Right aligned text.

textyjust=*yj* Sets vertical text alignment.

yj=0: Bottom aligned text (default).

yj=1: Middle aligned text.

yj=2: Top aligned text.

translate= *dx,dy* Shifts coordinate system by *dx* and *dy*. Units are in the current coordinate system. See **Coordinate Transformation**.

xcoord=abs X coordinates are absolute window coordinates (default for all windows except graphs where the default is xcoord=prel). The unit of measurement is **Control Panel Units** if the window is a panel, otherwise they are points. The left edge of the window (or of the printable area in a layout) is at x=0. See **Drawing Coordinate Systems** on page III-66 for details.

xcoord=axrel X coordinates are relative axis rectangle coordinates (graphs only). The axrel coordinate system was added in Igor Pro 9.00.

The axis rectangle is the plot rectangle expanded to include any axis standoff. x=0 is at the left edge of the rectangle; x=1 is at the right edge of the rectangle. This coordinate system ideal for objects that should maintain their size and location relative to the axes when axis standoff is used. See **Axis Relative (Graphs Only)** on page III-67 for details.

You can retrieve the axis rectangle's coordinates using GetWindow axSize.

xcoord=rel X coordinates are relative window coordinates. x=0 is at the left edge of the window; x=1 is at the right edge. See **Drawing Coordinate Systems** on page III-66 for details.

xcoord=prel X coordinates are relative plot rectangle coordinates (graphs only). x=0 is at the left edge of the rectangle; x=1 is at the right edge of the rectangle. This coordinate system ideal for objects that should maintain their size and location relative to the axes, and is the default for graphs. See **Plot Relative (Graphs Only)** on page III-67 for details.

You can retrieve the axis rectangle's coordinates using GetWindow pSize.

xcoord=*axisName* X coordinates are in terms of the named axis (graphs only).

ycoord=abs Y coordinates are absolute window coordinates (default for all windows except graphs where the default is ycoord=prel). The unit of measurement is **Control Panel Units** if the window is a panel, otherwise they are points. The top edge of the window (or the of the printable area in a layout) is at y=0. See **Drawing Coordinate Systems** on page III-66 for details.

xcoord=axrel Y coordinates are relative axis rectangle coordinates (graphs only). The axrel coordinate system was added in Igor Pro 9.00.

The axis rectangle is the plot rectangle expanded to include any axis standoff. y=0 is at the top edge of the rectangle; y=1 is at the bottom edge of the rectangle. This coordinate system ideal for objects that should maintain their size and location relative to the axes when axis standoff is used. See **Axis Relative (Graphs Only)** on page III-67 for details.

You can retrieve the axis rectangle's coordinates using GetWindow axSize.

ycoord=rel Y coordinates are relative window coordinates. y=0 is at the top edge of the window; y=1 is at the bottom edge. See **Drawing Coordinate Systems** on page III-66 for details.

ycoord=prel Y coordinates are relative plot rectangle coordinates (graphs only). y=0 is at the top edge of the rectangle; y=1 is at the bottom edge of the rectangle. This coordinate system ideal for objects that should maintain their size and location relative to the axes, and is the default for graphs. See **Plot Relative (Graphs Only)** on page III-67 for details.

You can retrieve the axis rectangle's coordinates using GetWindow pSize.

ycoord=*axisName* Y coordinates are in terms of the named axis (graphs only).

Flags

/W=*winName*	Sets the named window or subwindow for drawing. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Coordinate Transformation

The execution order for the translate, rotate, scale, and origin parameters is important. Translation followed by rotation is different than rotation followed by translation. When using multiple keywords in one SetDrawEnv operation, the order in which they are applied is origin, translate, rotate followed by scale regardless of the command order (with the exception of the rsabout parameter). Before using origin with the save keyword, you should use push to save the current draw environment and then use pop after drawing objects using the new origin.

Examples

Following is a simple example of arrow markers:

```
NewPanel
SetDrawEnv arrow= 1,arrowlen= 30,save
SetDrawEnv arrowsharp= 0.3
DrawLine 61,67,177,31
SetDrawEnv arrowsharp= 1
DrawLine 65,95,181,59
SetDrawEnv astyle= 1
DrawLine 69,123,185,87
SetDrawEnv arrowframe= 1
DrawLine 73,151,189,115
```

You can position objects in one coordinate system and then draw them in another with the origin keyword. In the following coordinate transformation example, we position arrows in axis units but size them in absolute units.

```
Make/O jack=sin(x/8)
Display jack
SetDrawEnv xcoord=bottom,ycoord=left,save
SetDrawEnv push
SetDrawEnv origin=50,0
SetDrawEnv xcoord=abs,ycoord=abs,arrow=1,arrowlen=20,arrowsharp=0.2,save
DrawLine 0,0,50,0          // arrow 50 points long pointing to the right
DrawLine 0,0,0,50          // arrow 50 points long pointing down
// now let's move over, rotate a bit and draw the same arrows:
SetDrawEnv translate=100,0
SetDrawEnv rotate=30,save
DrawLine 0,0,50,0
DrawLine 0,0,0,50
SetDrawEnv pop
```

Now try zooming in on the graph. You will see that the first pair of arrows always starts at 50 on the bottom axis and 0 on the left axis whereas the second pair is 100 points to the right of the first.

See Also

Chapter III-3, **Drawing**, and **DrawAction**.

SetDrawLayer

`SetDrawLayer [/K/W=`*`winName`*`]` ***`layerName`***

The SetDrawLayer operation makes all future drawing operations use the named layer.

Parameters

Valid *layerName*s for graphs:

ProgBack	UserBack	ProgAxes	UserAxes	ProgFront	UserFront	Overlay

Valid *layerName*s for page layouts:

ProgBack	UserBack	ProgFront	UserFront	Overlay

Valid *layerName*s for control panels:

ProgBack	UserBack	ProgFront	UserFront	Overlay

There are really only three layers for control panels. ProgFront is treated as an alias for ProgBack and UserFront is treated as an alias for UserBack.

Flags

/K   Kills (erases) the given layer.

/W=*winName*   Sets the named window or subwindow for drawing. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The Overlay layer is drawn above all else. It is not included when printing or exporting graphics and is provided for programmers who wish to add user-interface drawing elements without disturbing graphics drawing elements. Overlay was added in Igor Pro 7.00.

The back-to-front order of the layers is shown by the layer pop-up menu obtained by clicking the Layer icon in the drawing palette: . A checkmark indicates the current layer. Non-drawing layers are indicated with gray text.

Output Variables

SetDrawLayer sets S_Name to the name of the previously-selected drawing layer. You can use this to restore the active drawing layer after programmatic drawing.

See Also

Drawing Layers on page III-68 and the **DrawAction** operation.

SetEnvironmentVariable

`SetEnvironmentVariable(`*`varName, varValue`*`)`

The SetEnvironmentVariable function creates an environment variable in Igor's process and sets its value to *varValue*. If a variable named *varName* already exists, its value is set to *varValue*.

The function returns 0 if it succeeds or a nonzero value if it fails.

The SetEnvironmentVariable function was added in Igor Pro 7.00.

Parameters

varName	The name of an environment variable which does not need to actually exist. It must not be an empty string and may not contain an equals sign (=).
varValue	The new contents for the variable. On Windows, if *varValue* is an empty string, the variable is removed. On other platforms, the variable is always set to *varValue*.

Details

The environment of Igor's process is composed of a set of key=value pairs that are known as environment variables. Any child process created by calling ExecuteScriptText inherits the environment variables of Igor's process.

SetEnvironmentVariable changes the environment variables present in Igor's process and any future process created by ExecuteScriptText but does not affect any other processes already created.

On Windows, environment variable names are case-insensitive. On other platforms, they are case-sensitive.

Examples

```
Variable result
result = SetEnvironmentVariable("SOME_VARIABLE", "15")
result = SetEnvironmentVariable("SOME_OTHER_VARIABLE", "string value")
```

See Also

GetEnvironmentVariable, **UnsetEnvironmentVariable**

SetFileFolderInfo

SetFileFolderInfo [***flags***][***fileOrFolderNameStr***]

The SetFileFolderInfo operation changes the properties of a file or folder.

Parameters

fileOrFolderNameStr specifies the file or folder to be changed.

If you use a full or partial path for *fileOrFolderNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Folder paths should not end with single Path Separators. See the **MoveFolder Details** section.

If Igor can not determine the location of the file or folder from *fileOrFolderNameStr* and /P=*pathName*, it displays a dialog allowing you to specify the file to be deleted. Use /D to select a folder in this event, otherwise Igor prompts your for a file.

Flags

At least one of the seven following flags is required, or nothing is actually accomplished:

/CDAT=*cdate*	Specifies the number of seconds since midnight January 1, 1904 when the file or folder was first created. *cDate* is interpreted as local time or UTC depending on the /UTC flag.
/INV[=*inv*]	Sets the visibility of a file. *inv*=0: File is visible. *inv*=1: Default; file is invisible (*Macintosh*) or Hidden (*Windows*).
/MDAT=*mDate*	Specifies the number of seconds since midnight January 1, 1904 when the file or folder was modified most recently. *mDate* is interpreted as local time or UTC depending on the /UTC flag.
/RO[=*ro*]	Sets the read/write state of a file or folder. *ro*=0: File or folder is writable. *ro*=1: File or folder is locked (default).

On Macintosh, locking the file or folder is equivalent to setting the locked property manually using the Get Info window in the Finder.

On Windows, locking the file or folder is equivalent to setting the read-only property manually using the Properties window in Windows Explorer.

If *fileOrFolderNameStr* refers to a file (not a folder), SetFileFolderInfo updates the file properties to reflect values given with the following keywords:

/CRE8=*creatorStr* Sets the four-character creator code string, such as 'IGR0' (Igor Pro creator code).

Ignored on Windows, where files have no “creator code”; instead file extensions are “registered” or “owned” by one, and only one, application. You cannot change that ownership from Igor Pro.

/FTYP=*fTypeStr* Sets the four-character file type code, such as 'TEXT' or 'IGsU' (packed experiment).

Ignored on Windows. Use **MoveFile** to change the file extension.

/STA[=*st*] Specifies whether the file is a stationery file or not.

st=1:	Stationery file (default).
st=0:	Normal file.

Ignored on Windows. Use **MoveFile** to change the file extension.

Optional Flags

/D Uses the Select Folder dialog rather than Open File dialog when *pathName* and *fileOrFolderNameStr* do not specify an existing file or folder.

/P=*pathName* Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/R[=*r*] Recursively applies change(s) to all files or folders in the folder specified by /P=*pathName* or *fileOrFolderNameStr*, and the folder itself:

r=0:	No recursion. Same as no /R.
r=1:	Recursively apply changes to files.
r=2:	Recursively apply changes to folders, including the folder specified by *pathName* or *fileOrFolderNameStr*.
r=3:	Recursively apply changes to both files and folders (default).

/R requires /D and a folder specification.

/UTC[=*u*] If you include /UTC or /UTC=1, SetFileFolderInfo interprets the creation and modification dates as UTC (coordinated universal time). If you omit /UTC or specify /UTC=0, SetFileFolderInfo interprets the creation and modification dates as local time.

The default, used if you omit /UTC, is local time.

The /UTC flag was added in Igor Pro 9.00.

/Z[=*z*] Prevents procedure execution from aborting if SetFileFolderInfo tries to set information about a file or folder that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

/Z=0:	Same as no /Z at all.
/Z=1:	Used for setting information for a file or folder only if it exists. /Z alone has the same effect as /Z=1.
/Z=2:	Used for setting information for a file or folder if it exists and displaying a dialog if it does not exist.

Variables

SetFileFolderInfo returns information about the file or folder in the following variables:

`V_flag`	0:	File or folder was found.
	-1:	User cancelled the Open File dialog.
	>0:	An error occurred, such as the specified file or folder does not exist.
`S_path`		File system path of the selected file or folder.

Examples

Change the file creator code; no complaint if it doesn't exist:

```
SetFileFolderInfo/Z /CRE8="CWIE", "Macintosh HD:folder:afile.txt"
```

Set the file modification date:

```
Variable mDate= Date2Secs(2000,12,25) + hrs*3600+mins*60+secs
SetFileFolderInfo/P=myPath/MDAT=(mDate), "afile.txt"
```

Remove read-only property from a folder and everything within it:

```
SetFileFolderInfo/P=myPath/D/R/RO=0
```

See Also

The **GetFileFolderInfo**, **MoveFile**, and **FStatus** operations. The **IndexedFile**, **date2secs**, and **ParseFilePath** functions.

SetFormula

`SetFormula` ***`waveName`*****`,`** ***`expressionStr`***

`SetFormula` ***`variableName`*****`,`** ***`expressionStr`***

The SetFormula operation binds the named wave, numeric or string variable to the expression or, if the expression is `""`, unbinds it from any previous expression. In user functions, SetFormula must be used to create dependencies.

Parameters

expressionStr is a string containing a numeric or string expression, depending on the type of the bound object.

Pass an empty string (`""`) for *expressionStr* to clear any previous dependency expression associated with the wave or variable.

Details

The dependent object (the wave or variable) will depend on the objects referenced in the string expression. The expression will be reevaluated any time an object referred to in the expression is modified.

Besides being set from a string expression this differs from just typing:

```
name := expression
```

in that syntax errors in *expressionStr* are not reported and are not fatal. You end up with a dependency assignment that is marked as needing to be recompiled. The recompilation will be attempted every time an object is created or when the procedure window is recompiled.

Use the Object Status dialog in the Misc menu to check up on dependent objects.

Examples

This command makes the variable `v_sally` dependent on the user-defined function `anotherFunction`, waves `wave_fred` and `wave_sue`, and the system variable `K2`:

```
SetFormula v_sally, "anotherFunction(wave_fred[1]) + wave_sue[0] + K2"
```

This is equivalent to:

```
v_sally := anotherFunction(wave_fred[1]) + wave_sue[0] + K2
```

except that no error will be generated for the SetFormula if, for instance, wave_fred does not exist.

A string variable dependency can be created by a command such as:

```
SetFormula myStringVar, "note(wave_joe)"
```

observe that *expressionStr* is a string *containing* a string expression, and that:

```
SetFormula myStringVar,note(wave_joe)
```

is not the same thing. In this case the note of wave_joe would contain the expression that myStringVar would depend on! Also, wave_joe would have to exist for Igor to understand the statement.

See Also

Chapter IV-9, **Dependencies**, and the **GetFormula** function.

SetIdlePeriod

`SetIdlePeriod` ***`period`***

The SetIdlePeriod operation changes and reports the period of Igor's main idle loop. The units of *period* are milliseconds. Setting *period* to zero does not change the period.

SetIdlePeriod was added in Igor Pro 8.00.

The default idle period is 20 milliseconds. Setting the period higher may make a slight improvement in the performance of some computation-heavy tasks.

Setting the period lower can make some parts of Igor more responsive. In particular, background tasks can be made to run more often, as the minimum period between runs of a background task is determined by Igor's idle period. Reducing the period too far increases the likelihood that your background task will run too long. Setting the period to very low values can make Igor very sluggish.

Output Variables

SetIdlePeriod creates the output variable V_value and sets it to the idle period before the call was made. Yu can use this value to restore the idle period after temporarily changing it. If period is zero, the idle period is not changed, but the current value is returned in V_value.

See Also

Background Tasks on page IV-319

SetIgorHook

`SetIgorHook [/K/L] [`*`hookType`*` = [`*`procName`*`]]`

The SetIgorHook operation tells Igor to call a user-defined "hook" function at the following times:

- After procedures have been successfully compiled (**AfterCompiledHook**)
- After a file is opened (**AfterFileOpenHook**)
- After the MDI frame window is resized on Windows (**AfterMDIFrameSizedHook**)
- After a window is created (**AfterWindowCreatedHook**)
- Before the debugger is opened (**BeforeDebuggerOpensHook**)
- Before an experiment is saved (**BeforeExperimentSaveHook**)
- Before a file is opened (**BeforeFileOpenHook**)
- Before a new experiment is opened (**IgorBeforeNewHook**)
- Before Igor quits (**IgorBeforeQuitHook**)
- When a menu item is selected (**IgorMenuHook**)
- During Igor's quit processing (**IgorQuitHook**)
- When Igor starts or a new experiment is created (**IgorStartOrNewHook**)

The term “hook” is used as in the phrase “to hook into”, meaning to intercept or to attach.

Hook functions are typically used by a sophisticated procedure package to make sure that the package's private data is consistent.

In addition to using SetIgorHook, you can designate hook functions using fixed function names (see **User-Defined Hook Functions** on page IV-280). The advantage of using SetIgorHook over fixed hook names is that you don't have to worry about name conflicts.

You can designate hook functions for specific windows using window hooks (see **SetWindow** on page V-865).

Flags

/K Removes *procName* from the list of functions called for the *hookType* events.

If *procName* is not specified all *hookType* functions are removed.

If *hookType* is not specified all functions are removed for all *hookType* events, returning Igor to the pre-SetIgorHook state.

/L Executes *procName* last. Without /L, a newly added hook function runs before previously registered hook functions.

A function that has been previously registered with SetIgorHook can be moved from being called first to being called last by calling SetIgorHook again with /L.

To move a function from being called last to being called first requires removing the hook function with /K and then calling SetIgorHook without /L.

Parameters

hookType Specifies one of the fixed-name hook function names:

AfterCompiledHook

AfterFileOpenHook

AfterMDIFrameSizedHook

AfterWindowCreatedHook

BeforeDebuggerOpensHook

BeforeExperimentSaveHook

BeforeFileOpenHook

IgorBeforeNewHook

IgorBeforeQuitHook

IgorMenuHook

IgorQuitHook

IgorStartOrNewHook

See the note below about these *hookType* names.

hookType is required except with /K.

procName Names the user-defined hook function that is called for the *hookType* event.

Details

The parameters and return type of the user-defined function *procName* varies depending on the *hookType* it is registered for.

For example, a function registered for the AfterFileOpenHook type must have the same parameters and return type as the shown for the **AfterFileOpenHook** on page IV-282.

The *procName* function is called *after* any window-specific hook for these *hookType*s, and the *procName* function is called *before* any other hook functions previously registered by calling SetIgorHook *unless the /L flag is given*, in which case it still runs after window-specific hook functions, but also *after* all other previously registered hook functions.

The *procName* function should return a nonzero value (1 is typical) to prevent later functions from being called. Returning 0 allows successive functions to be called.

SetIgorHook does not work at Igor start or new experiment time, so SetIgorHook IgorStartOrNewHook is disallowed. Define a global or static fixed-name **IgorStartOrNewHook** function (see page IV-292).

The saved Igor experiment file remembers the SetIgorHooks that are in effect when the experiment is saved:

Hook Function Interactions

After all the SetIgorHook functions registered for *hookType* have run (and all have returned 0), any static fixed-name hook functions are called and then the (only) fixed-name user-defined hook function, if any, is called. As an example, when a menu event occurs, Igor handles the event by calling routines in this order:

1. The top window's hook function as set by **SetWindow**
2. Any SetIgorHook-registered hook functions
3. Any static fixed-named IgorMenuHook functions (in any independent module)
4. The one-and-only non-static fixed-named IgorMenuHook function (in only the ProcGlobal independent module)

1. SetWindow event	**2. SetIgorHook *hookType***	**3. User-defined Hook Function(s)**
(called first)	**(called second)**	**(called last)**
enableMenu	IgorMenuHook	IgorMenuHook
menu	IgorMenuHook	IgorMenuHook

Note: Although you can technically use one of the fixed-name functions, as described in **User-Defined Hook Functions** on page IV-280, for *procName*, the result would be that the function will be called twice: once as a registered named hook function and once as the fixed-named hook function. That is, don’t use SetIgorHook this way:

```
SetIgorHook AfterFileOpenHook=AfterFileOpenHook // NO
```

Variables

SetIgorHook returns information in the following variables:

`S_info` Semicolon-separated list of all current hook functions associated with *hookType*, listed in the order in which they are called. S_info includes the full independent module paths (e.g.,"ProcGlobal#MyMenuHook;MyIM#MyModule#MyMenuHook;").

Examples

This hook function invokes the Export Graphics menu item when Command-C (*Macintosh*) or Ctrl+C (*Windows*) is selected for a graph, preventing the usual Copy.

```
SetIgorHook IgorMenuHook=CopyIsExportHook

Function CopyIsExportHook(isSelection,menuName,itemName,itemNo,win,wType)
   Variable isSelection
   String menuName,itemName
   Variable itemNo
   String win
   Variable wType

   Variable handledIt= 0
   if( isSelection && wType==1 ) // menu was selected, window is graph
      if( Cmpstr(menuName,"Edit")==0 && CmpStr(itemName,"Copy")==0 )
         DoIgorMenu "Edit", "Export Graphics"       // dialog instead
         handledIt= 1           // don't call other IgorMenuHook functions.
      endif
   endif
   return handledIt
End
```

To unregister CopyIsExportHook as a hook procedure:

```
SetIgorHook/K IgorMenuHook=CopyIsExportHook // unregister CopyIsExportHook
```

To discover which functions are associated with a *hookType*, use a command such as:

```
SetIgorHook IgorMenuHook   // inquire about names registered for IgorMenuHook
Print S_info               // list of functions
```

To remove (or “unregister”) named hooks:

```
SetIgorHook/K                  // removes all hook functions for all hookTypes
SetIgorHook/K IgorMenuHook     // removes all IgorMenuHook functions
SetIgorHook/K IgorMenuHook=CopyIsExportHook// removes only this hook function
```

See Also

The **SetWindow** operation and **User-Defined Hook Functions** on page IV-280.

Independent Modules on page IV-238.

SetIgorMenuMode

`SetIgorMenuMode` ***`MenuNameStr`***, ***`MenuItemStr`***, ***`Action`***

The SetIgorMenuMode operation allows an Igor programmer to disable or enable Igor's built-in menus and menu items. This is useful for building applications that will be used by end-users who shouldn't have access to all Igor's extensive and confusing functionality.

Parameters

MenuNameStr The name of an Igor menu, like "File", "Graph", or "Load Waves".

MenuItemStr The text of an Igor menu item, like "Copy" (in the Edit menu) or "New Graph" (in the Windows menu). For menu items in submenus, such as the "Load Waves" submenu in the "Data" menu, *MenuItemStr* is the name of the submenu.

Action One of DisableItem, EnableItem, DisableAllItems, or EnableAllItems.

DisableItem and EnableItem disable or enable just the single item named by *MenuNameStr* and *MenuItemStr*. If *MenuItemStr* is `""`, then the menu itself is disabled.

DisableAllItems and EnableAllItems disable and enable all the items in the menu named by *MenuNameStr*.

Details

All menu names and menu item text are in English. This ensures that code developed for a localized version of Igor will run on all versions. Note that no trailing "..." is used in *MenuItemStr*.

The SetIgorMenuModeProc.ipf procedure file includes procedures and commands that disable or enable every menu and item possible. It is in your Igor Pro folder, in WaveMetrics Procedures:Utilities. It is not intended to be used as-is. You should make a copy and edit the copy to include just the parts you need.

The text of some items in the File menu changes depending on the type of the active window. In these cases you must pass generic text as the *MenuItemStr* parameter. Use "Save Window", "Save Window As", "Save Window Copy", "Adopt Window" and "Revert Window" instead of "Save Notebook" or "Save Procedure", etc. Use "Page Setup" instead of "Page Setup For All Graphs", etc. Use "Print" instead of "Print Graph", etc.

The Edit→Insert File menu item was previously named Insert Text. For compatibility reasons, you can specify either "Insert File" or "Insert Text" as MenuItemStr to modify this item.

See Also

DoIgorMenu, **ShowIgorMenus**, **HideIgorMenus**

The SetIgorMenuModeProc.ipf WaveMetrics procedure file contains SetIgorMenuMode commands for every menu and menu item. You can load it using

```
#include <SetIgorMenuModeProc>
```

SetIgorOption

`SetIgorOption` `[`***`mainKeyword`***`,]` ***`keyword`*****`=`** ***`value`***

`SetIgorOption` `[`***`mainKeyword`***`,]` ***`keyword`*****`= ?`**

The SetIgorOption operation makes unusual and temporary changes to Igor Pro's behavior. The behavior changes are of interest to advanced users only and last only until you end the Igor session.

The details of the syntax depend on the keyword and are documented where the alternate behaviors are described.

SetIgorOption is not compilable. To use it in a user-defined function, you need to use **Execute**.

In most cases the current value of a setting can be read using the keyword=? syntax.

For example, the IndependentModuleDev keyword is used to enable editing of procedure files that implement independent modules:

```
SetIgorOption IndependentModuleDev=?; Print V_Flag       // Query
SetIgorOption IndependentModuleDev=1                     // Set
```

Most SetIgorOption keywords are obscure and rarely of use. Here are the some of the more commonly-used SetIgorOption keywords:

IndependentModuleDev	See **SetIgorOption IndependentModuleDev=1** on page IV-239
PoundDefine	See **Conditional Compilation** on page IV-108
GraphicsTechnology	See **Graphics Technology** on page III-506
PanelResolution	See **SetIgorOption PanelResolution** on page III-456
DisableThreadSafe	See **Debugging ThreadSafe Code** on page IV-225

It is rarely necessary, but you can find the more obscure applications using Help→Search Igor Files to search for "SetIgorOption".

SetMarquee

SetMarquee [***/HAX=hAxisName /VAX=vAxisName /W=winName***] ***left, top, right, bottom***

The SetMarquee operation creates a marquee on the target graph or layout window or the specified window or subwindow.

Parameters

The *left*, *top*, *right*, and *bottom* coordinates are in units of points unless you specify /HAX or /VAX in which case they are in axis units. Axis units are allowed for graphs only, not for layouts.

If the coordinates are all 0, the marquee, if it exists, is killed.

Flags

/HAX=*hAxisName*	Specifies that the *left* and *right* parameters are in units of the axis named by *hAxisName*. The /HAX flag was added in Igor Pro 9.00.
/VAX=*vAxisName*	Specifies that the *top* and *bottom* parameters are in units of the axis named by *vAxisName*. The /VAX flag was added in Igor Pro 9.00.
/W=*winName*	Specifies the named window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

Igor stores marquee coordinates internally as integers in units of points. If you specify coordinates in axis units, there will be some roundoff error when Igor converts to integer points. This results in a small discrepancy between the coordinates you set using /HAX or /VAX and the coordinates returned by **GetMarquee**.

See Also

GetMarquee

SetProcessSleep

SetProcessSleep ***sleepTicks***

The SetProcessSleep operation is obsolete and does nothing as of Igor Pro 7.00. It is documented here in case you come across it in old Igor procedure code. Do not use it in new code.

The SetProcessSleep operation determines how much time Igor will give to background tasks or other Macintosh applications executing in the background. This operation does nothing on Windows.

Parameters

sleepTicks is the amount of time given to background tasks in sixtieths of a second. *sleepTicks* values between 0 and 60 are valid.

Details

Igor starts up with sleepTicks = 1. Use 0 to give Igor maximum time, use a larger number to give other applications more time.

Background tasks are used mainly by data acquisition programs.

See Also

Background Tasks on page IV-319 and the **SetBackground** operation.

SetRandomSeed

`SetRandomSeed` ***`seed`***

The SetRandomSeed operation seeds the random number generator used for the noise functions listed under **Noise Functions** on page III-390.

Use SetRandomSeed if you need “random” numbers that are reproducible. If you don’t use SetRandomSeed, the random number generator is initialized using the system clock when Igor starts. This almost guarantees that you will never get the same sequence twice unless you use SetRandomSeed.

Flags

/BETR[=better]	If better is absent or non-zero, a better method is used for seeding the Mersenne Twister random number generator. /BETR is ignored for all other random generators.

Parameters

seed should be a number in the interval (0, 1]. For any given *seed*, enoise or gnoise or any of the other random-number generator functions generates a particular sequence of pseudorandom numbers. Calling SetRandomSeed with the same seed restarts and repeats the sequence.

Details

Igor's noise functions are listed under **Noise Functions** on page III-390. The enoise and gnoise functions allow you to choose a random number generator. The other functions always use the Mersenne Twister generator.

How the seed is used internally depends on the generator. For the Linear Congruential Generator the seed is scaled to a 32-bit signed integer. For the Mersenne Twister the seed is scaled to a 32-bit unsigned integer. Both only use the lower 16-bits of the so scaled value for historic reasons.

The Xoshiro256** generator uses all available bits and scales it to an unsigned 64-bit integer.

All generators use the scaled seed value when initializing their internal state tables.

See Also

The **enoise** and **gnoise** functions. **Noise Functions** on page III-390.

SetScale

`SetScale` `[`**`/I/P`**`]` ***`dim`***, ***`num1`***, ***`num2`*** `[`, ***`unitsStr`***`]`, ***`waveName`*** `[`, ***`waveName`***`]`…

`SetScale` ***`d`***, ***`num1`***, ***`num2`*** `[`, ***`unitsStr`***`]`, ***`waveName`*** `[`, ***`waveName`***`]`…

The SetScale operation sets the dimension scaling or the data full scale for the named waves.

Parameters

The first parameter *dim* must be one of the following:

Character	Signifies
d	Data full scale.
t	Scaling of the chunks dimension (t scaling).
x	Scaling of the rows dimension (x scaling).
y	Scaling of the columns dimension (y scaling).
z	Scaling of the layers dimension (z scaling).

If setting the scaling of any dimension (*x, y, z,* or *t*), *num1* is the starting index value — the scaled index for the first point in the dimension. The meaning of *num2* changes depending on the /I and /P flags. If you use /P, then *num2* is the delta value — the difference in the scaled index from one point to the next. If you use /I, *num2* is the “ending value” — the index value for the last element in the dimension. If you use neither flag, *num2* is the “right value” — the index value that the element *after the last element in the dimension* would have.

These three methods are just three different ways to specify the two scaling values, the starting value and the delta value, that are stored for each dimension of each wave.

If setting the data full scale (*d*), then *num1* is the nominal minimum and *num2* is the nominal maximum data value for the waves. The data full scale values are not used. They serve only to document the minimum and maximum values the waves are expected to attain. No flags are used when setting the data full scale.

The *unitsStr* parameter is a string that identifies the natural units for the x, y, z, t, or data values of the named waves. Igor will use this to automatically label graph axes. This string must be one to 49 bytes such as “m” for meters, “g” for grams or “s” for seconds. If the waves have no natural units you can pass `""` for this parameter.

Setting *unitsStr* to "dat" (case-sensitive) tells Igor that the wave is a date/time wave containing data in Igor date/time format (seconds since midnight on January 1, 1904). Date/time waves must be double-precision.

Flags

At most one flag is allowed, and then only if dimension scaling (not data full scale) is being set:

/I	Inclusive scaling. *num2* is the ending index — the index value for the very last element in the dimension.
/P	Per-point scaling. *num2* is the delta index value — the difference in scaled index value from one element to the next.

Details

SetScale will not allow the delta scaling value to be zero. If you execute a SetScale command with a delta value of zero, it will set the delta value to 1.0.

If you do not use the /P flag, SetScale converts *num1* and *num2* into a starting index value and a delta index value. If you call SetScale on a dimension with fewer than two elements, it does this conversion as if the dimension had two elements.

Prior to Igor Pro 3.0, Igor supported only 1D waves. “SetScale x” was used to set the scaling for the rows dimensions and “SetScale y” was used to set the data full scale. With the addition of multidimensional waves, “SetScale y” is now used to set the scaling of the columns dimension and “SetScale d” is used to set the data full scale. For backward compatibility, “SetScale y” on a 1D wave sets the data full scale.

When setting the dimension scaling of a numeric wave, you can omit the *unitsStr* parameter. Igor will set the wave’s scaling but not change its units. However, when setting the dimension scaling of a text wave, you must supply a *unitsStr* parameter (use `""` if the wave has no units). If you don’t, Igor will think that the text wave is the start of a string expression and will attempt to treat it as the *unitsStr*.

See Also

See Also

CopyScales, **DimDelta**, **DimOffset**, **DimSize**, **WaveUnits**

For an explanation of waves and dimension scaling, see **Changing Dimension and Data Scaling** on page II-68.

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-85.

SetVariable

`SetVariable [/Z] ctrlName [keyword = value [, keyword = value …]]`

The SetVariable operation creates or modifies a SetVariable control in the target window.

A SetVariable control sets the value of a global numeric or string variable or a point in a wave when you type or click in the control. A SetVariable can also hold its own value without the need for a global or wave.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the SetVariable control to be created or changed.

The following keyword=value parameters are supported:

activate

Activates the control and selects the text that sets the value. Use **ControlUpdate** to deactivate the control and deselect the text.

align=*alignment*

Sets the alignment mode of the control. The alignment mode controls the interpretation of the *leftOrRight* parameter to the pos keyword. The align keyword was added in Igor Pro 8.00.

If *alignment*=0 (default), *leftOrRight* specifies the position of the left end of the control and the left end position remains fixed if the control size is changed.

If *alignment*=1, *leftOrRight* specifies the position of the right end of the control and the right end position remains fixed if the control size is changed.

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **Button** and **DefaultGUIControls** for more appearance details.

bodyWidth=*width*

Specifies an explicit size for the body (nontitle) portion of a SetVariable control. By default (bodyWidth=0), the body portion is the amount left over from the specified control width after providing space for the current text of the title portion. If the font, font size or text of the title changes, then the body portion may grow or shrink. If you supply a bodyWidth>0, then the body is fixed at the size you specify regardless of the body text. This makes it easier to keep a set of controls right aligned when experiments are transferred between Macintosh and Windows, or when the default font is changed.

disable=*d*

Sets user editability of the control.

d=0: Normal.

d=1: Hide.

d=2: No user input.

fColor=(*r*,*g*,*b*[,*a*])

Sets the initial color of the title. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

To further change the color of the title text, use escape sequences as described for title=*titleStr*.

focusRing=*fr*

Enables or disables the drawing of a rectangle indicating keyboard focus:

fr=0: Focus rectangle will not be drawn.

fr=1: Focus rectangle will be drawn (default).

On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.

font="*fontName*"

Sets the font used to display the value of the variable, e.g., `font="Helvetica"`.

format=*formatStr*

Sets the numeric format of the displayed value, e.g., format="%g". Not used with string variables. Never use leading text or the "%W" formats, because Igor reads the value back without interpreting the units. For a description of *formatStr*, see the **printf** operation.

frame=*f* — Sets the frame for the value readout.

f=0: Value unframed.

f=1: Value framed (default).

fsize=*s* — Sets the size of the type used to display the variable's value.

fstyle=*fs* — *fs* is a bitwise parameter with each bit controlling one aspect of the font style as follows:

Bit 0: Bold

Bit 1: Italic

Bit 2: Underline

Bit 4: Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*} — Sets the help for the control.

helpStr is limited to 1970 bytes (255 in Igor Pro 8 and before).

You can insert a line break by putting "\r" in a quoted string.

labelBack=(*r*,*g*,*b*[,*a*]) or 0 — Specifies the background fill color for labels. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is 0, which uses the window's background color.

limits={*low*,*high*,*inc*} — Sets the limits of the allowable values (*low* and *high*) for the variable. *inc* sets the amount by which the variable is incremented if you click the control's up/down arrows. This applies to numeric variables, not to string variables. If *inc* is zero then the up/down arrows will not be drawn.

live=*l* — Determines when the readout is updated.

l=0: Update only after variable changes (default).

l=1: Update as variable changes.

noedit=*val* — noedit=1 prevents the user from clicking (or tabbing into) a SetVariable control to directly edit its value. This is useful when you want to make a string read-only or when you want to restrict a numeric setting to those available only via the control's up or down arrow buttons.

noedit=0 reactivates user editing.

noedit=2 is deprecated as of Igor Pro 6.34 but still supported. It allows the use of formatting escape codes described under **Annotation Escape Codes**. Use styledText=1, instead.

noproc — No procedure is to execute when the control's value is changed.

pos={*leftOrRight*,*top*} — Sets the position in **Control Panel Units** of the top/left corner of the control if its alignment mode is 0 or the top/right corner of the control if its alignment mode is 1. See the align keyword above for details.

pos+={*dx*,*dy*} — Offsets the position of the control in **Control Panel Units**.

proc=*procName* — Sets the procedure to execute when the control's value is changed.

rename=*newName* — Gives control a new name.

styledText=val — styledText=1 allows the use of formatting escape codes described under **Annotation Escape Codes** on page III-53. This works for string SetVariable controls only, not for numeric controls.

For example:

```
SetVariable sv0 value=_STR:"\\JC\\K(65535,0,0)Centered Red
Text"
```

styledText=0 treats escape codes as plain text.

The styledText keyword was added in Igor Pro 6.34. For compatibility with earlier versions of Igor, the combination of noedit=1 and styledText=1 is recorded as noedit=2 in recreation macros.

size={*width*,*height*} Sets width of control in **Control Panel Units**. *height* is ignored.

textAlign=*t* Sets the alignment of the text displayed in the body of the control.

t=0:	Left (default)
t=1:	Center
t=2:	Right

The textAlign keyword was added in Igor Pro 9.00.

title=*titleStr* Sets the title of the control to the specified string expression. The title is displayed to the left of the control. If *titleStr* is empty (`""`), the name of the controlled variable is displayed as the title. Use `title=" "` (put a space within the quotation marks) to create a “blank” title.

Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.

userdata(*UDName*)=*UDStr*

Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

value=*varOrWaveName* Sets the numeric or string variable or wave element to be controlled.

If *varOrWaveName* references a wave, the point is specified using standard bracket notation with either a numeric point number or a row label, for example: `value=awave[4]` or `value=awave[%alabel]`.

You may also use a 2D, 3D, or 4D wave and specify a column, layer, and chunk index or dimension label in addition to the row index.

You can have the control store the value internally rather than in a global variable. In place of varName, use _STR:str or _NUM:num. For example:

```
NewPanel; SetVariable sv1,value=_NUM:123
```

valueColor=(*r*,*g*,*b*[,*a*]) Sets the color of the value text. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

valueBackColor=(*r*,*g*,*b*[,*a*]) Sets the background color under the value text. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

valueBackColor=0 Sets the background color under the value text to the default color, the standard document background color used on the current operating system, which is usually white.

win=*winName* Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Flags

/Z	No error reporting.

Details

The target window must be a graph or panel.

SetVariable Action Procedure

The action procedure for a SetVariable control takes a predefined WMSetVariableAction structure as a parameter to the function:

```
Function ActionProcName(SV_Struct) : SetVariableControl
   STRUCT WMSetVariableAction &SV_Struct
   …
   return 0
End
```

The “`: SetVariableControl`” designation tells Igor to include this procedure in the Procedure pop-up menu in the SetVariable Control dialog.

See **WMSetVariableAction** for details on the WMSetVariableAction structure.

Although the return value is not currently used, action procedures should always return zero.

You may see an old format SetVariable action procedure in old code:

```
Function procName(ctrlName,varNum,varStr,varName) : SetVariableControl
   String ctrlName
   Variable varNum     // value of variable as number
   String varStr       // value of variable as string
   String varName      // name of variable
   …
   return 0
End
```

This old format should not be used in new code.

Examples

Executing the commands:

```
Variable/G globalVar=99
SetVariable setvar0 size={120,20}
SetVariable setvar0 font="Helvetica", value=globalVar
```

creates a SetVariable control that displays the value of globalVar.

See Also

The **printf** operation for an explanation of *formatStr*, and **SetVariable** on page III-417.

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

The **GetUserData** function for retrieving named user data.

The **ControlInfo** operation for information about the control.

SetVariableControl

`SetVariableControl`

SetVariableControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined SetVariable control. See **Procedure Subtypes** on page IV-204 for details. See **SetVariable** for details on creating a SetVariable control.

SetWaveLock

`SetWaveLock` ***`lockVal`*****`,`** ***`waveList`***

The SetWaveLock operation locks a wave or waves and protects them from modification. Such protection is not absolute, but it should prevent most common attempts to change or kill a wave.

Parameters

lockVal can be 0, to unlock, or 1, to lock the wave(s).

waveList is a list of waves or it can be `allinCDF` to act on all waves in the current data folder.

See Also

WaveInfo to check if a wave is locked.

SetWaveTextEncoding

`SetWaveTextEncoding [`*`flags`*`]` ***`newTextEncoding`***`,` ***`elements`***`,` **`[`*`wave`*`,`** ***`wave`***`, ...]`**

The SetWaveTextEncoding operation changes the text encoding of the specified waves and/or the text encoding of all waves in the specified data folder.

Wave text encodings are mostly an issue in dealing with pre-Igor Pro 7 experiments containing non-ASCII text. Most users will have no need to worry about or change them. You should not use this operation unless you have a thorough understanding of text encoding issues or are instructed to use it by someone who has a thorough understanding.

See **Wave Text Encodings** on page III-472 for essential background information.

SetWaveTextEncoding can work on a list of specific waves or on all of the waves in a data folder (/DF flag). When working on a data folder, it can work on just the data folder itself or recursively on sub-data folders as well.

If /CONV is present, SetTextWaveEncoding actually converts the text to a different text encoding. You would use this, for example, to convert text stored as Shift JIS (Japanese non-Unicode) into UTF-8 (Unicode).

If /CONV is omitted, SetTextWaveEncoding merely causes Igor to reinterpret text. You would do this to tell Igor what text encoding is used for a wave created by Igor Pro 6 if Igor gets it wrong.

Conversion does not change the characters that make up text - it merely changes the numeric codes used to represent those characters. Reinterpretation does not change the numeric codes but does change the characters by changing the interpretation of the numeric codes.

In some cases it may be necessary to fix text encoding issues in Igor Pro 6.3x before opening an experiment in later versions.

Parameters

newTextEncoding specifies the text encoding to set the wave element to. See **Text Encoding Names and Codes** on page III-490 for a list of codes.

newTextEncoding can be the special value 255 which marks a text wave's content as really containing binary data, not text. See **Text Waves Containing Binary Data** on page III-475 below for details.

elements is a bitwise parameter that specifies one or more elements of a wave, as follows:

Bit	Value	Meaning
0	1	Wave name
1	2	Wave units
2	4	Wave note
3	8	Wave dimension labels
4	16	Text wave content

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

wave, *wave*, ... is a list of the targeted waves. The list is optional and typically should be omitted if you use the /DF flag. However, if you specify a data folder via /DF and you also list specific waves, SetWaveTextEncoding works on all waves in the data folder as well as the specifically listed waves.

Flags

/BINA={*markAsBinary*, *diagnosticsFlags*}

If *markAsBinary* is 1, SetWaveTextEncoding marks the content of any text wave as binary if the data contains control characters (codes less than 32) other than carriage return (13), linefeed (10), and tab (9). The number of waves so marked is returned via V_numMarkedAsBinary.

If *markAsBinary* is 0, SetWaveTextEncoding acts as if /BINA were omitted.

diagnosticsFlags is optional and defaults to 1. If you omit it you can also omit the braces (/BINA=1).

diagnosticsFlags is a bitwise parameter defined as follows:

Bit 0: Emit diagnostic message for each wave marked as binary.

All other bits are reserved for future use.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The /BINA=1 flag works with with the content of text waves only. It skips non-text waves. It is also independent of the *elements* parameter. That is, it marks the text wave content and only the text wave content as binary even if *elements* is something other than 16.

If you pass 255 for *newTextEncoding* when using the /BINA flag, this tells SetWaveTextEncoding to stop processing after marking binary waves. No further conversion or reinterpretation is done.

See **Text Waves Containing Binary Data** on page III-475 for further discussion.

/CONV={*errorMode* [, *defaultTextEncoding*, *diagnosticsFlags*, *conversionFlags*]}

Causes the text data to be converted to the specified text encoding. If /CONV is present, SetTextWaveEncoding actually converts the text. If it is omitted, SetTextWaveEncoding merely causes Igor to reinterpret it.

defaultTextEncoding, *diagnosticsFlags* and *conversionFlags* are optional and are further discussed below. If you omit them you can also omit the braces (/CONV=1).

If /CONV is specified and the original text encoding is binary (255) then SetWaveTextEncoding does nothing, because all conversions involving binary are NOPs.

If /CONV is specified and *newTextEncoding* is binary (255) then no conversion is done, because all conversions involving binary are NOPs, but the wave is marked as binary. Thus this amounts to the same thing as reinterpreting the wave's text data as binary.

If /CONV is specified, *defaultTextEncoding* is omitted or is -1, and the original text encoding is unknown (0) then SetWaveTextEncoding does nothing. That is, it does no reinterpretation or conversion.

errorMode determines how SetWaveTextEncoding behaves if the conversion can not be done because the text can not be mapped to the specified text encoding. This will occur if the text contains characters that can not be represented in the specified text encoding or if Igor's notion of the original text encoding is wrong. In the latter case, call SetWaveTextEncoding without /CONV to correct Igor's interpretation of the text and then call SetWaveTextEncoding with /CONV to do the conversion.

errorMode takes one of these values:

1: Generate error. SetWaveTextEncoding returns an error to Igor.

2: Use a substitute character for any unmappable characters. The substitute character for Unicode is the Unicode replacement character, U+FFFD. For most non-Unicode text encodings it is either control-Z or a question mark.

3: Skip unmappable input characters. Any unmappable characters will be missing in the output.

4: Use escape sequences representing any unmappable characters or invalid source text.

If the source text is valid in the source text encoding but can not be represented in the destination text encoding, unmappable characters are replaced with \uXXXX where XXXX specifies the UTF-16 code point of the unmappable character in hexadecimal.

If the conversion can not be done because the source text is not valid in the source text encoding, invalid bytes are replaced with \xXX where XX specifies the value of the invalid byte in hexadecimal.

defaultTextEncoding is optional. If it is present, not -1, and if the wave text element's original encoding is unknown (0), then the wave text element is treated as if it were the specified *defaultTextEncoding*. This allows you to convert the text of Igor Pro 6 waves that are set to unknown when you know that they are really some other text encoding. For example, if you know a wave's text data text encoding is Shift JIS, you can convert it to UTF-8 in one step, like this:

```
SetWaveTextEncoding /CONV={1,4} 1, 16, textWave0
```

Without the *defaultTextEncoding*, you would have to do two steps - the first to tell Igor what the real text encoding is and the second to do the conversion:

```
// Text encoding is Shift JIS
SetWaveTextEncoding 4, 16, textWave0
```

```
// Convert to UTF-8
SetWaveTextEncoding /CONV=1 1, 16, textWave0
```

Passing -1 for *defaultTextEncoding* acts the same as omitting it.

diagnosticsFlags is an optional bitwise parameter defined as follows:

Bit 0: Emit diagnostic message if text conversion succeeds.

Bit 1: Emit diagnostic message if text conversion fails.

Bit 2: Emit diagnostic message if text conversion is skipped.

Bit 3: Emit summary diagnostic message.

All other bits are reserved for future use and must be 0.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

diagnosticsFlags defaults to 2 (bits 1 set) if the /DF flag is not present and to 10 (bits 1 and 3 set) if the /DF flag is present.

Text conversion may be skipped because the wave element is marked as unknown text encoding, because it is marked as binary, because of the /ONLY or /SKIP flags, or because *newTextEncoding* is the same as the wave element's original text encoding.

conversionFlags is an optional bitwise parameter defined as follows:

Bit 0:	Do a dry run only.

All other bits are reserved for future use and must be 0.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The *conversionFlags* parameter was added in Igor Pro 9.00.

conversionFlags defaults to 0. If *conversionFlags* is 1, SetWaveTextEncoding goes through the entire conversion process but does not actually do any conversions. This feature was added to provide a way to determine if conversions are needed to create a pure UTF-8 experiment. For most purposes you can omit the *conversionFlags* parameter.

/DF={*dfr*, *recurse*, *excludedDFR*}

dfr is a reference to a data folder. SetWaveTextEncoding operate on all waves in the specified data folder. If *dfr* is null ($"") SetWaveTextEncoding acts as if /DF was omitted.

If *recurse* is 1, SetWaveTextEncoding works recursively on all sub-data folders. Otherwise it affects only the data folder referenced by *dfr*.

excludedDFR is an optional reference to a data folder to be skipped by SetWaveTextEncoding. For example, this command sets the text encoding of text wave data for all waves in all data folders except for root:Packages and its sub-data folders:

```
SetWaveTextEncoding /DF={root:,1,root:Packages} 1, 16
```

If *excludedDFR* is null ($"") SetWaveTextEncoding acts as if *excludedDFR* was omitted and no data folders are excluded.

/KIND=*kind* SetWaveTextEncoding changes only waves of the specified kind. *kind* is:

1:	Home waves only
2:	Shared waves only
3:	Home and shared waves (default)

A "home wave" is a wave that is stored in the experiment file for packed experiment or in the disk folder corresponding to the wave's data folder for an unpacked experiment.

A "shared wave" is a wave that is stored in a standalone file for packed experiment or in a standalone file outside the disk folder corresponding to the wave's data folder for an unpacked experiment.

See **References to Files and Folders** on page II-24 for an explanation of shared versus home waves.

/ONLY=*targetedTextEncoding*

SetWaveTextEncoding changes only wave elements that are currently set to *targetedTextEncoding*. For example, this command converts the content of all waves that are currently set to unknown text encoding (0) to UTF-8 (1), treating the waves originally marked as unknown as Shift JIS (4) waves:

```
SetWaveTextEncoding /DF={root:,1} /ONLY=0 /CONV={1,4} 1, 16
```

Passing -1 for *targetedTextEncoding* acts as if you omitted /ONLY altogether.

/SKIP=*skipTextEncoding*

SetWaveTextEncoding skips all wave elements that are currently set to *skipTextEncoding*. For example, this command converts the content of all waves to UTF-8 (1) except those that are set to Japanese (4):

```
SetWaveTextEncoding /DF={root:,1} /SKIP=4 /CONV=1 1, 16
```

As explained under /CONV, binary (255) wave elements are always skipped and unknown (0) wave elements are skipped if the /CONV defaultTextEncoding parameter is omitted.

Passing -1 for *skipTextEncoding* acts as if you omitted /SKIP altogether.

/TYPE=*type* SetWaveTextEncoding changes only waves of the specified type.

type is:

1:	Text waves only
2:	Non-text waves only
3:	All waves (default)

/Z[=z] Prevents procedure execution from aborting if SetWaveTextEncoding generates an error. Use /Z or the equivalent, /Z=1, if you want to handle errors in your procedures rather than having execution abort.

/Z does not suppress invalid parameter errors. It suppresses only errors in doing text encoding reinterpretation or conversion.

Details

Because SetWaveTextEncoding is intended for use by experts or by users instructed by experts, it does not respect the lock state of waves. That is, it will change waves even if they are locked using SetWaveLock.

For background information on wave text encodings, see **Wave Text Encodings** on page III-472.

A wave's text wave content text encoding can also be set to the special value 255. This marks a text wave as really containing binary data, not text. See **Text Waves Containing Binary Data** on page III-475 for details.

Using SetWaveTextEncoding

One of the main uses for SetWaveTextEncoding is to set the encoding settings to some value other than 0 (unknown) so that Igor's idea of the text encoding used for the items accurately reflects the actual text encoding. We call this "reinterpreting" the item. It does not change the numeric codes representing the text but rather just changes the setting that controls Igor's idea of how the text is encoded. This does change the meaning of the underlying numeric codes. In other words, it changes the characters represented by the text.

You would do reinterpretation if you load an Igor6 experiment and Igor interprets the waves using the wrong text encoding. For example, if you load an experiment that you know uses Shift JIS encoding but Igor interprets it as Windows-1252, you get garbage for Japanese text. Reinterpreting it as Shift JIS fixes this. An example is provided below.

The other use for SetWaveTextEncoding is to actually change the numeric codes representing the text - i.e., to convert the content to a different text encoding. For example, if you have text waves from Igor Pro 6 that are encoded in Japanese (Shift JIS), you may want to convert the text to UTF-8 (a form of Unicode) so that you can combine Japanese and non-Japanese characters or use other features that require Unicode. This also applies to western text containing non-ASCII characters encoded as MacRoman or Windows-1252. Converting changes the underlying numeric codes but does not change the characters represented by the text.

The main use for converting text is to convert Igor Pro 6 waves from whatever encoding they use, which typically will be MacRoman, Windows-1252, or Shift JIS, to UTF-8 (a form of Unicode) which is a more modern representation but is not backward compatible with Igor Pro 6.

If the /CONV flag is omitted SetWaveTextEncoding does reinterpretation. If the /CONV flag is present then SetWaveTextEncoding does text conversion except if the original text encoding is unknown (0) or binary (255) in which case it does nothing.

If a wave has mistakenly been marked as containing binary, use SetTextWaveEncoding without /CONV to set it to the correct text encoding.

If a wave's text encoding is set to unknown (0) but you know that it really contains text in some specific encoding, you can use SetTextWaveEncoding without /CONV to set it to the correct text encoding and then use SetTextWaveEncoding with /CONV to convert it to the desired final text encoding. Alternately you can combine these two steps by using /CONV and providing a value for the optional *defaultTextEncoding* parameter.

Output Variables

The SetWaveTextEncoding operation returns information in the following variables:

`V_numConversionsSucceeded` Set only when the /CONV flag is used. Zero otherwise.

V_numConversionsSucceeded is set to the number of successful text element conversions.

`V_numConversionsFailed` Set only when the /CONV flag is used. Zero otherwise.

V_numConversionsFailed is set to the number of unsuccessful text conversions. Because SetWaveTextEncoding quits when any error occurs, V_numConversionsFailed will be either 0 or 1.

`V_numConversionsSkipped` Set only when the /CONV flag is used. Zero otherwise.

V_numConversionsSkipped is set to the number of skipped text element conversions. Text conversion may be skipped because the wave element is marked as unknown text encoding, because it is marked as binary, because of the /ONLY or /SKIP flags or because newTextEncoding is the same as the wave element's original text encoding.

V_numConversionsSkipped does not count waves skipped because of the /TYPE flag.

`V_numMarkedAsBinary` Set only when the /BINA flag is used. Zero otherwise.

V_numMarkedAsBinary is set to the number of text waves whose text wave content was marked as binary.

V_numMarkedAsBinary was added in Igor Pro 9.00.

`V_numWavesAffected` V_numWavesAffected is set to the number of waves modified in any way, whether through reinterpretation, conversion of one or more elements, or marking as binary.

V_numWavesAffected was added in Igor Pro 9.00.

Examples

```
// Keep in mind that, if /CONV is present, SetWaveTextEncoding does nothing
// for wave text elements currently set to binary (255).
// Also, if /CONV is present, SetWaveTextEncoding does nothing
// for wave text elements currently set to unknown (0) if the /CONV
// optional defaultTextEncoding parameter is omitted.
// In the following examples 1 means UTF-8, 4 means Shift JIS, and 16
// means text wave content.

// Reinterpret specific text waves as Shift JIS if they are currently set to unknown
SetWaveTextEncoding /ONLY=0 4, 16, textWave0, textWave1

// Convert specific waves' content to UTF-8
SetWaveTextEncoding /CONV=1 1, 16, textWave0, textWave1

// Reinterpret all text waves as Shift JIS if they are currently set to unknown
SetWaveTextEncoding /DF={root:,1} /ONLY=0 4, 16

// Convert all waves' content to UTF-8
SetWaveTextEncoding /DF={root:,1} /CONV=1 1, 16

// Convert all text waves' content from Shift JIS to UTF-8
// if it is currently set to unknown
SetWaveTextEncoding /DF={root:,1} /ONLY=0 /TYPE=1 /CONV={1,4} 1, 16

// Same as before but exclude the root:Packages data folder
```

```
SetWaveTextEncoding /DF={root:,1,root:Packages} /ONLY=0 /TYPE=1 /CONV={1,4} 1, 16

// Mark a text wave as really containing binary data
SetWaveTextEncoding 255, 16, textWaveContainingBinaryData

// Convert Chinese, Japanese and Korean text wave data to UTF-8.
// This example illustrates how to do a conversion based on criteria
// that require inspecting each wave.
Function ConvertCJKToUTF8(dfr, recurse)
   DFREF dfr
   Variable recurse

   Variable index = 0
   do
      Wave/Z w = WaveRefIndexedDFR(dfr, index)
      if (!WaveExists(w))
         break
      endif
      if (WaveType(w) == 0)   // Text wave?
         Variable currentEncoding = WaveTextEncoding(w,5)// Wave content current
   encoding
         switch(currentEncoding)
            case 4:           // Japanese (Shift JIS)
            case 5:           // Traditional Chinese (Big5)
            case 6:           // Simplified Chinese (ISO-2022-CN)
            case 7:           // Macintosh Korean (EUC-KR)
            case 8:           // Windows Korean (Windows-949)
               SetWaveTextEncoding /CONV=1 1, 16, w
               break
         endswitch
      endif
      index += 1
   while(1)

   if (recurse)
      Variable numChildDataFolders = CountObjectsDFR(dfr, 4)
      Variable i
      for(i=0; i<numChildDataFolders; i+=1)
         String childDFName = GetIndexedObjNameDFR(dfr, 4, i)
         DFREF childDFR = dfr:$childDFName
         ConvertCJKToUTF8(childDFR, 1)
      endfor
   endif

   SetDataFolder saveDFR
End
```

See Also

Text Encodings on page III-459, **Wave Text Encodings** on page III-472, **Text Encoding Names and Codes** on page III-490, **Text Waves Containing Binary Data** on page III-475

WaveTextEncoding, **ConvertTextEncoding**, **ConvertGlobalStringTextEncoding**

SetWindow

`SetWindow` ***`winName`*** `[,` ***`keyword = value`***`]`…

The SetWindow operation sets the window note and user data for the named window or subwindow. SetWindow can also set hook functions for a base window or exterior subwindow (interior subwindows not supported).

Parameters

winName can be the name of any target window (graph, table, page layout, notebook, control panel, Gizmo, camera, or XOP target window). It can also be the keyword `kwTopWin` to specify the topmost target window.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

activeChildFrame=*f*

Determines if the frame indicating the active subwindow in the named window or subwindow is drawn.

f=-1: Default. Recursively check ancestors of this window. If one is found with a value of either 0 or 1, use that value. If the topmost window has a value of -1, then the frame is drawn.

f=1: If this subwindow or one of its descendants becomes active, then the frame is drawn.

f=0: If this subwindow or one of its descendants becomes active, then the frame is not drawn.

doScroll=*mode*

Controls scrolling mode for top-level graph and panel windows only. The doScroll keyword was added in Igor Pro 9.00.

mode=1: Turns scrolling mode on for the graph or panel.

When scrolling mode is on, the size of the content area of the graph or panel is fixed and the window no longer grows or shrinks when you resize the window. If you shrink the window smaller than the size of the content area, scroll bars appear which allow you to view different parts of the content area.

When scrolling mode is on, you can change the size of the content area of the graph or panel window using doScroll=(width, height).

mode=0: Turns scrolling mode off for the graph or panel.

When scrolling mode is off, scroll bars are not shown and the size of the content area of the graph or panel grows or shrinks when you resize the window.

doScroll=(*width*,*height*)

Controls scrolling mode for top-level graph and panel windows only. The doScroll keyword was added in Igor Pro 9.00.

Turns scrolling mode on if it is off. Sets the size of the content area of the graph or panel. *width* and *height* are in units of points.

graphicsTech=*t*

Sets the graphics technology used to draw the window. This flag may be useful in rare cases to work around graphics limitations. See **Graphics Technology** (see page III-506) for background information.

t=0: Default: The graphics technology specified in Miscellaneous Settings dialog, Miscellaneous category. This is Qt Graphics by default.

t=1: Native: Core Graphics on Macintosh, GDI+ on Windows.

t=2: Old: Core Graphics on Macintosh, GDI on Windows.

hide=*h*

Hides or unhides widows or subwindows.

h=0: Unhides a subwindow or base window.

h=1: Hides a subwindow or base window.

h=2: Unhides without restoring minimized windows (*Windows* only).

When unhiding subwindows, you should combine with needUpdate=1 if conditions require the subwindow to be redrawn since the window was hidden.

hook=*procName*

Sets the window hook function that Igor will call when certain events happen. Use `SetWindow hook=$""` to specify no hook function.

See **Unnamed Window Hook Functions** on page IV-305 for further details.

hook(*hName*)=*procName*

Defines a named window hook *hName* and sets the function that Igor will call when certain events happen. *hName* can be any legal name. Named hooks are called before any unnamed hooks.

Use $"" for *procName* to specify no hook.

See **Named Window Hook Functions** on page IV-295 for further details.

To hook a subwindow, see **Window Hooks and Subwindows** on page IV-294.

hookcursor=*number* Sets the mouse cursor. This keyword is antiquated. See **Setting the Mouse Cursor** on page IV-302 for the preferred technique.

hookevents=*flags* Bitfield of flags to enable certain events for the unnamed hook function:

Bit 0:	Mouse button clicks.
Bit 1:	Mouse moved events.
Bit 2:	Cursor moved events.

To set bit 0 and bit 1 (mouse clicks and mouse moved), use $2^0+2^1 = 1+2 = 3$ for *flags*. Use 7 to also enable cursor moved events. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

This keyword applies to the unnamed hook function only. It does not affect named hook functions which always receive all events.

markerHook= {*hookFuncName*, *start*, *end*}

Specifies a user function and marker number range for custom markers. The marker range can be any positive integers less than 1000 and can overlap built-in marker numbers. See **Custom Marker Hook Functions** on page IV-308 for details.

Use $"" for hookFuncName to specify no hook.

needUpdate= *n* Marks a window as needing an update (*n*=1) or takes no action (*n*=0).

note=*noteStr* Sets the window note to *noteStr*, replacing any existing note.

note+=*noteStr* Appends *noteStr* to current contents of the window note.

sizeLimit= {*minWidth*, *minHeight*, *maxWidth*, *maxHeight*}

Imposes limits on a window's size when resized with the mouse or by calling **MoveWindow**. The units of the limits are the same as those returned by **GetWindow** wsize.

The sizeLimit keyword was added in Igor Pro 7.00.

To allow the window width to grow essentially without bound, pass INF for maxWidth.

To allow the window height to grow essentially without bound, pass INF for maxHeight.

If *minWidth* > *maxWidth* or *minHeight* > *maxHeight*, the maximum dimensions are set to the minimum, effectively fixing the size of the window in that dimension. For control panels, it is better to use `ModifyPanel fixedSize=1`.

Combining these limits with ModifyGraph width and height modes leads to unexpected results and is discouraged.

If you first use sizeLimit and then execute `ModifyPanel fixedSize=1` on the same window, the fixedSize command takes precedence. If you execute `SetWindow sizeLimit` on a control panel that has `fixedSize=1`, Igor generates an error.

Igor includes a `SetWindow sizeLimit` command in a window recreation macro if necessary. Igor6 does not support `SetWindow sizeLimit` so this causes an error when recreating the window in Igor6. If you never set the `sizeLimit` property for a window, or if the minimum dimensions are very small and maximum dimensions are INF, then no SetWindow command is generated.

tooltipHook(*hName*)=*procName*

A tooltip hook function is like a window hook function (see **Named Window Hook Functions** on page IV-295), except that it allows you to alter the text of a tooltip in a graph or table. At present, a tooltip hook function is called for a graph when the mouse hovers over a trace or image, and for a table when the mouse hovers over the data for a wave.

Unlike a window hook function, a tooltip hook function can be set for either a top-level window or a subwindow.

For details, see **Tooltip Hook Functions** on page IV-310.

userdata=*UDStr*
userdata(*UDName*)=*UDStr*

Sets the window or subwindow user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata+=*UDStr*
userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current window or subwindow user data. Use the optional (*UDName*) to append to the named user data.

Details

For details on named window hooks, see **Window Hook Functions** on page IV-293.

Unnamed window hook functions are supported for backward compatibility only. New code should use named window hook functions. For details on unnamed window hooks, see **Unnamed Window Hook Functions** on page IV-305.

For details on marker hooks, see **Custom Marker Hook Functions** on page IV-308.

You can also attach user data to traces in graphs using the userData keyword of the ModifyGraph operation and to controls using the userData keyword of the various control operations.

See Also

The **GetWindow, SetIgorHook**, and **SetIgorMenuMode** operations and **AxisValFromPixel**, **NumberByKey**, **PopupContextualMenu**, and **TraceFromPixel** functions. The **GetUserData** operation for retrieving named user data.

ShowIgorMenus

`ShowIgorMenus` `[`***`MenuNameStr`***` [, `***`MenuNameStr`***`]` ...

The ShowIgorMenus operation shows the named built-in menus or, if none are explicitly named, shows all built-in menus in the menu bar.

User-defined menus attached to built-in menus are also affected by this operation.

Parameters

MenuNameStr	The name of an Igor menu, like “File”, “Data”, or “Graph”.

Details

See **HideIgorMenus** for details.

See Also

HideIgorMenus, **DoIgorMenu**, **SetIgorMenuMode**, **User-Defined Menus** on page IV-125

ShowInfo

`ShowInfo` `[`**`/CP=`*`num`*` /W=`*`winName`***`]`

The ShowInfo operation puts an information panel on the target or named graph. The information panel contains cursors and readouts of values associated with waves in the graph.

Flags

/CP=*num* Selects a cursor pair to display in the info panel.

num=0:	Selects cursor A and cursor B.
num=1:	Selects cursor C and cursor D.
num=2:	Selects cursor E and cursor F.
num=3:	Selects cursor G and cursor H.
num=4:	Selects cursor I and cursor J.

/CP={*n1*,*n2*,...} Allows you to select multiple cursor pairs to be displayed in the info panel. The numbers n1, n2, etc., are the same as the single-pair version of this flag.

This form of /CP was added in Igor Pro 7.00.

/SIDE=side Selects the side of the host window where the info panel should be attached:

side = 0: right side
side = 1: left side
side = 2: bottom side (default)
side = 3: top side

/V=num If num is non-zero, selects a vertical layout for the info panel. Most appropriate when /SIDE=0 or 1.

/W=*winName* Displays info panel in the named window.

See Also

Info Panel and Cursors on page II-319.

The **HideInfo** operation.

Programming With Cursors on page II-321.

ShowTools

`ShowTools [/A/W=winName][toolName]`

The ShowTools operation puts a tool palette for drawing along the left hand side of the target or named graph or control panel, and optionally activates the named tool.

Flags

/A	Sizes window automatically to make extra room for the tool palette. This preserves the proportion and size of the actual graph area.
/W=*winName*	Shows tool palette in the named window. This must be the first flag specified when used in a Proc or Macro or on the command line. *winName* must be either the name of a top-level window or a path leading to an exterior panel window (see **Exterior Control Panels** on page III-443).

Parameters

If you specify a *toolName* (which can be one of: normal, arrow, text, line, rect, rrect, oval, or poly) the named tool is activated. Specifying the "normal" tool has the same effect as issuing the GraphNormal command for a graph that has the drawing tools selected.

Details

The activated tool is not highlighted until the top graph or control panel becomes the topmost (activated) window. Use `DoWindow/F` to bring a window to the top (or "front").

See Also

The **DoWindow**, **GraphNormal**, **GraphWaveDraw**, **GraphWaveEdit**, and **HideTools** operations.

SinIntegral

`SinIntegral(z)`

The SinIntegral(z) function returns the sine integral of z.

If z is real, a real value is returned. If z is complex then a complex value is returned.

The SinIntegral function was added in Igor Pro 7.00.

Details

The sine integral is defined by

$$Si(z)=\int_0^z \frac{\sin(t)}{t}dt.$$

IGOR computes the SinIntegral using the expression:

$$Si(z)=z\,{}_1F_2\left(\frac{1}{2};\frac{3}{2},\frac{3}{2};-\frac{z^2}{4}\right).$$

References

Abramowitz, M., and I.A. Stegun, "Handbook of Mathematical Functions", Dover, New York, 1972. Chapter 5.

See Also

CosIntegral, **ExpIntegralE1**, **hyperGPFQ**

sign

`sign(num)`

The sign function returns -1 if *num* is negative or 1 if it is not negative.

Silent

`Silent num`

The Silent operation is largely obsolete. Only very obscure uses remain and most users can ignore this operation.

Prior to Igor Pro 7, Silent was used to enable or disable the display of macro commands in the command line as they were executed. It was also used to enable compatibility modes for very old experiments.

Parameters

If *num* is 2, commands issued by AppleEvents or ActiveX Automation are not shown in the history are of the command window. Use 3 to re-enable.

If *num* is 100, 101 or 102, all procedures are recompiled. For 102, the time to recompile is displayed in the history.

sin

`sin(angle)`

The sin function returns the sine of *angle* which is in radians.

In complex expressions, *angle* is complex, and sin(*angle*) returns a complex value:

$$\sin(x+iy)=\sin(x)\cosh(y)+i\cos(x)\sinh(y).$$

See Also

asin, **cos**, **tan**, **sec**, **csc**, **cot**

sinc

`sinc(num)`

The sinc function returns sin(*num*)/*num*. The sinc function returns 1.0 when *num* is zero. *num* must be real.

sinh

`sinh(num)`

The sinh function returns the hyperbolic sine of *num*:

$$\sinh(x)=\frac{e^x-e^{-x}}{2}.$$

In complex expressions, *num* is complex, and sinh(*num*) returns a complex value.

See Also

cosh, **tanh**, **coth**

Sleep

`Sleep [flags] timeSpec`

The Sleep operation puts Igor to sleep for a while. After the while is up, Igor continues execution.

You could use Sleep, for example, to give an instrument time to perform an action or to allow a user to admire a graph before proceeding.

More advanced programmers may prefer to use a background task as an alternative. See **Background Tasks** on page IV-319.

Parameters

The format of *timeSpec* depends on which flags, if any, are present.

If no flags are present, then *timeSpec* is in *hh*:*mm*:*ss* format and specifies the number of elapsed hours, minutes and seconds to sleep.

Flags

/A timeSpec is an absolute time in 24 hour format (e.g., 16:00:00).

/A/W Wait until tomorrow if absolute time has passed.

/B Stop sleeping if the user clicks the mouse button.

The /B flag is ignored if you use the /PROG flag.

/C=*cursor* Controls what kind of cursor to display during sleep.

cursor=-1:	No cursor change.
cursor=0:	Hour glass (default).
cursor=1:	Arrow.
cursor=2:	“Click”.
cursor=3:	Spinning beachball.
cursor=4:	Watch with spinning hands.
cursor=5:	Jacob’s ladder.
cursor=6:	Displays a progress dialog instead of changing the cursor.
cursor=7:	Spinning arrrows.
Other:	Watch.

cursor values 3 through 6 require Igor Pro 7.00 or later.

cursor value 7 requires Igor Pro 9.00 or later.

Under rare circumstances, cursors 0, 3, 4, and 5 may cause memory leaks.

/M=*message* If you use /C=6 or /PROG, the progress dialog displays *message* above the progress bar. By default the message reads "Sleeping".

/PROG={*cancelButtonTitleStr*, *continueButtonTitleStr*, *abortMode*}

Displays a progress dialog with user-settable titles for the Cancel and Continue buttons.

If you pass "" for *cancelButtonTitleStr*, the Cancel button is hidden. If you pass "" for *continueButtonTitleStr*, the Continue button is hidden. It is an error to pass "" for both buttons.

If *abortMode* is 0, the **User Abort Key Combinations** and the Cancel button abort any running procedure code. If it is 1, the user abort key combinations and the Cancel button terminate the Sleep operation but user procedure code continues to run.

The /B and /Q flags are ignored if you use the /PROG flag.

See **Displaying a Progress Dialog** below for further information.

/Q Continue executing the procedure containing the Sleep operation even if the **User Abort Key Combinations** were pressed.

The /Q flag is ignored if you use the /PROG flag.

/S *timeSpec* is a numeric expression in seconds.

/T *timeSpec* is a numeric expression in ticks (about 1/60 of a second).

Details

The Sleep operation does *not* let the user choose menus, move cursors, run procedures, draw in graphs, or do any other interactive task.

Normally *timeSpec* specifies an amount of elapsed time. If the /A flag is present, then *timeSpec* is an absolute time when sleep is to end. If the specified absolute time has already passed, no sleep occurs unless you also use /W, which makes it wait until tomorrow.

If you specify time in hh:mm:ss format, you can also specify the time indirectly through a string variable. See the examples.

You can end sleep by pressing the **User Abort Key Combinations**. Normally when you do this, it aborts any procedure that is running. However, if you use the /Q flag, the procedure continues running normally.

Displaying a Progress Dialog

When you specify /C=6 or if you use the /PROG flag, Sleep displays a progress dialog with a progress bar showing how much of the sleep time has passed. The dialog displays a prompt which you can control using the /M flag.

If you use /PROG, then /Q and /B are ignored.

If you use /C=6, then /Q and /B have special meanings:

/C=6	Progress dialog with Cancel button which aborts running procedures. The Abort key combinations abort running user procedures.
/C=6/B	Progress dialog with Abort button which aborts running procedures. The Abort key combinations abort running user procedures.
/C=6/Q	Progress dialog with Continue button which terminates the current sleep operation but allows procedures to continue. Abort key combinations allow running procedures to continue.
/C=6/B/Q	Progress dialog with Continue and Abort buttons. The Continue button terminates the current sleep operation but allows procedures to continue. The Abort button aborts running procedures. Abort key combinations allow running procedures to continue.

If you use the /PROG flag, you can provide your own titles for the Cancel and Continue buttons.

Examples

These examples assume the current time is 4 PM:

```
Sleep 00:01:30            // sleeps for 1 minute, 30 seconds
Sleep/A 23:30:00          // sleeps until 11:30 PM
Sleep/A 03:00:00          // doesn't sleep at all because time is past
Sleep/A/W 03:00:00        // sleeps until 3 AM tomorrow
String str1= "03:00:00"   // put wakeup call time in string
Sleep/A/W $str1           // sleeps until 3 AM tomorrow
Sleep/B/C=2/S/Q 60        // sleep 60 seconds, or until user clicks,
                          //  and keep going (don't abort)
```

The following function creates a graph and then periodically updates the displayed data. By default, it sleeps for a number of seconds specified by the sleepTime parameter.

```
Function SleepDemo(sleepTime, displayProgressDialog)
   Variable sleepTime                   // In seconds
   Variable displayProgressDialog       // 1 for progress dialog

   Make/N=200/O junk
   SetScale/I x 0, 2*pi, junk
   junk=sin(x)
   DoWindow/F SleepDemoGraph
   if (V_Flag == 0)
      Display/N=SleepDemoGraph junk
   endif
   DoUpdate

   try
      Variable i
      for (i = 0; i < 10; i+=1)
         if (displayProgressDialog)
            // Because the abortMode is 0, pressing the user abort key combinations
            // or pressing the Done button generates an abort instead of merely
            // terminating the current Sleep call.
            int abortMode = 0
            Sleep/S/PROG={"Done", "Continue", abortMode} sleepTime
         else
            // /B makes Sleep terminate if the user clicks.
            // Because the /Q flag is omitted, pressing the user abort key combinations
            // or pressing Igor's Abort button generates an abort instead of merely
            // terminating the current Sleep call.
            Sleep/S/C=2/B sleepTime
         endif
         junk = sin(x*(i+2))
         DoUpdate
      endfor
   catch
      Printf "An abort occurred with V_abortCode=%d\r", V_abortCode
   endtry
End
```

Slider

`Slider` `[`**`/Z`**`]` ***`controlName`*** `[`***`key`*** `[`**`=`** ***`value`***`]][,` ***`key`*** `[`**`=`** ***`value`***`]]…`

The Slider operation creates or modifies a Slider control in the target window.

A Slider control sets or displays a single numeric value. The user can adjust the value by dragging a thumb along the length of the Slider.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the Slider control to be created or changed.

The following keyword=value parameters are supported:

align=*alignment* Sets the alignment mode of the control. The alignment mode controls the interpretation of the *leftOrRight* parameter to the pos keyword. The align keyword was added in Igor Pro 8.00.

If *alignment*=0 (default), *leftOrRight* specifies the position of the left end of the control and the left end position remains fixed if the control size is changed.

If *alignment*=1, *leftOrRight* specifies the position of the right end of the control and the right end position remains fixed if the control size is changed.

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

Note: The Slider control reverts to `os9` appearance on Macintosh if thumbColor isn’t the default blue (0,0,65535).

See **Button** and **DefaultGUIControls** for more appearance details.

disable=*d* Sets user editability of the control.

d=0:	Normal.
d=1:	Hide.
d=2:	Draw in gray state; disable control action.

fColor=(*r*,*g*,*b*[,*a*]) Sets the color of the tick marks. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

focusRing=*fr* Enables or disables the drawing of a rectangle indicating keyboard focus:

fr=0:	Focus rectangle will not be drawn.
fr=1:	Focus rectangle will be drawn (default).

On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.

font="*fontName* " Sets the font used to display the tick labels, e.g., `font="Helvetica"`.

fsize=*s* Sets the size of the type for tick mark labels.

help={*helpStr*} Sets the help for the control.

helpStr is limited to 1970 bytes (255 in Igor Pro 8 and before).

You can insert a line break by putting "\r" in a quoted string.

limits= {*low*,*high*,*inc*}

low sets left or bottom value, *high* sets right or top value. Use *inc*=0 for continuous or use desired increment between stops.

live=*l* Controls updating of readout.

l=0:	Update only after mouse is released.
l=1:	Update as slider moves (default).

noproc Specifies that no procedure is to execute when the control's value is changed.

pos={*leftOrRight*,*top*} Sets the position in **Control Panel Units** of the top/left corner of the control if its alignment mode is 0 or the top/right corner of the control if its alignment mode is 1. See the align keyword above for details.

pos+={*dx*,*dy*} Offsets the position of the slider in **Control Panel Units**.

proc=*procName* Specifies the action procedure for the slider.

rename=*newName* Gives control a new name.

repeat={*style*,*springback*,*rate*,*restingValue*}

Set the control to call its action procedure repeatedly at timed intervals while the user clicks the thumb.

style can take one of the following values:

0:	No repeat (default). Use this to turn the repeat feature off.
1:	Slider repeats at a constant rate.
2:	Slider repeats at a rate proportional to the distance from the *restingValue*.

springback controls what happens to the slider value when the user releases the mouse. If *springback* is 1, the slider returns to the resting value. If it is 0, it remains at the last set value.

rate specifies the rate at which the action procedure is called in calls per second. The maximum rate accepted rate is 1000. If *rate* is 0, it sets style to 0 which turns the repeating feature off.

restingValue specifies the value to which the slider returns when the user releases the mouse. If *springback* is 1, the thumb automatically returns to *restingValue* when the mouse button is released. If *springback* is 0, *restingValue* has no effect.

side=*s* Controls slider thumb.

s=0:	Thumb is blunt and tick marks are suppressed.
s=1:	Thumb points right or down (default).
s=2:	Thumb points up or left.

size={*width*,*height*} Sets width or height of control in **Control Panel Units**. *height* is ignored if vert=0 and *width* is ignored if vert=1.

thumbColor=(*r*,*g*,*b*[,*a*]) If appearance={os9} is in effect, sets dominant foreground color of thumb. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. Alpha (*a*) is accepted but ignored.

ticks=*t* Controls slider ticks.

t=0: No ticks.

t=1: Number of ticks is calculated from limits (no ticks drawn if calculated value is less than 2 or greater than 100). Default value.

t>1: *t* is the number of ticks distributed between the start and stop position. Ticks are labeled using the same automatic algorithm used for graph axes. Use negative tick values to force ticks to not be labeled. Ticks are shown on the side specified by the side keyword and are not drawn if side=0.

tkLblRot= *deg* Rotates tick labels. *deg* is a value between -360 and 360.

userdata(*UDName*)=*UDStr*

Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

userTicks={*tvWave,tlblWave*}

User-defined tick positions and labels. *tvWave* contains the tick positions, and text wave *tlblWave* contains the labels. See ModifyGraph userticks for more info. Overrides normal ticking specified by ticks keyword.

value=*v* *v* is the new value for the Slider.

valueColor=(*r*,*g*,*b*[, *a*]) Sets the color of the tick labels. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

variable= *var* Sets the variable (*var*) that the slider will update. It is not necessary to connect a Slider to a variable — you can get a Slider's value using the **ControlInfo** operation.

vert=*v* Set vertical (*v* =1; default) or horizontal (*v* =0) orientation of the slider.

win=*winName* Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

The target window must be a graph or panel.

If you use negative ticks to suppress automatic labeling, you can label tick marks using drawing tools (panels only).

Slider Action Procedure

The action procedure for a Slider control takes a predefined `WMSliderAction` structure as a parameter to the function:

```
Function ActionProcName(S_Struct) : SliderControl
   STRUCT WMSliderAction &S_Struct
   …
   return 0
End
```

The "`: SliderControl`" designation tells Igor to include this procedure in the Procedure pop-up menu in the Slider Control dialog.

See **WMSliderAction** for details on the WMSliderAction structure.

See **Handling Slider Events** on page III-430 for a demonstration of slider event handling.

Although the return value is not currently used, action procedures should always return zero.

This old format for a slider action procedure should not be used in new code:

```
Function MySliderProc(name, value, event) : SliderControl
   String name            // name of this slider control
   Variable value         // value of slider
   Variable event         // bit field:bit 0:value set; 1:mouse down,
                          //  2:mouse up, 3:mouse moved

   return 0               // other return values reserved
End
```

Repeating Sliders

A repeating slider calls your action procedure periodically while the user is clicking the thumb. See **Repeating Sliders** on page III-418 for an overview.

When the repeat style is set to 2, the call rate is proportional to the slider value. See the Slider reference documentation in the online help for the formula.

Whereas the rate can be set as high as 1000 calls per second, if your action procedure takes more time to execute than the period of the slider repeat, you will simply lose repeats. It is unlikely that a rate of more than about 50 calls per second can realistically be achieved.

Examples

```
Function SliderExample()
   NewPanel /W=(150,50,501,285)
   Variable/G var1
   Execute "ModifyPanel cbRGB=(56797,56797,56797)"
   SetVariable setvar0,pos={141,18},size={122,17},limits={-Inf,Inf,1},value=var1
   Slider foo,pos={26,31},size={62,143},limits={-5,10,1},variable=var1
   Slider foo2,pos={173,161},size={150,53}
   Slider foo2,limits={-5,10,1},variable=var1,vert=0,thumbColor=(0,1000,0)
   Slider foo3,pos={80,31},size={62,143}
   Slider foo3,limits={-5,10,1},variable=var1,side=2,thumbColor=(1000,1000,0)
   Slider foo4,pos={173,59},size={150,13}
   Slider foo4,limits={-5,10,1},variable=var1,side=0,vert=0
   Slider foo4,thumbColor=(1000,1000,1000)
   Slider foo5,pos={173,90},size={150,53}
   Slider foo5,limits={-5,10,1},variable= var1,side=2,vert=0
   Slider foo5,ticks=5,thumbColor=(500,1000,1000)
End
```

See Also

Chapter III-14, **Controls and Control Panels**, **Handling Slider Events** on page III-430

ControlInfo, **GetUserData**

Demos

Choose File→Example Experiments→Feature Demos 2→Slider Labels.

Choose File→Example Experiments→Feature Demos 2→Slider Repeat Demo.

SliderControl

`SliderControl`

SliderControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined slider control. See **Procedure Subtypes** on page IV-204 for details. See **Slider** for details on creating a slider control.

Slow

`Slow` ***`ticks`***

The Slow operation is obsolete. Prior to Igor Pro 7 it slowed down execution of macros for debugging purposes. It now does nothing.

Smooth

Smooth [***flags***] ***num***, ***wave0*** [, ***wave1***]…

The Smooth operation smooths the named waves using binomial (Gaussian) smoothing, boxcar (sliding average) smoothing, Savitzky-Golay (polynomial) smoothing, or running-median filtering.

Parameters

num is the number of smoothing operations to be applied for binomial smoothing or the integer number of points in the smoothing window for boxcar, Savitzky-Golay, and running-median smoothing, or, if you use /BLPF, the low-pass cutoff frequency.

Each wave, which may be floating point or integer, is smoothed in-place, overwriting the values with the smoothed result.

If an input wave is complex, the real and imaginary parts are smoothed independently.

If an input wave contains NaNs, smoothing results are unsatisfactory unless running-median (/M) filtering is used with a median of 3 or more. The **Loess** operation can fill in NaNs and the **Interpolate2** operation ignores NaNs and INFs.

Flags

/B [=*b*]   Invokes boxcar smoothing algorithm. If given, *b* specifies the number of passes to use when smoothing the data with smoothing factor *num* (box width). The number of passes can be any value between 1 and 32767.

/BLPF[=*roundMode*]   Calculates the integer number of smoothing operations to be applied for binomial smoothing based on interpreting the *num* parameter as the low-pass cutoff frequency and then performs the smoothing. /BLPF was added in Igor Pro 8.00.

The calculated number of smoothing operations is returned in V_iterations:

$$\text{V\_iterations} = \ln(\sqrt{2})\left(\frac{1}{(\pi f_{cn})^2} - \frac{1}{6}\right)$$

where f_{cn} = *num* * DimDelta(wave0, *dim*)

If *roundMode* is 1, V_iterations is rounded down to an integer value. This is the default if you use /BLPF by itself without *roundMode*.

If *roundMode* is 2, V_iterations is rounded up to an integer value.

f_{cn} is the Nyquist-normalized cutoff frequency and should be ≤ 0.182028, which results in V_iterations = 1.

The cutoff frequency, also known as the "half power point", is the frequency at which the magnitude response of the smoothing filter declines to 1/sqrt(2) ≈ 0.707107.

/DIM=*dim*   Specifies the wave dimension to smooth.

dim=-1:	Treats entire wave as 1D (default)
dim=0:	Operates along rows
dim=1:	Operates along columns
dim=2:	Operates along layers
dim=3:	Operates along chunks

/E=*endEffect*   Determines how to handle the ends of the wave (w) when fabricating missing neighbor values.

endEffect=0:	Bounce method (default). Uses w[*i*] in place of the missing w[-*i*] and w[*n*-*i*] in place of the missing w[*n*+*i*].
endEffect=1:	Wrap method. Uses w[*n*-*i*] in place of the missing w[-*i*] and vice versa.
endEffect=2:	Zero method. Uses 0 for any missing value.
endEffect=3:	Repeat method. Uses w[0] in place of the missing w[-*i*] and w[*n*] in place of the missing w[*n*+*i*].

/EVEN [=*evenAllowed*]

Specifies the smoothing increment for boxcar smoothing (/B). Values are:

0:	Increments even values of *num* to the next odd value. Default when /EVEN omitted.
1:	Uses even values of *num* for boxcar smoothing despite the half-sample shifting this introduces in the smoothed output (prior to version 6, this shift was prevented). Same as/EVEN alone.

/F [=*f*]   Selects the boxcar or multipass binomial smoothing method:

f=0:	Slow, but accurate, method (default).
f=1:	Fast method. Same as /F alone.

/M=*threshold* Invokes running-median smoothing and specifies an absolute numeric threshold used to optionally replace “outliers”. Points that differ from the central median by an amount exceeding *threshold* are replaced, either with the *replacement* value specified by /R, or otherwise with the median value.

Special *threshold* values are:

0: Replace all values with running-median values or the *replacement* value.

(NaN): Replace only NaN input values with running-median values or the *replacement* value.

The smoothing factor *num* is the number of points in the smoothing window used to compute each median.

/MPCT=*percentile* Used with /M to compute a smoothed value that is a different percentile than the median. /M must be present if /MPCT is used.

percentile is a value from 0 to 100.

Roughly speaking, the smoothed value returned is the smallest value in the smoothing window that is greater than the smallest *percentile* % of the values. See "Median and Percentile Smoothing Details", below.

percentile=0: The smoothed value is the minimum value in the smoothing window.

percentile=50: The smoothed value is the median of the values in the smoothing window. This is the default if /MPCT is omitted.

percentile=100: The smoothed value is the maximum value in the smoothing window.

/R=*replacement* Specifies the value that replaces input values that exceed the central median by *threshold* (requires /M). *replacement* can be any value (including NaN or ±Inf if *waveName* is floating point).

/S=*sgOrder* Invokes Savitzky-Golay smoothing algorithm and specifies the smoothing order. *sgOrder* must be either 2 or 4.

Binomial Smoothing Details

For binomial smoothing, use no flags other than /BLPF, /DIM, /E, and /F.

If you are not using /BLPR, *num* is the number of smoothing operations between 1 and 32767.

If you are using /BLPF, *num* is the low-pass cutoff frequency which must be less than 0.182028 times the sampling frequency of *wave0*. The computed smoothing factor is returned in V_iterations. Very small values of f_{cn} (≈ 0.0007 or less) will fail with errors due to excessive number of smoothing operations or to insufficient range (too few wave points).

In Igor Pro 6 and later, the binomial smooth algorithm automatically switches to a nearly-equivalent but much faster multipass box smooth when *num* >= 50. You can prevent the switch to box smoothing by setting this global variable:

```
Variable/G root:V_doOrigBinomSmooth=1
```

The binomial smoothing algorithm does not detect and ignore NaNs in the input data. Use **Loess** or **Interpolate2** to fill in NaNs.

Boxcar Smoothing Details

For boxcar smoothing, use the /B flag and a *num* value from 1 to 32767.

For *num* < 2, no smoothing is done.

If *num* is even and /EVEN is not specified, *num* is incremented to the next (odd) integer.

If *num* is even and /EVEN is specified, each smoothed output is formed from one more previous value than future values.

The fast boxcar smoothing (/F) algorithm creates small errors when the data has a large offset. For some data sets you may want to subtract the mean of the data before smoothing and add it back in afterwards.

The boxcar smoothing algorithm detects and ignores NaNs in the input data. If *num* is less than the number of NaNs near the output point, then the result is NaN. Otherwise the average of the non-NaN neighboring points is used to compute the smoothed result.

Savitzky-Golay Smoothing Details

For Savitzky-Golay smoothing, use the /S flag and an odd *num* value from 5 to 25. An even value for *num* returns an error. If *sgOrder*=4, then *num*= 5 gives no smoothing at all so *num* should be at least 7.

The Savitzky-Golay smoothing algorithm does not detect and ignore NaNs in the input data.

Median and Percentile Smoothing Details

For running-median smoothing, use the /M flag and a *num* value from 1 to 32767. When *num* is 1, no smoothing is done.

If *num* is even, the median is the average of the two middle values.

For example, the median of 6 values around data[i] is the median of data[i-3], data[i-2], data[i-1], data[i], data[i+1], and data[i+2], and if these values were already sorted, the median would be the average of data[i-1] and data[i].

Use /M=0 to replace all values with the median over the smoothing window or use `/M=`*`threshold`*`/R=(NaN)` to replace outliers with NaNs.

Use `/M=(NaN)` to replace only NaN input values with the running-median values or the *replacement* value.

The running-median smoothing algorithm detects and ignores NaNs in the input data. If *num* is less than the number of NaNs near the output point, then the result is NaN. Otherwise the median of the non-NaN neighboring points is used to compute the smoothed result.

The running-median is a special case of running-percentile, with *percentile*=50.

The /M and /MPCT algorithm uses an interpolated rank to compute the value of percentiles other than 0 and 100.

Using Example 1 from <http://cnx.org/content/m10805/latest/> ("A Third Definition"), the 25th percentile (/MPCT=25) of the 8 values:

```
Make/O sortedData={3,5,7,8,9,11,13,15}// Already sorted, rank 1 to 8
```

The first step is to compute the rank (R) of the 25th percentile. This is done using the following formula: R=(*percentile*/100)*(*num*+1), where *percentile* is 25 and *num* is 8, so here R = 2.25.

If R were an integer, the Pth percentile would be the number with rank R; if R were 2 the result would be the 2nd value = 5.

Since R is not an integer, we compute the Pth percentile by interpolation as follows:

1. Define IR as the integer portion of R (the number to the left of the decimal point). For this example, IR=2.
2. Define FR as the fractional portion of R. For this example, FR=0.25
3. Find the values with Rank IR and with Rank IR+1. For this example, this means the values with Rank 2 and the score with Rank 3. The values are 5 and 7.
4. Interpolate by multiplying the difference between the values by FR and add the result to the lower values. For these data, this is 0.25(7-5)+5=5.5

Therefore, the 25th percentile is 5.5:

```
Smooth/M=0/MPCT=(percentile) 8, sortedData  // 8-point smoothing window
Print sortedData[3] // prints 5.5, the 25th percentile of all 8 values
```

Smoothing Window and End Effects Details

These smoothing algorithms compute the output value for a given point using each point’s neighbors. Except for running-median smoothing, each algorithm combines neighboring points before and after the point being smoothed. At the start or end of a wave some points will not have enough neighbors so some method for fabricating neighbor values must be implemented. The /E flag specifies the method.

The running-median filter, however, ignores /E. At each end of the data fewer values are included in the median calculation, so that values “beyond” the end of data are not needed.

The first output value is the median of `wave[0, floor((num-1)/2)]`. For example, if *num* = 7, then the first output value is the median of wave[0], wave[1], wave[2], and wave[3]. Because that is an even number of points, the median is the average of the two middle values. Continuing the example, if the values were 3, 1, 7, and 5, the two middle values are 3 and 5. The computed median would be (3+5)/2=4.

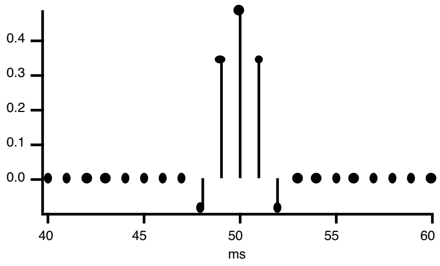
Examples

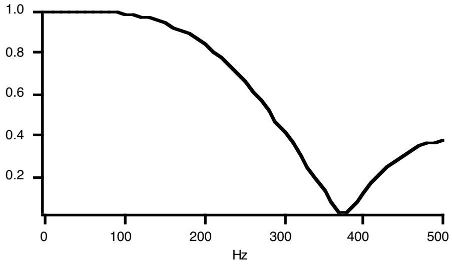
Box smoothing example:

```
Make/N=100 wv; Display wv
wv=gnoise(1)
Smooth/B/E=3 3,wv    // output[p] = average of wv[p-1], wv[p] and wv[p+1]
                     // /E=3 causes wv[0] = (w[0]+w[0]+w[1])/3
                     // and wv[n-1] = (w[n-2]+w[n-1]+w[n-1])/3
```

Demonstrate the impulse response of Savitzky-Golay Smoothing:

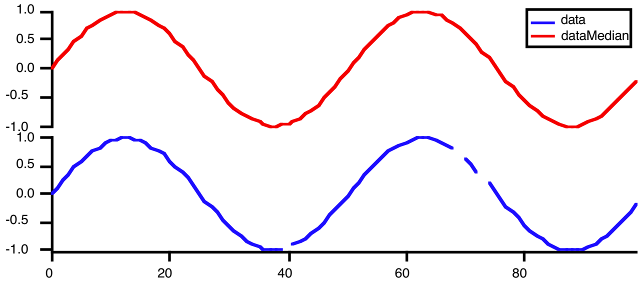
```
Make/O/N=100 wv
wv= p==50    // 1 at center of wave, 0 elsewhere; an impulse
SetScale/P x, 0, 1/1000, "s", wv            // 1000 Hz sampling rate
Smooth/S=2 5,wv
Display wv
ModifyGraph mode=8,marker=19
FFT/MAG/DEST=fftMag wv
Display fftMag
```





Replace NaN with median:

```
Make/O/N=100 data= enoise(1)>.9 ? NaN : sin(x/8)      // signal with NaNs
Duplicate/O data, dataMedian
Smooth/M=(NaN) 5, dataMedian      // replace (only) NaNs with 5-point median
```



Binomial Smoothing References

Marchand, P., and L. Marmet, *Revues of Scientific Instrumentation 54*, 1034, 1983.

Savitzky-Golay Smoothing References

Savitzky, A., and M.J.E. Golay, *Analytical Chemistry*, *36*, 1627-1639, 1964.

Steiner, J., Y. Termonia, and J. Deltour, *Analytical Chemistry*, *44*, 1906-1909, 1972.

Madden, H., *Analytical Chemistry*, *50*, 1386-1386, 1978.

Percentile References

<http://en.wikipedia.org/wiki/Percentile>

<http://cnx.org/content/m10805/latest/>

See Also

See the **Loess**, **MatrixConvolve**, and **MatrixFilter** operations for true 2D smoothing.

FilterFIR, **FilterIIR**, **Loess**, **Interpolate2**

Also see the "Smooth Operation Responses" example experiment.

SmoothCustom

`SmoothCustom [/E=`*`endEffect`*`] `*`coefsWaveName`*`, `*`waveName`*` [, `*`waveName`*`]`...

Note: SmoothCustom is obsolete. Use the **FilterFIR** operation instead. For multidimensional data use the **MatrixConvolve** or **MatrixFilter** operations.

The SmoothCustom operation smooths waves by convolving them with *coefsWaveName*.

Parameters

coefsWaveName must be single or double floating point, must not be one of the destination *waveName*s, must not be complex.

waveName is a numeric destination wave that is overwritten by the convolution of itself and *coefsWaveName*.

Flags

/E=*endEffect*	End effect method, a value between 0 and 3. See the **Smooth** operation for a description of the /E flag.

Details

The convolution is in the time domain. That is, the FFT is not employed. For this reason the length of *coefsWaveName* should be small or small in comparison to the destination waves.

SmoothCustom presumes that the middle point of *coefsWaveName* corresponds to the delay = 0 point. The "middle" point number = trunc(numpnts(*coefsWaveName*-1)/2). *coefsWaveName* usually contains the two-sided impulse response of a filter, and contains an odd number of points. This is the type of wave created by FilterFIR.

SmoothCustom ignores the X scaling of all the waves.

The SmoothCustom operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-95 for details.

Sort

`Sort [ /A /DIML /C /R ] `*`sortKeyWaves`*`, `*`sortedWaveName`*` [, `*`sortedWaveName`*`]`...

The Sort operation sorts the *sortedWaveName*s by rearranging their Y values to put the data values of *sortKeyWaves* in order.

Parameters

sortKeyWaves is either the name of a single wave, to use a single sort key, or the name of multiple waves in braces, to use multiple sort keys.

All waves must be of the same length.

The *sortKeyWaves* must not be complex.

Flags

/A[=*a*] Alphanumeric sort. When *sortKeyWaves* includes text waves, the normal sorting places "wave1" and "wave10" before "wave9".

The optional *a* parameter requires Igor Pro 7.00 or later.

Use /A or /A=1 to sort the number portion numerically, so that "wave9" is sorted before "wave10".

Use /A=2 to ignore + and - characters in the text so that "Text-09" sorts before "Text-10".

/C Case-sensitive sort. When *sortKeyWaves* includes text waves, the sort is case-insensitive unless you use the /C flag to make it case-sensitive.

/DIML Moves the dimension labels with the values (keeps any row dimension label with the row's value).

/LOC Performs a locale-aware sort.

When *sortKeyWaves* includes text waves, the text encoding of the text waves' data is taken into account and sorting is done according to the sorting conventions of the current system locale. This flag is ignored if the text waves' data encoding is unknown, binary, Symbol, or Dingbats. This flag cannot be used with the /A flag. See Details for more information.

The /LOC flag was added in Igor Pro 7.00.

/R Reversed sort; sort from largest to smallest.

Details

sortKeyWaves are not actually sorted unless they also appear in the list of destination waves.

The sort algorithm does not maintain the relative position of items with the same key value.

When the /LOC flag is used, the bytes stored in the text wave at each point are converted into a Unicode string using the text encoding of the text wave data. These Unicode strings are then compared using OS specific text comparison routines based on the locale set in the operating system. This means that the order of sorted items may differ when the same sort is done with the same data under different operating systems or different system locales.

When /LOC is omitted the sort is done on the raw text without regard to the waves' text encoding.

Examples

```
Sort/R myWave,myWave       // sorts myWave in decreasing order
Sort xWave,xWave,yWave     // sorts x wave in increasing order,
                           // corresponding yWave values follow.
Make/O/T myWave={"1st","2nd","3rd","4th"}
Make/O key1={2,1,1,1}      // places 2nd, 3rd, 4th before 1st.
Make/O key2={0,1,3,2}      // arranges 2nd, 3rd, 4th as 2nd, 4th, 3rd.
Sort {key1,key2},myWave    // sorts myWave in increasing order by key1.
                           // For equal key1 values, sorted by key2.
                           // Result is myWave={"2nd","4th","3rd","1st"}
Make/O/T tw={"w1","w10","w9","w-2.1"}
Sort/A tw,tw     // sorts tw in increasing number-aware order:
                 // Result is tw={"w-2.1","w1","w9","w10"}
```

See Also

Sorting on page III-132

MakeIndex, **IndexSort**, **Reverse**, **SortColumns**, **SortList**

FindDuplicates, **TextHistogram**

SortColumns

SortColumns [*flags*] keyWaves={*waveList*}, sortWaves={*waveList*}

The SortColumns operation rearranges data in columns of the *sortWaves* using the data movements that would sort the values of the *keyWaves* if they were sorted.

The SortColumns operation was added in Igor Pro 7.00.

Parameters

keyWaves is a lists of 1 or more wave references in braces separated by commas. The first listed wave is the primary sort key, the second is the secondary sort key, and so on. The *keyWaves* list can contain a maximum of 10 waves. The key waves can be either text or real numeric waves but all key waves must be of the same type and have the same number of points. Complex waves, wave reference waves and data folder reference waves can not be used as key waves.

sortWaves is a lists of one or more wave references in braces separated by commas. The *sortWaves* list can contain a maximum of 100 waves.

Flags

/A Alphanumeric sort.

When *keyWaves* includes text waves, or the /KNDX flag is used and the first wave in the *sortWaves* list is a text wave, the normal sorting places "wave1" and "wave10" before "wave9". Use /A to sort the number portion numerically, so that "wave9" is sorted before "wave10". /A cannot be used with the /LOC flag.

/C Case-sensitive sort. When *keyWaves* includes text waves, or the /KNDX flag is used and the first wave in the *sortWaves* list is a text wave, the sort is case-insensitive unless you use the /C flag to make it case-sensitive.

/DIML Moves the row dimension labels with the data values. Column dimension labels remain unchanged.

/KNDX={*c0*, *c1*, ... *c9*}

Specifies up to 10 columns of the first wave in the *sortWaves* list to use as the sort keys. This flag and the *keyWaves* keyword are mutually exclusive. If this flag is used then the first wave in the *sortWaves* list must be either a real numeric or text wave.

/LOC Locale aware sort.

When *keyWaves* includes text waves, or the /KNDX flag is used and the first wave in the *sortWaves* list is a text wave, the text encoding of the text waves' data is taken into account and sorting is done according to the sorting conventions of the current system locale.

/LOC is ignored if the text waves' data encoding is unknown, binary, Symbol, or Dingbats.

/LOC can not be used with the /A flag.

See Details for more information.

/R Reverses the sort, sorting from largest to smallest.

Details

Waves in the *keyWaves* list are not actually sorted unless they also appear in the *sortWaves* list.

All waves must have the same number of rows but can have different numbers of columns, layers and chunks.

keyWaves, or the first wave in the *sortWaves* list when /KNDX is used, must be either numeric or text waves.

When the *sortWaves* list includes 3D or 4D waves, the operation sorts all columns of all layers/chunks.

The sorting algorithm used does not maintain the relative position of rows with the same key value.

When the /LOC flag is used, the bytes stored in the text wave at each point are converted into a Unicode string using the text encoding of the text wave data. These Unicode strings are then compared using OS-specific text comparison routines based on the current locale as set in the operating system. This means that the order of sorted items may differ when the same sort is done with the same data under different operating systems or different system locales.

Examples

```
// Define a function that creates sample data
Function CreateSampleData()
   Make/O key1={3,1,0,2}
   Make/O/T text1={"Jack","Fred","Robin","Bob"}
   Make/O w1={{1,2,3,4},{11,12,13,14}}
End

// Create sample data and display in a table
CreateSampleData()
Edit key1,text1,w1

// Sort based on a numeric key
SortColumns keyWaves=key1,sortWaves=w1
```

```
// Revert the data
CreateSampleData()

// Sort based on text key
SortColumns keyWaves=text1,sortWaves=w1

// Revert the data
CreateSampleData()

// Sort using key index
SortColumns/kndx=0 sortWaves={text1,w1}
```

See Also

Sorting on page III-132, **Sort**, **Reverse**, **SortList**

SortList

SortList(*listStr* [, *listSepStr* [, *options*])

The SortList function returns *listStr* after sorting it according to the default or *listSepStr* and *options* parameters. *listStr* should contain items separated by *listSepStr*, such as "the first item;second item;".

Use SortList to sort the items in a string containing a list of items separated by a string, such as those returned by functions like **TraceNameList** or **WaveList**, or a line of text from a delimited text file, where listSepStr can be "\r" or "\r\n".

listSepStr and *options* are optional; their defaults are ";" and 0 (ascending alphabetic sort), respectively.

Details

listStr is treated as if it ends with a *listSepStr* even if it doesn’t. The returned list will always have an ending *listSepStr* string.

In Igor6, SortList used only the first byte of listSepStr. As of Igor7, it uses the whole string.

options controls the sorting method, as follows:

0:	Default sort (ascending case-sensitive alphabetic ASCII sort).
1:	Descending sort.
2:	Numeric sort.
4:	Case-insensitive sort.
8:	Case-sensitive alphanumeric sort.
16:	Case-insensitive alphanumeric sort that sorts wave0 and wave9 before wave10.
32:	Unique sort in which duplicates are removed. Added in Igor Pro 7.00.
64:	Ignore + and - in the alphanumeric sort so that “Text-09” sorts before “Text-10”. Set options to 80 or 81. Added in Igor Pro 7.00.

options may also be a bitwise combination of these values with the following restriction: only one of 2, 4, 8, or 16 may be specified. Thus the legal values are thus 0, 1, 2, 3, 4, 5, 8, 9, 16, 17, 32, 33, 34, 35, 36, 40, 41, 48, 49, 80 or 81. Other values will produce undefined sorting.

In a case-insensitive, unique sort (options=4+32), if two items differ only in case, which one is retained is not specified.

Examples

```
// Alphabetic sorts
Print SortList("c;a;a;b")                   // prints "a;a;b;c;"
Print SortList("you,me,More", ",", 0)       // prints "More,me,you,"
Print SortList("you,me,More", ",", 4)       // prints "me,More,you,"
Print SortList("9,93,91,33,15,3", ",")      // prints "15,3,33,9,91,93,"
Print SortList("Zx;abc;All;", ";", 0)       // prints "All;Zx;abc;"
Print SortList("Zx;abc;All;", ";", 8)       // prints "abc;All;Zx;"
Print SortList("w9;w10;w02;", ";", 16)      // prints "w02;w9;w10;"

// Unique sort
Print SortList("b;c;a;a;", ";", 32)         // prints "a;b;c;"
Print SortList("b;c;A;a;", ";", 4+32)       // prints "A;b;c;"
Print SortList("b;c;a;A;", ";", 4+32)       // prints "a;b;c;"

// Numeric sorts
Print SortList("9,93,91,33,15,3",",",2)     // prints "3,9,15,33,91,93,"
Print SortList("9,93,91,33,15,3",",",3)     // prints "93,91,33,15,9,3,"
```

See Also

Sort, **StringFromList**, **WaveList**, **RemoveEnding**

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

SoundInRecord

SoundInRecord [**/BITS=*bits* /Z**] ***wave***

The SoundInRecord operation records audio input at the sample rate obtained from the wave's X scaling and for the number of points determined by the length of the wave. The recording is done synchronously.

The number type of the wave must be a type supported by the sound input hardware as reported by the SoundInStatus operation. Use **SoundInStatus** with the format keyword to check if a particular format is supported.

On Windows 8-bit or 16-bit integer are typically supported. On Macintosh, 16-bit integer or 32-bit floating point (the Mac OS native type for sound) are typically supported.

To record in stereo, provide a 2 column wave. (The software is designed to handle any number of channels but has not been tested on more than 2.)

Flags

/BITS=*bits* Controls the number of bits used for each recorded sound sample.

Use /BITS=24 with a 32-bit integer wave for 24-bit sound data capable of representing values from -8,388,608 to +8,388,607.

If you omit /BITS or use /BITS=0, SoundInRecord uses the wave's data type and size to determine how many bits are recorded for each sound sample.

The /BITS flag was added in Igor Pro 9.00.

/Z Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SoundInRecord requires a computer with sound inputs. Several sample experiments using sound input can be found in your Igor Pro Folder in the Examples folder.

See Also

SoundInSet, **SoundInStartChart**, **SoundInStatus**

SoundInSet

SoundInSet [**/Z**][**gain=*g*, agc=*a***]

The SoundInSet operation is used to setup the input device for recording.

Parameters

SoundInSet can accept multiple *keyword =value* parameters on one line.

agc=*a* Turns automatic gain control mode on (*a*=1) or off (*a*=0). Will generate an error if device does not support setting agc. Use SoundInStatus to check or use /Z flag to make errors nonfatal.

Windows: This is not supported and V_SoundInAGC from the SoundInStatus command always returns -1.

gain=*g* Sets input gain, 0 is lowest gain and 1 is highest. Will generate an error if device does not support setting gain. Use SoundInStatus to check or use /Z flag to make errors nonfatal.

Windows: SoundInSet attempts to adjust the master gain of the sound input device but not all sound cards have a master gain. If V_SoundInGain from the SoundInStatus command returns -1, you will have to use your sound card software to adjust the input gain for the particular input source your are using. On some cards there are separate line-in and microphone-in sources.

Flags

/Z Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SoundInSet requires a computer with sound inputs. Several sample experiments using sound inputs are in your Igor Pro Folder in the Examples folder.

See Also

The **SoundInRecord**, **SoundInStartChart**, and **SoundInStatus** operations.

SoundInStartChart

`SoundInStartChart` `[`**`/Z`**`]` ***`buffersize , destFIFOname`***

The SoundInStartChart operation starts audio data acquisition into the given FIFO.

Parameters

buffersize is the number of bytes to allocate for the interrupt time buffer which then feeds into the given Igor named FIFO *destFIFOname*. The FIFO must be set up with the correct number of channels and number type - use **SoundInStatus** to find legal values. The sample rate is read from the FIFO also, so that also needs to be correct.

Flags

/Z Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SoundInStartChart requires a computer with sound inputs. Several sample experiments using sound inputs are in your Igor Pro Folder in the Examples folder.

On systems where 32-bit floating point data is supported, you can use NewFIFOChan with no flags and a range of -1 to 1.

See Also

The **SoundInRecord**, **SoundInSet**, **SoundInStatus** and **SoundInStopChart** operations, and **FIFOs and Charts** on page IV-313.

SoundInStatus

`SoundInStatus` `[`**`format={`*`intOrFloat`*`,`*`channels`*`,`*`bits`*`,`*`frequency`*`}`**`]`

The SoundInStatus operation creates and sets a set of variables and strings with information about the current sound input device.

Keywords

format={*intOrFloat*, *channels*, *bits*, *frequency*}

Checks if a specific sound input format is supported. This is useful because it is often the case that more sound input formats are supported than are reported via the V_SoundInSampSize and W_SoundInRates outputs.

V_Flag is set to 0 if the specified format is supported by the input device or to a non-zero error code otherwise.

The format keyword was added in Igor Pro 9.00.

intOrFloat is a keyword (unquoted), either int or float.

channels is 1 for mono, 2 for stereo. Additional channels may also work with some sound hardware.

bits may be any value between 8 and 64, though 8, 16, 24, and 32 are most likely to be supported by the operating system. CD audio samples use 16 bits.

frequency is the sampling rate in Hertz, with a minimum of 4800. CD audio is sampled at 44100 Hz. A practical upper limit is 192000.

Outputs

`V_Flag`	Set to 0 if the device is available or to a non-zero error code. Also, if you use the format keyword and the specified format is not supported then V_Flag is set to a non-zero error code. If V_Flag is non-zero then none of the following outputs are valid.
`S_SoundInName`	String with name of device.
`V_SoundInAGC`	Automatic gain control on or off (1 or 0). This is an optional item and if the current device does not support AGC then V_SoundInAGC will be set to -1.
`V_SoundInChansAv`	Available number of channels (e.g., 1 for mono, 2 for stereo).
`V_SoundInGain`	Current input gain. Ranges from 0 (lowest) to 1. This is an optional item and if the current device does not support gain then V_SoundInGain will be set to -1.
`V_SoundInSampSize`	Bits set depending on number of bits available in a sample. Bit 0: Set if 8-bit integer is supported. Bit 1: Set if 16-bit integer is supported. Bit 2: Set if 32-bit integer is supported. Bit 3: Set if 32-bit floating point is supported (range is -1 to 1). Bit 4: Set if 64-bit floating point is supported (range is -1 to 1).
`W_SoundInRates`	Wave containing sample rate information: If point 0 contains 0 then points 1 and 2 contain the lower and upper limits of a continuous range; otherwise point 0 contains the number of discrete rates which follow in the wave. The usual rates are 44100 Hz and 4800 Hz.

See Also

SoundInSet, **SoundInRecord**, **SoundInStartChart**

SoundInStopChart

`SoundInStopChart` `[`**`/Z`**`]`

The SoundInStopChart operation stops audio data acquisition started by SoundInStartChart.

Flags

/Z	Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SoundInStopChart requires a computer equipped with sound input hardware.

Audio data acquisition also stops automatically when an experiment is closed.

See Also

The **SoundInStartChart** and **SoundInStatus** operations.

SoundLoadWave

`SoundLoadWave [flags] waveName [ ,fileNameStr ]`

The SoundLoadWave operation loads sound data from the named file into a wave. Mono, stereo, surround-sound, and high-resolution sound formats are supported.

The SoundLoadWave operation was added in Igor Pro 7.00.

Parameters

waveName is the name of the wave to load the sound into.

If *fileNameStr* is omitted or is "", SoundLoadWave displays an Open File dialog.

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with pathName, or the name of a file in the folder associated with pathName. If SoundLoadWave can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to choose the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/I [=*filterStr*]	Force interactive mode. Use optional filter string to limit allowable file extensions. See **Open File Dialog File Filters** on page IV-149.
/O	Overwrite existing waves in case of a name conflict.
/P=*pathName*	Specifies the folder to load the file from. *pathName* is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\`". See **Symbolic Paths** on page II-22 for details.
/Q	Quiet: Doesn't print message to history area, and doesn't abort, if the sound can not be loaded. V_Error is set to the returned error code, which will be zero if there was no error.
/S=(*startT*,*endT*)	Load a subrange of the sound resource. *startT* and *endT* are in seconds, clipped to the duration of the loaded sound.
/TMOT= *timeOut*	Aborts load if *timeOut*, in seconds, is exceeded.

Details

SoundLoadWave uses Core Audio on Macintosh and Qt framework calls on Windows. Note that some files can not be loaded due to digital rights managment issues even though they can be played.

If *waveName* specifies a wave that does not exist, it is created. The wave is redimensioned to a wave type that maintains the numeric precision of the sound data. If the wave can not be created or resized to fit the loaded data then SoundLoadWave returns an error.

If *waveName* does exist, the wave is overwritten only if the /O flag is specified. Without the /O flag SoundLoadWave returns an error.

Multi-channel audio is loaded into sequential columns of the destination wave.

On Macintosh, Igor uses Core Audio to produce 32-bit or 64-bit floating point waves. The BITS value in S_info, described below, may be zero for some formats.

On Windows, SoundLoadWave uses the smallest Igor wave data type that preserves the number of bits in the audio. Igor doesn't have a 24-bit data type, so these values are stored in a 32-bit integer wave.

Output Variables

SoundLoadWave sets these output variables:

V_flag	Set to 1 if a sound is loaded and fits into available memory, 0 otherwise.
V_Error	Set if /Q is specified, V_Error is set to a non-zero error code if something went wrong or to zero on success. Negative returned codes are system-dependent, positive are Igor-defined errors. V_Error = 1 means there wasn't enough memory to load the (uncompressed) sound.
S_path	Set to the full file path of the loaded file, not including the file name.
S_fileName	Set to the name of the loaded file.
S_waveNames	Set to the name of loaded wave.
S_info	Information about the loaded sound.

If the sound file exists, SoundLoadWave sets the string variable S_info to:

```
"FILE:nameOfFile;FORMAT:soundFileFormat;CHANNELS:numChannels;CHANNEL_LAYOUT:ch
annelLayoutDescription;CHANNEL_ORDER:channelsList;BITS:numBits;SAMPLES:numSamp
les;RATE:samplesPerSec;"
```

The *soundFileFormat* and *channelLayoutDescription* values are text descriptions of the sound data in the file, and are written in the localized language. This information is available only on Macintosh and may or may not be present in a given sound file.

The *channelsList* value is a comma-separated list of channel names, always in English abbreviations, such as "L,R" or "L,R,C,LFE,Ls,Rs". The meaning of the abbrevations:

channelList Abbreviation	Channel or Speaker Names
L	Front Left
R	Front Right
C	Front Center
LFE	Low Frequency Effects
Ls	Left Surround (Back Left)
Rs	Right Surround (Back Right)
Lc	Left Center (Front Left of Center)
Rc	Right Center (Front Right of Center)
Cs	Center Surround (Back Center)
Lsd	Left Surround Direct (Side Left)
Rsd	Right Surround Direct (Side Right)
Ts	Top Center Surround (Top Center)
Vhl	Vertical Height Left (Top Front Left)
Vhc	Vertical Height Center (Top Front Center)
Vhr	Vertical Height Right (Top Front Right)
Rls	Rear Left Surround (Top Back Left)
Rcs	Rear Center Surround (Top Back Center)
Rrs	Rear Right Surround (Top Back Right)

Examples

```
// Display an Open File dialog and load the chosen file.
// Use file's name for wave, overwrite any pre-existing wave, print information to history
SoundLoadWave/O myDestWave

// SoundLoadWave stores following in S_Info and prints it to the history area
FILE:<file name>;FORMAT:MPEG Layer 3;CHANNELS:2;BITS:0;SAMPLES:524416;RATE:44100;

// Rename the wave to a cleaned up version of the file name
Rename myDestWave, $CleanupName(S_fileName,1)
```

See Also

SoundSaveWave, **PlaySound**

SoundSaveWave

SoundSaveWave [*flags*] *typeStr*, *waveName* [, *fileNameStr*]

The SoundSaveWave operation saves the named wave on disk as an Audio Interchange File Format (AIFF-C) or Microsoft WAVE sound file. AIFF-C is primarily used on Macintosh.

Parameters

typeStr must be either "AIFC" or "WAVE".

fileNameStr contains the name of the file in which the named wave is saved. If you omit fileNameStr , SoundSaveWave uses the wave name with the appropriate extension.

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with pathName, or the name of a file in the folder associated with pathName. If SoundSaveWave can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

Flags

/BITS=*bits* Controls the number of bits used for each sound sample written to the file.

Use /BITS=24 with a 32-bit integer wave to save 24-bit sound data capable of representing values from -8,388,608 to +8,388,607.

If you omit /BITS or use /BITS=0, SoundSaveWave uses the wave's data type and size to determine how many bits are written for each sound sample.

The /BITS flag was added in Igor Pro 9.00.

/I Presents a Save File dialog in which you can specify the file to be saved.

/O Overwrites the file if it already exists.

If you omit /O and the file exists, SoundSaveWave displays a Save File dialog.

/P=*pathName* Specifies the folder to store the file in. *pathName* is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\`". See **Symbolic Paths** on page II-22 for details.

/Q Suppresses the normal messages in the history area of the command window. At present nothing is written to the history even if /Q is omitted.

Details

The sound file is always an uncompressed AIFF-C or WAVE file, with as many channels as the wave contains columns.

The sound file format is determined by the wave's data type, typeStr, and the /BITS flag. Signed 8-, 16- and 24-bit integers are supported as are 32-bit and 64-bit floating point. When writing floating point waves, the wave data should be scaled to +/- 1.0 as full scale.

The SoundSaveWave operation was added in Igor Pro 7.00. 24-bit integer and 64-bit floating point support were added in Igor Pro 9.00.

Output Variables

SoundSaveWave sets these automatically created variables:

V_flag	Set to 0 if the wave was successfully saved to the file or to a non-zero error code.
S_fileName	Set to the name of the saved file.
S_path	Set to the full path to the file's directory.

Examples

```
// Create a simple sound (1000 Hz tone burst)
Make/O/N=10000 mySound                    // Single-precision wave, 10,000 values
SetScale/P x, 0, 1/8000, "" mySound       // 8000 Hz sampling frequency (1.25 seconds)
mySound= sin(2*pi*1000*x)                 // 1000 Hz tone
Hanning mySound                           // Fade in and out

// Save it to a file, chosen from the Save File dialog
SoundSaveWave "AIFC", mySound, "my sound.aif"

// Create a floating point stereo frequency sweep
Make/O/N=(20000,2) stereoSineSoundF32     // 32-bit float data
SetScale/P x,0,1e-4,stereoSineSoundF32    // Set sample rate to 10KHz
stereoSineSoundF32= sin(2*Pi*(1000 + (1-2*q)*150*x)*x)
NewPath sound                             // Create a symbolic path via dialog
SoundSaveWave/P=sound/O "WAVE", stereoSineSoundF32
```

See Also

SoundLoadWave, **PlaySound**, **WaveType**, **WaveInfo**

SpecialCharacterInfo

SpecialCharacterInfo(*notebookNameStr*, *specialCharacterNameStr*, *whichStr*)

The SpecialCharacterInfo function returns a string containing information about the named special character in the named notebook window.

Parameters

If *notebookNameStr* is "", the top visible notebook is used. Otherwise *notebookNameStr* contains either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-92 for details on host-child specifications.

specialCharacterNameStr is the name of a special character in the notebook.

If specialCharacterNameStr is "" and if exactly one special character is selected, the selected special character is used. If other than exactly one special character is selected, an error is returned.

whichStr identifies the information item you want. Because SpecialCharacterInfo can return several items that may contain semicolons, it does not return a semicolon-separated keyword-value list like other info functions. Instead it returns just one item as specified by *whichStr*.

Details

Here are the supported values for *whichStr*.

Keyword	Returned Information
NAME	The name of the special character.
FRAME	0: None 1: Single 2: Double 3: Triple 4: Shadow

Keyword	Returned Information
LOC	Paragraph and character position (e.g., 1,3).
SCALING	Horizontal and vertical scaling in units of one tenth of a percent (e.g., 1000,1000).
TYPE	Special character type is: Picture, Graph, Table, Layout, Action, ShortDate, LongDate, AbbreviatedDate, Time, Page, TotalPages, or WindowTitle.

These keywords apply to Igor-object pictures only. If the specified character is not an Igor-object picture, “” is returned.

Keyword	Returned Information
WINTYPE	1 for graphs, 2 for tables, 3 for layouts.
OBJECTNAME	The name of the window with which the special character is associated.

The remaining keywords apply to notebook action characters only. If the specified special character is not a notebook action character, `""` is returned.

Keyword	Returned Information
BGRGB	Background color in RGB format (e.g., 65535,65534,49151).
COMMANDS	Command string.
ENABLEBGRGB	1 if the action’s background color is enabled, 0 if not.
HELPTEXT	Help text string.
IGNOREERRORS	0 or 1.
LINKSTYLE	0 or 1.
PADDING	The value of the left, right, top, bottom and internal padding properties, in that order (.e.g, 4,4,4,4,8).
PICTURE	1 if the action has a picture, 0 if not.
PROCPICTNAME	The name of the action Proc Picture or `""` if none.
QUIET	0 or 1.
SHOWMODE	1: Title only 2: Picture only 3: Picture below title 4: Picture above title 5: Picture to the left of title 6: Picture to the right of title
TITLE	Title string.

If *whichStr* is an unknown keyword, SpecialCharacterInfo returns `""` but does not generate an error.

Examples

```
Function PrintSpecialCharacterInfo(notebookName, specialCharacterName)
   String notebookName, specialCharacterName

   String typeStr=SpecialCharacterInfo(notebookName, specialCharacterName, "TYPE")
   String locStr=SpecialCharacterInfo(notebookName, specialCharacterName, "LOC")

   Printf "TYPE: %s\r", typeStr
   Printf "LOC: %s\r", locStr
End
```

See Also

The **Notebook** and **NotebookAction** operations; the **SpecialCharacterList** function; **Using Igor-Object Pictures** on page III-18.

SpecialCharacterList

SpecialCharacterList(*notebookNameStr*, *separatorStr*, *mask*, *flags*)

The SpecialCharacterList function returns a string containing a list of names of special characters in a formatted text notebook.

Parameters

If *notebookNameStr* is "", the top visible notebook is used. Otherwise *notebookNameStr* contains either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-92 for details on host-child specifications.

separatorStr should contain a single ASCII character, usually semicolon, to separate the names.

mask determines which types of special characters are included. *mask* is a bitwise parameter with values:

1:	Pictures including graphs, tables and layouts.
2:	Notebook actions.
4:	All other special characters such as dates and times.

or a bitwise combination of the above for more than one type. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

flags is a bitwise parameter. Pass 0 to include all special characters or 1 to include only selected special characters. All other bits are reserved and should be passed as zero.

Details

Only formatted text notebooks have special characters. When called for a plain text notebook, SpecialCharacterList always returns "".

Examples

Print a list of all special characters in the top notebook:

```
Print SpecialCharacterList("", ";", -1, 0)
```

Prints a list of notebook action characters in Notebook0:

```
Print SpecialCharacterList("Notebook0", ";", 2, 0)
```

Print a list of selected notebook action characters in Notebook0:

```
Print SpecialCharacterList("Notebook0", ";", 2, 1)
```

See Also

The **Notebook** and **NotebookAction** operations; the **SpecialCharacterInfo** function.

SpecialDirPath

SpecialDirPath(*dirIDStr*, *domain*, *flags*, *createDir*)

The SpecialDirPath function returns a full path to a file system directory specified by *dirIDStr* and *domain*. It provides a programmer with a way to access directories of special interest, such as the preferences directory and the desktop directory.

The path returned always ends with a separator character which may be a colon, backslash, or forward slash depending on the operating system and the *flags* parameter.

SpecialDirPath depends on operating system behavior. The exact path returned depends on the locale, the operating system, the specific installation, the current user, and possibly other factors.

Parameters

dirIDStr is one of the following strings:

`"Packages"`	Place for advanced programmers to put preferences for their procedure packages.
`"Documents"`	The OS-defined place for users to put documents.
`"Preferences"`	The OS-defined place for applications to put preferences.
`"Desktop"`	The desktop.

`"Temporary"` The OS-defined place for applications to put temporary files.

`"Igor Application"` The Igor installation folder. This is typically:

`/Applications/Igor Pro X Folder (`*`Macintosh`*`)`

or

`C:\Program Files\WaveMetrics\Igor Pro X Folder (`*`Windows`*`)`

where “X” is the major version number.

Use only with domain = 0 (the current user).

`"Igor Executable"` The folder containing the current Igor executable. On Macintosh, this is the path to the executable itself, not to the application bundle.

Use only with domain = 0 (the current user).

Requires Igor Pro 7.00 or later.

`"Igor Preferences"` The folder in which Igor's own preference files are stored.

`"Igor Pro User Files"` A guaranteed-writable folder for the user to store their own Igor files, and to activate extensions, help, and procedure files by creating shortcuts or aliases in the appropriate subfolders. Use only with domain = 0 (the current user).

This is the folder opened using the Show Igor Pro User Files menu item in the Help menu.

domain permits discriminating between, for example, the preferences folder for all users versus the preferences folder for the current user. It is supported only for certain *dirIDStrs*. It is one of the following:

0: The current user (recommended value for most purposes).

1: All users (may generate an error or return the same path as 0).

2: System (may generate an error or return the same path as 1).

flags a bitwise parameter:

Bit 0: If set, the returned path is a native path (Macintosh-style on Mac OS 9, Unix-style on Mac OS X, Windows-style on Windows). If cleared, the returned path is a Macintosh-style path regardless of the current platform. In most cases you should set this bit to zero since Igor accepts Macintosh-style paths on all operating systems. You must set this bit to one if you are going to pass the path to an external script.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

createDir is 1 if you want the directory to be created if it does not exist or 0 if you do not want it to be created. This flag will not work if the current user does not have sufficient privileges to create the specified directory. In almost all cases it is not needed, you can’t count on it, and you should pass 0.

Details

The *domain* parameter has no effect in most cases. In almost all cases you should pass 0 (current user) for this parameter. For values other than 0, SpecialDirPath might return an error which you must be prepared to handle.

In the event of an error, SpecialDirPath returns a NULL string and sets a runtime error code. You can check for an error like this:

```
String fullPath = SpecialDirPath("Packages", 0, 0, 0)
Variable len = strlen(fullPath)          // strlen(NULL) returns NaN
if (numtype(len) == 2)                   // fullPath is NULL?
   Print "SpecialDirPath returned error."
endif
```

Here is sample output from `SpecialDirPath(“Packages”,0,0,0)`:

Mac OS X hd:Users:<*user*>:Library:Preferences:WaveMetrics:Igor Pro X:Packages:

Windows	C:Documents and Settings:<*user*>:Application Data:WaveMetrics:Igor Pro X:Packages:

where <*user*> is the name of the current user and "X" is the major version number..

Example

For an example using SpecialDirPath, see **Saving Package Preferences** on page IV-251.

sphericalBessJ

`sphericalBessJ(`*`n`*`, `*`x`*` [, `*`accuracy`*`])`

The sphericalBessJ function returns the spherical Bessel function of the first kind and order *n*.

$$j_n(x)=\sqrt{\frac{\pi}{2x}}J_{n+1/2}(x).$$

For example:

$$j_0(x)=\frac{\sin(x)}{x}$$

$$j_1(x)=\frac{\sin(x)}{x^2}-\frac{\cos(x)}{x}$$

$$j_2(x)=\left(\frac{3}{x^3}-\frac{1}{x}\right)\sin(x)-\frac{3}{x^2}\cos(x).$$

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **sphericalBessJD** and **sphericalBessY** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

sphericalBessJD

`sphericalBessJD(`*`n`*`, `*`x`*` [, `*`accuracy`*`])`

The sphericalBessJD function returns the derivative of the spherical Bessel function of the first kind and order *n*.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **sphericalBessJ** and **sphericalBessY** functions.

sphericalBessY

`sphericalBessY(`*`n`*`, `*`x`*` [, `*`accuracy`*`])`

The sphericalBessY function returns the spherical Bessel function of the second kind and order *n*.

$$y_n(x)=\sqrt{\frac{\pi}{2x}}Y_{n+1/2}(x).$$

$$y_0(x) = -\frac{\cos(x)}{x}$$

$$y_1(x) = -\frac{\cos(x)}{x^2} - \frac{\sin(x)}{x}$$

$$y_2(x) = \left(\frac{1}{x} - \frac{3}{x^3}\right)\cos(x) - \frac{3}{x^2}\sin(x).$$

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **sphericalBessYD** and **sphericalBessJ** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

sphericalBessYD

sphericalBessYD(*n*, *x* [, *accuracy*])

The sphericalBessYD function returns the derivative of the spherical Bessel function of the second kind and order *n*.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **sphericalBessJ** and **sphericalBessY** functions.

sphericalHarmonics

sphericalHarmonics(*L*, *M*, *q*, *f*)

The sphericalHarmonics function returns the complex-valued spherical harmonics

$$Y_L^M(\theta,\phi) = (-1)^M \sqrt{\frac{2L+1}{4\pi}\frac{(L-M)!}{(L+M)!}} P_L^M(\cos\theta) e^{iM\phi}$$

$$Y_L^M(\theta,\phi) = (-1)^M \sqrt{\frac{2L+1}{4\pi}\frac{(L-M)!}{(L+M)!}} P_L^M(\cos(\theta)) e^{iM\phi},$$

where $P_L^M(\cos(\theta))$ is the associated Legendre function.

See Also

The **legendreA** function. The NumericalIntegrationDemo.pxp experiment.

Demos

Choose File→Example Experiments→Visualization→SphericalHarmonicsDemo.

Choose File→Example Experiments→Analysis→NumericalIntegrationDemo.

References

Arfken, G., *Mathematical Methods for Physicists*, Academic Press, New York, 1985.

SphericalInterpolate

SphericalInterpolate ***triangulationDataWave*, *dataPointsWave*, *newLocationsWave***

The SphericalInterpolate operation works in conjunction with the SphericalTriangulate operation to calculate interpolated values on a surface of a sphere. Given a set of {x_i, y_i, z_i} points on the surface of a sphere with their associated values {v_i}, the SphericalTriangulate operation performs the Delaunay triangulation and creates an output that is used by the SphericalInterpolate operation to calculate values at any other point on the surface of a sphere. The interpolation calculation uses Voronoi polygons to weigh the contribution of the nearest neighbors to any given location on the sphere.

Parameters

triangulationDataWave is a 13 column wave that was created by the SphericalTriangulate operation.

dataPoints is a 4 column wave. The first 3 columns are the {x_i, y_i, z_i} locations that were used to create the triangulation, and the last column corresponds to the {v_i} values at the triangulation locations.

newLocationsWave is a 3 column wave that specifies the x, y, z locations on the sphere at which the interpolated values are calculated. Note that internally, each triplet is normalized to a point on the unit sphere before it is used in the interpolation.

Details

You will always need to use the SphericalTriangulate operation first to generate the *triangulationDataWave* input for this operation.

The result of the operation are put in the wave W_SphericalInterpolation.

See Also

SphericalTriangulate, **Triangulate3D**, **ImageInterpolate** with keyword Voronoi

Demo

Choose File→Example Experiments→Analysis→SphericalTriangulationDemo.

SphericalTriangulate

SphericalTriangulate [**/Z**] ***tripletWaveName***

The SphericalTriangulate operation triangulates an arbitrary XYZ triplet wave on a surface of a sphere.

It starts by normalizing the data to make sure that sqrt(x^2+y^2+z^2)=1, and then proceeds to calculate the Delaunay triangulation.

Flags

/Z	No error reporting.

Details

The result of the triangulation is the wave M_SphericalTriangulation. This 13 column wave is used in SphericalInterpolate to obtain the interpolated values.

Example

```
// Generates output waves that can be used in Gizmo to display the triangulation.
// triangulationData is the M_TriangulationData output from SphericalTriangulation.
// tripletWave is the source wave input to SphericalTriangulation.
// Output wave sphereTrianglesPath can be used to display the triangulation as a path.
// Output wave sphereTrianglesSurf can be used to display the triangulation as a surface.
Function BuildTriangleWaves(triangulationData,tripletWave)
    Wave triangulationData, tripletWave

    // Extract 3 columns from triangulationData that contain the index of the row.
    Duplicate/O/FREE/r=[][1,3] triangulationData,triIndices
    Variable finalNumTriangles=dimSize(triIndices,0),i,j,k

    // Initialize both waves to NaN so any unassigned point would appear as a hole.
    Make/O/N=(5*finalNumTriangles,3) sphereTrianglesPath=NaN
    Make/O/N=(3*finalNumTriangles,3) sphereTrianglesSurf=NaN

    // Assign the values of the vertices to the two waves:
    Variable rowIndex,rowIndex0,outRowCount=0,outcount2=0
    for(i=1;i<finalNumTriangles;i+=1)
```

```
        for(j=0;j<3;j+=1)
            rowIndex=triIndices[i][j]
            for(k=0;k<3;k+=1)
                sphereTrianglesPath[outRowCount][k]=tripletWave[rowIndex][k]
                sphereTrianglesSurf[outcount2][k]=tripletWave[rowIndex][k]
            endfor
            outRowCount+=1
            outcount2+=1
        endfor

        // Close the triangle path by returning to the first vertex:
        rowIndex0=triIndices[i][0]
        sphereTrianglesPath[outRowCount][0]=tripletWave[rowIndex0][0]
        sphereTrianglesPath[outRowCount][1]=tripletWave[rowIndex0][1]
        sphereTrianglesPath[outRowCount][2]=tripletWave[rowIndex0][2]
        outRowCount+=2          // Increment row count and skip the NaN
    endfor
End
```

See Also

SphericalInterpolate, **Triangulate3D**, **ImageInterpolate** with keyword Voronoi

Demo

Choose File→Example Experiments→Analysis→SphericalTriangulationDemo.

SplitString

`SplitString /E=`*`regExprStr str`* `[,` ***`substring1`*** `[,` ***`substring2`*`,…`*`substringN`***`]]`

The SplitString operation uses the regular expression *regExprStr* to split *str* into subpatterns. See **Subpatterns** on page IV-186 for details. Each matched subpattern is returned sequentially in the corresponding substring parameter.

Parameters

str is the input string to be split into subpatterns.

The *substring1*…*substringN* output parameters must be the names of existing string variables if you need to use the matched subpatterns. The first matched subpattern is returned in *substring1*, the second in *substring2*, etc.

Flags

/E=*regExprStr*	Specifies the Perl-compatible regular expression string containing subpattern definition(s).

Details

regExprStr is a regular expression with successive subpattern definitions, such as shown in the examples. (Subpatterns are regular expressions within parentheses.)

For unmatched subpatterns, the corresponding substring is set to `""`. If you specify more substring parameters than subpatterns, the extra parameters are also set to `""`.

The number of matched subpatterns is returned in V_flag.

The part of *str* that matches *regExprStr* (often all of *str*) is stored in S_value.

Examples

```
// Split the output of the date() function:
Print date()
  Mon, May 2, 2005

String expr="([[:alpha:]]+), ([[:alpha:]]+) ([[:digit:]]+), ([[:digit:]]+)"
String dayOfWeek, monthName, dayNumStr, yearStr
SplitString/E=(expr) date(), dayOfWeek, monthName, dayNumStr, yearStr
Print V_flag
  4
Print dayOfWeek
  Mon
Print monthName
  May
Print dayNumStr
  2
Print yearStr
```

```
  2005
Print S_value
  Mon, May 2, 2005

// Get the part of str that matches regExprStr
SplitString/E=",.*," "stuff in front,second value,stuff at end"
Print S_value
  ,second value,
```

See Also

Regular Expressions on page IV-176 and **Subpatterns** on page IV-186.

sscanf, **Grep**, **strsearch**, **str2num**, **RemoveEnding**, **TrimString**

SplitWave

`SplitWave [`*`flags`*`]` ***`srcWave`***

The SplitWave operation creates new waves containing subsets of the data in *srcWave* which must be 2D or greater.

The newly generated waves have lower dimensionality than *srcWave*. The operation is ideal for splitting 2D waves into constituent columns, 3D waves into their layers, etc.

Added in Igor Pro 7.00.

Flags

/DDF=*destDataFolder* Specifies the data folder where the generated waves are created. If the data folder does not exist the operation creates it. If the /DDF flag is not used, output goes into the current data folder.

/FREE Generates free output waves. The /OREF flag must also be used when the /FREE flag is used. When you use this flag there is no need to use either /N or /NAME.

/N=*baseName* Provides the base name for all output waves. The waves will be named sequentially, i.e., baseName0, baseName1...

/NAME=*strList* *strList* is a semicolon-separated list of wave names to be used as the names of the output waves.

If *strList* contains fewer names than the number needed, the operation terminates and returns an error.

If the output data folder is the data folder containing *srcWave* then *strList* must not contain the name of *srcWave*.

Only simple names, not full paths, are allowed in *strList*.

/NOTE Copies the wave note, if any, from *srcWave* to the output waves. The /NOTE flag was added in Igor Pro 8.00.

/O Permits overwriting of existing destination waves. Overwriting *srcWave* is not permitted.

/OREF=*waveRefWave* *waveRefWave* is a wave reference wave. SplitWave stores a wave reference for each of the output waves in *waveRefWave*.

If the specified *waveRefWave* already exists it is overwritten and its size is changed as appropriate. If it does not already exist, it is created by the operation.

/SDIM=*n* Specifies the dimensionality of the output waves. By default this is 1 less than the dimensionality of *srcWave*. The minimum value is *n*=1 which results in 1D output waves.

/Z[=z] /Z or /Z=1 prevents procedure execution from aborting if there is an error. Use /Z if you want to handle this case in your procedures rather than having execution abort.

/Z=0: Same as no /Z at all. This is the default.

/Z=1: Same as /Z alone.

Details

The SplitWave operation is in some ways the inverse of the Concatenate operation. *srcWave* is decomposed into waves of lower dimensionality.

Splitting a 2D 10x15 wave results in 15 waves of 10 rows each.

Splitting a 3D 10x15x4 using /SDIM=2 results in 4 2D waves of dimension 10x15.

Splitting a 3D 10x15x4 using /SDIM=1 results in 60 1D waves of 10 rows each.

The SplitWave operation works on all wave types. *srcWave* must be 2D or greater.

The operation creates the string variable S_waveNames which contains a semicolon separated list of the names of the output waves. However if you use /FREE then S_waveNames will be empty as free waves can not be accessed by name; use /OREF to access the created waves.

Examples

```
// Create sample input
Make/N=(5,4,3,2) wave1 = p + 10*q + 100*r + 1000*s

// Split chunks into 2 3D waves and store them in data folder Chunks
SplitWave/DDF=Chunks/N=chunk wave1

// Split layers into 6 2D waves and store them in data folder Layers
SplitWave/DDF=Layers/N=Layers/SDIM=2 wave1

// Split into 24 1D waves and store them in data folder Columns
SplitWave/DDF=Columns/N=Columns/SDIM=1 wave1
```

See Also

Duplicate, **Redimension**, **Concatenate**

sprintf

sprintf *stringName*, *formatStr* [**,** ***parameter***]…

The sprintf operation is the same as printf except it prints the formatted output to the string variable *stringName* rather than to the history area.

Parameters

formatStr	See printf.
parameter	See printf.
stringName	Results are “printed” into the named string variable.

See Also

The **printf** operation for complete format and parameter descriptions and **Creating Formatted Text** on page IV-259.

sqrt

sqrt(*num*)

The sqrt function returns the square root of *num* or NaN if *num* is negative.

In complex expressions, *num* is complex, and sqrt(*num*) returns the complex value x + *i*y.

sscanf

sscanf *scanStr*, *formatStr*, *var* [**,** ***var***]

The sscanf operation is useful for parsing text that contains numeric or string data. It is based on the C sscanf function and provides a subset of the features available in C.

Here is a trivial example:

```
Variable v1
sscanf "Value= 1.234", "Value= %f", v1
```

This skips the text “Value=” and the following space and then converts the text “1.234” (or whatever number appeared there) into a number and stores it in the local variable v1.

The sscanf operation sets the variable V_flag to the number of values read. You can use this as an initial check to see if the *scanStr* is consistent with your expectations.

Note: The sscanf operation is supported in user functions only. It is not available using the command line, using a macro, or using the Execute operation.

Parameters

scanStr contains the text to be parsed.

formatStr is a format string which describes how the parsing is to be done.

formatStr is followed by the names of one or more local numeric or string variables or NVARs (references to global numeric variables) or SVARs (references to global string variables), which are represented by *var* above.

sscanf can handle a maximum of 100 *var* parameters.

Details

The format string consists of the following:

- Normal text, which is anything other than a percent sign ("%") or white space.
- White space (spaces, tabs, linefeeds, carriage returns).
- A percent ("%") character, which is the start of a conversion specification.

The trivial example illustrates all three of these components.

```
Variable v1
sscanf "Value= 1.234", "Value= %f", v1
```

sscanf attempts to match normal text in the format string to the identical normal text in the scan string. In the example, the text "Value=" in the format string skips the identical text in the scan string.

sscanf matches a single white space character in the format string to 0 or more white space characters in the scan string. In the example, the single space skips the single space in the scan string.

When sscanf encounters a percent character in the format string, it attempts to convert the corresponding text in the scan string into a number or string, depending on the conversion character following the percent, and stores the resulting number or string in the corresponding variable in the parameter list. In the example, "%f" converts the text "1.234" into a number which it stores in the local variable v1.

A conversion specification consists of:

- A percent character ("%").
- An optional "*", which is a conversion suppression character.
- An optional number, which is a maximum field width.
- A conversion character, which specifies how to interpret text in the scan string.

Don't worry about the suppression character and the maximum width specification for now. They will be explained later.

The sscanf operation supports a subset of the conversion characters supported by the C sscanf operation. The supported conversion characters, which are case-sensitive, are:

d	Converts text representing a decimal number into an integer numeric value.
i	Converts text representing a decimal, octal or hexadecimal number into an integer value. If the text starts with "0x" (zero-x), it is interpreted as hexadecimal. Otherwise, if it starts with "0" (zero), it is interpreted as octal. Otherwise it is interpreted as decimal.
o	Converts text representing an octal number into an integer numeric value.
u	Converts text representing an unsigned decimal number into an integer numeric value.
x	Converts text representing a hexadecimal number into an integer numeric value.
c	Converts a single character into an integer value which is the ASCII code representing that character.
e	Converts text representing a decimal number into a floating point numeric value.
f	Same as e.

g	Same as e.
s	Stores text up to the next white space into a string.
[	Stores text that matches a list of specific characters into a string. The list consists of the characters inside the brackets ("%[abc]"). If the first character is "^", this means to match any character that is **not** in the list. You can specify a range of characters to match. For example `"%[A-Z]"` matches all of the upper case letters and `"%[A-Za-z]"` matches all of the upper and lower case letters.

Here are some simplified examples to illustrate each of these conversions.

```
Variable v1
String s1
```

Convert text representing a decimal number to an integer value:

```
sscanf "1234", "%d", v1
```

Convert text representing a decimal, octal, or hexadecimal number:

```
sscanf "1.234", "%i", v1          // Convert from decimal.
sscanf "01234", "%i", v1          // Convert from octal.
sscanf "0x123", "%i", v1          // Convert from hex.
```

Convert text representing an octal number:

```
sscanf "1234", "%o", v1
```

Convert text representing an unsigned decimal number:

```
sscanf "1234", "%u", v1
```

Convert text representing a hexadecimal number:

```
sscanf "1FB9", "%x", v1
```

Convert a single ASCII character:

```
sscanf "A", "%c", v1
```

Convert text representing a decimal number to an floating point value:

```
sscanf "1.234", "%e", v1
sscanf "1.234", "%f", v1
sscanf "1.234", "%g", v1
```

Copy a string of text up to the first white space:

```
sscanf "Hello There", "%s", s1
```

Copy a string of text matching the specified characters:

```
sscanf "+4.27", "%[+-]", s1
```

In a C program, you will sometimes see the letters "l" (ell) or "h" between the percent and the conversion character. For example, you may see "%lf" or "%hd". These extra letters are not needed or tolerated by Igor's sscanf operation.

When sscanf matches the format string to the scan string, it reads from the scan string until a character that would be inappropriate for the section of the format string that sscanf is trying to match. In the following example, sscanf stops reading characters to be converted into a number when it hits the first character that is not appropriate for a number.

```
Variable v1
String s1, s2
sscanf "1234Volts DC", "%d%s %s", v1, s1, s2
```

sscanf stops matching text for "%d" when it hits "V" and stores the converted number in v1. It stops matching text for the first "%s" when it hits white space and stores the matched text in s1. It then skips the space in the scan string because of the corresponding space in the format string. Finally, it matches the remaining text to the second "%s" and stores the text in s2.

The maximum field width must appear just before the conversion character ("d" in this case).

```
Variable v1, v2
sscanf "12349876", "%4d%4d", v1, v2
```

The suppression character ("*") is used in a conversion specification to skip values in the scan string. It parses the value, but sscanf does not store the value in any variable. In the following example, we read one number into local variable v1, skip a colon, and read another number into local variable v2, skip a colon, and read another number into local variable v3.

```
Variable v1, v2, v3
sscanf "12:30:45", "%d%*[:]%d%*[:]%d", v1, v2, v3
```

Here "%*[:]" means "read a colon character but don't store it anywhere". The "*" character must appear immediately after the percent. Note that there is nothing in the parameter list corresponding to the suppressed strings.

If the text in the scan string is not consistent with the text in the format string, sscanf may not read all of the values that you expected. You can check for this using the V_flag variable, which is set to the number of values read. This kind of inconsistency does not cause sscanf to return an error to Igor, which would cause procedure execution to abort. It is a situation that you can deal with in your procedure code.

The sscanf operation returns the following kinds of errors:

- Out-of-memory.
- The number of parameters implied by *formatStr* does not match the number of parameters in the *var* list.
- *formatStr* calls for a numeric variable but the parameter list expects a string variable.
- *formatStr* calls for a string variable but the parameter list expects a numeric variable.
- *formatStr* includes an unsupported, unknown or incorrectly constructed conversion specification.
- The *var* list references a global variable that does not exist.

Examples

Here is a simple example to give you the general idea:

```
Function SimpleExample()
   Variable v1, valuesRead
   sscanf "Value=1.234", "Value=%g", v1
   valuesRead = V_flag
   if (valuesRead != 1)
      Printf "Error: Expected 1 value, got %d values\r", valuesRead
   else
      Printf "Value read = %g\r", v1
   endif
End
```

For an example that uses sscanf to load data from a text file, choose File→Example Experiments→Programming→Load File Demo.

See Also

str2num, **strsearch**, **StringMatch**, **SplitString**

Stack

`Stack` [***`flags`***] [***`objectName`***][, ***`objectName`***]…

The Stack operation stacks the named layout objects in the top page layout.

Parameters

objectName is the name of a graph, table, picture or annotation object in the top page layout.

See Also

The **Tile** operation for details on the flags and parameters.

StackWindows

`StackWindows` [***`flags`***] [***`windowName`*** [, ***`windowName`***]…]

The StackWindows operation stacks the named windows on the desktop.

See Also

See the **TileWindows** operation for details on the flags and parameters.

StartMSTimer

`StartMSTimer`

The StartMSTimer function creates a new microsecond timer and returns a timer reference number.

Details

You can create up to ten different microsecond timers using StartMSTimer. A valid timer reference number is a number between 0 and 9. If StartMSTimer returns -1, there are no free timers available. StartMSTimer works in conjunction with StopMSTimer.

See Also

StopMSTimer, **ticks**, **DateTime**

Static

`Static constant` ***`objectName = value`***

`Static strconstant` ***`objectName = value`***

`Static Function` ***`funcName`*****`()`**

`Static Structure` ***`structureName`***

`Static Picture` ***`pictName`***

The Static keyword specifies that a constant, user-defined function, structure, or Proc Picture is local to the procedure file in which it appears. Static objects can only be used by other functions; they cannot be accessed from macros; they cannot be accessed from other procedure files or from the command line.

See Also

Static Functions on page IV-105, **Proc Pictures** on page IV-56, and **Constants** on page IV-51.

StatsAngularDistanceTest

`StatsAngularDistanceTest` `[`***`flags`***`][`***`srcWave1, srcWave2, srcWave3`***`…]`

The StatsAngularDistanceTest operation performs nonparametric tests on the angular distance between sample data and reference directions for two or more samples in individual waves. The angular distance is the shortest distance between two points on a circle (in radians). Specify the sample waves using /WSTR or by listing them following the flags. Set reference directions with /ANG, /ANGW, or the sample mean direction.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/ANG={*d1*, *d2*}	Sets reference directions (in radians) for two samples; for more than two samples use /ANGW.
/ANGM	Computes the mean direction of each sample and uses it as the reference direction.
/ANGW=*dWave*	Sets reference directions (in radians) for more than two samples using directions in *dWave*, which must be single or double precision.
/APRX=*m*	Controls the approximation method for computing the P-value in the case of two samples (Mann-Whitney Wilcoxon). See **StatsWilcoxonRankTest** for more details. The default value is 0, which may require long computation times if your sample size is large. Use /APRX=1 if you have a large sample and you expect ties in the data.
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.

/TAIL=*tail* — *tail* is a bitwise parameter that specifies the tails tested.

Bit 0: Lower tail.
Bit 1: Upper tail (default).
Bit 2: Two tail.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The P value corresponding to the last tail calculated will be entered in the table.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z — Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The inputs for StatsAngularDistanceTest are two or more waves each corresponding to individual sample. The waves must be single or double precision expressing the angles in radians. There is no restriction on the number of points or dimensionality of the waves but the data should not contain NaNs or INFs. We recommend that you use double precision waves, especially if there are ties in the data. The reference directions should also be in radians. For two samples, StatsAngularDistanceTest computes the angular distances between the input data and the reference directions and then uses the Mann-Whitney-Wilcoxon test (**StatsWilcoxonRankTest**). Results are stored in the W_WilcoxonTest wave and in the corresponding table. For more than two samples, StatsAngularDistanceTest uses the Kruskal-Wallis test, storing results in the wave W_KWTestResults wave in the current data folder.

V_flag will be set to -1 for any error and to zero otherwise.

References

See, in particular, Chapter 27 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWilcoxonRankTest** and **StatsKWTest**.

Examples:Statistics:Circular Statistics:AngularDistanceTest.pxp.

StatsANOVA1Test

`StatsANOVA1Test` `[`***`flags`***`] [`***`wave1, wave2,… wave100`***`]`

The StatsANOVA1Test operation performs a one-way ANOVA test (fixed-effect model). The standard ANOVA test results are stored in the M_ANOVA1 wave in the current data folder.

Flags

/ALPH=*val* — Sets the significance level (default 0.05).

/BF — Performs the Brown and Forsythe test computing F'' and degrees of freedom. The W_ANOVA1BnF wave in the current data folder contains the output.

/Q — No results printed in the history area.

/T=*k* — Displays results in a table; additional tables are created with /BF and /W.

k specifies the table behavior when it is closed.

k=0: Normal with dialog (default).
k=1: Kills with no dialog.
k=2: Disables killing.

/W — Performs the Welch test F' and computes degrees of freedom. The W_ANOVA1Welch wave in the current data folder contains the output.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Inputs to StatsANOVA1Test are two or more 1D numerical waves containing (one wave for each group of samples). Use NaN for missing entries or use waves with different numbers of points. The standard ANOVA results are in the M_ANOVA1 wave with corresponding row and column labels. Use /T to display the results in a table. In each case you will get the two degrees of freedom values, the F value, the critical value Fc for the choice of alpha and the degrees of freedom, and the P-value for the result. V_flag will be set to -1 for any error and to zero otherwise.

In some cases the ANOVA test may not be appropriate. For example, if groups do not exhibit sufficient homogeneity of variances. Although this may not be fatal for the ANOVA test, you may get more insight by performing the variances test in **StatsVariancesTest**.

If there are only two groups this test should be equivalent to **StatsTTest**.

You can evaluate the power of an ANOVA test for a given set of degrees of freedom and noncentrality parameter using:

```
power=1-StatsNCFCDF(StatsInvFCDF((1-alpha),n1,n2),n1,n2,delta)
```

Here n1 is the Groups' degrees of freedom, n2 is the Error degrees of freedom, and delta is the noncentrality parameter. For more information see ANOVA Power Calculations Panel and the associated example experiment.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsVariancesTest**, **StatsTTest**, **StatsNCFCDF**, and **StatsInvFCDF**.

StatsANOVA2NRTest

StatsANOVA2NRTest [***flags***] ***srcWave***

The StatsANOVA2NRTest operation performs a two-factor analysis of variance (ANOVA) on the data that has no replication where there is only a single datum for every factor level. *srcWave* is a 2D wave of any numeric type. Output is to the M_ANOVA2NRResults wave in the current data folder or optionally to a table.

Flags

/ALPH=*val* Sets the significance level (default 0.05).

/FOMD Estimates one missing value. You will also have to use a single or double precision wave for *srcWave* and designate the single missing value as NaN. The estimated value is printed to the history as well as the bias used to correct the sum of the squares of factor A.

/INT=*val* Sets the degree of interactivity.

Sets the degree of interactivity.

val=0: No interaction between the factors (default).

val=1: Significant interaction effect between factors.

Combination with /MODL determines which factors to test:

val	Model 1	Model 2	Model 3
1		A&B	A
0	A&B	A&B	A&B

As indicated in the table, factor B is not tested for significant interaction under Model 3 and neither factor A nor factor B are tested for Model 1. If you are willing to accept an increase in Type II error you can obtain the relevant values by specifying Model 2. None of the models support a test for interaction A x B.

/MODL=*m* Sets the model number.

m=1:	Factor A and factor B are fixed.
m=2:	Both factors are random.
m=3:	Factor A is fixed and factor B is random (default).

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Input to StatsANOVA2NRTest is a 2D wave in which the Factor A corresponds to rows and Factor B corresponds to columns. H0 provides that there is no difference in the means of the respective populations, i.e., if H0 is rejected for Factor A but accepted for Factor B that means that there is no difference in the means of the columns but the means of the rows are different.

NaN and INF entries are not supported although you may use a single NaN value in combination with the /FOMD flag. If srcWave contains dimension labels they will be used to designate the two factors in the output.

The contents of the M_ANOVA2NRResults output wave columns are as follows:

Column 0	Sum of the squares (SS) values
Column 1	Degrees of freedom (DF)
Column 2	Mean square (MS) values
Column 3	Computed F value for this test
Column 4	Critical F value (Fc) for the specified alpha
Column 5	Conclusion with 0 to reject H0 or 1 to accept it

The variable V_flag is set to zero if the operation succeeds or to -1 otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test** and **StatsANOVA2Test**.

StatsANOVA2RMTest

`StatsANOVA2RMTest [flags] srcWave`

The StatsANOVA2RMTest operation performs analysis of variance (ANOVA) on *srcWave* where replicates consist of multiple measurements on the same subject (repeated measures). *srcWave* is a 2D wave of any numeric type. Output is to the M_ANOVA2RMResults wave in the current data folder or optionally to a table.

Flags

/ALPH=*val* Sets the significance level (default 0.05).

/Q No results printed in the history area.

/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Input to StatsANOVA2RMTest is the 2D *srcWave* in which the factor A (Groups) are columns and the different subjects are rows. It does not support NaNs or INFs.

The contents of the M_ANOVA2RMResults output wave columns are: the first contains the sum of the squares (SS) values, the second contains the degrees of freedom (DF), the third contains the mean square (MS) values, the fourth contains the single F value for this test, the fifth contains the critical F value for the specified alpha and degrees of freedom, and the last column contains the conclusion with 0 to reject H_0 or 1 to accept it. In each case H_0 corresponds to the mean level, which is the same for all subjects.

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA2NRTest** and **StatsANOVA2Test**.

StatsANOVA2Test

StatsANOVA2Test [***flags***] ***srcWave***

The StatsANOVA2Test operation performs a two-factor analysis of variance (ANOVA) on *srcWave*. Output is to the M_ANOVA2Results wave in the current data folder or optionally to a table.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/FAKE=*num*	Specifies the number of points in *srcWave* obtained by “estimation”. *num* is subtracted from the total and error degrees of freedom.
/MODL=*m*	Sets the model number. *m*=1: Factor A and factor B are fixed. *m*=2: Both factors are random. *m*=3: Factor A is fixed and factor B is random (default).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Input to StatsANOVA2Test is the single or double precision 3D *srcWave* in which the factor A levels are columns, the factor B levels are rows, and the replicates are layers. If *srcWave* contains dimension labels they will be used to designate the factors in the output.

Ideally, the number of replicates must be equal for each factor and each level. StatsANOVA2Test supports both equal replication and proportional replication. Proportional replication allows for different number of data in each cell with missing data represented as NaN and the number of points in each cell is given by

```
Nij=(sum of data in row i)*(sum of data in column j)/number of samples.
```

If you have no replicates (a single datum per cell) use **StatsANOVA2NRTest** instead. If the number of replicates in your data does not satisfy these conditions you may be able to "estimate" additional replicates using various methods. In that case use the /FAKE flag so that the operation can account for the estimated data by reducing the total and error degrees of freedom. /FAKE only accounts for the number of estimates being used. You must provide an appropriate number of estimated values.

The contents of the M_ANOVA2Results output wave columns are: the first contains the sum of the squares (SS) values, the second the degrees of freedom (DF), the third contains the mean square (MS) values, the fourth contains the computed F value for this test, the fifth contains the critical Fc value for the specified alpha and degrees of freedom, and the last contains the conclusion with 0 to reject H_0 or 1 to accept it. In each case H_0 corresponds to the mean level, which is the same for all populations.

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test** and **StatsANOVA2NRTest**.

StatsBetaCDF

StatsBetaCDF(*x*, *p*, *q* [, *a*, *b*])

The StatsBetaCDF function returns the beta cumulative distribution function

$$\mathrm{F(x,p,q,a,b)}=\frac{1}{\mathrm{B(p,q)}}\int_0^{\frac{x-a}{b-a}} t^{p-1}(1-t)^{q-1}\,dt, \qquad \begin{array}{c} p,q>0 \\ a\le x\le b \end{array}$$

where $B(p,q)$ is the beta function

$$B(p,q)=\int_0^1 t^{p-1}(1-t)^{q-1}\,dt.$$

The defaults (a=0 and b=1) correspond to the standard beta distribution were a is the location parameter, (b-a) is the scale parameter, and p and q are shape parameters.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBetaPDF** and **StatsInvBetaCDF**.

StatsBetaPDF

StatsBetaPDF(*x*, *p*, *q* [, *a*, *b*])

The StatsBetaPDF function returns the beta probability distribution function

$$f(x;p,q,a,b)=\frac{\left(x-a\right)^{p-1}\left(b-x\right)^{q-1}}{B\left(p,q\right)\left(b-a\right)^{p+q-1}}, \qquad \begin{array}{c} a\le x\le b \\ p,q>0 \end{array}$$

where $B(p,q)$ is the beta function

$$B(p,q)=\int_0^1 t^{p-1}(1-t)^{q-1}\,dt.$$

The defaults (*a*=0 and *b*=1) correspond to the standard beta distribution were *a* is the location parameter, (*b*-*a*) is the scale parameter, and *p* and *q* are shape parameters. When *p*<1, *f*(*x*=*a*) returns Inf.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBetaCDF** and **StatsInvBetaCDF**.

StatsBinomialCDF

```
StatsBinomialCDF(x, p, N)
```

The StatsBinomialCDF function returns the binomial cumulative distribution function

$$F(x;p,N)=\sum_{i=1}^{x}\binom{N}{i}p^{i}(1-p)^{N-i}, \qquad x=1,2,...$$

where

$$\binom{N}{i}=\frac{N!}{i!(N-i)!}.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBinomialCDF** and **StatsBinomialPDF**.

StatsBinomialPDF

```
StatsBinomialPDF(x, p, N)
```

The StatsBinomialPDF function returns the binomial probability distribution function

$$f(x;p,N)=\binom{N}{x}p^{x}(1-p)^{N-x}, \qquad x=0,1,2,...$$

where

$$\binom{N}{x}=\frac{N!}{x!(N-x)!}.$$

is the probability of obtaining *x* good outcomes in *N* trials where the probability of a single successful outcome is *p*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBinomialCDF** and **StatsInvBinomialCDF**.

StatsCauchyCDF

```
StatsCauchyCDF(x, μ, σ)
```

The StatsCauchyCDF function returns the Cauchy-Lorentz cumulative distribution function

$$F(x;\mu,\sigma)=\frac{1}{2}+\frac{1}{\pi}\tan^{-1}\left(\frac{x-\mu}{\sigma}\right).$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCauchyCDF** and **StatsCauchyPDF**.

StatsCauchyPDF

`StatsCauchyPDF(`*`x`*`, μ, σ)`

The StatsCauchyPDF function returns the Cauchy-Lorentz probability distribution function

$$f(x;\mu,\sigma)=\frac{1}{\sigma\pi}\frac{1}{1+\left(\frac{x-\mu}{\sigma}\right)^2},$$

where μ is the location parameter and σ is the scale parameter. Use μ=0 and σ=1 for the standard form of the Cauchy-Lorentz distribution.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCauchyCDF** and **StatsInvCauchyCDF**.

StatsChiCDF

`StatsChiCDF(`*`x`*`, `*`n`*`)`

The StatsChiCDF function returns the chi-squared cumulative distribution function for the specified value and degrees of freedom *n*.

$$F(x;n)=\frac{\gamma\left(\frac{n}{2},\frac{x}{2}\right)}{\Gamma\left(\frac{n}{2}\right)}.$$

where is γ(*a*,*b*) the incomplete gamma function. The distribution can also be expressed as

$$F(x;n)=1-gammq\left(\frac{n}{2},\frac{x}{2}\right).$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsChiPDF**, **StatsInvChiCDF**, and **gammq**.

StatsChiPDF

`StatsChiPDF(`*`x`*`, `*`n`*`)`

The StatsChiPDF function returns the chi-squared probability distribution function for the specified value and degrees of freedom as

$$f(x;n)=\frac{\exp\left(-\frac{x}{2}\right)x^{\frac{n}{2}-1}}{2^{\frac{n}{2}}\Gamma\left(\frac{n}{2}\right)}.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsChiCDF** and **StatsChiPDF**.

StatsChiTest

`StatsChiTest [`*`flags`*`] `*`srcWave1`*`, `*`srcWave2`*

The StatsChiTest operation computes a χ^2 statistic for comparing two distributions or a χ^2 statistic for comparing a sample distribution with its expected values. In both cases the comparison is made on a bin-by-bin basis. Output is to the W_StatsChiTest wave in the current data folder or optionally to a table.

Flags

/ALZR Allows zero entries in source waves. If you are using /S zero entries in *srcWave2* are skipped.

/NCON=*nCon* Specifies the number of constraints (0 by default), which reduces the number degrees of freedom and the critical value by *nCon*.

/S Sets the calculation mode to a single distribution where *srcWave1* represents an array of binned measurements and *srcWave2* represents the corresponding expected values.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The source waves, *srcWave1* and *srcWave2*, must have the same number of points and can be any real numeric data type. Any nonpositive values (including NaN) in either wave removes the entry in both waves from consideration and reduces the degrees of freedom by one. The number degrees of freedom is initially the number of points in *srcWave1*-1-*nCon*. By default it is assumed that *srcWave1* and *srcWave2* represent two distributions of binned data.

When you specify /S, *srcWave1* must consist of binned values of measured data and *srcWave2* must contain the corresponding expected values. The calculation is:

$$\chi^2 = \sum_{i=0}^{n-1} \frac{\left(Y_i - V_i\right)^2}{V_i}.$$

Here Y_i is the sample point from *srcWave1*, V_i is the expected value of Y_i based on an assumed distribution (*srcWave2*), and n is the number of points in the each wave. If you do not use /S, it calculates:

$$\chi^2 = \sum_{i=0}^{n-1} \frac{\left(Y_{1i} - Y_{2i}\right)^2}{Y_{1i} + Y_{2i}},$$

where Y_{1i} and Y_{2i} are taken from *srcWave1* and *srcWave2* respectively.

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsContingencyTable**.

StatsCircularCorrelationTest

```
StatsCircularCorrelationTest [flags] waveA, waveB
```

The StatsCircularTwoSampleTest operation peforms a number of tests for two samples of circular data. Using the appropriate flags you can choose between parametric or nonparametric, unordered or paired tests. The input consists of two waves that contain one or two columns. The first column contains angle data expressed in radians and an optional second column contains associated vector lengths. The waves must be either single or double precision floating point. Results are stored in the W_StatsCircularCorrelationTest wave in the current data folder and optionally displayed in a table. Some flags generate additional outputs, described below.

Flags

/ALPH=*val* Sets the significance level (default 0.05).

/NAA Performs a nonparametric angular-angular correlation test.

/PAA Performs a parametric angular-angular correlation test.

/PAL Performs a parametric angular-linear correlation test. In this case the angle wave is *waveA* and the linear data corresponds to *waveB*.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/Z Ignores errors.

Details

The nonparametric test (/NAA) follows Fisher and Lee's modification of Mardia's statistic, which is an analogue of Spearman's rank correlation. The test ranks the angles of each sample and computes the quantities *r*' and *r*'' as follows:

$$r' = \frac{\left\{\sum_{i=0}^{n-1}\cos\left[\frac{2\pi}{n}\left(r_{ai}-r_{bi}\right)\right]\right\}^2 + \left\{\sum_{i=0}^{n-1}\sin\left[\frac{2\pi}{n}\left(r_{ai}-r_{bi}\right)\right]\right\}^2}{n^2},$$

$$r'' = \frac{\left\{\sum_{i=0}^{n-1}\cos\left[\frac{2\pi}{n}\left(r_{ai}+r_{bi}\right)\right]\right\}^2 + \left\{\sum_{i=0}^{n-1}\sin\left[\frac{2\pi}{n}\left(r_{ai}+r_{bi}\right)\right]\right\}^2}{n^2}.$$

Here n is the number of data pairs and r_{ai} and r_{bi} are the ranks of the *i*th member in the first and second samples respectively.

The test statistic is (n-1)(r'-r''), which is compared with the critical value (for one and two tails). The CDF of the statistic is a highly irregular function. The critical value is computed by a different methods according to *n*. For $3 \le n \le 8$, a built-in table of CDF transitions gives a "conservative" estimate of the critical value. For $9 \le n \le 30$, the CDF is approximated by a 7th order polynomial in the region $x > 0$. For $n \ge 30$, the CDF is from the asymptotic expression. For $3 \le n \le 30$, CDF values are obtained by Monte-Carlo simulations using 1e6 random samples for each *n*.

The parametric test for angular-angular correlation (/PAA) involves computation of a correlation coefficient r_{aa} and then evaluating the mean $\overline{r_{aa}}$ and variance $s^2_{r_{aa}}$ of equivalent correlation coefficients computed from the same data but by deleting a different pair of angles each time. The mean and variance are then used to compute confidence limits L1 and L2:

$$L1 = nr_{aa} - (n-1)\overline{r_{aa}} - Z_{\alpha(2)}\sqrt{\frac{s^2_{r_{aa}}}{n}},$$

$$L2 = nr_{aa} - (n-1)\overline{r_{aa}} + Z_{\alpha(2)}\sqrt{\frac{s^2_{r_{aa}}}{n}}$$

where $Z_{\alpha(2)}$ is the normal distribution two-tail critical value at the a level of significance. H_0 (corresponding to no correlation) is rejected if zero is not contained in the interval [L1,L2].

The parametric test for angular-linear correlation (/PAL) involves computation of the correlation coefficient r_{al} which is then compared with a critical value from χ^2 for alpha significance and two degrees of freedom.

$$r_{al} = \sqrt{\frac{r_{xc}^2 + r_{xs}^2 - 2r_{xc}r_{xs}r_{cs}}{1 - r_{cs}^2}},$$

where:

$$r_{xc} = \frac{\sum_{i=0}^{n-1} X_i \cos(a_i) - \frac{1}{n}\sum_{i=0}^{n-1} X_i \sum_{i=0}^{n-1} \cos(a_i)}{\sqrt{\left(\sum_{i=0}^{n-1} X_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1} X_i\right)^2\right)\left(\sum_{i=0}^{n-1} \cos^2(a_i) - \frac{1}{n}\left(\sum_{i=0}^{n-1} \cos(a_i)\right)^2\right)}},$$

$$r_{xs} = \frac{\sum_{i=0}^{n-1} X_i \sin(a_i) - \frac{1}{n}\sum_{i=0}^{n-1} X_i \sum_{i=0}^{n-1} \sin(a_i)}{\sqrt{\left(\sum_{i=0}^{n-1} X_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1} X_i\right)^2\right)\left(\sum_{i=0}^{n-1} \sin^2(a_i) - \frac{1}{n}\left(\sum_{i=0}^{n-1} \sin(a_i)\right)^2\right)}},$$

$$r_{cs} = \frac{\sum_{i=0}^{n-1} \cos(a_i)\sin(a_i) - \frac{1}{n}\sum_{i=0}^{n-1} \sin(a_i) \sum_{i=0}^{n-1} \cos(a_i)}{\sqrt{\left(\sum_{i=0}^{n-1} \sin^2(a_i) - \frac{1}{n}\left(\sum_{i=0}^{n-1} \sin(a_i)\right)^2\right)\left(\sum_{i=0}^{n-1} \cos^2(a_i) - \frac{1}{n}\left(\sum_{i=0}^{n-1} \cos(a_i)\right)^2\right)}}.$$

References

Fisher, N.I., and A.J. Lee, Nonparametric measures of angular-angular association, *Biometrica*, *69*, 315-321, 1982.

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvChiCDF**, **StatsInvNormalCDF**, and **StatsKendallTauTest**.

StatsCircularMeans

```
StatsCircularMeans [flags] srcWave
```

The StatsCircularMeans operation calculates the mean of a number of circular means, returning the mean angle (grand mean), the length of the mean vector, and optionally confidence interval around the mean angle. Output is to the history area and to the W_CircularMeans wave in the current data folder.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/CI	Calculates the confidence interval (labeled CI_t1 and CI_t2) around the mean angle.
/NSOA	Performs nonparametric second order analysis according to Moore's version of Rayleigh's test where H_0 corresponds to uniform distribution around the circle. Moore's test ranks entries by the lengths of the mean radii (second column of the input) from smallest (rank 1) to largest (rank *n*) and then computes the statistic:

$$R' = \sqrt{\frac{\left(\frac{1}{n}\sum_{i=0}^{n-1}(i+1)\cos(a_i)\right)^2 + \left(\frac{1}{n}\sum_{i=0}^{n-1}(i+1)\sin(a_i)\right)^2}{n}},$$

where a_i are the mean angle entries (from column 1) corresponding to vector length rank (i+1). The critical value is obtained from Moore's distribution **StatsInvMooreCDF**.

/PSOA Perform parametric second order analysis where H_0 corresponds to no mean population direction. It assumes that the second order quantities are from a bivariate normal distribution. If this is not the case, use /NSOA above. The test statistic is:

$$F = \frac{k(k-2)}{2}\left[\frac{\overline{X}^2 S_{y^2} - 2\overline{X}\overline{Y}S_{xy} + \overline{Y}^2 S_{x^2}}{S_{x^2}S_{y^2} - S_{xy}^2}\right.$$

where

$$\overline{X} = \frac{1}{n}\sum_{i=0}^{n-1} X_i = \frac{1}{n}\sum_{i=0}^{n-1} r_i\cos(a_i),$$

$$\overline{Y} = \frac{1}{n}\sum_{i=0}^{n-1} Y_i = \frac{1}{n}\sum_{i=0}^{n-1} r_i\sin(a_i),$$

$$S_{x^2} = \sum_{i=0}^{n-1} X_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1} X_i\right)^2,$$

$$S_{y^2} = \sum_{i=0}^{n-1} Y_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1} Y_i\right)^2,$$

$$S_{xy} = \sum_{i=0}^{n-1} X_iY_i - \frac{1}{n}\sum_{i=0}^{n-1} X_i\sum_{i=0}^{n-1} Y_i.$$

Here n is the number of means in *srcWave* and the critical value is computed from the F distribution, equivalent to executing:

```
Print StatsInvFCDF(1-alpha,2,n-2)
```

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The *srcWave* input to StatsCircularMeans must be a single or double precision two column wave containing in each row a mean angle (radians) and the length of a mean radius (the first column contains mean angles and the second column contains mean vector lengths). *srcWave* must not contain any NaNs or INFs. The confidence interval calculation follows the procedure outlined by Batschelet.

V_flag will be set to -1 for any error and to zero otherwise.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCircularMoments**, **StatsInvMooreCDF**, and **StatsInvFCDF**.

StatsCircularMoments

`StatsCircularMoments` `[`***`flags`***`]` ***`srcWave`***

The StatsCircularMoments operation computes circular statistical moments and optionally performs angular uniformity tests for the data in *srcWave*. The extent of the calculation is determined by the requested moment. The default results are stored in the W_CircularStats wave in the current data folder and are optionally displayed in a table. Additional results are listed under the corresponding flags.

Flags

/ALPH=*alpha* Sets an *alpha* value for computing confidence intervals (default is 0.05).

/AXD=*p* Designates the input as p-axial data. For example, if the input represents undirected lines then *p* =2 and the operation multiplies the angles by a factor *p* (after shifting /ORGN and accounting for /CYCL). It does not back-transform the mean or median axis.

/CYCL=*cycle* Specifies the length of the data cycle. You do not need to do so if you are using one of the built-in modes, but this is still a useful option, as for setting the length of a particular month when using /MODE=5.

/GRPD={*start*, *delta*}

Computes circular statistics for grouped data. In this case *srcWave* contains frequencies or the number of events that belong to a particular angle group. There are as many groups as there are elements in *srcWave*. The first group is centered at *start* radians and each consecutive group is centered *delta* radians away. You must set both the *start* and *delta* to sensible values. *srcWave* may contain NaNs but it is an error if all values are NaN. The only other flags that work in combination with this flag are /Q, /T, and /Z.

/KUPR[=*k*] Tests the uniformity of a circular distribution of ungrouped data using the Kuiper statistic. The data are converted into a set {x_i} by normalizing the input angles to the range [0,1], ranking the results then using the two quantities D_+ and D_- to compute the Kuiper statistic. Use k=0 for Fisher's version:

$$\mathrm{V}=\left(D_+ + D_-\right)\left(\sqrt{\mathrm{n}}+0.155+0.24/\sqrt{\mathrm{n}}\right),$$

Use k=1, added in Igor Pro 8.00, for the more common definition of the Kuiper statistic:

$$V=\left(D_+ + D_-\right).$$

Here

$$D_+ = \text{Max of: } \frac{1}{\mathrm{n}}\text{-x}_0,\ \frac{2}{\mathrm{n}}\text{-x}_1,\dots,1\text{-x}_{\mathrm{n-1}},$$

$$D_- = \text{Max of: } \mathrm{x}_0,\ \mathrm{x}_{1-\frac{1}{\mathrm{n}}},\dots,\ \mathrm{x}_{\mathrm{n-1}-\frac{\mathrm{n-1}}{\mathrm{n}}},\ ,$$

and *n* is the number of valid points in *srcWave*. You can find the results in the wave W_CircularStats under row label “Kuiper V” and “Kuiper CDF(V)”. See Fisher and Press *et al*. for more information.

/LOS Computes Linear Order Statistics by sorting the angle values from small to large, dividing each angle by 2π and shifting the origin so that the output range is [0,1]. The results are stored in the wave W_LinearOrderStats in the current data folder. The X scaling of the wave is set so that the offset and the delta are 1/(*n*+1) where *n* is the number of non-NaN points in the input.

/M=*moment* Computes specified moments. By default, it computes the second order moments as well as skewness, kurtosis, median, and mean deviation. Use /M=1 for the first moment. For higher moments, both the specified moment and all the default quantities are computed.

/MODE=*mode* Handles special types of data.

mode	**Data in *srcWave***
0	Angles in radians [0,2π]
1	Angles in radians [-π, π]
2	Angles in degrees [0,360]
3	Angles in degrees [-180,180]
4	Igor date format for one year cycles.
5	Igor date format for one month cycles.
6	Igor date format for one week cycles.
7	Igor date format for one day cycles.
8	Igor date format for one hour cycles.

/ORGN=*origin* Specifies the origin of the data (the value corresponding to an angle of zero degrees). For example, if you are using Igor date format and you want the origin to be the first second in year YYYY, use `/ORGN=(date2secs(YYYY,1,1))`.

/Q No results printed in the history area.

/RAYL[=*meanDirection*]

Performs the Rayleigh test for uniformity. If the “alternative” mean direction is specified (in radians), the test computes

```
r0Bar=rBar cos(tBar-meanDirection)
```

and then computes the significance probability of r0Bar. The null hypothesis H_0 corresponds to uniformity. It is rejected when r0Bar is too large. If the mean direction is not specified then r0Bar is rBar which is always calculated as part of the first moments so the operation only computes the relevant significance probability (P-Value). The critical values for both cases are computed according to Durand and Greenwood.

/SAW Saves the translated angle data in the wave W_AngleWave in the current data folder.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).
k=1: Kills with no dialog.
k=2: Disables killing.

The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results unless you moved the output wave to a different data folder. If the named table exists, but does not display the output wave from the current data folder, the table is renamed and a new table is created.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsCircularMoments is equivalent to **WaveStats** but it applies to circular data, which are distributed on the perimeter of a circle representing some period or cycle. If your data are not described by one of the built-in modes, you can specify the value of the origin (/ORGN), which is mapped to zero degrees and the size of a cycle or period.

When you use Igor date formats with the built-in modes for dates, the default origin is set to zero. The default cycle in the case of Mode 4 is 366. This is done in order to handle both leap and nonleap years. Similarly, Mode 5 uses a cycle of 31 days. Note that the internal conversion from Igor date to (year, month, day) is independent of the cycle specification and is therefore not affected by this choice. You should use the /CYCL flag if you use one of these modes with a fixed size of year or month.

The parameters listed below are computed and displayed (see row labels) in the table. Here *N* is the number of valid (non-NaN) angles $\{\theta_i\}$

$$\mathrm{C}=\sum_{i=1}^{n}\cos\theta_i$$

$$\mathrm{S}=\sum_{i=1}^{n}\sin\theta_i$$

$$R=\sqrt{C^2+S^2}$$

$$cBar=\overline{C}=C/n$$

$$sBar=\overline{S}=S/n$$

$$rBar=\overline{R}=R/n$$

$$tBar=\overline{\theta}=\begin{cases}\operatorname{atan}(S/C) & S>0, C>0\\ \operatorname{atan}(S/C)+\pi & C<0\\ \operatorname{atan}(S/C)+2\pi & S<0, C>0\end{cases}$$

$$V=1-\overline{R}$$

$$v=\sqrt{-2\ln(1-V)}$$

median is the value which minimizes

$$d(\theta)=\pi-\frac{1}{n}\sum_{i=1}^{n}\left|\pi-\left|\theta_i-\overline{\theta}\right|\right|$$

mean deviation = The minimum of the last equation when θ → median.

Higher order moments are denoted with the moment number such that t3Bar is the uncentered third moment of the angle while primed quantities are relative to mean direction tBar. Using this notation

$$\widehat{\rho_2}=\frac{1}{n}\sum_{i=1}^{n}\cos 2\left(\theta_i-\overline{\theta}\right)$$

$$circular\ \ dispersion=\frac{1-\widehat{\rho_2}}{2\overline{R}^2}$$

$$skewness = \frac{\widehat{\rho_2}\sin\left(\hat{\mu}_2' - 2\bar{\theta}\right)}{\left(1-\bar{R}\right)^{3/2}}$$

$$kurtosis = \frac{\widehat{\rho_2}\cos\left(\hat{\mu}_2' - 2\bar{\theta}\right) - \bar{R}^4}{\left(1-\bar{R}\right)^2}$$

where

$$\hat{\mu}_p' = \begin{cases} \operatorname{atan}\left(S_p/C_p\right) & S_p > 0, C_p > 0 \\ \operatorname{atan}\left(S_p/C_p\right) + \pi & C_p < 0 \\ \operatorname{atan}\left(S_p/C_p\right) + 2\pi & S_p < 0, C_p > 0 \end{cases}$$

and

$$C_p = \frac{1}{n}\sum_{i=1}^{n}\cos p\theta_i, \qquad S_p = \frac{1}{n}\sum_{i=1}^{n}\sin p\theta_i.$$

References

Fisher, N.I., *Statistical Analysis of Circular Data*, 295pp., Cambridge University Press, New York, 1995.

Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

Durand, D., and J.A. Greenwood, Modifications of the Rayleigh test for uniformity in analysis of two-dimensional orientation data, *J. Geol.*, *66*, 229-238, 1958.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

WaveStats, **StatsAngularDistanceTest**, **StatsCircularCorrelationTest**, **StatsCircularMeans**, **StatsHodgesAjneTest**, **StatsWatsonUSquaredTest**, **StatsWatsonWilliamsTest**, and **StatsWheelerWatsonTest**.

StatsCircularTwoSampleTest

`StatsCircularTwoSampleTest [`*`flags`*`]` ***`waveA, waveB`***

The StatsCircularTwoSampleTest operation performs second order analysis of angles. Using the appropriate flags you can choose between parametric or nonparametric, unordered or paired tests. The input consists of two waves that contain one or two columns. The first column contains angle data (mean angles) expressed in radians and an optional second column that contains associated vector lengths. The waves must be either single or double precision. Results are stored in the W_StatsCircularTwoSamples wave in the current data folder and optionally displayed in a table. Some of the tests may have additional outputs.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/NPR	Performs nonparametric paired-sample test (Moore). The input waves must contain paired angular data so both must have single column and the same number of points.
/NSOA	Perform nonparametric second order two-sample test. Input waves must each contain two columns.
/PPR	Performs parametric paired-sample test. Input waves must contain paired data and must have the same number of points.

/PSOA Performs parametric second order analysis of two samples. The input waves must each contain two columns.

/Q No information printed in the history area.

/T= *k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/Z Ignores any errors.

Details

The nonparametric paired-sample test (/NPR) is Moore's test for paired angles applied in second order analysis. The input can consist of one or two column waves. When both waves contain a single column the operation proceeds as if all the vector length were identically 1. The Moore statistic (H_0 → pair equality) is computed and compared to the critical value from the Moore distribution (see **StatsInvMooreCDF**).

The nonparametric second-order two-sample test (/NSOA) consists of pre-processing where the grand mean is subtracted from the two inputs followed by application of Watson's U^2 test (**StatsWatsonUSquaredTest**) with H_0 implying that the two samples came from the same population. The results of this test are stored in the wave W_WatsonUtest.

The parametric paired-sample test (/PPR) is due to Hotelling. In this test the input should consist of both angular and vector length data. The test statistic is compared with a critical value from the F distribution (**StatsInvFCDF**).

The parametric second order two-sample test (/PSOA) is an extension of Hotelling one-sample test to second order analysis where an F-like statistic is computed corresponding to H_0 of equal mean angles.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvMooreCDF**, **StatsWatsonUSquaredTest**, and **StatsInvFCDF**.

StatsCMSSDCDF

`StatsCMSSDCDF(`*`C`*`, `*`n`*`)`

The StatsCMSSDCDF function returns the cumulative distribution function of the C distribution (mean square successive difference), which is

$$f(C,n)=\frac{\Gamma(2m+2)}{a2^{2m+1}\left[\Gamma(m+1)\right]^2}\left(1-\frac{C^2}{a^2}\right)^m,$$

where

$$a^2=\frac{\left(n^2+2n-12\right)\left(n-2\right)}{\left(n^3-13n+24\right)},$$

$$m=\frac{\left(n^4-n^3-13n^2+37n-60\right)}{2\left(n^3-13n+24\right)}.$$

The distribution (*C*>0) can then be expressed as

$$F(C,n)=\frac{\Gamma(2m+2)}{a2^{2m+1}\left[\Gamma(m+1)\right]^2}C\,{}_2F_1\left(\frac{1}{2},-m,\frac{3}{2},\frac{C^2}{a^2}\right),$$

where ${}_2F_1$ is the hypergeometric function **hyperG2F1**.

References

Young, L.C., On randomness in ordered sequences, *Annals of Mathematical Statistics*, *12*, 153-162, 1941.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCMSSDCDF** and **StatsSRTest**.

StatsCochranTest

`StatsCochranTest [`*`flags`*`] [`*`wave1, wave2,… wave100`*`]`

The StatsCochranTest operation performs Cochran’s (Q) test on a randomized block or repeated measures dichotomous data. Output is to the M_CochranTestResults wave in the current data folder or optionally to a table.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results unless you moved the output wave to a different data folder. If the named table exists but it does not display the output wave from the current data folder, the table is renamed and a new table is created.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsCochranTest computes Cochran's statistic and compares it to a critical value from a Chi-squared distribution, which depends only of the significance level and the number of groups (columns). The null hypothesis for the test is that all columns represent the same proportion of the effect represented by a non-zero data.

The Chi-square distribution is appropriate when there are at least 4 columns and at least 24 total data points.

Dichotomous data are presumed to consist of two values 0 and 1, thus StatsCochranTest distinguishes only between zero and any nonzero value, which is considered to be 1; it does not allow NaNs or INFs. Input waves can be a single 2D wave or a list of 1D numeric waves, which can also be specified in a string list with /WSTR. In the standard terminology, data rows represent blocks and data columns represent groups. H_0 corresponds to the assumption that all groups have the same proportion of 1’s.

With the /T flag, it displays the results in a table that contains the number of rows, the number of columns, the Cochran statistic, the critical value, and the conclusion (1 to accept H_0 and 0 to reject it).

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFriedmanTest**.

StatsContingencyTable

StatsContingencyTable [*flags*] srcWave

The StatsContingencyTable operation performs contingency table analysis on 2D and 3D tables. Output is to the W_ContingencyTableResults wave in the current data folder or optionally to a table or the history area.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/COR=*mode* Sets the correction type for 2x2 tables. By default there is no correction. Use *mode*=1 for Yates and *mode*=2 for Haber correction.

/FEXT={*row*, *col*} Computes Fisher's Exact P-value with 2x2 contingency tables. *row* and *col* are zero-based indices of the table entry where it computes the probability of getting the results in the table or more extreme values. Without the /Q flag, it prints the probabilities of each individual table in the history.

Given the contingency table:

	Succeeded	**Failed**
Group1	11	8
Group2	4	9

Example 1: When you use `/FEXT={0,0}` the P-value represents the sum of the probabilities of the first group having in the Succeeded column 11 or more extreme values, i.e., 12, 13, 14, and 15. In each case the remaining table elements are adjusted so that row and column sums remain constant.

Example 2: When you needed to evaluate the sum of the probabilities of Group2 having 4 counts or less in the Succeeded column, then the appropriate flag is `/FEXT={1,1}`, which effectively computes the equivalent of having 9, 10, 11, 12, and 13 Failed counts. In each case it computes the upper, the lower, and the two-tail probabilities.

/HTRG Tests for heterogeneity between tables stored as layers of 3D wave.

/LLIK Computes log likelihood statistic.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results unless you moved the output wave to a different data folder. If the named table exists but it does not display the output wave from the current data folder, the table is renamed and a new table is created.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsContingencyTable supports 2D waves representing single contingency tables or 3D waves representing multiple 2D tables (where each table is a layer) or a single 3D table. Each entry in the wave must contain a frequency value and must be a positive number; it does not support 0's, NaNs, or INFs. In the special case of 2x2 tables, use the /COR flag to compute the statistic using either the Yates or Haber corrections. Except for the heterogeneity option you can also compute the log likelihood statistic. In all the tests, H_0 corresponds to independence between the tested variables.

For 3D tables StatsContingencyTable provides Chi-squared, degrees of freedom, the critical value, and optionally the log likelihood G statistic (/LLIK flag) for each of the following cases:

- Mutual independence by testing if all three variables are independent of each other.
- Partial dependence (rows) by testing if rows independent of columns and layers.
- Partial dependence (columns) by testing if columns independent of rows and layers.
- Partial dependence (layers) by testing if layers independent of rows and columns.

In each case you should compare the statistic with the critical value and reject H_0 if the statistic exceeds or equals the critical value.

You should examine the table entries to determine if the Chi-square statistic is appropriate (if the frequency is smaller than 6 for /ALPH=0.05 you should consider computing the Fisher exact test).

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvChiCDF**.

StatsCorrelation

`StatsCorrelation(`*`waveA`*` [, `*`waveB`*`])`

The StatsCorrelation function computes Pearson's correlation coefficient between two real valued arrays of data of the same length. Pearson r is give by:

$$r = \frac{\sum_{i=0}^{n-1}\left(waveA[i]-A\right)\left(waveB[i]-B\right)}{\sqrt{\sum_{i=0}^{n-1}\left(waveA[i]-A\right)^2 \sum_{i=0}^{n-1}\left(waveB[i]-B\right)^2}}$$

Here *A* is the average of the elements in *waveA*, *B* is the average of the elements of *waveB* and the sum is over all wave elements.

Details

If you use both *waveA* and *waveB* then the two waves must have the same number of points but they could be of different number type. If you use only the *waveA* parameter then *waveA* must be a 2D wave. In this case StatsCorrelation will return 0 and create a 2D wave M_Pearson where the (*i,j*) element is Pearson's r corresponding to columns *i* and *j*.

Fisher's z transformation converts Person's r above to a normally distributed variable *z*:

$$z = \frac{1}{2}\ln\left(\frac{1+r}{1-r}\right),$$

with a standard error

$$\sigma_z = \frac{1}{\sqrt{n-3}}.$$

You can convert between the two representations using the following functions:

```
Function pearsonToFisher(inr)
   Variable inr
   return 0.5*(ln(1+inr)-ln(1-inr))
End

Function fisherToPearson(inz)
   Variable inz
   return tanh(inz)
End
```

See Also

Correlate, **StatsLinearCorrelationTest**, and **StatsCircularCorrelationTest**.

StatsDExpCDF

StatsDExpCDF(*x*, *m*, *s*)

The StatsDExpCDF function returns the double-exponential cumulative distribution function

$$F(x;\mu,\sigma) = \begin{cases} \exp\left(\dfrac{x-\mu}{\sigma}\right) & \textit{when } x < \mu \\ 1-\dfrac{1}{2}\exp\left(-\left|\dfrac{x-\mu}{\sigma}\right|\right) & \textit{when } x \geq \mu \end{cases}$$

for σ>0. It returns NaN when σ=0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsDExpPDF** and **StatsInvDExpCDF**.

StatsDExpPDF

StatsDExpPdf(*x*, *m*, *s*)

The StatsDExpPdf function returns the double-exponential probability distribution function

$$\mathrm{f(x;}\mu,\sigma\mathrm{)}=\frac{1}{2\sigma}\exp\left[-\left|\frac{x-\mu}{\sigma}\right|\right],$$

where μ is the location parameter and σ>0 is the scale parameter. Use μ=0 and σ=1 for the standard form of the double exponential distribution. It returns NaN when σ=0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsDExpCDF** and **StatsInvDExpCDF**.

StatsDIPTest

StatsDIPTest [/Z] *srcWave*

The StatsDIPTest operation performs Hartigan test for unimodality.

Flags

/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The input to the operation *srcWave* is any real numeric wave. Outputs are: V_Value contains the dip statistic; V_min is the lower end of the modal interval; and V_max is the higher end of the modal interval. Percentage points or critical values for the dip statistic can be obtained from simulations using an identical sample size as in this example:

```
Function getCriticalValue(sampleSize,alpha)
Variable sampleSize,alpha

   Make/O/N=(sampleSize) dataWave
   Make/O/N=100000  dipResults
   Variable i
   for(i=0;i<100000;i+=1)
      dataWave=enoise(100)
      StatsDipTest dataWave
      dipResults[i]=V_Value
   endfor
   Histogram/P/B=4 dipResults              // Compute the PDF.
   Wave W_Histogram
   Integrate/METH=1 W_Histogram/D=W_INT    // Compute the CDF.
   Findlevel/Q  W_int,(1-alpha)            // Find the critical value.
   return V_LevelX
End
```

References

Hartigan, P. M., Computation of the Dip Statistic to Test for Unimodality, *Applied Statistics, 34*, 320-325, 1985.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsDunnettTest

StatsDunnettTest [***flags***] [***wave1, wave2,… wave100***]

The StatsDunnettTest operation performs the Dunnett test by comparing multiple groups to a control group. Output is to the M_DunnettTestResults wave in the current data folder or optionally to a table. StatsDunnettTest usually follows StatsANOVA1Test.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/CIDX=*cIndex* Specifies the (zero based) index of the input wave corresponding to the control group. The default is zero (the first wave corresponds to the control group).

/Q No results printed in the history area.

/SWN Creates a text wave, T_DunnettDescriptors, containing wave names corresponding to each row of the comparison table (Save Wave Names). Use /T to append the text wave to the last column.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

/TAIL=*tc* Specifies H_0.

tc=1:	One tailed test ($\mu_c \le \mu_a$)
tc=2:	One tailed test ($\mu_c \ge \mu_a$)
tc=4:	Two tailed test ($\mu_c = \mu_a$) (default)

Code combinations are not allowed.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsDunnettTest inputs are two or more 1D numeric waves (one wave for each group of samples). The input waves may contain different number of points, but they must contain two or more valid entries per wave.

For output to a table (using /T), each labelled row represents the results of the test for comparing the means of one group to the control group, and rows are ordered so that all comparisons are computed sequentially starting with the group having the smallest mean. The contents of the labeled columns are:

First	The difference between the group means
Second	SE (which is computed for possibly unequal number of points)
Third	The q statistic for the pair which may be positive or negative

Fourth	The critical q' value
Fifth	0 if the conclusion is to reject H_0 or 1 to accept H_0
Sixth	The P-value

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsTukeyTest**, **StatsANOVA1Test**, **StatsScheffeTest**, and **StatsNPMCTest**.

StatsErlangCDF

`StatsErlangCDF(`*`x`*`, `*`b`*`, `*`c`*`)`

The StatsErlangCDF function returns the Erlang cumulative distribution function

$$F(x;b,c) = 1 - \frac{\Gamma\left(c,\dfrac{x}{b}\right)}{\Gamma(c)}.$$

where b>0 (also as λ=1/b) is the scale parameter, c> 0 the shape parameter, $\Gamma(x)$ the **gamma** function, and $\Gamma(a,x)$ the incomplete gamma function **gammaInc**.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsErlangPDF**.

StatsErlangPDF

`StatsErlangPDF(`*`x`*`, `*`b`*`, `*`c`*`)`

The StatsErlangPDF function returns the Erlang probability distribution function

$$f(x;b,c) = \frac{\left(\dfrac{x}{b}\right)^{c-1} \exp\left(-\dfrac{x}{b}\right)}{b(c-1)!}.$$

where b>0 (also as λ=1/b) is the scale parameter and c> 0 the shape parameter.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsErlangCDF**.

StatsErrorPDF

`StatsErrorPDF(`*`x`*`, `*`a`*`, `*`b`*`, `*`c`*`)`

The StatsErrorPDF function returns the error probability distribution function or the exponential power distribution

$$f(x;a,b,c) = \frac{\exp\left[-\dfrac{1}{2}\left(\dfrac{|x-a|}{b}\right)^{\frac{2}{c}}\right]}{b2^{\frac{c}{2}+1}\Gamma\left(1+\dfrac{c}{2}\right)}.$$

where a is the location parameter, b> 0 is the scale parameter, c> 0 is the shape parameter, and $\Gamma(x)$ is the **gamma** function.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsEValueCDF

`StatsEValueCDF(x, μ, σ)`

The StatsEValueCDF function returns the extreme-value (type I, Gumbel) cumulative distribution function

$$F(x;\mu,\sigma) = 1 - \exp\left(-\exp\left(\frac{x-\mu}{\sigma}\right)\right),$$

where σ>0. This is also known as the "minimum" form or distribution of the smallest extreme. To obtain the distribution of the largest extreme reverse the sign of σ.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsEValuePDF, **StatsInvEValueCDF**, **StatsGEVCDF**, **StatsGEVPDF**

StatsEValuePDF

`StatsEValuePDF(x, μ, σ)`

The StatsEValuePDF function returns the extreme-value (type I, Gumbel) probability distribution function

$$F(x;\mu,\sigma) = 1 - \exp\left(-\exp\left(\frac{x-\mu}{\sigma}\right)\right),$$

where σ>0. This is also known as the "minimum" form or the distribution of the smallest extreme. To obtain the distribution of the largest extreme reverse the sign of σ.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsEValueCDF, **StatsInvEValueCDF**, **StatsGEVCDF**, **StatsGEVPDF**

StatsExpCDF

`StatsExpCDF(x, μ, σ)`

The StatsExpCDF function returns the exponential cumulative distribution function

$$F(x;\mu,\sigma) = 1 - \exp\left(-\frac{x-\mu}{\sigma}\right),$$

where $x \geq \mu$ and σ > 0. It returns NaN for σ = 0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsExpPDF** and **StatsInvExpCDF**.

StatsExpPDF

`StatsExpPDF(x, μ, σ)`

The StatsExpPDF function returns the exponential probability distribution function

$$f(x;\mu,\sigma) = \frac{1}{\sigma}\exp\left(-\frac{x-\mu}{\sigma}\right),$$

where μ is the location parameter and σ>0 is the scale parameter. Use μ=0 and σ=1 for the standard form of the exponential distribution. It returns NaN for σ=0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsExpCDF** and **StatsInvExpCDF**.

StatsFCDF

`StatsFCDF(x, n1, n2)`

The StatsFCDF function returns the cumulative distribution function for the F distribution with shape parameters *n1* and *n2*

$$F(x;n_1,n_2) = 1 - Betai\left(\frac{n_2}{2},\frac{n_1}{2},\frac{n_2}{n_2+n_1x}\right),$$

where *Betai* is the incomplete beta function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFPDF** and **StatsInvFCDF**.

StatsFPDF

`StatsFPDF(x, n1, n2)`

The StatsFPDF function returns the probability distribution function for the F distribution with shape parameters *n1* and *n2*

$$f(x;n_1,n_2) = \frac{\Gamma\left(\frac{n_1+n_2}{2}\right)\left(\frac{n_1}{n_2}\right)^{\frac{n_1}{2}} x^{\frac{n_1}{2}-1}}{\Gamma\left(\frac{n_1}{2}\right)\Gamma\left(\frac{n_2}{2}\right)\left(1+\frac{n_1x}{n_2}\right)^{\frac{n_1+n_2}{2}}}.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFCDF** and **StatsInvFCDF**.

StatsFriedmanCDF

`StatsFriedmanCDF(x, n, m, method, useTable)`

The StatsFriedmanCDF function returns the cumulative probability distribution of the Friedman distribution with *m* rows and *n* columns. The exact Friedman distribution is computationally intensive, taking on the order of $(n!)^m$ iterations. You may be able to use a range of precomputed exact values by passing a nonzero value for *useTable*, which will use *method* only if the value is not in the table. For large *m,* consider using the Chi-squared or the Monte-Carlo approximations. To abort execution, press the **User Abort Key Combinations**.

method	**What It Does**
0	Exact computation.
1	Chi-square approximation.
2	Monte-Carlo approximation.
3	Use built-table only and return NaN if not in table.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvFriedmanCDF** and **StatsFriedmanTest**.

StatsFriedmanTest

`StatsFriedmanTest` `[`***`flags`***`] [`***`wave1, wave2,… wave100`***`]`

The StatsFriedmanTest operation performs Friedman's test on a randomized block of data. It is a nonparametric analysis of data contained in either individual 1D waves or in a single 2D wave. Output is to the M_FriedmanTestResults wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/RW	Saves the ranking wave M_FriedmanRanks, which contains the rank values corresponding to each input datum.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results.
/WSTR=*waveListString*	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The Friedman test ranks the input data on a row-by-row basis, sums the ranks for each column, and computes the Friedman statistic, which is proportional to the sum of the squares of the ranks.

Input waves can be a single 2D wave or a list of 1D numeric waves, which can also be specified in a string list with /WSTR. All 1D waves must have the same number of points. A 2D wave must not contain any NaNs.

The critical value for the Friedman distribution is fairly difficult to compute when the number of rows and columns is large because it requires a number of permutations on the order of (numColumns!)^numRows. A certain range of these critical values are supported by precomputed tables. When the exact critical value is not available you can use one of the two approximations that are always computed: the Chi-squared approximation or the Iman and Davenport approximation, which converts the Friedman statistic is converted to a new value Ff then compares it with critical values from the F distribution using weighted degrees of freedom.

With the /T flag, it displays the results in a table that contains the number of rows, the number of columns, the Friedman statistic, the exact critical value (if available), the Chi-squared approximation, the Iman and Davenport approximation, and the conclusion (1 to accept H_0 and 0 to reject it).

V_flag will be set to -1 for any error and to zero otherwise.

References

Iman, R.L., and J.M. Davenport, Approximations of the critical region of the Friedman statistic, *Comm. Statist. A9*, 571-595, 1980.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFriedmanCDF** and **StatsInvFriedmanCDF**.

StatsFTest

`StatsFTest` `[`***`flags`***`]` ***`wave1, wave2`***

The StatsFTest operation performs the F-test on the two distributions in *wave1* and *wave2*, which can be any real numeric type, must contain at least two data points each, and can have an arbitrary number of dimensions. Output is to the W_StatsFTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/TAIL=*tc*	Specifies the tail tested. *tc*=1: Lower one-tail test with H_a: sigma1>sigma2. *tc*=2: Upper one-tail test with H_a: sigma1<sigma2. *tc*=3: Default; the null hypothesis H_0: sigma1=sigma2 with H_a: sigma1!=sigma2.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The F statistic is the ratio of the variance of *wave1* to the variance of *wave2*. We assume the waves have equal wave variances and that H_0 is sigma1=sigma2. For the upper one-tail test we reject H_0 if F is greater than the upper critical value or if F is smaller than the lower critical value in the lower one-tail test. In the two-tailed test we reject H_0 if F is either greater than the upper critical value or smaller than the lower critical value. The critical values are computed by numerically solving for the argument at which the cumulative distribution function (CDF) equals the appropriate values for the tests. The CDF is given by

$$F(x,n_1,n_2)=1-betai\left(\frac{n_2}{2},\frac{n_1}{2},\frac{n_2}{n_2+n_1x}\right),$$

where the degrees of freedom n_1 and n_2 equal the number of valid (non-NaN) points in each wave -1, and *betai* is the incomplete beta function. To get the critical value for the upper one-tail test we solve F(x)=1-alpha. For the lower one-tail test we solve F(x)=alpha. In the two-tailed test the lower critical value is a solution for F(x)=alpha/2 and the upper critical value is a solution for F(x)=1-alpha/2.

The F-test requires that the two samples are from normally distributed populations.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsVariancesTest**, **StatsFCDF**, and **betai**.

StatsGammaCDF

```
StatsGammaCDF(x, μ, σ, γ)
```

The StatsGammaCDF function returns the gamma cumulative distribution function

$$F(x;\mu,\sigma,\gamma)=\frac{\Gamma_{inc}\left(\gamma,\frac{x-\mu}{\sigma}\right)}{\Gamma(\gamma)}. \qquad \begin{array}{c} x\ge\mu \\ \sigma,\gamma>0 \end{array}$$

where Γ is the gamma function and Γ_{inc} is the incomplete gamma function **gammaInc**.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGammaPDF** and **StatsInvGammaCDF**.

StatsGammaPDF

StatsGammaPDF(*x*, μ, σ, γ)

The StatsGammaPDF function returns the gamma probability distribution function

$$f(x;\mu,\sigma,\gamma) = \frac{\left(\dfrac{x-\mu}{\sigma}\right)^{\gamma-1} \exp\left(-\dfrac{x-\mu}{\sigma}\right)}{\sigma\Gamma(\gamma)}. \qquad \begin{array}{c} x \ge \mu \\ \sigma,\gamma > 0 \end{array}$$

where μ is the location parameter, σ is the scale parameter, γ is the shape parameter, and Γ is the gamma function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGammaCDF** and **StatsInvGammaCDF**.

StatsGeometricCDF

StatsGeometricCDF(*x*, *p*)

The StatsGeometricCDF function returns the geometric cumulative distribution function

$$F(x,p) = 1-(1-p)^{x+1}.$$

where *p* is the probability of success in a single trial and *x* is the number of trials for $x \ge 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGeometricPDF** and **StatsInvGeometricCDF**.

StatsGeometricPDF

StatsGeometricPDF(*x*, *p*)

The StatsGeometricPDF function returns the geometric probability distribution function

$$f(x,p) = p(1-p)^x,$$

where the *p* is the probability of success in a single trial and *x* is the number of trials $x \ge 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGeometricCDF** and **StatsInvGeometricCDF**.

StatsHodgesAjneTest

StatsHodgesAjneTest [*flags*] *srcWave*

The StatsHodgesAjneTest operation performs the Hodges-Ajne nonparametric test for uniform distribution around a circle. Output is to the W_HodgesAjne wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/SA=*specAngle*	Uses the Batschelet modification of the Hodges-Ajne test to test for uniformity against the alternative of concentration around the specified angle. *specAngle* must be expressed in radians modulus 2π.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The input *srcWave* must contain angles in radians, can be any number of dimensions, can be single or double precision, and should not contain NaNs or INFs.

StatsHodgesAjneTest performs the standard Hodges-Ajne test, which simply tests for uniformity against the hypothesis that the population is not uniformly distributed around the circle. This test finds a diameter that divides the circle into two halves such that one contains the least number of data *m*, the test statistic.

Use /SA to perform the modified (Batschelet) test, which tests against the alternative that the population is concentrated somehow about the specified angle. The modified test counts the number of points *m*' in 90-degree neighborhoods around the specified angle. The test statistic is given by C=*n*-*m*' where *n* is the number of points in the wave. The critical value is computed from the binomial probability density.

In both cases H_0 is rejected if the statistic is smaller than the critical value.

V_flag will be set to -1 for any error and to zero otherwise.

References

Ajne, B., A simple test for uniformity of a circular distribution, *Biometrica*, *55*, 343-354, 1968.

See, in particular, Chapter 27 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsCircularMeans, **StatsCircularMoments**, **StatsWatsonUSquaredTest**, **StatsWatsonWilliamsTest**, and **StatsWheelerWatsonTest**.

StatsGEVCDF

StatsGEVCDF(*x*, μ, σ, ξ**)**

The StatsGEVCDF function returns the generalized extreme value cumulative distribution function.

$$F(x,\mu,\sigma,\xi) = \exp\left\{-\left[1+\xi\left(\frac{x-\mu}{\sigma}\right)^{-1/\xi}\right]\right\},$$

where

$$1+\xi\left(\frac{x-\mu}{\sigma}\right) > 0,$$

and σ>0.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsGEVPDF, **StatsEValuePDF**, **StatsEValueCDF**, **StatsInvEValueCDF**

StatsGEVPDF

StatsGEVPDF(*x*, μ, σ, ξ**)**

The StatsGEVPDF function returns the generalized extreme value probability distribution function.

$$f(x,\mu,\sigma,\xi)=\frac{1}{\sigma}\left[1+\xi\left(\frac{x-\mu}{\sigma}\right)\right]^{(-1/\xi)-1}\exp\left\{-\left[1+\xi\left(\frac{x-\mu}{\sigma}\right)^{-1/\xi}\right]\right\},$$

where

$$1+\xi\left(\frac{x-\mu}{\sigma}\right)>0,$$

and σ>0.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsGEVCDF, **StatsEValuePDF**, **StatsEValueCDF**, **StatsInvEValueCDF**

StatsHyperGCDF

`StatsHyperGCDF(`*`x`*`, `*`m`*`, `*`n`*`, `*`k`*`)`

The StatsHyperGCDF function returns the hypergeometric cumulative distribution function, which is the probability of getting *x* marked items when drawing (without replacement) *k* items out of a population of *m* items when *n* out of the *m* are marked.

Details

The hypergeometric distribution is

$$F(x;m,n,k)=\sum_{L=0}^{x}\frac{\begin{pmatrix} n \\ L \end{pmatrix}\begin{pmatrix} m-L \\ k-L \end{pmatrix}}{\begin{pmatrix} m \\ k \end{pmatrix}},$$

where $\binom{a}{b}$ is the **binomial** function. All parameters must be positive integers and must have *m*>*n* and *x*<*k*; otherwise it returns NaN.

References

Klotz, J.H., *Computational Approach to Statistics*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsHyperGPDF**.

StatsHyperGPDF

`StatsHyperGPDF(`*`x`*`, `*`m`*`, `*`n`*`, `*`k`*`)`

The StatsHyperGPDF function returns the hypergeometric probability distribution function, which is the probability of getting *x* marked items when drawing without replacement *k* items out of a population of *m* items where *n* out of the *m* are marked.

Details

The hypergeometric distribution is

$$f(x;m,n,k)=\frac{\binom{n}{x}\binom{m-n}{k-x}}{\binom{m}{k}},$$

where $\binom{a}{b}$ is the **binomial** function. All parameters must be positive integers and must have *m*>*n* and *x*<*k*.

References

Klotz, J.H., *Computational Approach to Statistics*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsHyperGCDF**.

StatsInvBetaCDF

`StatsInvBetaCDF(cdf, p, q [, a, b])`

The StatsInvBetaCDF function returns the inverse of the beta cumulative distribution function. There is no closed form expression for the inverse beta CDF; it is evaluated numerically.

The defaults (*a*=0 and *b*=1) correspond to the standard beta distribution.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBetaCDF** and **StatsBetaPDF**.

StatsInvBinomialCDF

`StatsInvBinomialCDF(cdf, p, N)`

The StatsInvBinomialCDF function returns the inverse of the binomial cumulative distribution function. The inverse function returns the value at which the binomial *CDF* with probability *p* and total elements *N*, has the value 0.95. There is no closed form expression for the inverse binomial CDF; it is evaluated numerically.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBinomialCDF** and **StatsBinomialPDF**.

StatsInvCauchyCDF

`StatsInvCauchyCDF(cdf, μ, σ)`

The StatsInvCauchyCDF function returns the inverse of the Cauchy-Lorentz cumulative distribution function

$$x=\mu+\sigma\tan\left[\pi\left(cdf-\frac{1}{2}\right)\right].$$

It returns NaN for *cdf* <0 or *cdf*> 1.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCauchyCDF** and **StatsCauchyPDF**.

StatsInvChiCDF

`StatsInvChiCDF(x, n)`

The StatsInvChiCDF function returns the inverse of the chi-squared distribution of *x* and shape parameter *n*. The inverse of the distribution is also known as the percent point function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsChiCDF** and **StatsChiPDF**.

StatsInvCMSSDCDF

```
StatsInvCMSSDCDF(cdf, n)
```

The StatsInvCMSSDCDF function returns the critical values of the C distribution (mean square successive difference distribution), which is given by

$$f(C,n) = \frac{\Gamma(2m+2)}{a2^{2m+1}\left[\Gamma(m+1)\right]^2}\left(1-\frac{C^2}{a^2}\right)^m,$$

where

$$a^2 = \frac{\left(n^2+2n-12\right)\left(n-2\right)}{\left(n^3-13n+24\right)},$$

$$m = \frac{\left(n^4-n^3-13n^2+37n-60\right)}{2\left(n^3-13n+24\right)}.$$

Critical values are computed from the integral of the probability distribution function.

References

Young, L.C., On randomness in ordered sequences, *Annals of Mathematical Statistics*, *12*, 153-162, 1941.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCMSSDCDF** and **StatsSRTest**.

StatsInvDExpCDF

```
StatsInvDExpCDF(cdf, μ, σ)
```

The StatsInvDExpCDF function returns the inverse of the double-exponential cumulative distribution function

$$x = \begin{cases} \mu + \sigma \ln(2cdf) & \text{when } cdf < 0.5 \\ \mu - \sigma \ln\left[2\left(1-cdf\right)\right] & \text{when } cdf \geq 0.5 \end{cases}$$

It returns NaN for *cdf* <0 or *cdf* > 1.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsDExpCDF** and **StatsDExpPDF**.

StatsInvEValueCDF

```
StatsInvEValueCDF(cdf, μ, σ)
```

The StatsInvEValueCDF function returns the inverse of the extreme-value (type I, Gumbel) cumulative distribution function

$$x = \mu - \sigma \ln\left(1-cdf\right)$$

where σ>0. It returns NaN for *cdf*<0 or *cdf*>1. This inverse applies to the “minimum” form of the distribution. Reverse the sign of σ to obtain the inverse distribution of the maximum form.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsEValueCDF, **StatsEValuePDF**, **StatsGEVCDF**, **StatsGEVPDF**

StatsInvExpCDF

`StatsInvExpCDF(`*`cdf`*`, μ, σ)`

The StatsInvExpCDF function returns the inverse of the exponential cumulative distribution function

$$x = \mu - \sigma \ln(1 - cdf).$$

It returns NaN for *cdf* <0 or *cdf* > 1.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsExpCDF** and **StatsExpPDF**.

StatsInvFCDF

`StatsInvFCDF(`*`x`*`, `*`n1`*`, `*`n2`*`)`

The StatsInvFCDF function returns the inverse of the F distribution cumulative distribution function for *x* and shape parameters *n1* and *n2*. The inverse is also known as the percent point function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFCDF** and **StatsFPDF**.

StatsInvFriedmanCDF

`StatsInvFriedmanCDF(`*`cdf`*`, `*`n`*`, `*`m`*`, `*`method`*`, `*`useTable`*`)`

The StatsInvFriedmanCDF function returns the inverse of the Friedman distribution cumulative distribution function of *cdf* with *m* rows and *n* columns. Use this typically to compute the critical values of the distribution

```
Print StatsInvFriedmanCDF(1-alpha,n,m,0,1)
```

where alpha is the significance level of the associated test.

The complexity of the computation of Friedman CDF is on the order of $(n!)^m$. For nonzero values of *useTable*, searches are limited to the built-in table for distribution values. If *n* and *m* are not in the table the calculation may still proceed according to the *method*.

method	What It Does
0	Exact computation(slow, not recommended).
1	Chi-square approximation.
2	Monte-Carlo approximation (slow).
3	Use built-in table only and return a NaN if not in table.

For large *m* and *n,* consider using the Chi-squared or the Iman and Davenport approximations. To abort execution, press the **User Abort Key Combinations**.

Note: Table values are different from computed values for both methods. Table values use more conservative criteria than computed values. Table values are more consistent with published values because the Friedman distribution is a highly irregular function with multiple steps of arbitrary sizes. The standard for published tables provides the X value of the next vertical transition to the one on which the specified P is found.

Precomputed tables use these values:

n	*m*
3	2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
4	2, 3, 4, 5, 6, 7, 8, 9
5	2, 3, 4, 5, 6

n	*m*
6	2, 3, 4, 5
7	2, 3, 4
8	2, 3
9	2, 3

References

Iman, R.L., and J.M. Davenport, Approximations of the critical region of the Friedman statistic, *Comm. Statist.*, *A9*, 571-595, 1980.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFriedmanCDF** and **StatsFriedmanTest**.

StatsInvGammaCDF

`StatsInvGammaCDF(`*`cdf`*`, μ, σ, γ)`

The StatsInvGammaCDF function returns the inverse of the gamma cumulative distribution function. There is no closed form expression for the inverse gamma distribution; it is evaluated numerically.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGammaCDF** and **StatsGammaPDF**.

StatsInvGeometricCDF

`StatsInvGeometricCDF(`*`cdf`*`, `*`p`*`)`

The StatsInvGeometricCDF function returns the inverse of the geometric cumulative distribution function

$$x = \frac{\ln(1-cdf)}{\ln(1-p)} - 1$$

where *p* is the probability of success in a single trial and *x* is the number of trials.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGeometricCDF** and **StatsGeometricPDF**.

StatsInvKuiperCDF

`StatsInvKuiperCDF(`*`cdf`*`)`

The StatsInvKuiperCDF function returns the inverse of Kuiper cumulative distribution function.

There is no closed form expression. It is mapped to the range of 0.4 to 4, with accuracy of 1e-10.

References

See in particular Section 14.3 of

Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsKuiperCDF**.

StatsInvLogisticCDF

`StatsInvLogisticCDF(`*`cdf`*`, `*`a`*`, `*`b`*`)`

The StatsInvLogisticCDF function returns the inverse of the logistic cumulative distribution function

$$x = a + b\log\left(\frac{cdf}{1-cdf}\right).$$

where the scale parameter *b*>0 and the shape parameter is *a*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogisticCDF** and **StatsLogisticPDF** functions.

StatsInvLogNormalCDF

`StatsInvLogNormalCDF(`*`cdf`*`, `*`sigma`*`, `*`theta`*`, `*`mu`*`)`

The StatsInvLogNormalCDF function returns the numerically evaluated inverse of the lognormal cumulative distribution function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogNormalCDF** and **StatsLogNormalPDF** functions.

StatsInvMaxwellCDF

`StatsInvMaxwellCDF(`*`cdf`*`, `*`k`*`)`

The StatsInvMaxwellCDF function returns the evaluated numerically inverse of the Maxwell cumulative distribution function. There is no closed form expression.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsMaxwellCDF** and **StatsMaxwellPDF** functions.

StatsInvMooreCDF

`StatsInvMooreCDF(`*`cdf`*`, `*`N`*`)`

The StatsInvMooreCDF function returns the inverse cumulative distribution function for Moore's R*, which is used as a critical value in nonparametric version of the Rayleigh test for uniform distribution around the circle. It supports the range $3 \le N \le 120$ and does not change appreciably for $N > 120$.

The inverse distribution is computed from polynomial approximations derived from simulations and should be accurate to approximately three significant digits.

References

Moore, B.R., A modification of the Rayleigh test for vector data, *Biometrica*, *67*, 175-180, 1980.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsCircularMeans** function.

StatsInvNBinomialCDF

`StatsInvNBinomialCDF(`*`cdf`*`, `*`k`*`, `*`p`*`)`

The StatsInvNBinomialCDF function returns the numerically evaluated inverse of the negative binomial cumulative distribution function. There is no closed form expression.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNBinomialCDF** and **StatsNBinomialPDF** functions.

StatsInvNCChiCDF

`StatsInvNCChiCDF(`*`cdf`*`, `*`n`*`, `*`d`*`)`

The StatsInvNCChiCDF function returns the inverse of the noncenteral chi-squared cumulative distribution function. It is computationally intensive because the inverse is computed numerically and involves multiple evaluations of the noncentral distribution, which is evaluated from a series expansion.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCChiCDF**, **StatsNCChiPDF**, **StatsChiCDF**, and **StatsChiPDF** functions.

StatsInvNCFCDF

`StatsInvNCFCDF(`*`cdf`*`, `*`n1`*`, `*`n2`*`, `*`d`*`)`

The StatsInvNCFCDF function returns the numerically evaluated inverse of the cumulative distribution function of the noncentral F distribution. *n1* and *n2* are the shape parameters and *d* is the noncentrality measure. There is no closed form expression for the inverse.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCFCDF** and **StatsNCFPDF** functions.

StatsInvNormalCDF

`StatsInvNormalCDF(`*`cdf`*`, `*`m`*`, `*`s`*`)`

The StatsInvNormalCDF function returns the numerically computed inverse of the normal cumulative distribution function. There is no closed form expression.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNormalCDF** and **StatsNormalPDF** functions.

StatsInvParetoCDF

`StatsInvParetoCDF(`*`cdf`*`, `*`a`*`, `*`c`*`)`

The StatsInvParetoCDF function returns the inverse of the Pareto cumulative distribution function

$$x = \frac{a}{\left(1 - cdf\right)^{(1/c)}}$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsParetoCDF** and **StatsParetoPDF** functions.

StatsInvPoissonCDF

`StatsInvPoissonCDF(`*`cdf`*`, λ)`

The StatsInvPoissonCDF function returns the numerically evaluated inverse of the Poisson cumulative distribution function. There is no closed form expression for the inverse Poisson distribution.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPoissonCDF** and **StatsPoissonPDF** functions.

StatsInvPowerCDF

`StatsInvPowerCDF(`*`cdf`*`, `*`b`*`, `*`c`*`)`

The StatsInvPowerCDF function returns the inverse of the Power Function cumulative distribution function

$$x = b \,/\, cdf^{(1/c)}.$$

where the scale parameter *b* and the shape parameter *c* satisfy *b*,*c*>0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPowerCDF**, **StatsPowerPDF** and **StatsPowerNoise** functions.

StatsInvQCDF

`StatsInvQCDF(`*`cdf`*`, `*`r`*`, `*`c`*`, `*`df`*`)`

The StatsInvQCDF function returns the critical value of the Q cumulative distribution function for *r* the number of groups, *c* the number of treatments, and *df* the error degrees of freedom (*df*=*r***c**(*n*-1) with sample size *n*).

Details

The Q distribution is the maximum of several Studentized range statistics. For a simple Tukey test, use *r*=1.

Examples

The critical value for a Tukey test comparing 5 treatments with 6 samples and 0.05 significance is:

```
Print StatsInvQCDF(1-0.05,1,5,5*(6-1))
```

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTukeyTest** function.

StatsInvQpCDF

`StatsInvQpCDF(`*`ng`*`, `*`nt`*`, `*`df`*`, `*`alpha`*`, `*`side`*`, `*`sSizeWave`*`)`

The StatsInvQpCDF function returns the critical value of the Q' cumulative distribution function for *ng* the number of groups, *nt* the number of treatments, and *df* the error degrees of freedom. *side*=1 for upper-tail or *side*=2 for two-tailed critical values.

sSizeWave is an integer wave of *ng* columns and *nt* rows specifying the number of samples in each treatment. If *sSizeWave* is a null wave (`$""`) StatsInvQpCDF computes the number of samples from *df*=*ng***nt**(*n*-1) with *n* truncated to an integer.

Details

StatsInvQpCDF is a modified Q distribution typically used with Dunnett’s test, which compares the various means with the mean of the control group or treatment.

StatsInvQpCDF differs from other StatsInv*XXX* functions in that you do not specify a *cdf* value for the inverse (usually 1-*alpha* for the critical value). Here *alpha* selects one- or two-tailed critical values.

It is computationally intensive, taking longer to execute for smaller *alpha* values.

Examples

The critical value for a Dunnett test comparing 4 treatments with 4 samples and (upper tail) 0.05 significance is:

```
// n=4 because 12=1*4*(4-1).
Print StatsInvQpCDF(1,4,12,0.05,1,$"")
  2.28734
```

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsDunnettTest** and **StatsInvQCDF** functions.

StatsInvRayleighCDF

`StatsInvRayleighCDF(`*`cdf`*` [, `*`s`*` [, `*`m`*`]])`

The **`StatsInvRayleighCDF`** function returns the inverse of the Rayleigh cumulative distribution functiongiven by

$$x = \mu + \sigma\sqrt{-2\ln(1-cdf)},$$

with defaults *s*=1 and *m*=0. It returns NaN for *s* ≤ 0 and zero for x ≤ *m*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRayleighCDF** and **StatsRayleighPDF** functions.

StatsInvRectangularCDF

StatsInvRectangularCDF(*cdf*, *a*, *b*)

The StatsInvRectangularCDF function returns the inverse of the rectangular (uniform) cumulative distribution function

$$x = a + cdf(b - a), \qquad a < b.$$

where $a< b$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRectangularCDF** and **StatsRectangularPDF** functions.

StatsInvSpearmanCDF

StatsInvSpearmanCDF(*cdf*, *N*)

The StatsInvSpearmanCDF function returns the inverse cumulative distribution function for Spearman's *r*, which is used as a critical value in rank correlation tests.

The inverse distribution is computed by finding the value of *r* for which it attains the *cdf* value. The result is usually lower than in published tables, which are more conservative when the first derivative of the distribution is discontinuous.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRankCorrelationTest**, **StatsSpearmanRhoCDF**, and **StatsKendallTauTest** functions.

StatsInvStudentCDF

StatsInvStudentCDF(*cdf*, *n*)

The StatsInvStudentCDF function returns the numerically evaluated inverse of Student cumulative distribution function. There is no closed form expression.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentCDF** and **StatsStudentPDF** functions.

StatsInvTopDownCDF

StatsInvTopDownCDF(cdf, N)

The StatsInvTopDownCDF function returns the inverse cumulative distribution function for the top-down distribution. For $3 \le N \le 7$ it uses a lookup table CDF and returns the next higher value of *r* for which the distribution value is larger than *cdf*. For $8 \le N \le 50$ it returns the nearest value for which the built-in distribution returns *cdf*. For N>50 it returns the scaled normal approximation.

Tabulated values are from Iman and Conover who pick as the critical value the very first transition of the distribution following the specified *cdf* value. These tabulated values tend to be slightly higher than calculated values for $7<N<15$.

References

Iman, R.L., and W.J. Conover, A measure of top-down correlation, *Technometrics*, *29*, 351-357, 1987.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRankCorrelationTest** and **StatsTopDownCDF** functions.

StatsInvTriangularCDF

StatsInvTriangularCDF(*cdf*, *a*, *b*, *c*)

The StatsInvTriangularCDF function returns the inverse of the triangular cumulative distribution function

$$x = \begin{cases} a + \sqrt{cdf(b-a)(c-a)} & 0 \le cdf \le \dfrac{c-a}{b-a} \\ b - \sqrt{(1-cdf)(b-a)(b-c)} & \dfrac{c-a}{b-a} \le cdf \le 1 \end{cases}$$

where $a<c<b$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTriangularCDF** and **StatsTriangularPDF** functions.

StatsInvUSquaredCDF

`StatsInvUSquaredCDF(`*`cdf`*`, `*`n`*`, `*`m`*`, `*`method`*`, `*`useTable`*`)`

The StatsInvUSquaredCDF function returns the inverse of Watson's U^2 cumulative distribution function integer sample sizes *n* and *m*. Use a nonzero value for *useTable* to search a built-in table of values. If *n* and *m* cannot be found in the table, it will proceed according to *method*:

method	**What It Does**
0	Exact computation using Burr algorithm (could be slow).
1	Tiku approximation using chi-squared.
2	Use built-in table only and return a NaN if not in table.

For large *n* and *m*, consider using the Tiku approximation. To abort execution, press the **User Abort Key Combinations**. Because *n* and *m* are interchangeable, *n* should always be the smaller value. For *n*>8 the upper limit in the table matched the maximum that can be computed using the Burr algorithm. There is no point in using method 0 with *m* values exceeding these limits.

The inverse is obtained from precomputed tables of Watson's U^2 (see **StatsUSquaredCDF**).

Note: Table values are different from computed values. These values use more conservative criteria than computed values. Table values are more consistent with published values because the U^2 distribution is a highly irregular function with multiple steps of arbitrary sizes. The standard for published tables provides the X value of the next vertical transition to the one on which the specified P is found. See **StatsInvFriedmanCDF**.

References

Burr, E.J., Small sample distributions of the two sample Cramer-von Mises' W^2 and Watson's U^2, *Ann. Mah. Stat. Assoc.*, *64*, 1091-1098, 1964.

Tiku, M.L., Chi-square approximations for the distributions of goodness-of-fit statistics, *Biometrica*, *52*, 630-633, 1965.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWatsonUSquaredTest** and **StatsUSquaredCDF** functions.

StatsInvVonMisesCDF

`StatsInvVonMisesCDF(`*`cdf`*`, `*`a`*`, `*`b`*`)`

The StatsInvVonMisesCDF function returns the numerically evaluated inverse of the von Mises cumulative distribution function where the value of the integral of the distribution matches *cdf*. Parameters are as for **StatsVonMisesCDF**.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsVonMisesPDF** and **StatsVonMisesNoise** functions.

StatsInvWeibullCDF

```
StatsInvWeibullCDF(cdf, m, s, g)
```

The StatsInvWeibullCDF function returns the inverse of the Weibull cumulative distribution function

$$x = \mu + \sigma\left[-\ln\left(1 - cdf\right)\right]^{1/\gamma}.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWeibullCDF** and **StatsWeibullPDF** functions.

StatsJBTest

```
StatsJBTest [flags] srcWave
```

The StatsJBTest operation performs the Jarque-Bera test on *srcWave*. Output is to the W_JBResults wave in the current data folder.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsJBTest computes the Jarque-Bera statistic

$$JB = \frac{n}{6}\left(S^2 + \frac{K^2}{4}\right),$$

where *S* is the skewness, *K* is the kurtosis, and *n* is the number of points in the input wave. We can express *S* and *K* terms of the *j*th moment of the distribution for n samples X_i

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^j$$

as

$$S = \frac{\mu_3}{\left(\mu_2\right)^{3/2}},$$

and

$$K = \frac{\mu_4}{\left(\mu_2\right)^2} - 3.$$

The Jarque-Bera statistic is asymptotically distributed as a Chi-squared with two degrees of freedom. For values of *n* in the range [7,2000] the operation provides critical values obtained from Monte-Carlo simulations. For further details or if you would like to run your own simulation to obtain critical values for other values of *n*, use the JarqueBeraSimulation example experiment.

StatsJBTest reports the number of finite data points, skewness, kurtosis, Jarque-Bera statistic, asymptotic critical value, and the critical value obtained from Monte-Carlo calculations as appropriate; it ignores NaNs and INFs.

References

Jarque, C., and A. Bera, A test of normality of observations and regression residuals, *International Statistical Review*, *55*, 163-172, 1987.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsKSTest**, **WaveStats**, and **StatsCircularMoments**.

StatsKDE

```
StatsKDE [flags] srcWave
```

StatsKDE can be used to estimate a PDF from original data distribution. Unlike histograms, this method produces a smooth result as it constructs the PDF from a normalized superposition of kernel functions.

The StatsKDE operation was added in Igor Pro 7.00.

Flags

/BWM=*m* Sets the bandwidth selection method.

- *m*=0: User-specified via /H flag
- *m*=1: Silverman
- *m*=2: Scott
- *m*=3: Bowmann and Azzalini

/DEST=*destWave* Specifies the output destination. Creates a real wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-72 for details.

/FREE Makes the destination wave (specified by /DEST) a free wave.

/H=bw Specifies a fixed user-defined bandwidth.

/KT=*kernel* Specifies the kernel type.

- *kernel*=1: Epanechnikov
- *kernel*=2: Bi-weight
- *kernel*=3: Tri-weight
- *kernel*=4: Triangular
- *kernel*=5: Gaussian
- *kernel*=6: Rectangular

/Q No results printed in the history area. In the case of univariate KDE this flags suppresses the printing of the bandwidth value.

/S={*x0*,*dx*,*xn*} Specifies the range of the output starting from x=*x0* to x=*xn* in increments of *dx*.

/Z	Ignores errors. V_flag is set to zero if there are no errors.

Details

StatsKDE estimates the PDF of a distribution of values using a smoothing kernel and a bandwidth parameter which affects the degree of smoothing.

Theory suggests that the Epanechnikov kernel is the most efficient but many expressions for the optimal bandwidth are derived for the Gaussian kernel. If *srcWave* contains N points and the requested output (/S flag) has M points then the computational complexity is O(NM). For large problems it may be beneficial to use the Gaussian kernel via the FastGaussTransform operation.

References

Wand M.P. and Jones M.C. (1995) Monographs on Statistics and Applied Probability, London: Chapman and Hall

Bowman, A.W., and Azzalini, A. (1997), Applied Smoothing Techniques for Data Analysis, London: Oxford University Press.

See Also

Statistics on page III-383, **Histogram**, **FastGaussTransform**

StatsKendallTauTest

StatsKendallTauTest [***flags***] ***wave1*** [, ***wave2***]

The StatsKendallTauTest operation performs the nonparametric Mann-Kendall test, which computes a correlation coefficient τ (similar to Spearman's correlation) from the relative order of the ranks of the data. Output is to the W_StatsKendallTauTest wave in the current data folder.

Flags

/Q	No results printed in the history area.	
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed.	
	k=0:	Normal with dialog (default).
	k=1:	Kills with no dialog.
	k=2:	Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.	

Details

Inputs may be a pair of XY (1D) waves of any real numeric type or a single 1D wave, which is equivalent to using a pair of XY waves where the X wave is monotonically increasing function of the point number. StatsKendallTauTest ignores wave scaling.

Kendall's τ is 1 for a monotonically increasing input and -1 for monotonically decreasing input. The significance of the test is computed from the normal approximation

$$Var(\tau) = \frac{4n+10}{9n(n-1)},$$

where *n* is the number of data points in each wave. The significance is expressed as a P-value for the null hypothesis of no correlation.

References

Kendall, M.G., *Rank Correlation Methods*, 3rd ed., Griffin, London, 1962.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsRankCorrelationTest**.

For small values of *n* you can compute the exact probability using the procedure WM_KendallSProbability().

StatsKSTest

StatsKSTest [***flags***] ***srcWave*** [, ***distWave***]

The StatsKSTest operation performs the Kolmogorov-Smirnov (KS) goodness-of-fit test for two continuous distributions. The first distribution is *srcWave* and the second distribution can be expressed either as the optional wave *distWave* or as a user function with /CDFF. Output is to the W_KSResults wave in the current data folder.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/CDFF=*func* Specifies a user function expressing the cumulative distribution function. See Details.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The Kolmogorov-Smirnov (KS) goodness-of-fit test applies only to continuous distributions and cases where the compared distribution (expressed as a user function) is completely specified without estimating parameters from the data. It compares the cumulative distribution function (CDF) of two distributions and sets the test statistic D to the largest difference between the CDFs. Because CDFs are in the range [0,1], D is also bound by this range.

When specifying the distributions with two waves, StatsKSTest first sorts the data in the waves and then computes the CDFs and D. You can also specify one of the distributions with a user function. For example, the following function tests if the data in *srcWave* is normally distributed with zero mean and stdv=5:

```
Function GetUserCDF(inX) : CDFFunc
    Variable inX
    return StatsNormalCDF(inX,0,5)
End
```

The “`: CDFFunc`” designation, which requires Igor7 or later, tells Igor to make the function accessible from the Kolmogorov-Smirnov Test dialog.

Outputs are the number of elements, the KS statistic D, and the critical value. When both distributions are specified by waves, the number of elements is the weighted value (n1*n2)/(n1+n2).

References

Critical values are based on:

Birnbaum, Z. W., and Fred H. Tingey, One-sided confidence contours for probability distribution functions, *The Annals of Mathematical Statistics*, *22*, 592–596, 1951.

A statistically more powerful modification of the classic KS test can be found in:

Khamis, H.J., The two-stage delta-corrected Kolmogorov-Smirnov test, *Journal of Applied Statistics*, *27*, 439-450, 2000.

StatsKSTest implements the original KS test. The difficulty in implementing the modified tests for all the cases defined by Stephens is in obtaining the critical values which have to be derived by time consuming Monte-Carlo simulations.

Critiques can be found in:

D’Agostino, R.B., and M. Stephens, eds., *Goodness-Of-Fit Techniques*, Marcel Dekker, New York, 1986.

NIST/SEMATECH, Kolmogorov-Smirnov Goodness-of-Fit Test, in *NIST/SEMATECH e-Handbook of Statistical Methods*, <`http://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm`>, 2005.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsJBTest**, **WaveStats**, and **StatsCircularMoments**.

StatsKuiperCDF

`StatsKuiperCDF(V)`

The StatsKuiperCDF function returns the Kuiper cumulative distribution function

$$F(V) = 1 - 2\sum_{j=1}^{\infty}\left(4j^2V^2 - 1\right)\exp\left(-2j^2V^2\right).$$

Accuracy is on the order of 1e-15. It returns 0 for values of *V*<0.4 or 1 for *V*>3.1.

References

See in particular Section 14.3 of

Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvKuiperCDF**.

StatsKWTest

`StatsKWTest [flags] [wave1, wave2,… wave100]`

The StatsKWTest operation performs the nonparametric Kruskal-Wallis test which tests variances using the ranks of the data. Output is to the W_KWTestResults wave in the current data folder.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/E Computes the exact P-value using the Klotz and Teng algorithm, which may require long computation times for large data sets. You can stop the calculation by pressing the **User Abort Key Combinations** after which all remaining results remain valid and the exact P-value is set to NaN.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Inputs are two or more 1D numerical waves (one for each group of samples). Use NaNs for missing data or use waves with different number of points.

StatsKWTest always computes the critical values using both the Chi-squared and Wallace approximations. If appropriate (small enough data set) you can also use /E to obtain the exact P value. When the calculation involves many waves or many data points the calculation of the exact critical value can be very lengthy. All the results are saved in the wave W_KWTestResults in the current data folder and are optionally displayed in a table (/T). The wave contains the following information:

H_0 for the Kruskal-Wallis test is that all input waves are the same. If the test fails and the input consisted of more than two waves, there is no indication for possible agreement between some of the waves. See **StatsNPMCTest** for further analysis.

Row	Data
0	Number of groups
1	Number of valid data points (excludes NaNs)
2	Alpha
3	Kruskal-Wallis Statistic H
4	Chi-squared approximation for the critical value Hc
5	Chi-squared approximation for the P value
6	Wallace approximation for the critical value Hc
7	Wallace approximation for the P value
8	Exact P value (requires /E)

V_flag will be set to -1 for any error and to zero otherwise.

References

Klotz, J.H., *Computational Approach to Statistics*.

Klotz, J., and Teng, J., One-way layout for counts and the exact enumeration of the Kruskal-Wallis H distribution with ties, *J. Am. Stat. Assoc*, *72*, 165-169, 1977.

Wallace, D.L., Simplified Beta-Approximation to the Kruskal-Wallis H Test, *J. Am. Stat. Assoc.*, *54*, 225-230, 1959.

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWilcoxonRankTest**, **StatsNPMCTest**, and **StatsAngularDistanceTest**.

StatsLinearCorrelationTest

StatsLinearCorrelationTest [***flags***] ***waveA***, ***waveB***

The StatsLinearCorrelationTest operation performs correlation tests on *waveA* and *waveB*, which must be real valued numeric waves and must have the same number of points. Output is to the W_StatsLinearCorrelationTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/CI	Computes confidence intervals for the correlation coefficient.
/Q	No results printed in the history area.
/RHO=*rhoValue*	Tests hypothesis that the correlation has a nonzero value $\|r\| \le 1$.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors.

Details

The linear correlation tests start by computing the linear correlation coefficient for the *n* elements of both waves:

$$r = \frac{\sum_{i=1}^{n} X_i Y_i - \frac{1}{n}\sum_{i=1}^{n} X_i \sum_{i=1}^{n} Y_i}{\sqrt{\left(\sum_{i=1}^{n} X_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} X_i\right)^2\right)\left(\sum_{i=1}^{n} Y_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} Y_i\right)^2\right)}}$$

Next it computes the standard error of the correlation coefficient

$$sr = \sqrt{\frac{1-r^2}{n-2}}$$

The basic test is for hypothesis H_0: the correlation coefficient is zero, in which case *t* and *F* statistics are applicable. It computes the statistics:

$$t = r / sr$$

and

$$F = \frac{1+|r|}{1-|r|},$$

and then the critical values for one and two tailed hypotheses (designated by t_{c1}, t_{c2}, F_{c1}, and F_{c2} respectively). Critical value for *r* are computed using

$$\mathrm{rc}_{\mathrm{i}} = \sqrt{\frac{t_c^2}{t_c^2 + n}}$$

where *i* takes the values 1 or 2 for one and two tailed hypotheses. Finally, it computes the power of the test at the alpha significance level for both one and two tails (Power1 and Power2).

If you use /RHO it uses the Fisher transformation to compute

$$\mathrm{FisherZ} = \frac{1}{2}\ln\left(\frac{1+r}{1\text{-}r}\right)$$

$$\mathrm{zeta} = \frac{1}{2}\ln\left(\frac{1+\rho}{1\text{-}\rho}\right)$$

the standard error approximation

$$\mathrm{sigmaZ} = \sqrt{\frac{1}{n-3}},$$

$$\mathrm{Zstatistic} = \frac{FisherZ - zeta}{sigmaZ},$$

and the critical values from the normal distribution Z_{ci}.

The confidence intervals are calculated differently depending on the hypothesis for the value of the correlation coefficient. If /RHO is not used the confidence intervals are computed using the critical value F_{c2}, otherwise they are computed using the critical Z_{c2} and sigmaZ.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCircularCorrelationTest**, **StatsMultiCorrelationTest**, and **StatsRankCorrelationTest**.

StatsLinearRegression

`StatsLinearRegression` `[`***`flags`***`] [`***`wave0, wave1,`***`…]`

The StatsLinearRegression operation performs regression analysis on the input wave(s). Output is to the W_StatsLinearRegression wave in the current data folder or optionally to a table. Additionally, the M_DunnettMCElevations, M_TukeyMCSlopes, and M_TukeyMCElevations waves may be created as specified.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/B=*beta0* Tests the hypothesis that the slope *b*= *beta0* (default is 0). The results are expressed by the t-statistic, which can be compared with the tc value for the two-tailed test. Get the critical value for a one-tailed test using `StatsStudentCDF(1-alpha,N-2)`. It does not work with /MYVW.

/BCIW Computes two confidence interval waves for the high side and the low side of the confidence interval. The new waves are named with _CH and _CL suffixes respectively appended to the Y wave name and are created in the current data folder. For multiple runs a numeric suffix will also be appended to the names.

/BPIW[=*mAdditional*]

Computes prediction interval waves for the high side and the low side of the confidence interval on a single additional measurement (default). Use *mAdditional* to specify additional measurements. The new waves are named with _PH and _PL suffixes respectively appended to the Y wave name and are created in the current data folder. For multiple runs a numeric suffix will also be appended to the names.

/DET=*controlIndex* Performs Dunnett’s multicomparison test for the elevations. The test requires more than two Y waves for regression, the test for the slopes should not reject the equal slope hypothesis, and the test for the elevations should reject the equal elevation hypothesis. *controlIndex* is the zero-based index of the Y wave representing the control (X waves do not count in the index specification). The test compares the elevation of every Y wave with the specified control.

Output is to the M_DunnettMCElevations wave in the current data folder or optionally to a table. For every Y wave and control Y wave combination, the results include SE, q, q' (shown as qp), and the conclusion with 1 to accept the hypothesis of equal elevations or 0 to reject it. Use /TAIL to determine the critical value and the sense of the test. If you use /TUK you will also get the Tukey test for the set of elevations.

/MYVW={*xWave*, *yWave*}

Specifies that the input consists of multiple Y values for each X value. It ignores all other inputs and the results are appropriate only for multiple Y values at each X point.

yWave is a 2D wave of values arranged in columns. Use NaNs for padding where rows do not have the same number of entries as others. It will use the X scaling of *yWave* when *xWave* is null, `/MYVW={*,`*`yWave`*`}`.

It first tests the hypothesis (H_0) that the population regression is linear in an analysis of variance calculation. It generates results 1-7 (see Details) as well as: Among Groups SS, Among Groups DF, Within Groups SS, Within Groups DF, Deviations from Linearity SS, Deviations from Linearity DF, F statistic defined by the ratio of Deviation from Linearity MS to Within Groups MS, and the critical value Fc.

Next, it tests the hypothesis that the slope beta=0. If the original H_0 was accepted, the new F statistic=regressionMS/residualMS. Otherwise the with the critical F=regressionMS/WithinGroupsMS with a corresponding critical value. Finally, it reports the values of the coefficient of determination r2 and the standard error of the estimate S_{YX}.

/PAIR Specifies that the input waves are XY pairs, where each pair must be an X wave followed by a Y wave.

/Q No results printed in the history area.

/RTO Reflects the regression through the origin.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).
k=1: Kills with no dialog.
k=2: Disables killing.

/TAIL=*tCode* Sets the sense of the test when applying Dunnett's test (see /DET). *tCode* is 1 or 2 for a one-tail critical value and 4 for a two-tail critical value.

/TUK Performs a Tukey-type test on multiple regressions on two or more Y waves. There are two possible Tukey-type tests: The first is performed if the hypothesis of equal slopes is rejected. It compares all combinations of two Y waves to identify if some of the waves have equal slopes. Output is to the M_TukeyMCSlopes wave in the current data folder or optionally to a table. For every Y wave pair, the results include the difference between slopes (absolute value), q, the critical value qc, and the conclusion set to 1 for accepting the equality of the pair of slopes or 0 for rejecting the hypothesis.

The second Tukey-type test is performed if all the slopes are the same but the elevations are not. The test (see /DET) compares all possible pairs of elevations to determine which satisfy the hypothesis of equality. Output is to the M_TukeyMCElevations wave in the current data folder.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors.

Details

Inputs may consist of Y waves or XY wave pairs. If X data are not used, the X values are inferred from the Y wave scaling. For multiple waves where only some have pairs, use the /PAIR flag and enter * in each place where the X values should be computed.

For each input StatsLinearRegression calculates:

1. Least squares regression line y=a+b*x.
2. Mean value of X: xBar.
3. Mean value of Y: yBar.
4. Sum of the squares (x_i-xBar)^2.
5. Sum of the squares (y_i-yBar)^2.
6. Sum of the product ($x_i y_i$-xyBar).
7. Standard error of the estimate $S_{YX}^2 = \frac{\sum\left(Y_i - \hat{Y}_i\right)^2}{n-2}$.
8. F statistic for the hypothesis beta=0.
9. Critical F value Fc.
10. Coefficient of determination r2.
11. Standard error of the regression coefficient S_b.
12. t-statistic for the hypothesis beta=*beta0*, NaN if /B is not specified.
13. Critical value tc for the t-statistic above (used to calculate L1 and L2).
14. Lower confidence interval boundary (L1) for the regression coefficient.
15. Upper confidence interval boundary (L2) for the regression coefficient.

For two Y waves with the same slope, it computes a common slope (bc) and then tests the equality of the elevations (a). In both cases it computes a t-statistic and compares it with a critical value. If the elevations are also the same then it computes the common elevation (ac) and the pooled means of X and Y in (xp) and (yp).

For more than two Y waves it computes:

$$A_c = \sum_{j=1}^{W} A_j; \qquad A_j \equiv \sum x_i^2 = \sum_{i=0}^{n_j-1} X_i^2 - \frac{1}{n_j}\left(\sum_{i=0}^{n_j-1} X_i\right)^2$$

$$B_c = \sum_{j=1}^{W} B_j; \qquad B_j \equiv \sum xy = \sum_{i=0}^{n_j-1} XY - \frac{1}{n_j}\left(\sum_{i=0}^{n_j-1} X_i\right)\left(\sum_{i=0}^{n_j-1} Y_i\right)$$

$$C_c = \sum_{j=1}^{W} C_j; \qquad C_j \equiv \sum y^2 = \sum_{i=0}^{n_j-1} Y_i^2 - \frac{1}{n_j}\left(\sum_{i=0}^{n_j-1} Y_i\right)^2$$

$$SSp = \sum_{j=1}^{W} C_j - \frac{B_j^2}{A_j}$$

$$SSc = Cc - \frac{B_c^2}{A_c^2}$$

$$SSt = \sum_{j=1}^{W}\sum_{i=0}^{n_j} Y_{ji}^2 - \frac{1}{N}\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} Y_{ji}\right)^2 - \frac{\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} X_{ji}Y_{ji} - \frac{1}{N}\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} X_{ji}\right)\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} Y_{ji}\right)\right)^2}{\sum_{j=1}^{W}\sum_{i=0}^{n_j} X_{ji}^2 - \frac{1}{N}\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} X_{ji}\right)^2}$$

$$DFp = \sum_{j=1}^{W} \left(n_i - 2\right)$$

$$DFt = \sum_{j=1}^{W} n_i - 2$$

Here *W* is the number of Y-waves and $N = \sum_{j=1}^{W} n_j$ is the total number of data points in all Y-waves.

The test statistic F for equality of slopes is given by:

$$F = \left(\frac{SSc - SSp}{numWaves - 1} \right) \Big/ \frac{SSp}{DFp}.$$

Fc is the corresponding critical value.

Output is to the W_LinearRegressionMC wave in the current data folder.

V_flag is set to -1 for any error and to zero otherwise.

S_waveNames is set to a semicolon-separated list of the names of the waves created by the operation.

References

See, in particular, Chapter 18 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; curvefit.

StatsLogisticCDF

`StatsLogisticCDF(`*`x`*`, `*`a`*`, `*`b`*`)`

The StatsLogisticCDF function returns the logistic cumulative distribution function

$$F(x;a,b) = \frac{1}{1 + \exp\left(-\frac{x-a}{b} \right)}.$$

where the scale parameter *b*>0 and the shape parameter is *a*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogisticPDF** and **StatsInvLogisticCDF** functions.

StatsLogisticPDF

`StatsLogisticPDF(`*`x`*`, `*`a`*`, `*`b`*`)`

The StatsLogisticPDF function returns the logistic probability distribution function

$$f(x;a,b) = \frac{\exp\left(-\frac{x-a}{b} \right)}{b\left[1 + \exp\left(-\frac{x-a}{b} \right) \right]^2},$$

where the scale parameter *b*>0 and the shape parameter is *a*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogisticCDF** and **StatsInvLogisticCDF** functions.

StatsLogNormalCDF

```
StatsLogNormalCDF(x, σ [, θ, μ])
```

The StatsLogNormalCDF function returns the lognormal cumulative distribution function

$$F(x;\sigma,\theta,\mu)=\frac{1}{\sigma\sqrt{2\pi}}\int_0^x \frac{1}{t-\theta}\exp\left\{-\left[\ln\left(\frac{t-\theta}{\mu}\right)\right]^2 \Big/ 2\sigma^2\right\}dt,$$

for $x > \theta$ and σ, μ>0. The standard lognormal distribution is for θ=0 and μ=1, which are the optional parameter defaults.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogNormalPDF** and **StatsInvLogNormalCDF** functions.

StatsLogNormalPDF

```
StatsLogNormalPDF(x, σ [, θ, μ])
```

The StatsLogNormalPDF function returns the lognormal probability distribution function

$$f(x;\sigma,\theta,\mu)=\frac{1}{\sigma\sqrt{2\pi}}\frac{1}{x-\theta}\exp\left\{-\left[\ln\left(\frac{x-\theta}{\mu}\right)\right]^2 \Big/ 2\sigma^2\right\},$$

for $x > \theta$ and σ, μ > 0, where θ is the location parameter, μ is the scale parameter and, σ is the shape parameter. The standard lognormal distribution is for θ=0 and μ=1, which are the optional parameter defaults.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogNormalCDF** and **StatsInvLogNormalCDF** functions.

Reference

The expression for the PDF follows the NIST definition at: https://www.itl.nist.gov/div898/handbook/eda/section3/eda3669.htm. Note that alternate definitions use μ differently.

StatsMaxwellCDF

```
StatsMaxwellCDF(x, k)
```

The StatsMaxwellCDF function returns the Maxwell cumulative distribution function

$$F(x;k)=gammp\left(\frac{3}{2},\frac{kx^2}{2}\right), \qquad x>0.$$

where **gammp** is the regularized incomplete gamma function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsMaxwellPDF** and **StatsInvMaxwellCDF** functions.

StatsMaxwellPDF

StatsMaxwellPDF(*x*, *k*)

The StatsMaxwellPDF function returns Maxwell's probability distribution function

$$f(x;k) = \sqrt{\frac{2}{\pi}} k^{3/2} x^2 \exp\left(-\frac{kx^2}{2}\right), \qquad x > 0.$$

The Maxwell distribution describes, for example, the speed distribution of molecules in an ideal gas.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsMaxwellCDF** and **StatsInvMaxwellCDF** functions.

StatsMedian

StatsMedian(*waveName*)

The StatsMedian function returns the median value of a numeric wave *waveName*, which must not contain NaNs.

Example

```
Make/N=5 sample1={1,2,3,4,5}
Print StatsMedian(sample1)
3
Make/N=6 sample2={1,2,3,4,5,6}
Print StatsMedian(sample2)
3.5
```

See Also

Chapter III-12, **Statistics** for a function and operation overview

median, **WaveStats**, **StatsQuantiles**

StatsMooreCDF

StatsMooreCDF(*x*, *N*)

The StatsMooreCDF function returns the cumulative distribution function for Moore's R*, which is used in a nonparametric version of the Rayleigh test for uniform distribution around the circle. It supports the range 3 ≤ *N* ≤ 120 and does not change appreciably for *N*>120.

The distribution is computed from polynomial approximations derived from simulations and should be accurate to approximately three significant digits.

References

Moore, B.R., A modification of the Rayleigh test for vector data, *Biometrica*, *67*, 175-180, 1980.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsCircularMeans** function.

StatsMultiCorrelationTest

StatsMultiCorrelationTest [*flags*] *corrWave*, *sizeWave*

The StatsMultiCorrelationTest operation performs various tests on multiple correlation coefficients. Inputs are two 1D waves: *corrWave*, containing correlation coefficients, and *sizeWave*, containing the size (number of elements) of the corresponding samples. Although you can do all the tests at the same time, it rarely makes sense to do so.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/CON={*controlRow*,*tails*}	

Performs a multiple comparison test using the *controlRow* element of *corrWave* as a control. It is one- or two-tailed test according to the tails parameter. Output is to the M_ControlCorrTestResults wave in the current data folder.

/CONT=*cWave* Performs a multiple contrasts test on the correlation coefficients. The contrasts wave, *cWave*, contains the contrast factor, c_i, entry for each of the *n* correlation coefficients r_i in *corrWave*, and satisfying the condition that the sum of the entries in *cWave* is zero. H_0 corresponds to

$$\sum_{i=0}^{n-1} c_i r_i = 0.$$

The test statistic *S* is

$$S = \frac{1}{\sqrt{\frac{c_i^2}{n_i - 3}}} \left| \sum_{i=0}^{n-1} c_i z_i \right|,$$

where z_i is the Fisher z transform of the correlation coefficient r_i:

$$z_i = \frac{1}{2} \ln\left(\frac{1 + r_i}{1 - r_i} \right).$$

It produces the SE value, the contrast statistic S, and the critical value, which are labeled ContrastSE, ContrastS, and Contrast_Critical, respectively, in the W_StatsMultiCorrelationTest wave.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/TUK Performs a Tukey-type multi comparison testing between the correlation coefficients by comparing every possible combination of pairs of correlation coefficients, computing the difference in their z-transforms, the SE, and the *q* statistic:

$$q = \frac{\left| z_j - z_i \right|}{\sqrt{\frac{1}{2}\left(\frac{1}{n_i - 3} + \frac{1}{n_j - 3} \right)}}.$$

The critical value is computed from the *q* CDF (**StatsInvQCDF**) with degrees of freedom *numWaves* and infinity. Output is to the M_TukeyCorrTestResults wave in the current data folder or optionally to a table.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Without any flags, StatsMultiCorrelationTest computes χ^2 for the correlation coefficients and compares it with the critical value.

$$\chi^2 = \sum_{i=0}^{n-1} z_i^2\left(n_i - 3\right) - \frac{\left(\sum_{i=0}^{n-1} z_i\left(n_i - 3\right)\right)^2}{\sum_{i=0}^{n-1}\left(n_i - 3\right)},$$

where z_i is the Fisher's z transform of the correlation coefficients and n_i is the corresponding sample size. It computes the common correlation coefficient rw and its transform zw.

$$z_w = \frac{\sum_{i=0}^{n-1} z_i\left(n_i - 3\right)}{\sum_{i=0}^{n-1}\left(n_i - 3\right)}$$

These values are calculated even when not appropriate, such as when χ^2 exceeds the critical value and H_0 (all samples came from populations of identical correlation coefficients) is rejected.

The operation also computes ChiSquaredP (due to S.R. Paul), a different variant of χ^2 that is corrected for bias and should be compared with the same critical value. Output is to the W_StatsMultiCorrelationTest wave in the current data folder or optionally to a table.

References

See, in particular, Chapters 19 and 11 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsLinearCorrelationTest, **StatsCircularCorrelationTest**, **StatsDunnettTest**, **StatsTukeyTest**, **StatsInvQCDF**, and **StatsScheffeTest**.

StatsNBinomialCDF

`StatsNBinomialCDF(x, k, p)`

The StatsNBinomialCDF function returns the negative binomial cumulative distribution function

$$F(x;k,p) = Betai(k, x+1; p),$$

where **betai** is the regularized incomplete beta function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNBinomialPDF** and **StatsInvNBinomialCDF** functions.

StatsNBinomialPDF

`StatsNBinomialPDF(x, k, p)`

The StatsNBinomialPDF function returns the negative binomial probability distribution function

$$f(x;k,p) = \binom{x+k-1}{k-1} p^k (1-p)^x, \qquad x = 0,1,2...$$

where $\binom{a}{b}$ is the **binomial** function.

The binomial distribution expresses the probability of the *k*th success in the *x*+*k* trial for two mutually exclusive results (success and failure) and *p* the probability of success in a single trial.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNBinomialCDF** and **StatsInvNBinomialCDF** functions.

StatsNCChiCDF

`StatsNCChiCDF(x, n, d)`

The StatsNCChiCDF function returns the noncentral chi-squared cumulative distribution function

$$F(x;n,d) = \sum_{i=1}^{\infty} \exp(d/2) \frac{(d/2)^i}{i!} F_c(x;n+2i),$$

where *n*>0 corresponds to degrees of freedom, *d* ≥ 0 is the noncentrality parameter, and F_c is the central chi-squared distribution.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsChiCDF**, **StatsNCChiPDF**, and **StatsChiPDF** functions.

StatsNCChiPDF

`StatsNCChiPDF(x, n, d)`

The StatsNCChiPDF function returns the noncentral chi-squared probability distribution function

$$f(x;n,d) = \frac{\sqrt{d}\exp\left(-\frac{x+d}{2}\right)x^{(n-1)/2}}{2(dx)^{n/4}} I_{n/2-1}\left(\sqrt{dx}\right).$$

where *n*>0 is the degrees of freedom, *d* ≥ 0 is the noncentrality parameter, and $I_k(x)$ is the modified Bessel function of the first kind, **bessI**.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCChiCDF**, **StatsInvNCChiCDF**, **StatsChiCDF**, and **StatsChiPDF** functions.

StatsNCFCDF

`StatsNCFCDF(x, n1, n2, d)`

The StatsNCFCDF function returns the cumulative distribution function of the noncentral F distribution. *n1* and *n2* are the shape parameters and *d* is the noncentrality measure. There is no closed form expression for the distribution.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCFPDF** and **StatsInvNCFCDF** functions.

StatsNCFPDF

`StatsNCFPDF(x, n1, n2, d)`

The StatsNCFPDF function returns the probability distribution function of the noncentral F distribution

$$f(x;n_1,n_2,d) = \frac{\exp\left(-d/2\right)}{B\left(\frac{n_1}{2},\frac{n_2}{2}\right)} x^{n_1/2-1}(xn_1+n_2)^{-(n_1+n_2)/2} n_1^{n_1/2} n_2^{n_2/2} {}_1F_1\left(\frac{n_1+n_2}{2},\frac{n_1}{2},\frac{xdn_1}{2\left(xn_1+n_2\right)}\right),$$

where $B()$ is the **beta** function and ${}_1F_1()$ is the hypergeometric function **hyperG1F1**.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCFCDF** and **StatsInvNCFCDF** functions.

StatsNCTCDF

`StatsNCTCDF(x, df, d)`

The StatsNCTCDF function returns the cumulative distribution function of the noncentral Student-T distribution. *df* is the degrees of freedom (positive integer) and *d* is the noncentrality measure. There is no closed form expression for the distribution.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentCDF**, **StatsStudentPDF**, and **StatsNCTPDF** functions.

StatsNCTPDF

`StatsNCTPDF(x, df, d)`

The StatsNCTPDF function returns the probability distribution function of the noncentral Student-T distribution. *df* is the degrees of freedom (positive integer) and *d* is the noncentrality measure.

$$f(x;n,\delta) = \frac{n^{n/2}n!}{2^n e^{\delta^2/2}(n+x^2)^{n/2}\Gamma\left(\frac{\mathrm{n}}{2}\right)}\left\{\frac{\sqrt{2}\delta x\,{}_1F_1\left(\frac{n}{2}+1;\frac{3}{2};\frac{\delta^2x^2}{2(n+x^2)}\right)}{(n+x^2)\Gamma\left(\frac{n+1}{2}\right)}+\frac{{}_1F_1\left(\frac{n+1}{2};\frac{1}{2};\frac{\delta^2x^2}{2(n+x^2)}\right)}{\sqrt{(n+x^2)}\Gamma\left(\frac{n}{2}+1\right)}\right\}$$

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentPDF**, **StatsStudentCDF**, and **StatsNCTCDF** functions.

StatsNormalCDF

`StatsNormalCDF(x, m, s)`

The StatsNormalCDF function returns the normal cumulative distribution function

$$F(x,\mu,\sigma) = \frac{1}{2}+\frac{1}{2}erf\left(\frac{x-\mu}{\sigma\sqrt{2}}\right),$$

where *erf* is the error function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **erf**, **StatsNormalPDF** and **StatsInvNormalCDF** functions.

StatsNormalPDF

`StatsNormalPDF(x, m, s)`

The StatsNormalPDF function returns the normal probability distribution function

$$f(x,\mu,\sigma)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNormalCDF** and **StatsInvNormalCDF** functions.

StatsNPMCTest

`StatsNPMCTest [flags] [wave1, wave2,… wave100]`

The StatsNPMCTest operation performs a number of nonparametric multiple comparison tests. Output waves are saved in the current data folder according to the test(s) performed. Some tests are only appropriate when you have the same number of samples in all groups. StatsNPMCTest usually follows **StatsANOVA1Test** or **StatsKWTest**.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/CIDX=*controlIndex* Performs nonparametric multiple comparisons on a control group specified by the zero-based *controlIndex* wave in the input list. Output is to the M_NPCCResults wave in the current data folder or optionally to a table. The output column contents are: the first contains the difference between the rank sums of the control and each of the other waves; the second contains the standard error (SE); the third contains the statistic q, defined as the ratio of the difference in rank sums to SE; the fourth contains the critical value which also depends on the tails specification (see /TAIL); and the fifth contains the conclusion with 0 to reject H_0 and 1 to accept it. One version of this test applies when all inputs contain the same number of samples. When that is not the case, it uses the Dunn-Hollander-Wolfe approach to compute an appropriate SE and to handle possible ties.

/CONW=*cWave* Performs a nonparametric multiple contrasts tests. *cWave* has one point for each input wave. The *cWave* value is 1 to include the corresponding (zero based) input wave in the first group, 2 to include the wave in the second group, or zero to exclude the wave.

The contrast is defined as the difference between the normalized sum of the ranks of the first group and that of the second group. If *cWave*={0,1,1,1,2}, then the contrast is computed as

$$\text{contrast: } \frac{1}{3}[R_{n1}+R_{n2}+R_{n3}]-R_{n4}.$$

where R_{ni} is the normalized rank sum of the samples from the corresponding input wave. Note the significance of allowing zeros in the contrast wave because the actual ranking is performed on the pool of all the samples.

Output is to the M_NPMConResults wave in the current data folder or optionally to a table. The output column contents are: the first is the contrast value; the second is the standard error (SE); the third is the statistic S, which is the ratio of the absolute value of the contrast to SE; the fourth is the critical value (from χ^2 the approximation); and the fifth is the conclusion with 0 to reject H_0 and 1 indicating acceptance.

This test supports input waves with different number of samples and can also handle tied ranks. Note that the contrast wave used here is structured differently than for **StatsMultiCorrelationTest**.

/DHW Performs the Dunn-Holland-Wolfe test, which supports unequal number of samples and accounts for ties in the rank sums. Output is to the M_NPMCDHWResults wave in the current data folder or optionally to a table. The output column contents are: the first contains the difference between the means of the rank sums (rank sums divided by the number of samples in the group), the second contains the standard error (SE), the third contains the DHW statistic Q, the fourth contains the critical value, and the fifth contains the conclusion (0 to reject H_0 and 1 to accept).

/Q No results printed in the history area.

/SWN Creates a text wave containing wave names corresponding to each row of the comparison table. Depending on your choice of tests, the following wave names are created:

/CIDX test: T_NPCCResultsDescriptors

/DHW test: T_NPMCDHWDescriptors

/SNK test: T_NPMCSNKResultsDescriptors

/TUK test: T_NPMCTukeyDescriptors

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results.

/TAIL=*tc* Specifies H_0 with /CIDX.

tc=1:	One tailed test ($\mu_c \leq \mu_a$).
tc=2:	One tailed test ($\mu_c \geq \mu_a$).
tc=4:	Default; two tailed test ($\mu_c = \mu_a$).

Code combinations are not allowed.

/SNK Performs a nonparametric variation on the Student-Newman-Keuls test where the standard error SE is a function of p (the rank difference). This test requires equal numbers of samples in all groups; use /DHW for unequal sizes.

Output is to the M_NPMCSNKResults wave in the current data folder. The output column contents are: the first contains the difference between rank sums, the second contains the standard error (SE), the third contains the p value (rank difference), the fourth the statistic, the fifth contains the critical value, and the sixth contains the conclusion (0 to reject H_0 and 1 to accept). This test is more sensitive to differences than the Tukey test (/TUK).

/TUK	Perform a Tukey-type (Nemenyi) multiple comparison test using the difference between the rank sums. This is the default that is performed if you do not specify any of the test flags. This test requires equal numbers of points in all waves; use /DHW for unequal sizes. Output is to the M_NPMCTukeyResults wave in the current data folder. The output column contents are: the first contains the difference between the rank sums, the second contains the SE values, the third contains the statistic q, the fourth contains the critical value for this specific alpha and the number of groups; and the last contains a conclusion flag with 0 indicating a rejection of H_0 and 1 indicating acceptance. H_0 postulates that the paired means are the same.
/WSTR=*waveListString*	
	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.
/Z	Ignores errors.

Details

Inputs to StatsNPMCTest are two or more 1D numerical waves (one wave for each group of samples) containing two or more valid entries. The waves must have the same number of points for the use /SNK and /TUK tests, otherwise, for waves of differing lengths you must use the Dunn-Hollander-Wolfe test (/DHW).

V_flag will be set to zero for no execution errors. Individual tests may fail if, for example, there are different number of samples in the input waves for a test that requires an equal number of points. StatsNPMCTest skips failed tests and V_flag will be a binary combination identifying the failed test(s):

`V_flag & 1`	Tukey method failed (/TUK).
`V_flag & 2`	Student-Newman-Keuls failed (/SNK).

V_flag will be set to -1 for any other errors.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test** and **StatsKWTest**.

For multiple comparisons in parametric tests see: **StatsDunnettTest** and **StatsScheffeTest**.

StatsNPNominalSRTest

`StatsNPNominalSRTest` `[`***`flags`***`] [`***`srcWave`***`]`

The StatsNPNominalSRTest operation performs a nonparametric serial randomness test for nominal data consisting of two types. The null hypothesis is that the data are randomly distributed. Output is to the W_StatsNPSRTest wave in the current data folder.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/P={*m*,*n*,*u*}	Provides a summary of the data instead of providing the nominal series. *m* is the number of elements of the first type, *n* is the number of elements of the second type, and *u* is the number of runs or contiguous sequences of each type. Do not use *srcWave* with /P.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors.

Details

The input wave to StatsNPNominalSRTest is specified with *srcWave* or /P. The wave must contain exactly two values. If *srcWave* is a text wave, then each type can be designated by a letter or by a short string (less than 200 bytes). If *srcWave* is numeric, you should avoid the usual floating point waves, which can give rise to internal representations of more than two distinct values. Output to W_StatsNPSRTest includes the total number of points (*N*), the number of occurrences (*m*) of the first variable, the number of occurrences (*n*) of the second variable, and the number of runs (*u*). When both *m* and *n* are less than 300, it computes the P value (probability P(*u*'<*u*)) and the critical values using the Swed and Eisenhart algorithm. When *m* or *n* are larger than 300, it computes the mean and standard deviation of an equivalent normal distribution with the corresponding critical value.

References

Swed, F.S., and C. Eisenhart, Tables for testing randomness of grouping in a sequence of alternatives, *Ann. Math. Statist.*, *14*, 66-87, 1943.

See, in particular, Chapter 25 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsSRTest**.

StatsParetoCDF

`StatsParetoCDF(x, a, c)`

The StatsParetoCDF function returns the Pareto cumulative distribution function

$$F(x;a,c) = 1 - \left(\frac{a}{x}\right)^c .$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsParetoPDF** and **StatsInvParetoCDF** functions.

StatsParetoPDF

`StatsParetoPDF(x, a, c)`

The StatsParetoPDF function returns the Pareto probability distribution function

$$\mathrm{f(x;a,c)}=\frac{\mathrm{c}}{\mathrm{x}}\left(\frac{a}{x}\right)^c, \qquad \begin{matrix} a,c > 0 \\ x \ge a. \end{matrix}$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsParetoCDF** and **StatsInvParetoCDF** functions.

StatsPermute

`StatsPermute(waveA, waveB, dir)`

The StatsPermute function permutes elements in *waveA* based on the lexicographic order of *waveB* and the direction *dir*. It returns 1 if a permutation is possible and returns 0 otherwise. Use *dir*=1 for the next permutation and *dir*=-1 for a previous permutation.

Details

Both *waveA* and *waveB* must be numeric. The lexicographic order of elements in the index wave is set so that permutations start with the index wave *waveB* in ascending order and end in descending order. Elements of *waveA* are permuted in place according to the order of the indices in *waveB* which are clipped (after permutation) to the valid range of entries in *waveA*. *waveB* is also permuted in place in order to allow you to obtain sequential permutations. If *waveA* consists of real numbers you can permute them using the lexicographic value of the entries directly. To do so pass `$""` for *waveB*. Whenever it returns 0, neither *waveA* and *waveB* are changed.

Examples

```
Function AllPermutations(num)
   Variable num

   Variable i,nf=factorial(num)
   Make/O/N=(num) wave0=p+1,waveA,waveB=p

   Print wave0
   for(i=0;i<nf;i+=1)
      waveA=wave0
      if(statsPermute(waveA,waveB,1)==0)
         break
      endif
      print waveA
   endfor
end

Executing AllPermutations(3) prints:
  wave0[0]= {1,2,3}
  waveA[0]= {1,3,2}
  waveA[0]= {2,1,3}
  waveA[0]= {2,3,1}
  waveA[0]= {3,1,2}
  waveA[0]= {3,2,1}
```

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsPoissonCDF

StatsPoissonCDF(*x*, λ)

The StatsPoissonCDF function returns the Poisson cumulative distribution function

$$F(x;\lambda)=\sum_{i=0}^{x}\frac{\exp(-\lambda)\lambda^i}{i!}, \qquad x=0,1,2...$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPoissonPDF** and **StatsInvPoissonCDF** functions.

StatsPoissonPDF

StatsPoissonPDF(*x*, λ)

The StatsPoissonPDF function returns the Poisson probability distribution function

$$f(x;\lambda)=\frac{\exp(-\lambda)\lambda^x}{x!}, \qquad x=0,1,2...$$

where λ is the shape parameter.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPoissonCDF** and **StatsInvPoissonCDF** functions.

StatsPowerCDF

StatsPowerCDF(*x*, *b*, *c*)

The StatsPowerCDF function returns the Power Function cumulative distribution function

$$F(x;b,c)=\left(\frac{x}{b}\right)^c$$

where the scale parameter *b* and the shape parameter *c* satisfy $b,c > 0$ and $b \geq x \geq 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPowerPDF**, **StatsInvPowerCDF** and **StatsPowerNoise** functions.

StatsPowerNoise

`StatsPowerNoise(`*`b`*`, `*`c`*`)`

The StatsPowerNoise function returns a pseudorandom value from the power distribution function with probability distribution:

$$f(x;b,c) = \frac{c}{x}\left(\frac{x}{b}\right)^{c}.$$

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable “random” numbers, use SetRandomSeed. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

The **StatsPowerPDF StatsInvPowerCDF** and **StatsInvPowerCDF** functions.

Noise Functions on page III-390.

Chapter III-12, **Statistics** for a function and operation overview.

StatsPowerPDF

`StatsPowerPDF(`*`x`*`, `*`b`*`, `*`c`*`)`

The StatsPowerPDF function returns the Power Function probability distribution function

$$f(x,b,c) = \frac{|c|}{x}\left(\frac{\mathrm{x}}{b}\right)^{c},$$

where b is a scale parameter and c is a shape parameter.

For b,c > 0, x is drawn from b >= x >= 0.

For b>0, c<0, x is drawn from x>b.

For b<0, c>0, x is drawn from -b <= x <= 0.

For b<0, c<0, x is drawn from x<-b.

Note that for -1<c<0 the average diverges and the magnitude of a mean calculated from N samples will increase indefinitely with N.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPowerCDF**, **StatsInvPowerCDF** and **StatsPowerNoise** functions.

StatsQCDF

`StatsQCDF(`*`q`*`, `*`r`*`, `*`c`*`, `*`df`*`)`

The StatsQCDF function returns the value of the Q cumulative distribution function for *r* the number of groups, *c* the number of treatments, and *df* the error degrees of freedom (*f*=*rc*(*n*-1) with sample size *n*).

Details

The Q distribution is the maximum of several Studentized range statistics. For a simple Tukey test, use *r*=1.

References

Copenhaver, M.D., and B.S. Holland, Multiple comparisons of simple effects in the two-way analysis of variance with fixed effects, *Journal of Statistical Computation and Simulation*, *30*, 1-15, 1988.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTukeyTest** function.

StatsQpCDF

StatsQpCDF(*q*, *nr*, *nt*, *dt*, *side*, *sSizeWave*)

The StatsQpCDF function returns the Q' cumulative distribution function associated with Dunnett's test.

Here *nr* is the number of groups (should be set to 1), *nt* is the number of treatments, *df* is the error degrees of freedom.

Set *side*=1 for upper-tail or *side*=2 for two-tailed CDF.

sSizeWave is an integer wave of nt rows specifying the number of samples in each treatment.

Details

StatsQpCDF is a modified Q distribution typically used with Dunnett's test, which compares the various means with the mean of the control group or treatment

References

"Algorithm AS 251: Multivariate Normal Probability Integrals with Product Correlations Structure", C. W. Dunnett, *Appl. Stat.*, 38 (1989) 564-579.

A short correction for the algorithm was published in: *Appl. Stat.*, 42 (1993) 709.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsDunnettTest**, **StatsInvQpCDF**, and **StatsInvQCDF** functions.

StatsQuantiles

StatsQuantiles [***flags***] ***srcWave***

The StatsQuantiles operation computes quantiles and elementary univariate statistics for a set of data in *srcWave*.

Flags

/ALL Invokes all flags except /Q, /QM, and /Z.

/BOX Computes parameters necessary to construct a box plot.

/iNaN Ignores NaNs, which are sorted to the end of the array by default.

/IW Creates an index wave W_QuantilesIndex. W_QuantilesIndex[*i*] corresponds to the position of *srcWave*[*i*] when sorted from minimum to maximum.

/Q No information printed in the history area.

/QM=*qMethod* Specifies the method for computing quartiles. *qMethod* has one of these values:

0: Tukey (default).

1: Minitab.

2: Moore and McCabe.

3: Mendenhall and Sincich.

See Details for more information.

/QW Creates a single precision wave W_QuantileValues containing the quantile value corresponding to each entry in *srcWave*.

/STBL Uses a stable sort, which may require significant computation time for multiple entries with the same value.

/T=*k* Displays the result wave W_StatsQuantiles in a table and specifies window behavior when the user attempts to close the table.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

If you use /K=2 you can still kill the window using the **KillWindow** operation.

/TM Computes the tri-mean: 0.25*(V_Q25+2*median+V_Q75).

/TRIM=*tVal* Computes the trimmed mean which is the mean value of the entries between the quantiles *tVal* (in %) and 100-*tVal*. By default *tVal*=25 and the trimmed mean corresponds to the midmean.

/Z Ignores any errors.

Details

StatsQuantiles produces quick five-number summaries or more detailed results for univariate data. Values are returned in the wave W_StatsQuantiles and in the variables:

V_min Minimum value.

V_max Maximum value.

V_Median Median value.

V_Q25 Lower quartile.

V_Q75 Upper quartile.

V_IQR Inter-quartile range V_Q75-VQ25, which is also known as the H-spread.

V_MAD Median absolute deviation.

V_mode The most frequent value.

If there is a tie and several values have the highest frequency then the lowest value among them is returned as the mode.

If all values in srcWave are unique or if the number of points in srcWave is less than 3, V_mode is set to NaN.

This output was added in Igor Pro 7.00.

Entries in the wave W_StatsQuantiles depend on your choice of flags. Each row has a row label explicitly defining its value. If you use the /ALL flag, W_StatsQuantiles will contain the following row labels:

minValue

maxValue

Median

Q25

Q75

IQR

MedianAbsoluteDeviation

lowerInnerFence

lowerOuterFence

upperInnerFence

upperOuterFence

triMean

trimmedMean

Otherwise, W_StatsQuantiles will contain the first five entries and any additionally requested value. You should always access values using the dimension labels (see **Dimension Labels** on page II-93).

There is frequently some confusion in comparing statistical results computed by different programs because each may use a different definition of quartiles. You can specify the method of computing the quartiles as you prefer with the /QM flag. If you neglect to choose a method, StatsQuantiles uses Tukey's method, which computes quartiles (also called hinges) as the lower and upper median values between the median of the data and the edges of the array. The Moore and McCabe method is similar to Tukey's method except you do not include the median itself in computing the quartiles. Mendenhall and Sincich compute the quartiles using 1/4 and 3/4 of (numDataPoints+1) and round to the nearest integer (if the fraction part is exactly 0.5 they round up for the lower quartile and down for the upper quartile). Minitab uses the same expressions but instead of rounding it uses linear interpolation.

StatsQuantiles uses a stable index sorting routine so that

```
IndexSort W_QuantilesIndex,srcWave
```

is a monotonically increasing wave.

References

Tukey, J. W., *Exploratory Data Analysis*, 688 pp., Addison-Wesley, Reading, Massachusetts, 1977.

Mendenhall, W., and T. Sincich, *Statistics for Engineering and the Sciences*, 4th ed., 1008 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1995.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **WaveStats**, **StatsMedian**, **Sort**, and **MakeIndex**.

StatsRankCorrelationTest

StatsRankCorrelationTest [***flags***] ***waveA***, ***waveB***

The StatsRankCorrelationTest operation performs Spearman's rank correlation test on *waveA* and *waveB*, 1D waves containing the same number of points. Output is to the W_StatsRankCorrelationTest wave in the current data folder.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/P=*method* Controls the computation of the P-value.

The /P flag was added in Igor Pro 9.00.

method=0: If the number of data points is less than or equal to 6 then an exact calculation is made. This is the default if /P is omitted.

method=1: The P-value is computed using the Edgeworth approximation. The P-value reported corresponds to a two tails calculation.

method=2: The P-value is computed using the Student-T approximation. This is appropriate when the number of data points is large.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/Z Ignores errors.

Details

StatsRankCorrelationTest ranks *waveA* and *waveB* and then computes the sum of the squared differences of ranks for all rows. Ties are assigned an average rank and the corrected Spearman rank correlation coefficient is computed with ties. It reports the sum of the squared ranks (sumDi2), the sums of the ties coefficients (sumTx and sumTy respectively), the Spearman rank correlation coefficient (in the range [-1,1]), and the critical value. H_0 corresponds to zero correlation against the alternative of nonzero correlation. The critical value is usually lower than the one in published tables. When the first derivative of the CDF is discontinuous, tables tend to use a more conservative value by choosing the next transition of the CDF as the critical value. StatsRankCorrelationTest is not as powerful as **StatsLinearCorrelationTest**.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsLinearCorrelationTest, **StatsCircularCorrelationTest**, **StatsKendallTauTest**, **StatsSpearmanRhoCDF**, and **StatsInvSpearmanCDF**.

StatsRayleighCDF

StatsRayleighCDF(*x* [, *s* [, *m*]])

The StatsRayleighCDF function returns the Rayleigh cumulative distribution function

$$F(x;\sigma,\mu)=1-\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \qquad \sigma>0, x>\mu.$$

with defaults *s*=1 and *m*=0. It returns NaN for *s* ≤ 0 and zero for x ≤ *m*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRayleighPDF** and **StatsInvRayleighCDF** functions.

StatsRayleighPDF

StatsRayleighPDF(*x* [, *s* [, *m*]])

The StatsRayleighPDF function returns the Rayleigh probability distribution function

$$f(x;\sigma,\mu)=\frac{x-\mu}{\sigma^2}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \qquad \sigma>0, x>\mu.$$

with defaults *s*=1 and *m*=0. It returns NaN for *s* ≤ 0 and zero for x ≤ *m*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRayleighCDF** and **StatsInvRayleighCDF** functions.

StatsRectangularCDF

StatsRectangularCDF(*x*, *a*, *b*)

The StatsRectangularCDF function returns the rectangular (uniform) cumulative distribution function

$$F(x,a,b)=\begin{cases}0 & x\le a\\ \dfrac{x-a}{b-a} & a\le x\le b\\ 1 & x\ge b\end{cases}$$

where *a*< *b*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRectangularPDF** and **StatsInvRectangularCDF** functions.

StatsRectangularPDF

StatsRectangularPDF(*x*, *a*, *b*)

The StatsRectangularPDF function returns the rectangular (uniform) probability distribution function

$$f(x;a,b)=\begin{cases}\dfrac{1}{b-a} & a\le x\le b\\ 0 & otherwise\end{cases}$$

where $a < b$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRectangularCDF** and **StatsInvRectangularCDF** functions.

StatsResample

`StatsResample /N=`*`numPoints`* `[`*`flags`*`]` *`srcWave`*

The StatsResample operation resamples *srcWave* by drawing (with replacement) *numPoints* values from *srcWave* and storing them in the wave W_Resampled or M_Resampled if /MC is used. You can iterate the process and compute various statistics on the data samples.

Flags

/ITER=*n* Repeats the resampling for *n* iterations, which is useful only when combined with /WS or /SQ.

The /ITER flag is ignored by the Jack-Knife analysis (/JCKN).

/JCKN=*ufunc* Performs Jack-Knife analysis. Here ufunc is a user function of the format:

```
Function ufunc(inWave)
    wave inWave
    ... compute some statistic for inWave
    return someValue
End
```

The results are stored in the wave W_JackKnifeStats in the current data folder. Use

```
Edit W_JackKnifeStats.ld
```

to display the wave with dimension labels.

In the Jack-Knife method the operation runs N iterations where N is the number of points in *srcWave*. In each iteration the opeation calls the user-defined function ufunc(*inWave*) passing it an internal wave which contains (N-1) samples from *srcWave*. The function computes some user-defined statistic, say "Z", and stores it in *inWave*. At the end of iterations the operation uses the Z values in *inWave* to compute various Jack-Knife estimates. The standard estimator is defined as:

$$Z = ufunc(srcWave).$$

The Jack-Knife estimator is simply:

$$\hat{z} = \frac{1}{n}\sum_{i=1}^{n} z_i .$$

The Jack-Knife t-estimator is slightly less biased. It is given by:

$$t = nZ - (n-1)\hat{z},$$

The estimate of the standard error is given by:

$$\hat{\sigma}_{\hat{z}} = \sqrt{\frac{n-1}{n}\sum_{i=1}^{n}(z_i - \hat{z})^2} .$$

The Jack-Knife analysis ignores the /N and /ITER flags. The number of points and the number of iterations are determined by the number of points in *srcWave*.

/K Kills W_Resampled after passing it to WaveStats. When /ITER is used, W_Resampled is not saved.

/MC Use /MC when you want to sample random (complete) rows from a multi-column 2D *srcWave*. The combination of /N=n with /MC results in the wave M_Resampled in the current data folder. M_Resampled will have n rows, the same number of columns and the same data type as *srcWave*.

/N=*numPoints* Specifies the number of points sampled from *srcWave*.

The /N flag is ignored by the Jack-Knife analysis (/JCKN).

/Q No information printed in the history area.

/SQ=*m* Uses StatsQuantiles to compute the data quartiles. The methods are:

m=0: Tukey (default).

m=1: Minitab.

m=2: Moore and McCabe.

m=3: Mendenhall and Sincich.

See Details for information about how the results are stored.

The default trim value is 25%.

/WS=*m* Uses WaveStats operation to calculate data statistics.

m=0: Creates a new wave containing the samples (default).

m=1: Creates the new wave and passes it to `WaveStats/Q/M=1`.

m=2: Creates the new wave and passes it to `WaveStats/Q/M=2`.

See Details for information about how the results are stored.

/Z Ignores any errors.

Details

StatsResample can perform Bootstrap Analysis, permutations tests, and Monte-Carlo simulations. It draws the specified number of data points (with replacement) from *srcWave* and places them in a destination wave W_Resampled.

Specify /WS or /SQ to use the WaveStats or StatsQuantiles operations, respectively, to compute results directly from the data. StatsResample normally creates the wave W_Resampled and, optionally, the M_WaveStats and W_StatsQuantiles waves. Both options also create various V_ variables described below. If you use more than one iteration, StatsResample creates instead the waves M_WaveStatsSamples and M_StatsQuantilesSamples for the results.

M_WaveStatsSamples (with /WS) contains a column for each iteration. Each column is equivalent to the contents of M_WaveStats for that iteration. You can use the command

```
Edit M_WaveStatsSamples.ld
```

to display the results in a table using row labels, and, for example, to display a graph of the rms of the samples as a function of iteration number execute:

```
Display M_WaveStatsSamples[5][]
```

M_StatsQuantilesSamples (with /SQ) contains a column for each iteration. Each column consists of the contents of W_StatsQuantiles for the corresponding data. Here again you can execute the command

```
Edit M_StatsQuantilesSamples.ld
```

to display the wave in a table using row labels. To display a graph of the median as a function of iteration execute:

```
Display M_statsQuantilesSamples[2][]
```

Output Variables

StatsResample creates the following variables: V_Median, V_Q25, V_Q75, V_IQR, V_min, V_max, V_numNaNs, V_numINFs, V_avg, V_sdev, V_rms, V_adev, V_skew, V_kurt, and V_Sum.

These variables are valid only if you use either /SQ or /WS, but not both, and only if you do not use /ITER. Unused variables are set to NaN.

If you use /SQ the operation sets V_Median, V_Q25, V_Q75, V_IQR, V_min, and V_max.

If you use /WS the operation sets V_min, V_max, V_numNaNs, V_numINFs, V_avg, V_sdev, V_rms, V_adev, V_skew, V_kurt, and V_Sum.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsSample**, **WaveStats** and **StatsQuantiles**.

StatsSample

`StatsSample /N=`*`numPoints`* [***`flags`***] ***`srcWave`***

StatsSample creates a random, non-repeating sample from *srcWave*.

It samples *srcWave* by drawing without replacement *numPoints* values from *srcWave* and storing them in the output wave W_Sampled or M_Sampled if /MC or /MR are used.

The /N flag is required.

Flags

/ACMB — Creates a wave containing all unique combinations of *numPoints* values from srcWave. It is assumed that *srcWave* is a 1D numeric wave containing more than *numPoints* elements. The results are stored in the wave M_Combinations in the current data folder. Each row in the result wave corresponds to a unique combination of samples.

Added in Igor Pro 7.00.

/CMPL — Stores all data elements from srcWave that were excluded from the random sample in the wave W_CompWave or M_CompWave in the current data folder. /CMPL was added in Igor Pro 8.00.

/N=*numPoints* — Specifies the number of points sampled from *srcWave*. When combined with /MC, *numPoints* is the number of sampled rows and when combined with /MR, it is the number of sampled columns.

/MC — Use /MC (multi-column) to randomly sample full rows from *srcWave*, i.e., the output consists of all columns of each selected row. /MC and /MR are mutually exclusive flags.

/MR — Use /MR (multi-row) to randomly sample full columns from *srcWave*, i.e., the output consists of all rows of each of the selected columns. /MC and /MR are mutually exclusive flags.

/Z — Ignores errors.

Details

If you omit /MC and /MR, the output is a 1D wave named W_Sampled where the samples are chosen from *srcWave* without regard to its dimensionality.

If you use either /MC or /MR the output is a 2D wave named M_Sampled which will have either the same number of columns (/MC) as *srcWave* or the same number of rows (/MR) as *srcWave*.

See Also

Chapter III-12, **Statistics**, **StatsResample**

StatsRunsCDF

`StatsRunsCDF(`*`n`*`, `*`r`*`)`

The StatsRunsCDF function returns the cumulative distribution function for the up and down runs distribution for total number of runs *r* in a random linear arrangement of *n* unequal elements. There is no closed form expression. It is computed numerically from the recursion of the probability density

$$f(r,n)=\frac{rf(r,n-1)+2f(r-1,n-1)+(n-r)f(r-2,n-1)}{n},$$

with the initial condition

$$f(1,n)=\frac{2}{n!}.$$

References

Bradley, J.V., *Distribution-Free Statistical Tests*, Prentice Hall, Englewood Cliffs, New Jersey, 1968.

Olmstead, P.S., Distribution of sample arrangements for runs up and down, *Annals of Mathematical Statistics*, *17*, 24-33, 1946.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsSRTest** function.

StatsScheffeTest

`StatsScheffeTest [`***`flags`***`] [`***`wave1, wave2,… wave100`***`]`

The StatsScheffeTest operation performs Scheffe’s test for the equality of the means. It supports two basic modes: the default tests all possible combinations of pairs of waves; the second tests a single combination where the precise form of H_0 is determined by the coefficients of a contrast wave (see /CONT). Output is to the M_ScheffeTestResults wave in the current data folder.

Flags

/ALPH=*val* Sets the significance level (default 0.05).

/CONW=*cWave* Performs a multiple contrasts test. *cWave* has one point for each input wave. The *cWave* value is 1 to include the corresponding (zero based) input wave in the first group, 2 to include the wave in the second group, or zero to exclude the wave.

The contrast is defined as the difference between the normalized sum of the ranks of the first group and that of the second group. If *cWave*={0,1,1,1,2}, then the contrast hypothesis H_0 corresponds to:

$$\frac{\overline{X}_1+\overline{X}_2+\overline{X}_3}{3}-\overline{X}_4=0.$$

For each pair of waves (*i, j*) with *i* ¦ *j*, it computes

$$SE_{ij}=\sqrt{s^2\left(\frac{1}{n_j}+\frac{1}{n_i}\right)},\qquad s^2=\sum_{i=1}^{W}\sum_{j=0}^{n_j-1}X_j^2-\sum_{i=1}^{W}\frac{\left(\sum_{j=0}^{n_j-1}X_j\right)^2}{n_j},$$

the statistic

$$S=\frac{\left|\sum_{i=0}^{n-1}c_i\overline{X}_i\right|}{SE},$$

the critical value, and a result field which is set to 1 if H_0 should be accepted or 0 if it should be rejected. W is the total number of waves, n_i and $\bar{X}_i$ are respectively the number of data points and the average of wave *i*.

/Q No results printed in the history area.

/SWN Creates a text wave, T_ScheffeDescriptors, containing wave names corresponding to each row of the comparison table (Save Wave Names). Use /T to append the text wave to the last column.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The default of StatsScheffeTest (also known as the S test) tests the hypotheses of equality of means for each possible pair of samples. It is not as powerful as Tukey's test (**StatsTukeyTest**) and is more useful for hypotheses formulated as multiple contrasts (see /CONT).

References

See, in particular, Chapter 11 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test**, **StatsDunnettTest** and **StatsTukeyTest**.

StatsShapiroWilkTest

StatsShapiroWilkTest [*flags*] *srcWave*

The StatsShapiroWilkTest computes Shapiro-Wilk statistic W and its associated P-value and stores them in V_statistic and V_prob respectively.

Flags

/Q No results printed in the history area.

/Z Ignores errors.

Details

The Shapiro-Wilk tests the null hypothesis that the population is normally distributed. If the P-value is less than the selected alpha then the null hypothesis, normality, is rejected.

The test is valid only for waves containing 3 to 5000 data points. The operation ignores any NaNs or INFs in *srcWave*.

Example

```
// Test normally distributed data
Make/O/N=(200) ggg=gnoise(5)
StatsShapiroWilkTest ggg
W=0.995697 p=0.846139           // p>alpha so accept normality
```

```
// Test uniform distribution
Make/O/N=(200) eee=enoise(5)
StatsShapiroWilkTest eee
W=0.959616 p=1.7979e-05          // p<alpha so reject normality
```

StatsSignTest

`StatsSignTest [`*`flags`*`]` ***`wave1, wave2`***

The StatsSignTest operation performs the sign test for paired-sample data contained in *wave1* and *wave2*.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The input waves must be the equal length, real numeric waves and must not contain any NaNs or INFs. Results are saved in the wave W_SignTest and are optionally displayed in a table. StatsSignTest computes the differences in each pair and counts the total number of entries with positive and negative differences, and tests the results using a binomial distribution. When the number of data pairs exceeds 1024 it uses a normal approximation to the binomials for calculating the probabilities and the power of the test.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsWilcoxonRankTest

StatsSpearmanRhoCDF

`StatsSpearmanRhoCDF(`*`r, N`*`)`

The StatsSpearmanRhoCDF function returns the cumulative distribution function for Spearman's *r,* which is used in rank correlation test. It is valid for *N*>1 and -1 ≤ *r* ≤ 1. The distribution is mostly computed using the Edgeworth series expansion.

References

Algorithm AS 89, *Appl. Statist.*, *24*, 377, 1975.

van de Wiel, M.A., and A. Di Bucchianico, Fast computation of the exact null distribution of Spearman's rho and Page's L statistic for samples with and without ties, *J. of Stat. Plan. and Inference*, *92*, 133-145, 2001.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRankCorrelationTest**, **StatsInvSpearmanCDF**, and **StatsKendallTauTest** functions.

StatsSRTest

`StatsSRTest [`*`flags`*`]` ***`srcWave`***

The StatsSRTest operation performs a parametric or nonparametric serial randomness test on *srcWave*, which must contain finite numerical data. The null hypothesis of the test is that the data are randomly distributed. Output is to the W_StatsSRTest wave in the current data folder.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/GCD Tests the output of a random number generator (RNG). *srcWave* consists of values between 0 and 2^{32} (converted to unsigned 32-bit integers). GCD computes the gcd for consecutive pairs of data in *srcWave*. The number of steps in the GCD and the distribution of the GCD's are compared with ideal distributions and corresponding P values are reported. This test is part of Marsaglia's Die-Hard battery of tests. P-values close to either 0 or 1 indicate a nonideal RNG. You should use the reported minimum and maximum values to check that the input is indeed in the proper range. Typically *srcWave* consists of at least1e6 entries.

/NAPR Use the normal approximation even when the number of points is below 150.

/NP Performs a nonparametric serial randomness test by counting the numbers of runs up and down and computing the probability that such a value is obtained by chance.

/P Performs a parametric serial randomness test.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/Z Ignores errors.

Details

The parametric test for serial randomness is according to Young. *C* is given by

$$C = 1 - \frac{\sum_{i=0}^{n-2}\left(X_i - X_{i+1}\right)^2}{2\sum_{i=0}^{n-1}\left(X_i - \overline{X}\right)^2},$$

where $\bar{X}$ is the mean and n is the number of points in *srcWave*. The critical value is obtained from mean square successive difference distribution **StatsInvCMSSDCDF**. For more than 150 points, StatsSRTest uses the normal approximation and provides the critical values from the normal distribution. For samples from a normal distribution, *C* is symmetrically distributed about 0 with positive values indicating positive correlation between successive entries and negative values corresponding to negative correlation.

The nonparametric test consists of counting the number of runs that are successive positive or successive negative differences between sequential data. If two sequential data are the same it computes two numbers of runs by considering the two possibilities where the equality is replaced with either a positive or a negative difference. The results of the operation include the number of runs up and down, the number of unchanged values (the number of places with no difference between consecutive entries), the size of the longest run and its associated probability, the number of converted equalities, and the probability that the number of runs is less than or equal to the reported number (**StatsRunsCDF**). When equalities are encountered the operation computes the probabilities that the computed number of runs or less can be found in an equivalent random sequence.

Converted equalities are those with the same sign on both sides so that when we replace the equality by the opposite sign we increase the number of runs. The equalities that are not converted are found between two different signs and therefore regardless of the sign that we give them they do not affect the total number of runs. We implicitly assume that the data does not contain more than one sequential equalities.

The longest run is determined without taking into account equalities or their conversions. The probability of the longest run is computed from Equation 6 of Olmstead, which is accurate within 0.001 when the number of runs is 5 or more. This probability applies to either positive or negative differences and should be divided by two if a specific sign is selected.

References

Bradley, J.V., *Distribution-Free Statistical Tests*, Prentice Hall, Englewood Cliffs, New Jersey, 1968.

P.S., Distribution of sample arrangements for runs up and down, *Annals of Mathematical Statistics*, *17*, 24-33, 1946.

Wallis, W.A., and G.H. Moore, A significance test for time series, *J. Amer. Statist. Assoc.*, *36*, 401-409, 1941.

Young, L.C., On randomness in ordered sequences, *Annals of Mathematical Statistics*, *12*, 153-162, 1941.

See, in particular, Chapter 25 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsNPNominalSRTest** and **StatsRunsCDF**.

StatsStudentCDF

`StatsStudentCDF(t, n)`

The StatsStudentCDF function returns the Student (uniform) cumulative distribution function

$$F(t,n)=\begin{cases}\frac{1}{2}\left\{1+I\left(\frac{n}{2},\frac{1}{2};1\right)-I\left(\frac{n}{2},\frac{1}{2};\frac{n}{n+t^2}\right)\right\} & t>0\\ \frac{1}{2}\left\{1+I\left(\frac{n}{2},\frac{1}{2};\frac{n}{n+t^2}\right)-I\left(\frac{n}{2},\frac{1}{2};1\right)\right\} & t<0\\ \frac{1}{2} & t=0\end{cases}$$

where *n*>0 is degrees of freedom and is the incomplete beta function **betai**.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentPDF** and **StatsInvStudentCDF** functions.

StatsStudentPDF

`StatsStudentPDF(t, n)`

The StatsStudentPDF function returns the Student (uniform) probability distribution function

$$f(t,n)=\frac{\left(\frac{n}{n+t^2}\right)^{(n+1)/2}}{\sqrt{n}B\left(\frac{n}{2},\frac{1}{2}\right)}.$$

where *n*>0 is degrees of freedom and *B*() is the **beta** function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentCDF** and **StatsInvStudentCDF** functions.

StatsTopDownCDF

`StatsTopDownCDF(r, N)`

The StatsTopDownCDF function returns the cumulative distribution function for the top-down correlation coefficient. It is computationally intensive because it must evaluate many permutations [O((n!)^2)]. It exactly calculates the distribution for $3 \le N \le 7$; outside this range it uses Monte-Carlo estimation for $8 \le N \le 50$ and asymptotic Normal approximation for N>50. The Monte-Carlo estimate uses 1e6 random permutations fitted with two 9-order polynomials for the range [-1,0] and [0,1]. The results are within 0.2% of exact values where known.

References

Iman, R.L., and W.J. Conover, A measure of top-down correlation, *Technometrics*, *29*, 351-357, 1987.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRankCorrelationTest** and **StatsInvTopDownCDF** functions.

StatsTriangularCDF

`StatsTriangularCDF(`*`x`*`, `*`a`*`, `*`b`*`, `*`c`*`)`

The StatsTriangularCDF function returns the triangular cumulative distribution function

$$F(x;a,b,c)=\begin{cases} 0 & x \le a \\ \dfrac{(x-a)^2}{(b-a)(c-a)} & a < x \le c \\ 1-\dfrac{(b-x)^2}{(b-a)(b-c)} & c < x < b \\ 1 & x \ge b \end{cases}$$

where $a<c<b$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTriangularPDF** and **StatsInvTriangularCDF** functions.

StatsTriangularPDF

`StatsTriangularPDF(`*`x`*`, `*`a`*`, `*`b`*`, `*`c`*`)`

The StatsTriangularPDF function returns the triangular probability distribution function

$$f(x;a,b,c)=\begin{cases} \dfrac{2(x-a)}{(b-a)(c-a)} & a \le x < c \\ \dfrac{2(b-x)}{(b-a)(b-c)} & c < x \le b \\ 0 & Otherwise \end{cases}$$

where $a<c<b$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTriangularCDF** and **StatsInvTriangularCDF** functions.

StatsTrimmedMean

`StatsTrimmedMean(`*`waveName`*`, `*`trimValue`*`)`

The StatsTrimmedMean function returns the mean of the wave *waveName* after removing *trimValue* fraction of the values from both tails of the distribution. *trimValue* is a number in the range [0, 0.5]. *waveName* can be any real numeric type.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsQuantiles** and **mean**.

StatsTTest

`StatsTTest [`*`flags`*`] `*`wave1`*` [, `*`wave2`*`]`

The StatsTTest operation performs two kinds of T-tests: the first compares the mean of a distribution with a specified mean value (/MEAN) and the second compares the means of the two distributions contained in *wave1* and *wave2*, which must contain at least two data points, can be any real numeric type, and can have an arbitrary number of dimensions. Output is to the W_StatsTTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/CI Computes the confidence intervals for the mean(s).

/DFM=*m* Specifies method for calculating the degrees of freedom.

m=0: Default; computes equivalent degrees of freedom accounting for possibly different variances.

m=1: Computes equivalent degrees of freedom but truncates to a smaller integer.

m=2: Computes degrees of freedom by $DF=n_1+n_2-2$, where *n* is the sum of points in the wave. Appropriate when variances are equal.

/MEAN=*meanV* Compares *meanV* with the mean of the distribution in *wave1*. Outputs are the number of points in the wave, the degrees of freedom (accounting for any NaNs), the average, standard deviation (σ),

$$s_{\overline{X}} = \frac{\sigma}{\sqrt{DF+1}},$$

the statistic

$$t = \frac{\overline{X} - meanV}{s_{\overline{X}}}$$

and the critical value, which depends on /TAIL.

/PAIR Specifies that the input waves are pairs and computes the difference of each pair of data to get the average difference $\bar{d}$ and the standard error of the difference $S_{\bar{d}}$. The t statistic is the ratio of the two

$$t = \frac{\bar{d}}{s_{\bar{d}}}.$$

In this case H_0 is that the difference $\bar{d}$ is zero.

This mode does not support /CI and /DFM.

/Q No results printed in the history area.

/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed.
	k=0: Normal with dialog (default).
	k=1: Kills with no dialog.
	k=2: Disables killing.
	The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.
/TAIL=*tailCode*	Specifies H_0.
	tailCode=1: One tailed test ($\mu_1 \leq \mu_2$).
	tailCode=2: One tailed test ($\mu_1 \geq \mu_2$).
	tailCode=4: Default; two tailed test ($\mu_1 = \mu_2$).
	When performing paired tests using /PAIR:
	tailCode=1: One tailed test ($\mu_d \leq 0$).
	tailCode=2: One tailed test ($\mu_d \geq 0$).
	tailCode=4: Default; two tailed test ($\mu_d = 0$).
	Here μ_d is the mean of the difference population.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

When comparing the mean of a single distribution with a hypothesized mean value, you should use /MEAN and only one wave (*wave1*). If you use two waves StatsTTest performs the T-test for the means of the corresponding distributions (which is incompatible with /MEAN).

When comparing the means of two distributions, the default t-statistic is computed from Welch's approximate t:

$$t' = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}},$$

where s_i^2 are variances, n_i the number of samples, and $\bar{X}_i$ the averages of the respective waves. This expression is appropriate when the number of points and the variances of the two waves are different. If you want to compute the t-statistic using pooled variance you can use the /AEVR flag. In this case the pooled variance is given by

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2},$$

and the t-statistic is

$$t = \frac{\bar{x}_1 - \bar{x}_2}{s_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}.$$

The different test are:

H_0	Rejection Condition
$\mu_1 = \mu_2$	$\lvert t \rvert \geq$ Tc(alpha,ν)
$\mu_1 > \mu_2$	$t \leq$ Tc(alpha, ν)
$\mu_1 < \mu_2$	$t \geq$ Tc(alpha, ν)

Tc is the critical value and ν is the effective number of degrees of freedom (see /DFM flag).When accounting for possibly unequal variances, ν is given by

$$\nu = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2 - 1}}.$$

The critical values (Tc) are computed by numerically by solving for the argument at which the cumulative distribution function (CDF) equals the appropriate values for the tests. The CDF is given by

$$F(x) = \begin{cases} \frac{1}{2} betai\left(\frac{\nu}{2}, \frac{1}{2}, \frac{\nu}{\nu + x^2}\right) & x < 0 \\ 1 - \frac{1}{2} betai\left(\frac{\nu}{2}, \frac{1}{2}, \frac{\nu}{\nu + x^2}\right) & x \geq 0. \end{cases}$$

To get the critical value for the upper one-tail test we solve F(x)=1-alpha. For the lower one-tail test we solve for x the equation F(x)=alpha. In the two-tailed test the lower critical value is a solution for F(x)=alpha/2 and the upper critical value is a solution for F(x)=1-alpha/2.

The T-test assumes both samples are randomly taken from normal population distributions.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsStudentCDF**, **StatsStudentPDF**, and **StatsInvStudentCDF**.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999. See in particular Section 8.1.

StatsTukeyTest

`StatsTukeyTest [`*`flags`*`] [`*`wave1, wave2,… wave100`*`]`

The StatsTukeyTest operation performs multiple comparison Tukey (HSD) test and optionally the Newman-Keuls test. Output is to the M_TukeyTestResults wave in the current data folder. StatsTukeyTest usually follows **StatsANOVA1Test**.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/NK	Computes the Newman-Keuls test.
/Q	No results printed in the history area.
/SWN	Creates a text wave, T_TukeyDescriptors, containing wave names corresponding to each row of the comparison table (Save Wave Names). Use /T to append the text wave to the last column.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors.

Details

Inputs to StatsTukeyTest are two or more 1D numeric waves (one wave for each group of samples) containing any numbers of points but with at least two or more valid entries.

The contents of the M_TukeyTestResults columns are: the first contains the difference between the group means $\bar{X}_i - \bar{X}_i$, the second contains SE (supports unequal number of points), the third contains the q statistic for the pair, and the fourth contains the critical q value, the fifth contains the conclusion with 0 to reject H_0 ($\mu_i == \mu_j$) or 1 to accept H_0, with /NK, the sixth contains the *p* values

$$p = rank[\bar{X}_i] - rank[\bar{X}_j] + 1,$$

the seventh contains the critical values, and the eighth contains the Newman-Keuls conclusion (with 0 to reject and 1 to accept H_0). The order of the rows is such that all possible comparisons are computed sequentially starting with the comparison of the group having the largest mean with the group having the smallest mean.

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test**, **StatsScheffeTest**, and **StatsDunnettTest**.

StatsUSquaredCDF

`StatsUSquaredCDF(`*`u2`*`, `*`n`*`, `*`m`*`, `*`method`*`, `*`useTable`*`)`

The StatsUSquaredCDF function returns the cumulative distribution function for Watson’s U^2 with parameters *u2* (U^2 statistic) and integer sample sizes *n* and *m*. The calculation is computationally intensive, on the order of binomial(*n*+*m*, *m*). Use a nonzero value for *useTable to* search a built-in table of values. If *n* and *m* cannot be found in the table, it will proceed according to *method*:

method	**What It Does**
0	Exact computation using Burr algorithm (could be slow).
1	Tiku approximation using chi-squared.
2	Use built-in table only and return a NaN if not in table.

For large *n* and *m*, consider using the Tiku approximation. To abort execution, press the **User Abort Key Combinations**.

Precomputed tables, using the algorithm described by Burr, contain these values:

n	***m***
4	4-30
5	5-30
6	6-30
7	7-30

n	*m*
8	8-26
9	9-22
10	10-18
11	11-16
12	12-14
13	13

Because *n* and *m* are interchangeable, *n* should always be the smaller value. For *n*>8 the upper limit in the table matched the maximum that can be computed using the Burr algorithm. There is no point in using method 0 with *m* values exceeding these limits.

References

Burr, E.J., Small sample distributions of the two sample Cramer-von Mises' W^2 and Watson's U^2, *Ann. Mah. Stat. Assoc.*, *64*, 1091-1098, 1964.

Tiku, M.L., Chi-square approximations for the distributions of goodness-of-fit statistics, *Biometrica*, *52*, 630-633, 1965.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWatsonUSquaredTest** and **StatsInvUSquaredCDF** functions.

StatsVariancesTest

StatsVariancesTest [***flags***] [***wave1, wave2,… wave100***]

The StatsVariancesTest operation performs Bartlett's or Levene's test to determine if wave variances are equal. Output is to the W_StatsVariancesTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/METH=*m* Specifies the test type.

- *m*=0: Bartlett test (default).
- *m*=1: Levene's test using the mean.
- *m*=2: Modified Levene's test using the median.
- *m*=3: Modified Levene's test using the 10% trimmed mean.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

- *k*=0: Normal with dialog (default).
- *k*=1: Kills with no dialog.
- *k*=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

All tests define the null hypothesis by

$$H_0: \qquad \sigma_1^2 = \sigma_2^2 = \ldots = \sigma_k^2,$$

against the alternative

$$H_a: \qquad \sigma_i^2 \neq \sigma_j^2 \ \textit{for at least one } i \neq j.$$

Bartlett’s test computes:

$$T = \frac{(n-k)\ln\left(\sigma_w^2\right) - \sum_{i=1}^{k}(n_i - 1)\ln\left(\sigma_i^2\right)}{1 + \frac{1}{3(k-1)}\left[\sum_{i=1}^{k}\frac{1}{n_i - 1} - \frac{1}{N-k}\right]}.$$

Here $\sigma^2{}_i$ is the variance of the *i*th wave, *N* is the sum of the points of all the waves, n_i is the number of points in wave *i*, and *k* is the number of waves. The weighted variance is given by

$$\sigma_w^2 = \sum_{i=1}^{k}\frac{(n_i - 1)\sigma_i^2}{N-k}.$$

H_0 is rejected if T is greater than the critical value taken from the χ^2 distribution computed by solving for *x*:

$$1 - alpha = 1 - gammq\left(\frac{k-1}{2}, \frac{x}{2}\right).$$

Levene’s test computes:

$$W = \frac{(N-k)\sum_{i=1}^{k} n_i\left(\overline{Z}_i - \overline{Z}\right)^2}{(k-1)\sum_{i=1}^{k}\sum_{j=1}^{k}\left(Z_{ij} - \overline{Z}_i\right)^2},$$

where

$$Z_{ij} = \left|Y_{ij} - \overline{Y}_i\right|,$$

$$\overline{Z}_i = \frac{1}{n_i}\sum_{j=1}^{k} Z_{ij},$$

$$\overline{Z} = \frac{1}{N}\sum_{i=1}^{k}\sum_{j=1}^{k} Z_{ij}.$$

$\bar{Y}_i$ depends on /METH.

H_0 is rejected if *W* is greater than the critical value from the F distribution computed by solving for *x*:

$$1 - alpha = 1 - betai\left(\frac{v_2}{2}, \frac{v_1}{2}, \frac{v_2}{v_2 + v_1 x}\right).$$

References

NIST/SEMATECH, Bartlett's Test, in *NIST/SEMATECH e-Handbook of Statistical Methods*, <`http://www.itl.nist.gov/div898/handbook/eda/section3/eda357.htm`>, 2005.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsVonMisesCDF

`StatsVonMisesCDF(`*`x`*`, `*`a`*`, `*`b`*`)`

The StatsVonMisesCDF function returns the von Mises cumulative distribution function

$$F(\theta;a,b)=\frac{1}{2\pi I_0(b)}\int_0^{\theta}\exp\big(b\cos(x-a)\big)dx.$$

where $I_0(b)$ is the modified Bessel function of the first kind (**bessI**), and

$$0<\theta\le 2\pi$$
$$0<a\le 2\pi$$
$$b>0.$$

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsVonMisesPDF**, **StatsInvVonMisesCDF**, and **StatsVonMisesNoise** functions.

StatsVonMisesNoise

`StatsVonMisesNoise(`*`a`*`, `*`b`*`)`

The StatsVonMisesNoise function returns a pseudo-random number from a von Mises distribution whose probability density is

$$f(\theta;a,b)=\frac{\exp\left[b\cos(\theta-a)\right]}{2\pi I_0(b)},$$

where I_0 is the zeroth order modified Bessel function of the first kind.

References

Best, D.J., and N. I. Fisher, Efficient simulation of von Mises distribution, *Appl. Statist.*, *28*, 152-157, 1979.

See Also

StatsVonMisesCDF, **StatsVonMisesPDF**, and **StatsInvVonMisesCDF**.

Noise Functions on page III-390.

Chapter III-12, **Statistics** for a function and operation overview

StatsVonMisesPDF

`StatsVonMisesPDF(`*`q`*`, `*`a`*`, `*`b`*`)`

The StatsVonMisesPDF function returns the von Mises probability distribution function

$$f(\theta;a,b)=\frac{\exp\big(b\cos(\theta-a)\big)}{2\pi I_0(b)}.$$

where $I_0(b)$ is the modified Bessel function of the first kind **bessI**, and

$$0 < \theta \le 2\pi$$
$$0 < a \le 2\pi$$
$$b > 0.$$

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsVonMisesCDF**, **StatsInvVonMisesCDF**, and **StatsVonMisesNoise** functions.

StatsWaldCDF

`StatsWaldCDF(`*`x`*`, `*`m`*`, `*`l`*`)`

The StatsWaldCDF function returns the numerically evaluated inverse Gaussian or Wald cumulative distribution function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWaldPDF** function.

StatsWaldPDF

`StatsWaldPDF(`*`x`*`, `*`m`*`, `*`l`*`)`

The StatsWaldPDF function returns the inverse Gaussian or Wald probability distribution function

$$f(x;\mu,\lambda) = \sqrt{\frac{\lambda}{2\pi x^3}}\exp\left[-\frac{\lambda(x-\mu)^2}{2\mu^2 x}\right]$$

where *x*, *m*, *l*> 0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWaldCDF** function.

StatsWatsonUSquaredTest

`StatsWatsonUSquaredTest [`*`flags`*`] `*`srcWave1, srcWave2`*

The StatsWatsonUSquaredTest operation performs Watson’s nonparametric two-sample U^2 test for samples of circular data. Output is to the W_WatsonUtest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.
/Z	Ignores errors.

Details

The input waves, *srcWave1* and *srcWave2*, each must contain at least two angles in radians (mod 2π), can have any number of dimensions, and can be single or double precision. They must not contain any NaNs or INFs.

The Watson U^2 H_0 postulates that the two samples came from the same population against the different populations alternative. In the calculation, StatsWatsonUSquaredTest ranks the two inputs, accounts for possible ties, computes the test statistic U^2, and compares it with the critical value. Because of the difficulty of computing the critical values, it always computes first the approximation due to Tiku and if possible it computes the exact critical value using the method outlined by Burr. You can evaluate the U2 CDF to get more information about the critical region.

V_flag will be set to -1 for any error and to zero otherwise.

References

We have found that this method leads to slightly different results depending on the compiler and the system on which it is implemented:

Burr, E.J., Small sample distributions of the two sample Cramer-von Mises' W2 and Watson's U2, *Ann. Mah. Stat. Assoc.*, *64*, 1091-1098, 1964.

Tiku, M.L., Chi-square approximations for the distributions of goodness-of-fit statistics, *Biometrica*, *52*, 630-633, 1965.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWatsonWilliamsTest**, **StatsWheelerWatsonTest**, **StatsUSquaredCDF**, and **StatsInvUSquaredCDF**.

StatsWatsonWilliamsTest

```
StatsWatsonWilliamsTest [flags] [srcWave1, srcWave2, srcWave3,…]
```

The StatsWatsonWilliamsTest operation performs the Watson-Williams test for two or more sample means. Output is to the W_WatsonWilliams wave in the current data folder or optionally to a table.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors.

Details

The StatsWatsonWilliamsTest must have at least two input waves, which contain angles in radians, can be single or double precision, and can be of any dimensionality; the waves must not contain any NaNs or INFs.

The Watson-Williams H_0 postulates the equality of the means from all samples against the simple inequality alternative. The test computes the sums of the sines and cosines from which it obtains a weighted r value (rw). According to Mardia, you should use different statistics depending on the size of rw: for rw>0.95 use the simple F statistic, but for 0.95>rw>0.7 you should use the F-statistic with the K correction factor. Otherwise you should use the t-statistic. StatsWatsonWilliamsTest computes both the (corrected) F-statistic and the t-statistic as well as their corresponding critical values.

V_flag will be set to -1 for any error and to zero otherwise.

References

See, in particular, Section 6.3 of:

Mardia, K.V., *Statistics of Directional Data*, Academic Press, New York, New York, 1972.

See, in particular, Chapter 27 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWatsonUSquaredTest** and **StatsWheelerWatsonTest**.

StatsWeibullCDF

`StatsWeibullCDF(`*`x`*`, `*`m`*`, `*`s`*`, `*`g`*`)`

The StatsWeibullCDF function returns the Weibull cumulative distribution function

$$F(x;\mu,\sigma,\gamma) = 1 - \exp\left[-\left(\frac{x-\mu}{\sigma}\right)^{\gamma}\right], \qquad x \ge \mu \text{ and } \sigma,\gamma > 0.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWeibullPDF** and **StatsInvWeibullCDF** functions.

StatsWeibullPDF

`StatsWeibullPDF(`*`x`*`, `*`m`*`, `*`s`*`, `*`g`*`)`

The StatsWeibullPDF function returns the Weibull probability distribution function

$$f(x;\mu,\sigma,\gamma) = \frac{\gamma}{\sigma}\left(\frac{x-\mu}{\sigma}\right)^{\gamma-1} \exp\left[-\left(\frac{x-\mu}{\sigma}\right)^{\gamma}\right],$$

where *m* is the location parameter, *s* is the scale parameter, and *g* is the shape parameter with *x* ≥ *m* and *s*, *g* > 0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWeibullCDF** and **StatsInvWeibullCDF** functions.

StatsWheelerWatsonTest

`StatsWheelerWatsonTest` [***`flags`***] [**`srcWave1, srcWave2, srcWave3,`**…]

The StatsWheelerWatsonTest operation performs the nonparametric Wheeler-Watson test for two or more samples. Output is to the W_WheelerWatson wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors.

Details

The StatsWatsonWilliamsTest must have at least two input waves, which contain angles in radians (mod 2π), can be single or double precision, and can be of any dimensionality; the waves must not contain any NaNs or INFs.

The Wheeler-Watson H_0 postulates that the samples came from the same population. The extension of the test to more than two samples is due to Mardia. The Wheeler-Watson test is not valid for data with ties, in which case you should use Watson's U^2 test.

V_flag will be set to -1 for any error and to zero otherwise.

References

Mardia, K.V., *Statistics of Directional Data*, Academic Press, New York, New York, 1972.

See, in particular, Chapter 27 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWatsonUSquaredTest** and **StatsWheelerWatsonTest**.

StatsWilcoxonRankTest

StatsWilcoxonRankTest [***flags***] ***waveA*, *waveB***

The StatsWilcoxonRankTest operation performs the nonparametric Wilcoxon-Mann-Whitney two-sample rank test or the Wilcoxon Signed Rank test (for paired data) on *waveA* and *waveB*. Output is to the W_WilcoxonTest wave in the current data folder or optionally to a table.

waveA and *waveB* must not contain NaNs or INFs.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/APRX=*m* Sets the approximation method. It computes an exact critical value by default.

m=1: Standard normal approximation with ties (Zar P. 151).

m=2: Improved normal approximation (Zar P. 152).

Approximations may be appropriate for large sample sizes when computation may take a long time.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0: Normal with dialog (default).

k=1: Kills with no dialog.

k=2: Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/TAIL=*tail* *tail* is a bitwise parameter that specifies the tails tested.

Bit 0: Lower tail.

Bit 1: Upper tail (default).

Bit 2: Two tail.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

You can perform any combination of tests by adding their corresponding tail values (/TAIL=7 tests all tail possiblities). Note that H0 changes according to the selected tail.

/WSRT Performs the Wilcoxon Signed Rank Test for paired data. The test computes statistics Tp and Tm, lower-tail, upper-tail, and two-tail P-values. If the number of samples is less than 200 it computes exact P-values, otherwise they are computed using the normal approximation. Do not use /ALPH, /APRX, and /TAIL with this flag.

/Z Ignores errors.

Details

The Wilcoxon-Mann-Whitney test combines the two samples and ranks them to compute the statistic U. If *waveA* has *m* points and *waveB* has *n* points, then *U* is given by

$$U = mn + \frac{m(m+1)}{2} - R_1,$$

with the corresponding statistic *U*' given by

$$U' = nm + \frac{n(n+1)}{2} - R2.$$

where R_i is the ranks of data in the *i*th wave (ranked in ascending order).

The distribution of *U* is difficult to compute, requiring the number of possible permutations of *m* elements of *waveA* and *n* elements of *waveB* that give rise to *U* values that do not exceed the one computed. The distribution is computed according to the algorithm developed by Klotz. With increasing sample size one can avoid the time consuming distribution computation and use a normal approximation instead. Klotz recommends this approximation for *N*=*m*+*n*~100.

Use /APRX=2 for the best approximation. The two approximations are discussed by Zar.

The Wilcoxon Signed Rank Test, or Wilcoxon Paired-Sample Test, ranks the difference between pairs of values and computes the sums of the positive ranks (Tp) and the negative ranks (Tm). It calculates Tp and Tm and P-values for all tail combinations. The P-values are:

P_lower_tail	P(Wp<=Tp)
P_upper_tail	P(Wp>=Tp)
P_two_tail	2*Min(P_lower_tail,P_upper_tail)

Wp is the generic symbol for the sum of positive ranks for the given number of pairs.

V_flag will be set to -1 for any error and to zero otherwise.

In both Wilcoxon-Mann-Whitney two-sample rank test and the Wilcoxon Signed Rank test H0 is that the data in the two input waves are statistically the same.

References

Cheung, Y.K., and J.H. Klotz, The Mann Whitney Wilcoxon distribution using linked lists, *Statistica Sinica*, *7*, 805-813, 1997.

See in particular Chapter 15 of:

Klotz, J.H., *Computational Approach to Statistics*.

Streitberg, B., and J. Rohmel, Exact distributions for permutations and rank tests: An introduction to some recently published algorithms, *Statistical Software Newsletter*, *12*, 10-17, 1986.

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsAngularDistanceTest, **StatsKWTest**, **StatsWilcoxonRankTest**

StatsWRCorrelationTest

`StatsWRCorrelationTest [`*`flags`*`] `*`waveA, waveB`*

The StatsWRCorrelationTest operation performs a Weighted Rank Correlation test on *waveA* and *waveB*, which contain the ranks of sequential factors. The waves are 1-based, integer ranks of factors in the range 1-2^31.

StatsWRCorrelationTest computes a top-down correlation coefficient using Savage sums as well as the critical and P-values. Output is to the W_StatsWRCorrelationTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val* Sets the significance level (default *val*=0.05).

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/Z Ignores errors.

Details

The StatsWRCorrelationTest input waves must be one-dimensional and have the same length. The waves are 1-based, integer ranks of factors corresponding to the point number. Ranks may have ties in which case you should repeat the rank value. For example, if the second and third entries have the same rank you should enter {1,2,2,4}. H_0 stipulates that the same factors are most important in both groups represented by *waveA* and *waveB*.

The top-down correlation is the sum of the product of Savage sums for each row:

$$r_{TD} = \frac{\sum_{i=1}^{n} S_{iA} S_{iB} - n}{n - S_1},$$

where n is the number of rows and the Savage sum S_i is

$$S_i = \sum_{j=i}^{n} \frac{1}{j},$$

and S_{iA} corresponds to the S_i value of the rank of the data in row (i-1) of *waveA*.

References

Iman, R.L., and W.J. Conover, A measure of top-down correlation, *Technometrics*, *29*, 351-357, 1987.

See, in particular, Chapter 19 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsLinearCorrelationTest**, **StatsRankCorrelationTest**, **StatsTopDownCDF**, and **StatsInvTopDownCDF**.

STFT

`STFT [flags] *srcWave*`

The STFT operation computes the Short-Time Fourier Transform of *srcWave*. STFT was added in Igor Pro 8.00.

Output is stored in the wave M_STFT in the current data folder or in a wave specified by the /DEST flag.

Flags

/DB=*dbMode* — *dbMode* determines if output is scaled in decibels:

0:	No dB scaling (default)
1:	Scale the output to standard dB using 20*log(wave)
2:	Compute dBFS (dB relative to full scale where 0 is the maximum value)

/DEST=*destWave* — Specifies the output wave created by the FFT operation.

It is an error to attempt specify the same wave as both *srcWave* and *destWave*.

The default output wave name M_STFT is used if you omit /DEST.

When used in a function, the STFT operation by default creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-72 for details.

/HOPS=*hopSize* — Specifies the offset in points between centers of consecutive source segments. By default this value is 1 and the transform is computed for segments that are offset by a single points from each other.

/OUT=*mode* — Sets the output wave format.

mode=1:	Complex output (default)
mode=2:	Real output
mode=3:	Magnitude
mode=4:	Magnitude square
mode=5:	Phase
mode=6:	Scaled magnitude
mode=7:	Scaled magnitude squared

The scaled quantities apply to transforms of real valued inputs where the output is normally folded in the first dimension (because of symmetry). The scaling applies a factor of 2 to the squared magnitude of all components except the DC. The scaled transforms should be used whenever Parseval's relation is expected to hold.

/PAD=*newSize* — Converts each segment of *srcWave* into a padded array of length *newSize*. The padded array contains the original data at the center of the array with zeros elements on both sides.

/RP=[*startPoint*, *endPoint*]

Specifies the range of *srcWave* from which data are sampled in point numbers. *startPoint* is the first point at which segments are centered. Wave data from points preceding *startPoint* are used as needed for the left parts of beginning segments.

/RX=(*startX*, *endX*) — Specifies the range of *srcWave* from which data are sampled in X values. The operation expects *startX*<*endX*. *startX* corresponds to the first point at which segments are centered. Data from points preceding *startX* are used as necessary to fill left parts of the beginning segments.

/SEGS=*segSize* — Sets the length of the segment sampled from *srcWave* in points. The segment is optionally padded to a larger dimension (see /PAD) and multiplied by a window function prior to FFT. The default segment size is 128 when the number of points in *srcWave* is greater than 128. Otherwise it is set to one less than the number of points in the *srcWave*. The operation requires that *segSize* is at least 32 points.

/WINF=*windowKind*

Premultiplies a data seggment with the selected window function. The default window is Hanning. See **Window Functions** on page V-225 for details.

/Z	Ignores errors. V_flag is set to -1 for any error and to zero otherwise.

Details

The Short-Time Fourier Transform is a time-frequency representation for a 1D array. The squared magnitude of the transform is known as the "spectrogram" for time series or "sonogram" in the case of sound input. The operation comprises the following steps:

1. Sampling the data.

   A segment of size specified by /SEGS is sampled from *srcWave* centered about the first point in the wave or as specified by /RX or /RP flags. When the first point is at the beginning of the wave the centering implies that the first half of the segment is set to zero. If the first point is somewhere else the operation uses as many points as are available in the wave and sets the rest to zero. End effects are mitigated by scaling the result by a factor that accounts for the actual number of source data used in the segment. Subsequent segments are each centered at *hopSize* points from the previous center.

2. Apply windowing.

   The selected segment is multiplied by the specified window function.

3. Apply padding.

   If padding is specified the windowed data are centered onto a zero padded array. Padding may be used to simulate longer arrays for improved spectral resolution.

4. Compute the FFT.

   Initial complex transform may be further processed according to the /OUT flag.

See Also

Fourier Transforms on page III-270, **FFT**, **CWT**, **WignerTransform**, **DSPPeriodogram**, **LombPeriodogram**

StopMSTimer

StopMSTimer(*timerRefNum*)

The StopMSTimer function frees up the timer associated with the *timerRefNum* and returns the number of elapsed microseconds since StartMSTimer was called for this timer.

Parameters

timerRefNum is the value returned by StartMSTimer or the special values -1 or -2. If *timerRefNum* is not valid then StopMSTimer returns 0.

On Windows, passing -1 returns the clock frequency of the timer and. On Macintosh, it returns NaN.

Passing -2 returns the time in microseconds since the computer was started.

Details

If you want to make sure that all timers are free, call StopMSTimer ten times with *timerRefNum* equal to 0 through 9. It is OK to stop a timer that you never started.

The function result may exclude the time the system has spent in sleep states. As of this writing this applies to Macintosh only, but this behavior may change in the future since it is determined by the operating system.

Examples

How long does an empty loop take on your computer?

```
Function TestMSTimer()
   Variable timerRefNum
   Variable microSeconds
   Variable n
```

```
    timerRefNum = StartMSTimer
    if (timerRefNum == -1)
        Abort "All timers are in use"
    endif
    n=10000
    do
        n -= 1
    while (n > 0)
    microSeconds = StopMSTimer(timerRefNum)
    Print microSeconds/10000, "microseconds per iteration"
End
```

See Also

StartMSTimer, **ticks**, **DateTime**

str2num

`str2num(`*`str`*`)`

The str2num function returns a number represented by the string expression *str*.

Details

str2num returns NaN if *str* does not contain the text for a number.

str2num skips leading spaces and tabs and then reads up to the first non-numeric character.

See Also

The **char2num**, **num2char** and **num2str** functions.

The **sscanf** operation for more complex parsing jobs.

StrConstant

`StrConstant `*`ksName`*`="`*`literal string`*`"`

The StrConstant declaration defines the string *literal string* under the name *ksName* for use by other code, such as in a switch construct.

See Also

The **Constant** keyword for numeric types, **Constants** on page IV-51, and **Switch Statements** on page IV-43.

String

`String [/G] `*`strName`*`[/N=`*`name`*`][=`*`strExpr`*`][, `*`strName`*`[/N=`*`name`*`][=`*`strExpr`*`]…]`

The String operation creates string variables and gives them the specified names.

Flags

/G	Creates a global string. Overwrites any existing string with the same name.
/N=*name*	Specifies a local name for the global string variable. /N was added in Igor Pro 8.00 and is available in user-defined functions only. See **SVAR Creation** below for details.

Details

The string variable is initialized when it is created if you supply the =*strExpr* initializer. However, when String is used to declare a function parameter, it is an error to attempt to initialize it.

You can create more than one string variable at a time by separating the names and optional initializers with commas.

If used in a procedure, the new string is local to that procedure unless the /G flag is used. If used on the command line, String is equivalent to String/G.

strName can optionally include a data folder path.

SVAR Creation

In a user-defined function, you need a local SVAR reference to access a global string variable. If you use a simple name rather than a path, Igor automatically creates an SVAR:

```
String/G sVar1                    // Creates an SVAR named sVar1
```

If you use a path or a $ expression, Igor does not create an automatic SVAR reference. You can explicitly create SVARs like this:

```
String/G root:sVar2
SVAR sVar2 = root:sVar2          // Creates an SVAR named sVar2

String path = "root:sVar3"
String/G $path
SVAR sVar3 = $path               // Creates an SVAR named sVar3
```

In Igor Pro 8.00 and later, you can explicitly create an SVAR reference in a user-defined function using the /N flag, like this:

```
String/G sVar4/N=sVar4           // Creates an SVAR named sVar4

String/G root:sVar5/N=sVar5      // Creates an SVAR named sVar5

String path = "root:sVar6"
String/G $path/N=sVar6           // Creates an SVAR named sVar6
```

The name used for the SVAR does not need to be the same as the name of the global variable:

```
String/G sVar7/N=sv7             // Creates an SVAR named sv7

String/G root:sVar8/N=sv8        // Creates an SVAR named sv8
String path = "root:sVar9"

String/G $path/N=sv9             // Creates an SVAR named sv9
```

See Also

String Variables on page II-105, **Working With Strings** on page IV-13, **Accessing Global Variables and Waves** on page IV-65

StringByKey

StringByKey(*keyStr*, *kwListStr* [, *keySepStr* [, *listSepStr* [, *matchCase*]]])

The StringByKey function returns a substring extracted from *kwListStr* based on the specified key contained in *keyStr*. *kwListStr* should contain keyword-value pairs such as `"KEY=value1,KEY2=value2"` or `"Key:value1;KEY2:value2"`, depending on the values for *keySepStr* and *listSepStr*.

Use StringByKey to extract a string value from a string containing a `"key1:value1;key2: value2;"` style list such as those returned by functions like **AxisInfo** or **TraceInfo**.

If the key is not found or if any of the arguments is `""` then a zero-length string is returned.

keySepStr, *listSepStr,* and *matchCase* are optional; their defaults are ":", ";", and 0 respectively.

Details

kwListStr is searched for an instance of the key string bound by *listSepStr* on the left and a *keySepStr* on the right. The text up to the next *listSepStr* is returned.

kwListStr is treated as if it ends with a *listSepStr* even if it doesn’t.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are usually case-insensitive. Setting the optional *matchCase* parameter to 1 makes the comparisons case sensitive.

In Igor6, only the first byte of *keySepStr* and *listSepStr* was used. In Igor7 and later, all bytes are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *matchCase* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print StringByKey("BKEY", "AKEY:hello;BKEY:nok-nok")  // prints "nok-nok"
Print StringByKey("KY", "KX=1;ky=hello", "=")         // prints "hello"
Print StringByKey("KY", "KX:1,KY:joey,", ":", ",")    // prints "joey"
Print StringByKey("kz", "KZ:1st,kz:2nd,", ":", ",")   // prints "1st"
Print StringByKey("kz", "KZ:1st,kz:2nd,", ":", ",", 1)// prints "2nd"
```

See Also

The **NumberByKey**, **RemoveByKey**, **ReplaceNumberByKey**, **ReplaceStringByKey**, **ItemsInList**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

StringCRC

StringCRC(*inCRC*,*str*)

The StringCRC function returns a 32-bit cyclic redundancy check value of bytes in *str* starting with *inCRC*.

Pass 0 for *inCRC* the first time you call StringCRC for a particular stream of bytes as represented by the string data.

Pass the last-returned value from StringCRC for *inCRC* if you are creating a CRC value for a given stream of bytes through multiple calls to StringCRC.

Details

Polynomial used is:

x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x+1

See crc32.c in the public domain source code for zlib for more information.

See Also

Hash, **WaveHash**, **WaveCRC**

StringFromList

StringFromList(*index*, *listStr* [, *listSepStr*] [, *offset*])

The StringFromList function returns the *index*th substring extracted from *listStr* starting *offset* bytes into *listStr*. *listStr* should contain items separated by *listSepStr* which typically is ";".

Use StringFromList to extract an item from a string such as those returned by functions like **TraceNameList** and **AnnotationList**.

Parameters

index is the zero-based index of the list item that you want to get. If *index* < 0, or *index* ≥ the number of items in list, or if *listStr* or *listSepStr* is `""`, then a zero-length string is returned.

listStr contains a series of text items separated by *listSepStr*. The trailing separator is optional though recommended. For example, these are both valid lists:

```
"First;Second;"
"First;Second"
```

listSepStr is optional. If omitted it defaults to ";". Prior to Igor Pro 7.00, only the first byte of *listSepStr* was used. Now all bytes are used.

offset is optional and requires Igor Pro 7.00 or later. If omitted it defaults to 0. The search begins *offset* bytes into *listStr*. When iterating through lists containing large numbers of items, using the *offset* parameter provides dramatically faster execution.

Details

For optimal performance, especially with lists larger than 100 items, provide the *separatorStr* and *offset* parameters as shown in the DemoStringFromList example below. When using this technique, the *index* parameter must be 0 and the *offset* parameter controls which list item is returned.

Examples

```
Print StringFromList(0, "wave0;wave1;")              // Prints "wave0"
Print StringFromList(2, "wave0;wave1;")              // Prints ""
Print StringFromList(1, "wave0;;wave2")              // Prints ""

// Iterate quickly over a list using the offset parameter
Function DemoQuickStringFromList(list)
    String list      // A semicolon-separated string list

    String separator = ";"
    Variable separatorLen = strlen(separator)
    Variable numItems = ItemsInList(list, separator)

    Variable offset = 0
    Variable i
    for(i=0; i<numItems; i+=1)
        // When using offset, the index parameter is always 0
        String item = StringFromList(0, list, separator, offset)
        // Do something with item
        offset += strlen(item) + separatorLen
    endfor
End
```

See Also

The **AddListItem**, **ItemsInList**, **FindListItem**, **RemoveListItem**, **RemoveFromList**, **WaveList**, **WhichListItem**, **StringByKey**, **ListMatch**, **ControlNameList**, **TraceNameList**, **StringList**, **VariableList**, and **FunctionList** functions.

StringList

`StringList(`*`matchStr`*`, `*`separatorStr`*` [, `*`dfr`*` ])`

The StringList function returns a string containing a list of global string variables selected based on the *matchStr* parameter. The string variables listed are all in the current data folder or the data folder specified by *dfr*.

Details

For a string variable name to appear in the output string, it must match *matchStr*. *separatorStr* is appended to each string variable name as the output string is generated.

The name of each string variable is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all string variable names
"xyz"	Matches name xyz only
"*xyz"	Matches names which end with xyz
"xyz*"	Matches names which begin with xyz
"*xyz*"	Matches names which contain xyz
"abc*xyz"	Matches names which begin with abc and end with xyz

The returned list contains names only, without data folder paths. Thus, they are not suitable for accessing string variables outside the current or specified data folder.

matchStr may begin with the ! character to return items that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches variable names which *do not* end with xyz

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

dfr is an optional data folder reference: a data folder name, an absolute or relative data folder path, or a reference returned by, for example, **GetDataFolderDFR**. The *dfr* parameter requires Igor Pro 9.00 or later.

Examples

`StringList("*",";")`	Returns a list of all string variables in the current data folder.
`StringList("S_*", ";")`	Returns a list of all string variables in the current data folder whose names begin with “S_”.

See Also

VariableList, **WaveList**, **DataFolderList**

StringMatch

`StringMatch(`*`string`*`, `*`matchStr`*`)`

The StringMatch function tests *string* for a match to *matchStr*. You may include asterisks in *matchStr* as a wildcard character.

StringMatch returns 1 to indicate a match, 0 for no match or NaN if it ran out of memory.

Details

matchStr is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches any string.
"xyz"	Matches the string “xyz” only.
"*xyz"	Matches strings ending with “xyz”, for instance “abcxyz”.
"xyz*"	Matches strings beginning with xyz, for instance “xyzpqr”.
"*xyz*"	Matches strings containing xyz, for instance “abcxyzpqr”.
"abc*xyz"	Matches strings beginning with abc and ending with xyz, for instance “abcpqrxyz”.

If *matchStr* begins with the ! character, a match is indicated if *string* does *not* match *matchStr*. For example:

"!*xyz"	Matches strings which *do not* end with xyz.

The ! character is considered to be a normal character if it appears anywhere else.

Note that matching is case-insensitive, so “xyz” also matches “XYZ” or “Xyz”.

Also note that it is impossible to match an asterisk in *string*: use **GrepString** instead.

Among other uses, the StringMatch function can be used to build your own versions of the **WaveList** function, using **NameOfWave** and stringmatch to qualify names of waves found by **WaveRefIndexedDFR**.

See Also

The **GrepString**, **CmpStr**, **strsearch**, **Demo**, **ListMatch**, and **ReplaceString** functions and the **sscanf** operation.

StringToUnsignedByteWave

`StringToUnsignedByteWave(`*`str`*`)`

The StringToUnsignedByteWave function returns a free unsigned byte wave containing the contents of *str*.

If *str* is an empty string, a zero point wave is returned. If *str* is null, a null wave reference is returned.

The StringToUnsignedByteWave function was added in Igor Pro 9.00.

Parameters

str is a string.

Details

The StringToUnsignedByteWave function returns a free wave so it can't be used on the command line or in a macro. If you need to convert the free wave to a global wave use **MoveWave**.

Using StringToUnsignedByteWave makes it possible to use commands that work on waves to manipulate the data originally stored in a string. This can be faster and more convenient than manipulating the data directly in the string. If necessary, you can create a string containing the manipulated data using WaveDataToString.

StringToUnsignedByteWave stores each byte of *str* in an element of the output wave. It does not do ASCII-to-binary conversion. For example, if *str* contains "123", it returns an unsigned binary wave containing three elements with values 49, 50, and 51 (0x31, 0x32, and 0x33). It does not return the numeric value 123.

Example

```
Function DemoStringToUnsignedByteWave()
   String theStr = "123"
```

```
    // This requires Igor 9.00
    WAVE/B/U w1 = StringToUnsignedByteWave(theStr)
    Print w1

    // This works in older versions of Igor
    Variable len = strlen(theStr)
    Make/B/U/FREE/N=(len) w1
    w1 = char2num(theStr[p])
    Print w1
End
```

See Also

WaveDataToString, **MoveWave**, **Free Waves** on page IV-91, **Working With Binary String Data** on page IV-175

strlen

strlen(*str*)

The strlen function returns the number of bytes in the string expression *str*.

strlen returns NaN if the *str* is NULL. A local string variable or a string field in a structure that has never been set is NULL. NULL is not the same as zero length. Use **numtype** to test if the result from strlen is NaN.

Examples

```
String zeroLength = ""
String neverSet
Print strlen(zeroLength), strlen(neverSet)

// Test if a string is null
Variable len = strlen(neverSet)   // NaN if neverSet is null
if (numtype(len) == 2)            // strlen returned NaN?
    Print "neverSet is null"
endif
```

See Also

Characters Versus Bytes on page III-483, **Character-by-Character Operations** on page IV-173

strsearch

strsearch(*str*, *findThisStr*, *start*[, *options*])

The strsearch function returns the byte position of the string expression *findThisStr* in the string expression *str*.

Details

strsearch performs a case-sensitive search.

strsearch returns -1 if *findThisStr* does not occur in *str*.

The search starts from the byte position in *str* specified by *start*; 0 references the start of *str*.

strsearch clips *start* to one less than the length of *str* in bytes, so it is useful to use `Inf` for *start* when searching backwards to ensure that the search is from the end of *str*.

options is an optional bitwise parameter specifying the search options:

1:	Search backwards from *start*.
2:	Ignore case.
3:	Search backwards and ignore case.

Examples

```
String str="This is a test isn't it?"
Print strsearch(str,"test",0)                 // prints 10
Print strsearch(str,"TEST",0)                 // prints -1
Print strsearch(UpperStr(str),"TEST",0)       // prints 10
Print strsearch(str,"TEST",0,2)               // prints 10
Print strsearch(str,"is",0)                   // prints 2
Print strsearch(str,"is",3)                    // prints 5
Print strsearch(str,"is",Inf,1)                // prints 15
```

See Also

sscanf, **FindListItem**, **ReplaceString**, **Character-by-Character Operations**

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

strswitch-case-endswitch

```
strswitch(<string expression>)
   case <literal><constant>:
      <code>
      [break]
   [default:
      <code>]
endswitch
```

A strswitch-case-endswitch statement evaluates a string expression and compares the result to the case labels using a case-insensitive comparison. If a case label matches *string expression*, then execution proceeds with *code* following the matching case label. When none of the cases match, execution will continue at the default label, if it is present, or otherwise the strswitch will be exited with no action taken. Note that although the break statement is optional, in almost all case statements it will be required for the strswitch to work correctly.

See Also

Switch Statements on page IV-43, **default** and **break** for more usage details.

STRUCT

STRUCT *structureName localName*

STRUCT is a reference that creates a local reference to a Structure accessed in a user-defined function. When a Structure is passed to a user function, it can only be passed by reference, so in the declaration within the function you must use `&localStructName` to define the function input parameter.

See Also

Structures in Functions on page IV-99 for further information.

See the **Structure** keyword for creating a Structure definition.

StructFill

StructFill [/AC=*createFlags* /SDFR=*dfr*] *structVar*

StructFill is a programmer-convenience operation that initializes NVAR, SVAR and WAVE fields in a structure. At run time, it scans through the fields in the specified structure and attempts to set all null NVAR, SVAR and WAVE fields by looking up corresponding same-named globals in the current data folder or in the specified data folder.

StructFill was added in Igor Pro 8.00.

Parameters

structVar is the name of a STRUCT variable.

Flags

/AC=*createFlags*	Enables auto-creation of globals. Bit 0 enables creation of numeric and string global variables which are set to 0 and "" respectively. Variables are auto-created only if they do not already exist. Bit 1 enables creation of waves which are created with zero points. Waves are auto-created only if they do not already exist.
/SDFR=*dfr*	Specifies a data folder. If you omit /SDFR, the current data folder used. See **The /SDFR Flag** on page IV-80 for details.

Output Variables

StructFill sets the following output variables:

V_Flag	The number of NVAR, SVAR and WAVE fields that were successfully initialized.
V_Error	The number of NVAR, SVAR and WAVE fields that could not be initialized.

If auto-creation is off, a field can not be initialized if the corresponding global variable, string variable, or wave does not exist. If auto-creation is on, a field can not be initialized if there was an error creating the global variable or wave.

When using auto-creation, errors are not reported other than via V_Error.

If you unexpectedly get non-zero for V_Error, you can print the structure to see which fields were left null.

See Also

Structures in Functions on page IV-99

StructGet

StructGet [**/B=*b***] ***structVar*, *waveStruct***[**[*colNum*]**]

StructGet /S [**/B=*b***] ***structVar*, *strStruct***

The StructGet operation reads binary numeric data from a specified column of a wave or from a string variable and copies the data into the designated structure variable. The source wave or string will have been filled beforehand by **StructPut**.

Parameters

structVar is the name of an existing structure that is to be filled with new data values.

waveStruct is the name of a wave containing binary numeric data that will be used to fill *structVar*. Use the optional *colNum* parameter to specify a column from the structure wave. The contents of *waveStruct* are created beforehand using StructPut.

strStruct is the name of a string variable containing binary numeric data. The contents of *strStruct* are created beforehand using StructPut.

Flags

/B=*b* Sets the byte ordering for reading of structure data.

b=0:	Reads in native byte order.
b=1:	Reads bytes in reversed order.
b=2:	Default; reads data in big-endian, high-byte-first order (Motorola).
b=3:	Reads data in little-endian, low-byte-first order (Intel).

/S Reads binary data from a string variable, which was set previously with StructPut.

Details

The data that are stored in *waveStruct* and *strStruct* are in binary format so you can not directly view a meaningful representation of their contents by printing them or viewing the wave in a table. To view the contents of *waveStruct* or *strStruct* you must use StructGet to export them back into a structure and then retrieve the members.

If *colNum* is out of bounds it will be clipped to valid values and an error reported. If the row dimension does not match the structure size, as much data as possible will be copied to the structure.

By default, data are read in big-endian, high-byte order (Motorola). This allows data written on one platform to be read on the other.

See Also

The **StructPut** operation for writing structure data to waves or strings.

StructPut

`StructPut [/B=`*`b`*`] `*`structVar`*`, `*`waveStruct`*`[[`*`colNum`*`]]`

`StructPut /S [/B=`*`b`*`] `*`structVar`*`, `*`strStruct`*

The StructPut operation copies the binary numeric data in a structure variable to a specified column in a wave or to a string variable. The data in the wave or string can be read out into another structure using **StructGet**.

Parameters

structVar is the name of a structure from which data will be exported.

waveStruct is the name of an existing wave to which data will be exported. Use the optional *colNum* parameter to specify a column in *waveStruct* to contain the data. The first column of *waveStruct* will be filled if *colNum* is omitted.

strStruct is the name of an existing string variable to which data will be exported.

Flags

/B=*b*	Sets the byte ordering for writing of structure data.	
	b=0:	Writes in native byte order.
	b=1:	Writes bytes in reversed order.
	b=2:	Default; writes data in big-endian, high-byte-first order (Motorola).
	b=3:	Writes data in little-endian, low-byte-first order (Intel).
/S	Writes binary data to a string variable.	

Details

The structure fields to be exported must contain only numeric data in either integer, floating point, or double precision format. If the structure contains any objects such as String, NVAR, WAVE, etc., then only the numeric data at the end of the structure is copied. If there is no suitable data at the end, an error is generated at compile time. Prior to Igor Pro 8, the presence of any illegal field would result in an error.

If needed, StructPut will redimension *waveStruct* to unsigned byte format, will set the number of rows to equal the size of the structure, and set the column dimension large enough to accommodate the size specified by *colNum*. You can think of *waveStruct* as a one-dimensional array of structure contents indexed by *colNum* although the wave is actually two-dimensional with each column containing a copy of a separate structure.

By default, data are written in big-endian, high-byte order (Motorola). This allows data written on one platform to be read on the other.

After you have exported the structure data to *waveStruct* or *strStruct* they will contain binary data that you cannot inspect directly. To view the contents of *waveStruct* or *strStruct*, you must use the original structure or use StructGet to export them into another structure.

See Also

The **StructGet** operation for reading structure data from waves or strings.

Structure

Structure *structureName*
    memType memName [***arraySize***] [, ***memName*** [***arraySize***]]
    ...
EndStructure

The Structure keyword introduces a structure definition in a user function. Within the body of the structure you declare the member type (*memType*) and the corresponding member name(s) (*memName*). Each *memName* may be declared with an optional array size.

Details

Structure member types (*memType*) can be any of the following Igor objects: Variable, String, WAVE, NVAR, SVAR, DFREF, FUNCREF, or STRUCT.

Igor structures also support additional member types, as given in the next table, for compatibility with C programming structures and disk files.

Igor Member Type	C Equivalent	Size	Note
char	signed 8-bit int	1 byte	
uchar	unsigned 8-bit int	1 byte	
int16	signed 16-bit int	2 bytes	
uint16	unsigned 16-bit int	2 bytes	
int32	signed 32-bit int	4 bytes	
uint32	unsigned 32-bit int	4 bytes	
int64	signed 64-bit int	8 bytes	Requires Igor Pro 7.00 or later
uint64	unsigned 64-bit int	8 bytes	Requires Igor Pro 7.00 or later
float	float	4 bytes	
double	double	8 bytes	

The Variable and double types are identical although Variable can be also specified as complex (using the /C flag).

Each structure member may have an optional *arraySize* specification, which gives the number of elements contained by the structure member. The array size is an integer number from 1 to 400 except for members of type STRUCT for which the upper limit is 100.

See Also

Structures in Functions on page IV-99 for further information.

See the **STRUCT** declaration for creating a local reference to a Structure.

StrVarOrDefault

StrVarOrDefault(*pathStr*, *defStrVal*)

The StrVarOrDefault function checks to see if *pathStr* points to a string variable and if so, it returns its value. If the string variable does not exist, returns *defStrVal* instead.

Details

StrVarOrDefault initializes input values of macros so they can remember their state without needing global variables to be defined first. Numeric variables use the corresponding numeric function, **NumVarOrDefault**.

Examples

```
Macro foo(nval,sval)
   Variable nval=NumVarOrDefault("root:Packages:mypack:nvalSav",2)
   String sval=StrVarOrDefault("root:Packages:mypack:svalSav","Hi")

   DFREF dfSav= GetDataFolderDFR()
   NewDataFolder/O/S root:Packages
```

```
    NewDataFolder/O/S mypack
    Variable/G nvalSav= nval
    String/G svalSav= sval
    SetDataFolder dfSav
End
```

StudentA

StudentA(*t, DegFree*)

Note: This function is deprecated. New code should use the more accurate **StatsStudentCDF**.

The StudentA function returns the area from -*t* to *t* under the Student's T distribution having *DegFree* degrees of freedom. That is, it returns the probability that a random sample from Student's T is between -*t* and *t*.

Note that this is the *bi-tail* result. That is, it gives the area from -*t* to *t*, rather than the cumulative area from -∞ to *t*. It is this latter number that is commonly tabulated- StudentA returns the probability 1-α where the area from -∞ to t is the probability 1-α/2.

StudentA tests whether a normally-distributed statistic is significantly different from a certain value. You could use it to test whether an intercept from a line fit is significantly different from zero:

```
Make/O/N=20 Data=0.5*x+2+gnoise(1)      // line with Gaussian noise
Display Data
CurveFit line Data /D
Print "Prob = ", StudentA(W_coef[0]/W_sigma[0], V_npnts-2)
```

Because the noise is random, the results will differ slightly each time this is tried. When we did it, the result was:

```
Prob = 0.999898
```

which indicates that the intercept of the line fit was different from zero with 99.99 per cent probability.

See Also

StatsStudentCDF, **StatsStudentPDF**, **StatsInvStudentCDF**

StudentT

StudentT(*Prob, DegFree*)

Note: This function is deprecated. New code should use the more accurate **StatsInvStudentCDF**.

The StudentT function returns the t value corresponding to an area *Prob* under the Student's T distribution from -*t* to *t* for *DegFree* degrees of freedom.

Note that this is a *bi-tail* result, which is what is usually desired. Tabulated values of the Student's T distribution are commonly the one-sided result.

StudentT calculates confidence intervals from standard deviations for normally-distributed statistics. For instance, you can use it to calculate a confidence interval for the coefficients from a curve fit:

```
Make/O/N=20 Data=0.5*x+2+gnoise(1)      // line with Gaussian noise
Display Data
CurveFit line Data /D
print "intercept = ", W_coef[0], "±", W_sigma[0]*StudentT(0.95, V_npnts-2)
print "slope = ", W_coef[1], "±", W_sigma[1]*StudentT(0.95, V_npnts-2)
```

See Also

StatsStudentCDF, **StatsStudentPDF**, **StatsInvStudentCDF**

Submenu

Submenu *menuNameStr*

The Submenu keyword introduces a submenu definition. It is used inside a Menu definition. See Chapter IV-5, **User-Defined Menus** for further information.

sum

sum(*waveName* [, *x1*, *x2*])

The sum function returns the sum of the wave elements for points from x=*x1* to x=*x2*.

Details

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with `pnt2x(waveName,pointNumber1)`, and a similar expression for *x2*.

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If the points nearest to *x1* or *x2* are not within the point range of 0 to numpnts(*waveName*)-1, sum limits them to the nearest of point 0 or point numpnts(*waveName*)-1.

If any values in the point range are NaN, sum returns NaN.

Examples

```
Make/O/N=100 data; SetScale/I x 0,Pi,data
data=sin(x)
Print sum(data,0,Pi)          // the entire point range, and no more
Print sum(data)               // same as -infinity to +infinity
Print sum(data,Inf,-Inf)      // +infinity to -infinity
```

The following is printed to the history area:

```
Print sum(data,0,Pi)          // the entire point range, and no more
  63.0201
Print sum(data)               // same as -infinity to +infinity
  63.0201
Print sum(data,Inf,-Inf)      // +infinity to -infinity
  63.0201
```

See Also

mean, **area**, **SumSeries**, **SumDimension**

SumDimension

SumDimension [*flags*] *srcWave*

The SumDimension operation sums values in *srcWave* along the specified dimension.

The SumDimension operation was added in Igor Pro 7.00.

Flags

/D=*dimension* Specifies a zero-based dimension number.

dimension=0:	Rows
dimension=1:	Columns
dimension=2:	Layers
dimension=3:	Chunks

If you omit /D the operation sums the highest dimension in the wave.

/DEST=*destWave* Specifies the output wave created by the operation. If *destWave* already exists it is overwritten by the new results.

If you omit /DEST the operation saves the data in W_SumDimension if the output wave is 1D or M_SumDimension otherwise.

/Y=type Specifies the data type of the output wave. See **WaveType** for the supported values of type.

If you omit /Y, the output wave is double precision.

Pass -1 for type to force the output wave to have the same data type as *srcWave*.

Details

The operation sums one dimension of an N dimensional wave producing an output wave with N-1 dimensions except if *srcWave* is 1D wave in which case SumDimension produces a single point 1D output wave. For example, given a 4D wave of dimensions dim0 x dim1 x dim2 x dim3 and the command:

```
SumDimension/D=1/DEST=wout wave4d
```

creates a wave wout that satisfies

$$wout[i][k][l] = \sum_{j=0}^{dim1-1} wave4d[i][j][k][l],$$

and wout has dimensions dim0 x dim2 x dim3.

If any values in *srcWave* are NaN, the corresponding sum element will be NaN.

See Also

sum

MatrixOp keywords sumRows, sumCols, sumBeams

ImageTransform keywords sumAllCols, sumAllRows, sumPlane, sumPlanes

SumSeries

`SumSeries [`*`flags`*`]` ***`keyword=value`***

The SumSeries operation computes the sum of the results returned from a user-defined function for input values between two specified index limits.

SumSeries was added in Igor Pro 7.00.

Flags

/CCNT=*nc* When summing with one or two infinite limits you can use this flag to specify the minimum number of calls to the summand function which, when added to the sum, produce a change that is less than the tolerance. By default *nc*=10.

If you are summing a well-behaved monotonic series it is sufficient to set *nc*=1. In some pathological cases it is useful to check that the sum remains effectively unchanged even after many terms are added to the series.

/INAN Ignore NaNs returned from the user function. In the case of a complex valued summand, a NaN in either the real or imaginary components excludes the contribution of the term to the sum.

/Q Quiet mode; do not print in the history.

/Z[=z] /Z or /Z=1 prevents reporting any errors. If the operation encounters an error it sets V_Flag to the error code.

Keywords

lowerLimit=*n1* Specifies the starting index at which the summand is evaluated. *n1* must be either an integer -INF.

series=*userFunc* Specifies the name of the user function that returns the summand (i.e., a single term in the sum that corresponds to the input index). See **The SumSeries Summand Function** below for details.

upperLimit=*n2* Specifies the last value at which the summand is evaluated. *n2* must be either an integer INF.

tolerance=*tol* Specifies a tolerance value used when one or both of the limits are infinite. By default, the tolerance value is 1e-10. *tol* must be finite. If both limits are finite this keyword is ignored.

paramWave=*pw* *pw* is a single-precision or double-precision wave that is passed to the summand function. This is useful if you need to provide the summand function external/global data.

If you omit the paramWave keyword then the summand function receives a null wave as the parameter wave.

The SumSeries Summand Function

You specify the summand function using the series keyword. The form of the user-defined summand function is:

```
Function summandReal(inW,index)
    Wave inW
    Variable index
    ... compute something
    return result
End
```

The index changes by 1 for each successive call to the summand.

You can also define a complex summand function:

```
Function/C summandComplex(inW,index)
    Wave inW
    Variable index
    ... compute something
    Variable/C result
    return result
End
```

Details

The SumSeries operation is primarily intended for use with one or two infinite limits. If both limits are finite the operation performs the straightforward sum by calling the summand function once for every index from lowerLimit to upperLimit, inclusive.

If one limit is infinite the sum is evaluated by starting from the finite limit and proceeding in the direction of the infinite limit index until convergence is reached. Convergence in this context is defined as multiple (*nc*) consecutive calls to the summand which do not change the value of the sum by more than the tolerance value. By default *nc*=10 but you can change it using the /CCNT flag.

When both limits are infinite the operation first computes the sum for indices 0 to INF and then the sum from -1 to -INF. The two calculations are independent and require that the same convergence condition is met independently in each case. When the summand function is complex the convergence condition must hold for the real and imaginary components independently.

The operation does not perform any test on the summand function to estimate its rate of convergence. If you provide a non-converging summand function the operation can run indefinitely. You can abort it by pressing the **User Abort Key Combinations** or by clicking the Abort button.

The result of the sum is stored in V_resultR and, if the summand function returns a complex result, V_resultI.

If the calculation completes without error V_Flag is set to 0. Otherwise it contains an error code.

Examples

A simple test case is the geometric series for powers of 1/2. The sum of x^i for i=0 to i=INF where 0<x<1 is given by 1/(1-x). For x=1/2, this sum is 2.

```
Function s1(paramWave, index)
    Wave/Z paramWave        // Not used
    Variable index

    return 0.5^index
End

// Execute:
SumSeries series=s1,lowerLimit=0,upperLimit=INF
Print V_resultR
```

In the next example we use the series expansion of cosine and sine to evaluate exp(i*pi).

```
Function/C s2(paramWave, index)
    Wave/Z paramWave        // Not used
    Variable index

    Variable n2=2*index
    Variable xx=pi^n2
    Variable sn=(-1)^index
    Variable fn=Factorial(n2)
    return cmplx(sn*xx/fn,sn*xx*pi/(fn*(n2+1)))
End
```

```
// Execute:
SumSeries series=s2,lowerLimit=0,upperLimit=INF
Print V_resultR,V_resultI
```

In the next example we sum 18 terms in the series for exp(x) using a parameter wave to pass the value of x.

```
Function fex(paramWave, index)
    Wave/Z paramWave       // Contains x value at which to evaluate exp(x)
    Variable index

    Variable xx = paramWave[0]
    if (index == 0)
        return 1
    endif
    return (xx^index) / factorial(index)
End
```

```
// Execute:
Make/D/O exponent = {-1}          // The value of x at which to evaluate exp(x)
SumSeries series=fex, lowerLimit=0, upperLimit=17, paramWave=exponent
Print/D V_resultR - exp(-1)       // Compare with built-in exp function
```

See Also

Integrate1D, **sum**

SVAR

SVAR [**/Z**][**/SDFR**=*dfr*] ***localName*** [**=** ***pathToStr***][**,** ***localName1*** [**=** ***pathToStr1***]]…

SVAR is a declaration that creates a local reference to a global string variable accessed in a user-defined function.

The SVAR reference is required when you access a global string variable in a function. At compile time, the SVAR statement specifies a local name referencing a global string variable. At runtime, it makes the connection between the local name and the actual global variable. For this connection to be made, the global string variable must exist when the SVAR statement is executed.

When *localName* is the same as the global string variable name and you want to reference a global variable in the current data folder, you can omit *pathToStr*.

pathToStr can be a full literal path (e.g., root:FolderA:var0), a partial literal path (e.g., :FolderA:var0) or $ followed by string variable containing a computed path (see **Converting a String into a Reference Using $** on page IV-62).

You can also use a data folder reference or the /SDFR flag to specify the location of the string variable if it is not in the current data folder. See **Data Folder References** on page IV-78 and **The /SDFR Flag** on page IV-80 for details.

If the global variable may not exist at runtime, use the /Z flag and call **SVAR_Exists** before accessing the variable. The /Z flag prevents Igor from flagging a missing global variable as an error and dropping into the Igor debugger. For example:

```
SVAR/Z nv=<pathToPossiblyMissingStringVariable>
if( SVAR_Exists(sv) )
    <do something with sv>
endif
```

Note that to create a global string variable, you use the **String**/G operation.

Flags

/SDFR=*dfr*	Specifies the source data folder. See **The /SDFR Flag** on page IV-80 for details.
/Z	An SVAR reference to a null string variable does not cause an error or a debugger break.

See Also

SVAR_Exists function.

Accessing Global Variables and Waves on page IV-65.

Converting a String into a Reference Using $ on page IV-62.

SVAR_Exists

`SVAR_Exists(`*`name`*`)`

The SVAR_Exists function returns 1 if the specified SVAR reference is valid or 0 if not. It can be used only in user-defined functions.

For example, in a user function you can test if a global string variable exists like this:

```
SVAR /Z str1 = gStr1       // /Z prevents debugger from flagging bad SVAR
if (!SVAR_Exists(str1))    // No such global string variable?
   String/G gStr1 = ""     // Create and initialize it
endif
```

See Also

WaveExists, **NVAR_Exists**, and **Accessing Global Variables and Waves** on page IV-65.

switch-case-endswitch

```
switch(<numeric expression>)
   case <literal><constant>:
      <code>
      [break]
   [default:
      <code>]
endswitch
```

A switch-case-endswitch statement evaluates the numeric expression and rounds the result to the nearest integer. If a case label matches *numerical expression*, then execution proceeds with *code* following the matching case label. When no cases match, execution continues at the default label, if present, or otherwise the switch exits with no action taken. Although the break statement is optional, in almost all case statements it is required for the switch to work correctly.

Literal numbers used as case labels are required by the compiler to be integers. Numeric constants used as case labels are rounded by the compiler to the nearest integers.

See Also

Switch Statements on page IV-43, **default** and **break** for more usage details.

t

`t`

The t function returns the T value for the current chunk of the destination wave when used in a multidimensional wave assignment statement. T is the scaled chunk index while **s** is the chunk index itself.

Details

Unlike **x**, outside of a wave assignment statement, t does not act like a normal variable.

See Also

Waveform Arithmetic and Assignments on page II-74.

For other dimensions, the **p**, **q**, **r**, and **s** functions.

For scaled dimension indices, the **x**, **y**, and **z** functions.

TabControl

`TabControl` `[`**`/Z`**`]` ***`ctrlName`*** `[`***`keyword = value`*** `[,` ***`keyword = value`*** `…]]`

The TabControl operation creates tab panels for controls.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the TabControl to be created or changed.

The following keyword=value parameters are supported:

align=*alignment* Sets the alignment mode of the control. The alignment mode controls the interpretation of the *leftOrRight* parameter to the pos keyword. The align keyword was added in Igor Pro 8.00.

If *alignment*=0 (default), *leftOrRight* specifies the position of the left end of the control and the left end position remains fixed if the control size is changed.

If *alignment*=1, *leftOrRight* specifies the position of the right end of the control and the right end position remains fixed if the control size is changed.

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **DefaultGUIControls Default Fonts and Sizes** for how enclosed controls are affected by native TabControl appearance.

See **Button** for more appearance details.

disable=*d* Sets user editability of the control.

d=0:	Normal.
d=1:	Hide.
d=2:	Draw in gray state; disable control action.

fColor=(*r*,*g*,*b*[,*a*]) Sets the initial color of the tab labels. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. Use of transparency is discouraged. The default is opaque black.

To further change the color of the tab labels text, use escape sequences in the text specified by the tabLabel keyword.

focusRing=*fr* Enables or disables the drawing of a rectangle indicating keyboard focus:

fr=0:	Focus rectangle will not be drawn.
fr=1:	Focus rectangle will be drawn (default).

On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.

font= "*fontName*" Sets the font used for tabs, e.g., `font="Helvetica"`.

fsize= *s* Sets the font size for tabs.

fstyle=*fs* *fs* is a bitwise parameter with each bit controlling one aspect of the font style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*} Sets the help for the control.

helpStr is limited to 1970 bytes (255 in Igor Pro 8 and before).

You can insert a line break by putting “\r” in a quoted string.

labelBack=(*r*,*g*,*b*[,*a*]) or 0

Sets fill color for current tab and the interior. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

If not set, then interior is transparent and the current tab is filled with the window background, though this style is platform-dependent.

If you use an opaque fill color, drawing objects will not be seen because they will be covered up.

noproc

Specifies that no function is to run when clicking a tab.

pos={*leftOrRight*,*top*}

Sets the position in **Control Panel Units** of the top/left corner of the control if its alignment mode is 0 or the top/right corner of the control if its alignment mode is 1. See the align keyword above for details.

pos+={*dx*,*dy*}

Offsets the position of the control in **Control Panel Units**.

proc=*procName*

Specifies the function to run when the tab is pressed. Your function must hide and show other controls as desired. The TabControl does not do this automatically.

size={*width*,*height*}

Sets TabControl size in **Control Panel Units**.

tabLabel(*n*)=*lbl*

Sets *n*th tab label to *lbl*.

Set the label of the last tab to `""` to remove the last tab.

Using escape codes you can change the font, size, style, and color of the label. See **Annotation Escape Codes** on page III-53 or details.

userdata(*UDName*)=*UDStr*

Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

value=*v*

Sets current tab number. Tabs count from 0.

win=*winName*

Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Tab Control Action Procedure

The action procedure for a TabControl takes a predefined `WMTabControlAction` structure as a parameter to the function:

```
Function ActionProcName(TC_Struct) : TabControl
   STRUCT WMTabControlAction &TC_Struct
   …
   return 0
End
```

The "`: TabControl`" designation tells Igor to include this procedure in the Procedure pop-up menu in the Tab Control dialog.

See **WMTabControlAction** for details on the WMTabControlAction structure.

Although the return value is not currently used, action procedures should always return zero.

When clicking a TabControl with the selector arrow, click in the title region. The control is not selected if you click in the body. This is to make it easier to select controls in the body rather than the TabControl itself.

Example

Designing a TabControl with all the accompanying interior controls can be somewhat difficult. Here is a suggested technique:

First, create and set the size and label for one tab. Then create the various controls for this first tab. Before starting on the second tab, create the TabControl's procedure so that it can be used to hide the first set of controls. Then add the second tab, click it to run your procedure and start adding controls for this new tab. When done, update your procedure so the new controls are hidden when you start on the third tab.

Here is an example:

1. Create a panel and a TabControl:

```
NewPanel /W=(150,50,478,250)
ShowTools
TabControl MyTabControl,pos={29,38},size={241,142},tabLabel(0)="First Tab",value=0
```

2. Add a few controls to the interior of the TabControl:

```
Button button0,pos={52,72},size={80,20},title="First"
CheckBox check0,pos={52,105},size={102,15},title="Check first",value=0
```

3. Write an action procedure:

```
Function TabActionProc(tc) : TabControl
   STRUCT WMTabControlAction& tc

   switch(tc.eventCode)
      case 2:                // Mouse up
      Button button0, disable=(tc.tab!=0)
      CheckBox check0, disable=(tc.tab!=0)
      break
   endswitch
End
```

4. Set the action procedure and add a new tab:

```
TabControl MyTabControl,proc=TabActionProc,tabLabel(1)="Second Tab"
```

5. Click the second tab, which hides the first tab's controls, and then add new controls like this:

```
Button button1,pos={58,73},size={80,20},title="Second"
CheckBox check1,pos={60,105},size={114,15},title="Check second",value= 0
```

6. Finally, change the action procedure by adding these lines at the end:

```
Button button1,disable=(tc.tab!=1)
CheckBox check1,disable=(tc.tab!=1)
```

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

The **GetUserData** function for retrieving named user data.

The **ControlInfo** operation for information about the control.

The **ModifyControl** and **ModifyControlList** operations.

TabControl

`TabControl`

TabControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined tab control. See **Procedure Subtypes** on page IV-204 for details. See **TabControl** for details on creating a tab control.

Table

`Table`

Table is a procedure subtype keyword that identifies a macro as being a table recreation macro. It is automatically used when Igor creates a window recreation macro for a table. See **Procedure Subtypes** on page IV-204 and **Killing and Recreating a Table** on page II-241 for details.

TableStyle

```
TableStyle
```

TableStyle is a procedure subtype keyword that puts the name of the procedure in the Style pop-up menu of the New Table dialog and in the Table Macros menu. See **Table Style Macros** on page II-272 for details.

TableInfo

`TableInfo(`*`winNameStr`*`, `*`itemIndex`*`)`

The TableInfo function returns a string containing a semicolon-separated list of keywords and values that describe a column in a table or overall properties of the table. The main purpose of TableInfo is to allow an advanced Igor programmer to write a procedure which formats or arranges a table or which manipulates the table selection.

Parameters

winNameStr is the name of an existing table window or `""` to refer to the top table.

itemIndex is one of the following:

***itemIndex* Value**	**Returns**
-2	Information about the table as a whole.
-1	Information about the Point column
≥0	Information about a column other than the Point column. 0 refers to the first column after the Point column, 1 refers to the second column after the Point column, and so on.

TableInfo returns `""` in the following situations:

- *winNameStr* is `""` and there are no table windows.
- *winNameStr* is a name but there are no table windows with that name.
- *itemIndex* not -2 and is out of range for an existing column.

Details

If *itemIndex* is -2, the returned string describes the table as a whole and contains the following keywords, with a semicolon after each keyword-value pair.

Keyword	**Information Following Keyword**
TABLENAME	The name of the table.
HOST	The host specification of the table's host window if it is a subwindow or `""` if it is a top-level table window.
ROWS	Number of used rows in the table.
COLUMNS	Number of used columns in the table including the Point column.
SELECTION	A description of the table selection as you would specify it when invoking the ModifyTable operation's selection keyword.
FIRSTCELL	An identification of the first visible data cell in the top/left corner of the table in row-column format. The first data cell is at location 0, 0.
LASTCELL	An identification of the last visible data cell in the bottom/right corner of the table in row-column format.

Keyword	Information Following Keyword
TARGETCELL	An identification of the target (highlighted) data cell in row-column format.
ENTERING	1 if an entry has been started in the entry line, 0 if not.
ENTRYTEXT	The text displayed in the entry line. If the user is editing the text in the entry line, this is the text as edited so far. If the user is not editing, it is the value of the first selected cell, as text. The ENTRYTEXT keyword is the last keyword-value pair in the returned string. It is not terminated with a trailing ";" character unless the entry line text itself ends with ";". The ENTRYTEXT keyword was added in Igor Pro 9.00.

If *itemIndex* is -1 up to but not including the number of used columns to the right of the Point column, the returned string describes the specified column and contains the following keywords, with a semicolon after each keyword-value pair.

Keyword	Information Following Keyword
TABLENAME	The name of the table.
HOST	The host specification of the table's host window if it is a subwindow or "" if it is a top-level table window.
COLUMNNAME	Name of the column as you would specify it to the Edit operation if you were creating a table showing just the column of interest.
TYPE	Column's type which will be one of the following: Unused, Point, Index, Label, Data, RealData, ImagData. "Index" identifies a index column such as the X values of a wave. "Label" identifies a column of dimension labels. "Data" identifies a data column of a scalar wave. RealData and ImagData identify a real or imaginary column of a complex wave.
INDEX	Column's position. -1 refers to the Point column, 0 to the first data column, and so on.
DATATYPE	Numeric data type of the wave or zero for text waves. See WaveType for a definition of data type codes.
WAVE	A full data folder path to the wave displayed in the column or "" for the Point column.
COLUMNS	The total number of columns in the table from the wave for the column for which you are getting information. This can be used to skip over all of the columns of a multidimensional wave.
HDIM	The wave dimension displayed horizontally as you move from one column to the next. 0 means rows, 1 means columns, 2 means layers, 3 means chunks.
VDIM	The wave dimension displayed vertically in the column. 0 means rows, 1 means columns, 2 means layers, 3 means chunks.
TITLE	As specified for the ModifyTable operation's title keyword.
WIDTH	Column's width in points.
FORMAT	As specified for the ModifyTable operation's format keyword.
DIGITS	As specified for the ModifyTable operation's digits keyword.
SIGDIGITS	As specified for the ModifyTable operation's sigDigits keyword.
TRAILINGZEROS	As specified for the ModifyTable operation's trailingZeros keyword.
SHOWFRACSECONDS	As specified for the ModifyTable operation's showFracSeconds keyword.

Keyword	Information Following Keyword
FONT	The name of the column's font.
SIZE	Column's font size.
STYLE	As specified for the ModifyTable operation's style keyword.
ALIGNMENT	0=left, 1=center, 2=right.
RGB	The column's color in R,G,B format.
RGBA	The column's color in R,G,B,A format. Added in Igor Pro 9.00.
ELEMENTS	As specified for the ModifyTable operation's elements keyword.

Examples

This example makes the table's target cell advance by one position within the range of selected cells each time it is called. To try it, create a table, select a range of cells and then run the function using the Macros menu.

```
Menu "Macros"
   "Test/1", /Q, AdvanceTargetCell("")
End

Function AdvanceTargetCell(tableName)
   String tableName              // Name of table or "" for top table.

   String info = TableInfo(tableName, -2)
   if (strlen(info) == 0)
      return -1                  // No such table
   endif

   String selectionInfo
   selectionInfo = StringByKey("SELECTION", info)

   Variable fRow, fCol, lRow, lCol, tRow, tCol
   sscanf selectionInfo, "%d,%d,%d,%d,%d,%d", fRow, fCol, lRow, lCol, tRow, tCol

   tCol += 1
   if (tCol > lCol)
      tCol = fCol
      tRow += 1
      if (tRow > lRow)
         tRow = fRow
      endif
   endif

   ModifyTable selection=(-1, -1, -1, -1, tRow, tCol)
End
```

See Also

The **ModifyTable** operation.

Tag

Tag [***flags***] [***taggedObjectName***, ***xAttach*** [, ***textStr***]]

The Tag operation puts a tag on the target or named graph window or subwindow. A tag is an annotation that is attached to a particular point on a trace, image, waterfall plot, or axis in a graph.

The Tag operation can be invoked in several ways as illustrated by these examples:

```
// Make a wave and a graph
Make wave0 = sin(x/8); Display wave0

// Tag command with all optional parameters included
Tag/N=tag0 wave0, 0, "Point 0 on wave0"        // Add a tag to a trace

// Tag command with all optional parameters omitted
Tag/C/N=tag0/F=0                               // Change frame using /F flag

// Tag command with optional text parameter omitted
Tag/C/N=tag0 wave0, 50                         // Change the tagged point
```

taggedObjectName and *xAttach* can be omitted only when changing an existing tag using `Tag/C/N=<tag name>`. *textStr* can be included only if *taggedObjectName* and *xAttach* are included.

Parameters

taggedObjectName is a trace, image or axis name which identifies the object to which the tag is to be attached. The name can be optionally followed by the # character and an instance number to distinguish multiple trace or image instances of the same wave. An axis name can be one of the standard axis names (bottom, top, left, or right) or a user-defined free axis name.

The *taggedObjectName* parameter is a name so, if you have a string variable containing the name, you must precede the string variable name with the $ operator.

xAttach identifies the point on the trace, image, or axis, to which the tag is to be attached. See ***xAttach Parameter*** below for details.

textStr is the text that is to appear in the tag.

xAttach Parameter

xAttach identifies the point on the trace, image, or axis, to which the tag is to be attached.

For a trace tag, *xAttach* is the X wave scaling value of the attachment point.

For an image tag, *xAttach* is the X index in terms of the image wave's X scaling of the wave element to be tagged treating the wave as if it were 1D.

For a horizontal axis tag, *xAttach* is the X axis value of the point on the axis to which the tag is to be attached. Specifying NaN for xAttach centers the tag on the axis.

For a vertical axis tag, *xAttach* is the Y axis value of the point on the axis to which the tag is to be attached. Specifying NaN for xAttach centers the tag on the axis.

Flags

/A=*anchorCode*

Specifies position of tag anchor point. *anchorCode* is a literal, *not* a string.

LT	left top
LC	left center
LB	left bottom
MT	middle top
MC	middle center (default)
MB	middle bottom
RT	right top
RC	right center
RB	right bottom

The anchor point is on the tag itself. Any line or arrow drawn from the tag to the wave starts at the tag’s anchor point. The anchor point also determines the precise spot on the tag which represents its position.

/AO=*ao*

Sets the text's auto-orientation mode. A non-zero *a0* value overrides the /O value.

/AO is for trace tags only. Setting /AO for any other kind of annotation has no effect.

An auto-oriented tag's text rotates whenever it is redrawn, usually when the underlying data changes, the graph is resized, or when the tag is attached to a new point.

The values for *ao* are:

ao=0: No auto-orientation. Use the /O value (default).

ao=1: Tangent to the trace line at the attachment point.

ao=2: Tangent to the trace line, snaps to vertical or horizontal if within 2 degrees of vertical or horizontal.

ao=3: Perpendicular to the trace line.

ao=4: Perpendicular to the trace line, snaps to vertical or horizontal if within 2 degrees of vertical or horizontal.

/AXS[=*isAxisTag*] Specifies that *taggedObjectName* is to be interpreted as an axis name. This is useful when the axis name is the same as one of the graph's trace or image instance names. /AXS is the same as /AXS=1. The /AXS flag was added in Igor Pro 9.00.

/B=(*r*,*g*,*b*[,*a*]) Sets color of the tag's background. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

/B=*b* Controls the tag background.

b=0: Opaque background.

b=1: Transparent background.

b=2: Same background as the graph plot area background.

b=3: Same background as the window background.

/C Changes the existing tag.

/F=*frame* Controls the tag frame.

frame=0: No frame.

frame=1: Underline frame.

frame=2: Box frame.

/G=(*r*,*g*,*b*[,*a*]) Sets color of the text in the tag. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

/H=*legendSymbolWidth*

legendSymbolWidth sets width of the legend symbol (the sample line or marker) in points. Use 0 for the default, automatic width.

/I=*i* Controls the tag visibility.

i=1: Tag will be invisible if it is "off screen". "Off screen" means that its attachment point or any part of the tag's text is off screen. This is esthetically pleasing but gives you nothing to grab if you want to drag the tag back on screen.

i=0: Tag will always be visible. If it is "off screen", it appears at the extreme edge of the graph.

/IMG[=*isAxisTag*] Specifies that *taggedObjectName* is to be interpreted as an image instance name. This is useful when the image instance name is the same as one of the graph's trace instance names or one of the axis names. /IMG is the same as /IMG=1. The /IMG flag was added in Igor Pro 9.00.

/K Kills existing tag.

/L=*line* Controls the line attaching the tag to the tagged point.

line=0: No line from tag to attachment point.

line=1: Line connecting tag to attachment point.

line=2: Line with arrow pointing from tag to attachment point.

line=3: Line with arrow pointing from attachment point to tag.

line=4: Line with arrows at both ends.

/LS= *linespace*	Specifies a tweak to the normal line spacing where *linespace* is points of extra (plus or minus) line spacing. For negative values, a blank line may be necessary to avoid clipping the bottom of the last line.
/M[=*sameSize*]	/M or /M=1 specifies that legend markers should be the same size as the marker in the graph. /M=0 turns same-size mode off so that the size of the marker in the legend is based on text size.
/N=*name*	Specifies name of the tag to create or change.
/O=*rot*	Sets the text's rotation. *rot* is in (integer) degrees, counterclockwise and must be a number from -360 to 360. 0 is normal horizontal left-to-right text, 90 is vertical bottom-to-top text. If the tag is attached to a trace (not an image or axis), any non-zero /AO value will overwrite this rotation value.
/P=*tipOffset*	Sets the offset from the tip of a tag's line or arrow to the point on the wave that it is tagging. *tipOffset* is a positive number from 0 to 200 in points. If *tipOffset*=0 (default), it automatically chooses an appropriate offset.
/Q[=*contourInstance*]	Associates a tag with a particular contour level trace in a graph recreation macro. Of interest mainly to hard-core programmers. When "=*contourInstance*" is present, /Q associates the tag with the contour wave. Igor will feel free to change or delete the tag, as appropriate, when it recalculates the contour (because you changed the contour data or appearance, the graph size or the axis range). *contourInstance* is a contour instance name, such as zWave or zWave#1 if you have the same wave contoured twice in the graph. /Q by itself, with "=*contourInstance*" not present, disassociates the tag from the contour wave. Igor will no longer modify or delete the tag (unless the contour level to which it is attached is deleted). If you manually tweak a contour label, using the Modify Annotation dialog, Igor uses this flag.
/R=*newName*	Renames the tag.
/S=*style*	Controls the tag frame style. *style*=0: Single frame. *style*=1: Double frame. *style*=2: Triple frame. *style*=3: Shadow frame.
/T=*tabSpec*	*tabSpec* is a single number in points, such as `/T=72`, for evenly spaced tabs or a list of tab stops in points such as `/T={50, 150, 225}`.
/TL=*extLineSpec*	Specifies extended tag line parameters similar to the **SetDrawEnv** arrow settings. *extLineSpec* = {*keyword* = *value*,…} or zero to turn off all extended specifications.

Valid *keyword-value* pairs are:

len=*l* Length of arrow head in points (*l*=0 for auto).

fat=*f* Width to length ratio of arrow head (default is 0.5 same as *f*=0).

style=*s* Sets barb side mode (see **SetDrawEnv** astyle for values).

shar =*s* Sets sharpness between -1 and 1 (default is 0; blunt).

frame=*f* Sets frame thickness in outline mode.

lThick=*l* Sets line thickness in points (default is 0.5 for *l*=0).

lineRGB=(*r*,*g*,*b*[,*a*]) Sets color for lines. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

dash=*d* Specifies dash pattern number between 0 and 17 (see **SetDashPattern** for patterns).

/W=*winName* Operates on the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/V=*vis* Controls annotation visibility.

vis=0: Invisible annotation; not selectable. The annotation is still listed in **AnnotationList**.

vis=1: Visible annotation (default).

/X=*xOffset* Distance from point to tag as percentage of graph width. For axis tags, the offsets are proportional to the size of the text used for the axis labels.

/Y=*yOffset* Distance from point to tag as percentage of graph height. For axis tags, the offsets are proportional to the size of the text used for the axis labels.

/Z=*freeze* Controls freezing of tag position.

freeze=1: Freezes tag position (you can’t move it with the mouse).

freeze=0: Unfreezes it.

Details

If the /C flag is used, it must be the first flag in the command and must be followed immediately by the /N=*name* flag.

If the /K flag is used, it must be the first flag in the command and must be followed immediately by the /N=*name* flag with no further flags or parameters.

taggedObjectName and *xAttach* can be omitted only when changing an existing tag using `Tag/C/N=<tag name>`. *textStr* can be included only if *taggedObjectName* and *xAttach* are included. This syntax allows changes to the tag to be made through the *flags* parameters without needing to respecify the other parameters. Similarly, the tag’s attachment point can be changed without needing to respecify the *textStr* parameter.

A tag can have at most 100 lines.

textStr can contain escape codes which affect subsequent characters in the text. An escape code is introduced by a backslash character. In a literal string, you must enter two backslashes to produce one. See **Backslashes in Annotation Escape Sequences** on page III-58 for details.

Using escape codes you can change the font, size, style and color of text, create superscripts and subscripts, create dynamically-updated text, insert legend symbols, and apply other effects. See **Annotation Escape Codes** on page III-53 for details.

"`\r`" inserts a carriage-return character which starts a new line of text in the annotation.

Some escape codes insert text based on the wave point or axis to which a tag is attached. See **Tag Escape Codes** on page III-55 and **Axis Label Escape Codes** on page III-57 for details.

The characters “<??>” in a tag indicate that you specified an invalid escape code or used a font that is not available.

Examples

Following is an example of various ways in which axis tags can be used:

```
Make/O jack=sin(x/8)
SetScale x,0,14e9,"y" jack
Display jack
Label bottom "\\u#2"                  // turn off default axis label
ModifyGraph axOffset(bottom)=1.16667 // make room for tag (manual adustment)
Tag/N=axisTag0/F=0/A=MT/X=0.20/Y=-4.29/L=0 bottom, Nan, "\\JCTime (\\U)\r2nd line"

// Now tag a few important points
Tag/N=axisTag1/F=0/A=LB/X=1.20/Y=3.00 bottom, 0, "Big Bang"
Tag/N=axisTag2/F=0/A=MB/X=0.00/Y=2.86 bottom, 8000000000, "Earth formed"
Tag/N=axisTag3/F=0/A=RB/X=-0.80/Y=4.71 bottom, 13040000000, "Dinosaurs ruled"
```

See Also

TextBox, **Legend**, **AppendText**, **AnnotationInfo**, **AnnotationList**

TagVal, **TagWaveRef**

Annotation Escape Codes on page III-53

Label, **Axis Labels** on page II-318

Trace Names on page II-282, **Programming With Trace Names** on page IV-87

TagVal

`TagVal(`*`code`*`)`

TagVal is a very specialized function that is only valid when called from within the text of a tag as part of a \{} dynamic text escape sequence. It returns a number reflecting some property of the tag and helps you to display information about the tagged wave. The property is selected by the *code* parameter:

code	**Return Value**
0	Similar to \OP, returns the tag attach point number.
1	Similar to \OX, returns the X coordinate of tag attachment in the graph. When a tag is attached to an XY pair of traces, the X coordinate will most likely be different than the tag’s X scaling attachment value specified in the Tag command.
2	Similar to \OY, returns the Y coordinate of tag attachment in the graph or the Y axis value in a Waterfall plot.
3	Similar to \OZ, returns the Z coordinate of tag attachment in a contour, image, or Waterfall plot.
4	Similar to \Ox, returns the trace x offset.
5	Similar to \Oy, returns the trace y offset.
6	Returns the X muloffset (with the not set value 0 translated to 1).
7	Returns the Y muloffset (with the not set value 0 translated to 1).

Because TagVal returns a numeric value, the result can be formatted any way you wish using the **printf** formatting codes. In contrast, the \O codes insert preformatted text, and you don’t have control over the format.

TagVal is sometimes used in conjunction with the **TagWaveRef** function. For example, you might write a user-defined function that calculates a value as a function of a wave and a point number.

Examples

```
Tag wave0, 0, "Y value is \\{\"%g\",TagVal(2)}"
Tag wave0, 0, "Y value is \\{\"%g\",TagWaveRef()[TagVal(0)]}"
Tag wave0, 0, "Y value is \\OY"
```

These examples all produce identical results.

See Also

The **Tag** operation, the **TagWaveRef** function.

For a discussion of wave references, see **Wave Reference Functions** on page IV-197.

TagWaveRef

`TagWaveRef()`

TagWaveRef is a very specialized function that is only valid when called from within the text of a tag as part of a \{} dynamic text escape sequence. It returns a wave reference to the wave that the tag is on and helps you to display information about the tagged wave. It is often used in conjunction with the **TagVal** function. You can pass the result of TagWaveRef to any function that takes a Wave parameter.

Examples

Show the name of the data folder containing the tagged wave:

```
Tag wave0, 0,"\\ON is in \\{\"%s\",GetWavesDataFolder(TagWaveRef(),0)}"
```

See Also

The **Tag** operation, the **TagVal** function

For a discussion of wave references, see **Wave Reference Functions** on page IV-197.

tan

`tan(`*`angle`*`)`

The tan function returns the tangent of *angle* which is in radians.

In complex expressions, *angle* is complex, and tan(*angle*) returns a complex value:

$$\tan(x+iy)=\frac{\sin(\mathrm{x+iy})}{\cos(x+iy)}=\frac{\sin(2x)+i\sinh(2y)}{\cos(2x)+\cosh(2y)}.$$

See Also

atan, **atan2**, **sin**, **cos**, **sec**, **csc**, **cot**

tanh

`tanh(`*`num`*`)`

The tanh function returns the hyperbolic tangent of *num*:

$$\tanh(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}.$$

In complex expressions, *num* is complex, and tanh(*num*) returns a complex value.

See Also

sinh, **cosh**, **coth**

Text2Bezier

`Text2Bezier[ `*`flags`*` ] `*`fontNameStr`*`, `*`fstyle`*`, `*`textStr`*`, `*`xWaveName`*`, `*`yWaveName`*

The Text2Bezier operation creates the data for a Bezier curve corresponding to the outline of some text using the supplied font information. The output waves are formatted to be drawn using Igor's DrawBezier operation.

Parameters

fontNameStr A string expression containing the name of a font to be used for generating outlines.

fstyle *fstyle* is a bitwise parameter with each bit controlling one aspect of the font style as follows::

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

textStr A string expression containing the text to be transformed into Bezier outlines.

xWaveName Specifies X output wave to receive the Bezier curve data.

yWaveName Specifies Y output wave to receive the Bezier curve data.

Flags

/FS=*fs* *fs* is the font size to apply while generating the outlines. Without this flag, glyphs are scaled to unit size. The scaling parameters for DrawBezier are limited to a maximum of 20, so you will probably need to use this flag if you want very large text.

/O Allow overwriting the output waves.

Details

To draw the text outline, pass the output waves from Text2Bezier to the DrawBezier operation. The coordinates are such that the DrawBezier origin will be the left baseline of the text. In most cases you will want to use the SetDrawEnv operation to set the coordinate system to absolute. If you wish to fill the text with color or gradient, you will need to use the SetDrawEnv subpaths keyword.

Output Variables

V_Flag 0 for success, 1 for general failure, 2 for extraction failure.

Examples

```
// Extract text to Bezier data
Text2Bezier/O/FS=20 "Helvetica", 0, "Text2Bezier", textx, texty
```

```
// Draw outline in a graph window with no fill
Display /W=(100,100,700,450)
SetDrawLayer UserFront
SetDrawEnv fillpat=0,xcoord= abs,ycoord= abs// fillpat=0 specifies no fill
DrawBezier 100,100,1,1,textx,texty
```

```
// Draw outline three times larger and filled with a gradient shading
// subpaths=1 draws the entire Bezier as a series of subpaths for correct filling
SetDrawEnv xcoord= abs,ycoord= abs,subpaths= 1,
                       gradient= {0, 0, 0, 1, 0, (65535,65535,0), (65535,0,0)}
DrawBezier 100,150,3,3,textx,texty
```

See Also

DrawBezier, **SetDrawEnv**

TextBox

`TextBox` `[`***`flags`***`] [`***`textStr`***`]`

The TextBox operation puts a textbox on the target or named graph window. A textbox is an annotation that is not associated with any particular trace.

Parameters

textStr is the text that is to appear in the textbox. It is optional.

Flags

/A=*anchorCode* Specifies position of textbox anchor point.

anchorCode	Position	*anchorCode*	Position
LT	left top	RT	right top
LC	left center	RC	right center
LB	left bottom	RB	right bottom
MT	middle top		
MC	middle center		
MB	middle bottom		

anchorCode is a literal, *not* a string.

For interior textboxes, the anchor point is on the rectangular edge of the plot area of the graph window (where the left, bottom, right, and top axes are drawn).

For exterior textboxes, the anchor point is on the rectangular edge of the entire graph window.

/B=(*r*,*g*,*b*[,*a*]) Sets the color of the textbox background. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

/B=*b* Controls the textbox background.

b=0:	Opaque background.
b=1:	Transparent background.
b=2:	Same background as the graph plot area background.
b=3:	Same background as the window background.

/C Changes existing textbox.

/D={*thickMult* [, *shadowThick* [, *haloThick*]]}

thickMult multiplies the normal frame thickness of a text-box. The thickness may be set using just /D=*thickMult*.

shadowThick, if present, overrides Igor's normal shadow thickness. It is in units of fractional points.

haloThick governs the annotation's halo thickness (a surrounding band of the annotation's background color), which can be -1 to 10 points wide.

The default *haloThick* value is -1, which preserves the behavior of previous versions of Igor where the halo of all annotations was set by the global variable root:V_TBBufZone. Any negative value of *haloThick* (-0.5, for example) will be overridden by V_TBBufZone if it exists, otherwise the absolute value of *haloThick* will be used. A zero or positive value overrides V_TBBufZone.

Any of the parameters may be missing. To set *haloThick* to 0 without changing other parameters, use `/D={,,0}`.

/E[=*exterior*] /E or /E=1 forces textbox (or legend) to be exterior to graph (provided *anchorCode* is not MC) and pushes the graph margins away from the anchor edge(s). /E=2 also forces exterior mode but does not push the margins.

/E=0 returns it to the default (an "interior textbox" which can be anywhere in the graph window).

/F=*frame* Controls the textbox frame.

frame=0: No frame.

frame=1: Underline frame.

frame=2: Box frame.

/G=(*r*,*g*,*b*[,*a*]) Sets color of the text in the textbox. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

/H=*legendSymbolWidth*

legendSymbolWidth sets width of the legend symbol (the sample line or marker) in points. Use 0 for the default, automatic width.

/K Kills existing textbox.

/LS= *linespace* Specifies a tweak to the normal line spacing where *linespace* is points of extra (plus or minus) line spacing. For negative values, a blank line may be necessary to avoid clipping the bottom of the last line.

/M[=*sameSize*] /M or /M=1 specifies that legend markers should be the same size as the marker in the graph.

/M=0 turns same-size mode off so that the size of the marker in the legend is based on text size.

/N=*name* Specifies the name of the textbox to change or create.

/O=*rot* Sets the text's rotation. *rot* is in (integer) degrees, counterclockwise and must be a number from -360 to 360.

0 is normal horizontal left-to-right text, 90 is vertical bottom-to-top text.

/R=*newName* Renames the textbox.

/S=*style* Controls the textbox frame style.

style=0: Single frame.

style=1: Double frame.

style=2: Triple frame.

style=3: Shadow frame.

/T=*tabSpec* *tabSpec* is a single number in points, such as `/T=72`, for evenly spaced tabs or a list of tab stops in points such as `/T={50, 150, 225}`.

/V=*vis* Controls annotation visibility.

vis=0: Invisible annotation; not selectable. The annotation is still listed in **AnnotationList**.

vis=1: Visible annotation (default).

/W=*winName* Operates in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

/X=*xOffset* For interior textboxes *xOffset* is the distance from anchor to textbox as a percentage of the plot area width.

For exterior textboxes *xOffset* is the distance from anchor to textbox as a percentage of the graph window width. See /E and /A.

/Y=*yOffset* *yOffset* is the distance from anchor to textbox as a percentage of the plot area height (interior textboxes) or graph window height (exterior textboxes). See /E and /A.

/Z=*freeze*	Controls freezing of textbox position.	
	freeze=1:	Freezes textbox position (you can't move it with the mouse).
	freeze=0:	Unfreezes it.

Details

Use the optional /W=*winName* flag to specify a specific graph or layout window. When used on the command line or in a Macro, Proc, or Window procedure, /W must precede all other flags.

If the /C flag is used, it must be the first flag in the command (except that if may follow an initial /W) and must be followed immediately by the /N=*name* flag.

If the /K flag is used, it must be the first flag in the command (or follow an initial /W) and must be followed immediately by the /N=*name* flag with no further flags or parameters.

textStr is optional. If missing, the textbox text is unchanged. This allows changes to the textbox to be made through the flags without changing the text.

A textbox can have at most 100 lines.

textStr can contain escape codes which affect subsequent characters in the text. An escape code is introduced by a backslash character. In a literal string, you must enter two backslashes to produce one. See **Backslashes in Annotation Escape Sequences** on page III-58 for details.

Using escape codes you can change the font, size, style and color of text, create superscripts and subscripts, create dynamically-updated text, insert legend symbols, and apply other effects. See **Annotation Escape Codes** on page III-53 for details.

The characters "<??>" in a textbox indicate that you specified an invalid escape code or used a font that is not available.

Examples

```
TextBox/C/N=t1/X=25/Y=50
```

moves the textbox named t1 to the location defined by X=25 and Y=50.

```
TextBox/C/N=t1 "New Text"
```

changes the text for t1.

See Also

Tag, **Legend**, **AppendText**, **AnnotationInfo**, **AnnotationList**

Annotation Escape Codes on page III-53

See the **printf** operation for formatting codes used in *formatStr*.

Programming with Annotations on page III-52.

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

TextEncoding

```
#pragma TextEncoding = "<text encoding name>"
```

#pragma TextEncoding is a compiler directive that tells Igor the text encoding used by a procedure file. Igor needs to know this in order to correctly interpret non-ASCII characters in the file. We recommend that you add a TextEncoding pragma to your procedure files.

All new procedure files should use the UTF-8 text encoding:

```
#pragma TextEncoding = "UTF-8"
```

See **Text Encoding Names and Codes** on page III-490 for a list of accepted text encoding names.

This statement must be flush against the left edge of the procedure file with no indentation. It is usually placed at or near the top of the file.

The TextEncoding pragma was added in Igor Pro 7.00 and is ignored by earlier versions.

See **The TextEncoding Pragma** on page IV-55 for further explanation.

TextEncodingCode

`TextEncodingCode(`*`textEncodingNameStr`*`)`

The TextEncodingCode function returns the Igor text encoding code for the named text encoding or 0 if the text encoding is unknown.

The TextEncodingCode function was added in Igor Pro 7.00.

Parameters

textEncodingNameStr is an Igor text encoding name as listed under **Text Encoding Names and Codes** on page III-490.

Details

Igor ignores all non-alphanumeric characters in text encoding names so "Shift JIS", "ShiftJIS", "Shift_JIS" and "Shift-JIS" are equivalent.

It also ignores leading zeros in numbers embedded in text encoding names so "ISO-8859-1" and "ISO-8859-01" are equivalent.

TextEncodingCode does a case-insensitive comparison.

See Also

Text Encodings on page III-459, **Text Encoding Names and Codes** on page III-490, **TextEncodingName**

TextEncodingName

`TextEncodingName(`*`textEncoding`*`, `*`index`*`)`

The TextEncodingName function returns one or more text encoding names corresponding to the specified text encoding code. The result is returned as a string value.

If *textEncoding* is not a valid Igor text encoding code or if index is out of range, TextEncodingName returns "Unknown".

This function is mainly useful for providing a human-readable string corresponding to a given text encoding code for display purposes. You might use it to generate some Internet-compatible text, such as an HTML page, if you need a string to specify the charset.

The TextEncodingName function was added in Igor Pro 7.00.

Parameters

textEncoding is an Igor text encoding code as listed under **Text Encoding Names and Codes** on page III-490.

index specifies which text encoding name you want. A given text encoding can be identified by more than one name. Normally you will pass 0 to get the first text encoding name for the specified text encoding code. This is the preferred text encoding name. You can pass 1 for the second name, if any, 2 for the third, if any, and so on. You can pass -1 to get a semicolon-separated list of all text encoding names for the specified text encoding.

Details

Internally Igor has a table of text encoding codes and the corresponding text encoding names. For a given code there may be more than one acceptable name. For example, for the code 2 (MacRoman), the names "macintosh", "MacRoman" and "x-macroman" are accepted, with "macintosh" being preferred. The TextEncodingName function returns a text encoding name from the internal table.

The preferred name is usually the name recognized by the Internet Assigned Numbers Authority (IANA) as listed at http://www.iana.org/assignments/character-sets.

Examples

```
// Get the preferred name for the MacRoman text encoding (2)
String firstName = TextEncodingName(2, 0); Print firstName

// Get the second name for the MacRoman text encoding (2)
String secondName = TextEncodingName(2, 1); Print secondName

// Get a semicolon-separated list of all text encoding names for MacRoman
String names = TextEncodingName(2, -1); Print names
```

See Also

Text Encodings on page III-459, **Text Encoding Names and Codes** on page III-490, **TextEncodingCode**

TextFile

`TextFile(`*`pathName`*`, `*`index`*` [, `*`creatorStr`*`])`

Note: TextFile is antiquated. Use **IndexedFile** instead.

The TextFile function returns a string containing the name of the *index*th TEXT file from the folder specified by *pathName*.

On Macintosh, TextFile returns only files whose file type property is TEXT, regardless of the file's extension.

On Windows, Igor considers files with ".txt" extensions to be of type TEXT.

Details

TextFile returns an empty string ("") if there is no such file.

pathName is the name of an Igor symbolic path; it is *not* a string.

index starts from zero.

creatorStr is an optional string argument containing four ASCII characters such as "IGR0". Only files of the specified Macintosh creator code are indexed. Set *creatorStr* to "????" to index all text files (or omit the argument altogether). This argument is ignored on Windows systems.

The order of files in a folder is determined by the operating system.

Examples

You can use TextFile in a procedure to sequence through each TEXT file in a folder, put the name of the text file into a string variable, and use this string variable as a parameter to the **LoadWave** or **Open** operations:

```
Function/S PrintFirstLineOfTextFiles(pathName)
    String pathName                     // Name of an Igor symbolic path.

    Variable refNum, index
    String str, fileName
    index = 0
    do
        fileName = TextFile($pathName, index)
        if (strlen(fileName) == 0)
            break                       // No more files
        endif
        Open/R/P=$pathName refNum as fileName
        FReadLine refNum, str           // Read first line including CR/LF
        Print fileName +":" + str       // Print file name and first line
        Close refNum
        index += 1                      // Next file
    while (1)
End
```

See Also

See the **IndexedFile** function, which is similar to TextFile but works on files of any type, and also **IndexedDir**. Also see the **LoadWave** and **Open** the operations.

TextHistogram

`TextHistogram [flags] `*`srcTextWave`*

The TextHistogram operation computes the histogram of a text wave where the output bins represent the count of occurrences of each unique string found in *srcTextWave*.

The TextHistogram operation was added in Igor Pro 9.00.

Flags

/CI	Performs case-insensitive string comparison. If you omit /CI TextHistogram performs case-sensitive comparison unless you include /LOC.
/DN=*binsCountsWave*	Specifies the numeric output wave that contains the count for each bin. If you omit /DN, the numeric output wave is created in the current data folder and named W_TextHistogram.
/DT=*binsTextWave*	Specifies the text output wave that contains the strings corresponding to each bin. If you omit /DT, the text output wave is created in the current data folder and named T_TextHistogram.
/FREE	Creates all output waves as a free waves. /FREE is permitted in user-defined functions only. If you use /FREE then all output wave parameters must be simple names, not paths or $ expressions. See **Free Waves** on page IV-91 for details on free waves.
/LOC	Performs case-insensitive string comparison following locale-aware rules. This option results in significantly slower performance.
/SORT=*mode*	Sets the order of the output bins. *mode*=0: The output waves are ordered from the bin with the largest count to the bin with the smallest count. This is the default if you omit /SORT. *mode*=1: The output waves are ordered from the bin with the longest string to the bin with the shortest string.
/Z	Suppress errors. You can use V_Flag to detect and handle errors yourself.

Details

TextHistogram does case-sensitive string comparisons unless you specify /CI or /LOC in which case it does case-insensitive string comparisons.

TextHistogram scans *srcTextWave* and counts matching string entries. It treats text waves of all dimensions as if they were 1D waves.

The output of the operation consists of two waves: a text wave containing the text corresponding to each bin and a numeric wave containing the bin counts. You can specify the destination waves using the /DT and /DN flags. If you omit /DT the text output wave is created in the current data folder and named T_TextHistogram. If you omit /DN the numeric output wave is created in the current data folder and named W_TextHistogram. The output waves overwrite any previously-existiing waves with the same names.

The operation creates automatic wave references for the output waves specified by /DT and /DN. See **Automatic Creation of WAVE References** on page IV-72 for details. It does not create wave references for the default output waves created if you omit /DT or /DN so you need explicit wave references.

When using case-insensitive mode the output text wave may use any one of the equivalent forms of the text represented by the bin.

Output Variables

TextHistogram sets the following output variable:

V_flag	0 if the operation succeeded or a non-zero error code.

See Also

Histogram, **FindDuplicates**, **Sort**

ThreadGroupCreate

`ThreadGroupCreate(`*`nt`*`)`

The ThreadGroupCreate function creates a thread group containing *nt* threads and returns a thread ID number. Use the number of computer processors for *nt* when trying to improve computation speed using parallel threads. A background worker might use just one thread regardless of the number of processors.

See Also

ThreadSafe Functions on page IV-106 and **ThreadSafe Functions and Multitasking** on page IV-329.

ThreadGroupGetDF

`ThreadGroupGetDF(`*`tgID`*`, `*`waitms`*`)`

ThreadGroupGetDFR should be used instead of ThreadGroupGetDF which causes memory leaks.

The ThreadGroupGetDF function retrieves a data folder path string from a thread group queue and removes the data folder from the queue.

When called from a preemptive thread it returns a data folder from the thread group's input queue. When called from the main thread it returns a data folder from the thread group's output queue.

tgID is a thread group ID returned by **ThreadGroupCreate**. You can pass 0 for *tgID* when calling ThreadGroupGetDF from a preemptive thread. You must pass a valid thread group ID when calling ThreadGroupGetDF from the main thread.

waitms is the maximum number of milliseconds to wait for a data folder to become available in the queue. Pass 0 to test if a data folder is available immediately. Pass INF to wait indefinitely or until a user abort.

ThreadGroupGetDF returns "" if the timeout period specified by *waitms* expires and no data folder is available in the queue.

See Also

ThreadSafe Functions on page IV-106 and **ThreadSafe Functions and Multitasking** on page IV-329.

The **ThreadGroupGetDFR** function.

ThreadGroupGetDFR

`ThreadGroupGetDFR(`*`tgID`*`, `*`waitms`*`)`

The ThreadGroupGetDF function retrieves a data folder reference from a thread group queue and removes the data folder from the queue. The data folder becomes a free data folder.

When called from a preemptive thread it returns a data folder from the thread group's input queue. When called from the main thread it returns a data folder from the thread group's output queue.

tgID is a thread group ID returned by **ThreadGroupCreate**. You can pass 0 for *tgID* when calling ThreadGroupGetDFR from a preemptive thread. You must pass a valid thread group ID when calling ThreadGroupGetDFR from the main thread.

waitms is the maximum number of milliseconds to wait for a data folder to become available in the queue. Pass 0 to test if a data folder is available immediately. Pass INF to wait indefinitely or until a user abort.

ThreadGroupGetDFR returns a NULL data folder reference if the timeout period specified by *waitms* expires and no data folder is available in the queue. You can test for NULL using **DataFolderRefStatus**.

See Also

ThreadSafe Functions on page IV-106, **ThreadSafe Functions and Multitasking** on page IV-329 and **Free Data Folders** on page IV-96.

ThreadGroupPutDF

`ThreadGroupPutDF `*`tgID`*`, `*`datafolder`*

The ThreadGroupPutDF operation posts data to a preemptive thread group.

Parameters

tgID is thread group ID returned by **ThreadGroupCreate**, datafolder is the data folder you wish to send to the thread group.

datafolder can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

Details

When you call it from the main thread, **ThreadGroupPutDF** removes *datafolder* from the main thread's data hierarchy and posts to the input queue of the thread group specified by *tgID*.

When you call it from a preemptive thread, use 0 for *tgID* and the data folder will be posted to the output queue of thread group to which thread belongs.

Input and output data folders may be retrieved from the queues by calling the string function **ThreadGroupGetDF** or **ThreadGroupGetDFR**.

Warning: Take care not to use any stale WAVE, NVAR, or SVAR variables that might contain references to objects in the data folder. Use **WAVEClear** on all WAVE reference variables that might contain references to waves that are in the data folder being posted before calling **ThreadGroupPutDF**. An error will occur if any waves in the data folder are in use or referenced in a WAVE variable.

Warning: Any DFREF variables that refer to the data folder (or any child thereof) must be cleared prior to executing this command. You can clear a DFREF using `dfr=$""`.

From the standpoint of the source thread, ThreadGroupPutDF is conceptually similar to KillDataFolder and, like KillDataFolder, if the current data folder is within *datafolder*, the current data folder is set to the parent of datafolder. You can not pass `root:` as *datafolder*.

See Also

The **ThreadGroupCreate** function, **ThreadSafe Functions** on page IV-106, and **ThreadSafe Functions and Multitasking** on page IV-329.

ThreadGroupRelease

`ThreadGroupRelease(`*`tgID`*` [, `*`beGraceful`*`])`

The ThreadGroupRelease function releases the thread group referenced by *tgID*. *tgID* is the thread group ID returned by **ThreadGroupCreate**. Any data folders remaining in the group's input or output queues are discarded.

Specifying *tgID*=-2 kills all running threads in all thread groups.

Normally you call ThreadGroupRelease after all threads in the group have completed. In the event that threads are still running, they are aborted. An attempt is made to safely stop running threads but, if they continue to run, they are killed.

The *beGraceful* parameter, added in Igor Pro 9.00, affects thread worker functions that include catch blocks to handle aborts gracefully. If *beGraceful* is omitted or is 0, ThreadGroupRelease performs aborts in a way that interferes with such catch blocks. If *beGraceful* is 1, the catch blocks run normally. See **Aborting Threads** on page IV-337 for details.

ThreadGroupRelease returns zero if successful, -1 if an error occurred (probably invalid *tgID*), or -2 if a force quit was needed. In the latter case, you should restart Igor Pro.

See Also

The **ThreadGroupCreate** function, **ThreadSafe Functions** on page IV-106, and **ThreadSafe Functions and Multitasking** on page IV-329.

ThreadGroupWait

`ThreadGroupWait(`*`tgID`*`, `*`waitms`*`)`

The ThreadGroupWait function returns index+1 of the first thread found still running after *waitms* milliseconds or returns zero if all are done.

tgID is the thread group ID returned by **ThreadGroupCreate** and *waitms* is milliseconds to wait.

If any of the threads of the group encountered a runtime error, the first such error will be reported now.

Use zero for *waitms* to just test or provide a large value to cause the main thread to sleep until the threads are finished. You can use INF to wait forever or until a user abort. If you know the maximum time the threads should take, you can use that value so you can print an error message or take other action if the threads don't return in time.

When ThreadGroupWait is called, Igor updates certain internal variables including variables that track whether a thread has finished and what result it returned. Therefore you must call ThreadGroupWait before calling **ThreadReturnValue**.

ThreadGroupWait updates the internal state of all threads in the group.

Finding a Free Thread

If you pass -2 for *waitms*, ThreadGroupWait returns index+1 of the first free (not running) thread or 0 if all threads in the group are running.

This allows you to dispatch a thread anytime a free thread is available. See **Parallel Processing - Thread-at-a-Time Method** on page IV-332 for an example.

See Also

The **ThreadGroupCreate** function, **ThreadSafe Functions** on page IV-106, and **ThreadSafe Functions and Multitasking** on page IV-329.

ThreadProcessorCount

`ThreadProcessorCount`

The ThreadProcessorCount function returns the number of processors in your computer. For example, on a Macintosh Core Duo, it would return 2.

ThreadReturnValue

`ThreadReturnValue(`*`tgID`*`, `*`index`*`)`

The ThreadReturnValue function returns the value that the specified thread function returned when it exited. Returns NAN if thread is still running. *tgID* is the thread group ID returned by **ThreadGroupCreate** and *index* is the thread number.

When **ThreadGroupWait** is called, Igor updates certain internal variables including variables that track whether a thread has finished and what result it returned. Therefore you must call ThreadGroupWait before calling ThreadReturnValue.

See Also

The **ThreadGroupCreate** function, **ThreadSafe Functions** on page IV-106, and **ThreadSafe Functions and Multitasking** on page IV-329.

ThreadSafe

`ThreadSafe Function `*`funcName`*`()`

The ThreadSafe keyword declaration specifies that a user function can be used for preemptive multitasking background tasks on multiprocessor computer systems.

A ThreadSafe function is one that can operate correctly during simultaneous execution by multiple threads. Such functions are generally limited to numeric or utility functions. Functions that access windows are not ThreadSafe. To determine if an operation is ThreadSafe, use the Command Help tab of the Help Browser and choose ThreadSafe from the pop-up menu.

ThreadSafe functions can call other ThreadSafe functions but may not call non-ThreadSafe functions. Non-ThreadSafe functions can call ThreadSafe functions.

See Also

ThreadSafe Functions on page IV-106 and **ThreadSafe Functions and Multitasking** on page IV-329.

ThreadStart

`ThreadStart `*`tgID`*`, `*`index`*`, `*`WorkerFunc`*`(`*`param1`*`, `*`param2`*`,…)`

The ThreadStart operation starts the specified function running in a preemptive thread.

Parameters

tgID is thread group ID returned by **ThreadGroupCreate**, *index* is the desired thread of the group to set up to execute the specified ThreadSafe *WorkerFunc*.

Details

The worker function starts running immediately.

The worker function must be defined as ThreadSafe and must return a real or complex numeric result.

The worker function's return value can be obtained after the function finishes by calling **ThreadReturnValue**. Igor records the fact that a thread has terminated when you call ThreadGroupWait so you must call ThreadGroupWait before calling ThreadReturnValue.

The worker function can take variable and wave parameters. It can not take pass-by-reference parameters or data folder reference parameters.

Any waves you pass to the worker are accessible to both the main thread and to your preemptive thread. Such waves are marked as being in use by a thread and Igor will refuse to perform any manipulations that could change the size of the wave.

See Also

The **ThreadGroupCreate** and **ThreadReturnValue** functions; **ThreadSafe Functions** on page IV-106, and **ThreadSafe Functions and Multitasking** on page IV-329.

ticks

```
ticks
```

The ticks function returns the number of ticks (approximately 1/60 second) elapsed since the operating system was initialized.

See Also

The **StopMSTimer** function.

TickWavesFromAxis

```
TickWavesFromAxis [ /W=graphName /DEST= {textWaveName, numericWaveName} /O
   /AUTO=mode ] axisName
```

The TickWavesFromAxis operation generates a pair of waves suitable for use as user tick waves (see **User Ticks from Waves** on page II-313). This allows you to programmatically determine the tick marks and tick labels that Igor would create for a given graph axis in auto mode.

TickWavesFromAxis was added in Igor Pro 8.00.

TickWavesFromAxis generates two waves: a two-column text wave containing tick labels and the names of tick types, and a numeric wave giving the positions of the ticks along the axis. By default, the information stored in the output waves reflects the automatically generated ticks based on the current axis settings but you can change this using the /AUTO flag.

By default the waves are given names derived from the graph window and axis name: <*graphname*>_<*axisname*>*_labels* and <*graphname*>_<*axisname*>*_values*. Use /DEST to give the waves custom names.

Parameters

axisName is the name of the axis for which the tick waves are generated. This will usually be left, bottom, right or top, but may be the name of a free axis.

Flags

/AUTO=*mode* Controls the axis settings that are used when generating the output waves:

mode = 0	Computed manual tick settings are used.
mode = 1	The automatic tick settings are used (default).
mode = 2	Either computed manual ticks or auto ticks are used, depending on the current setting of the graph.

If you use /AUTO=0 and the graph is currently in auto mode, the results could be surprising if you haven't actually set the computed manual ticks settings.

/AUTO was added in Igor Pro 9.00.

/DEST={*textWaveName*, *numericWaveName*}

Specifies custom names for the generated waves. *textWaveName* is the name of the text wave containing the tick labels. *numericWaveName* is the name of the numeric wave containing the axis position values for the ticks. If the waves don't already exist, they are created.

You may use data folder syntax as long as the data folders already exist.

The operation may create wave references for the destination waves if called in a user-defined function. See **Automatic Creation of WAVE References** on page IV-72 for details.

/O Tells TickWavesFromAxis that it can overwrite existing waves.

Igor returns an error if you attempt to overwrite a numeric wave with a text wave or a text wave with a numeric wave.

/W=winName Specifies the graph containing the axis. This may be a subwindow path. If you omit /W, the top graph window is used.

Details

The TickWavesFromAxis operation depends on drawing the axis to generate the list of ticks. It causes a screen refresh, equivalent to calling **DoUpdate**, twice during its execution.

The TickWavesFromAxis operation honors your axis format settings and manual range. It works on regular numeric axes, log axes and date/time axes. At this time it does not work on category axes or axes using user ticks from waves.

See Also

User Ticks from Waves on page II-313 , **ModifyGraph (axes)** UserTicks, ManTicks and ManMinor keywords

Tile

`Tile` [***`flags`***] [***`objectName`*** [, ***`objectName`***]…]

The Tile operation tiles the specified objects in the top page layout.

Parameters

objectName is the name of a graph, table, picture or annotation object in the top page layout.

Flags

/A=(*rows*,*cols*) Specifies number of rows/columns in which to tile objects.

/BBOX[=*ubb*] Specifies that you want to use the bounding box of the objects to be tiled as the tiling area. /BBOX overrides /W. See **Specifying the Tiling Area** on page II-489 for details. Added in Igor Pro 9.00.

/G=*grout* Specifies grout, the spacing between window tiles, in prevailing coordinates (points unless preceded by /I, /M or /R).

/I Specifies coordinates in inches.

/M Specifies coordinates in centimeters.

/O=*objTypes* Adds objects of type(s) specified by bitwise mask to list of objects to be tiled:

Bit 0:	Tile graphs.
Bit 1:	Tile tables.
Bit 3:	Tile pictures.
Bit 5:	Tile textboxes.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/PA[=*preserve*] /PA and /PA=1 specify that you want to preserve the rough arrangement of the objects to be tiled. See **Preserving Your Rough Arrangement** on page II-490 for details. Added in Igor Pro 9.00.

/R Specifies coordinates measured in percent of the printable page.

/S Adds selected objects to objects to be tiled.

/W=(*left*,*top*,*right*,*bottom*)

Specifies page layout area in which to tile objects. Coordinates are in points unless /I, /M or /R are specified before /W. /BBOX overrides /W.

Details

If /A=(*rows*,*cols*) is not used, Tile uses an appropriate number of rows and columns. If /A=(*rows*,*cols*) is used, objects are tiled in a grid of that many rows and columns. If *rows* or *cols* is zero, it substitutes an appropriate number for the zero parameter.

Objects to be tiled are determined by the /S and /O=*objTypes* flags and by any *objectName*s.

If no /S or /O flags are present and there are no *objectName*s, then all objects in the layout are tiled.

Otherwise the objects to be tiled are determined as follows:

- All objects specified by *objectName*s are tiled.
- If the /S flag is present, the selected objects, if any, are also tiled.
- If the /O=*objTypes* flag is present then any objects specified by *objTypes* are also tiled. *objTypes* is a bitwise mask, so /O=3 tiles both graphs and tables.

See Also

The **Stack** operation.

TileWindows

`TileWindows [`*`flags`*`] [`*`windowName`*` [, `*`windowName`*`]…]`

The TileWindows operation tiles the specified windows on the desktop (*Macintosh*) or in the Igor frame window (*Windows*).

Flags

/A=(*rows*,*cols*) Specifies number of rows/columns in which to tile windows.

/C Adds the command window to the windows to be tiled.

/G=*grout* Specifies grout, the spacing between tiles, in prevailing units (points unless /I or /M are used).

/I Specifies coordinates in inches.

/M Specifies coordinates in centimeters.

/O=*objTypes* Adds windows of types specified by *objTypes* to windows to be tiled.

objTypes is a bitwise mask where:

Bit 0:	Graphs
Bit 1:	Tables
Bit 2:	Page layouts
Bit 4:	Notebooks
Bit 6:	Control panels
Bit 7:	Procedure windows
Bit 9:	Help windows
Bit 12:	XOP target windows
Bit 14:	Camera windows
Bit 16:	Gizmo windows

Other bits should always be zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/P Adds the main procedure window to the windows to be tiled.

/R Specifies coordinates measured as % of tiling rectangle.

/W=(*left*,*top*,*right*,*bottom*)

Specifies tiling rectangle on the screen. Coordinates are in points unless /I, /M, or /R are specified before /W.

/WINS=*windowListStr*

Specifies the windows to be tiled using a semicolon-separated list of window names. Added in Igor Pro 9.00.

Details

If you omit the /W flag, the default tiling area is used. This is the area above your preferred command window position. You can set this using Misc→Command Buffer→Capture Prefs or Misc→History Area→Capture Prefs.

The windows to be tiled are determined by the /WINS, /C, /P, and /O=*objTypes* flags and by the *windowName*s. If none of these flags are present and there is no *windowName*s then all windows are tiled.

Otherwise the windows to be tiled are determined as follows:

- All visible named windows are tiled.
- All visible windows specified by /WINS are tiled.
- If the /C flag is present and the command window is visible, the command window is also tiled.
- If the /P flag is present and the procedure window is visible, the procedure window is also tiled.
- If the /O=*objTypes* flag is present, any visible windows specified by *objTypes* are also tiled.

Examples

To tile all the visible procedure windows, including the main one, use:

```
TileWindows/P/O=128        // 2^7=128
```

See Also

The **StackWindows** operation.

time

`time()`

The time function returns a string containing the current local time. The empty parentheses are required.

See Also

The **date**, **date2secs** and **DateTime** functions.

TitleBox

TitleBox [**/Z**] ***ctrlName*** [***keyword = value*** [, ***keyword = value*** …]]

The TitleBox operation creates the named title box in the target window.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the TitleBox control to be created or changed.

The following keyword=value parameters are supported:

anchor= *hv*
Specifies the anchor mode using a two letter code, *hv*. *h* may be L, M, or R for left, middle, and right. *v* may be T, C, or B for top, center and bottom. Default is LT.

If fixedSize=1, the anchor code sets the positioning of text within the frame.

align=*alignment*
Sets the alignment mode of the control. The alignment mode controls the interpretation of the *leftOrRight* parameter to the pos keyword. The align keyword was added in Igor Pro 8.00.

If *alignment*=0 (default), *leftOrRight* specifies the position of the left end of the control and the position of the control depends on the anchor keyword.

If *alignment*=1, *leftOrRight* specifies the position of the right end of the control and the right end position remains fixed if the control size is changed.

There is a conflict between the align keyword and the anchor keyword. See **TitleBox Positioning** on page V-1040.

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **Button** and **DefaultGUIControls** for more appearance details.

disable=*d*
Sets user editability of the control.

d=0: Normal.
d=1: Hide.
d=2: Draw in gray state.

fColor=(*r*,*g*,*b*[,*a*])
Sets color of the titlebox. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

fixedSize=*f*
Controls title box sizing:

f =0: The titlebox automatically sizes itself to fit the title text (default).
f =1: The size settings are honored, and the titlebox does not automatically size itself to fit the title text.

font="*fontName*"
Sets the font used for the control, e.g., `font="Helvetica"`.

frame= *f*
Sets frame style:

f=0: No frame.
f=1: Default (same as *f*=3).
f=2: Simple box.
f=3: 3D sunken frame.
f=4: 3D raised frame.
f=5: Text well.

fsize=*s*
Sets font size.

fstyle=*fs*

fs is a bitwise parameter with each bit controlling one aspect of the font style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*}

Sets the help for the control.

helpStr is limited to 1970 bytes (255 in Igor Pro 8 and before).

You can insert a line break by putting “\r” in a quoted string.

labelBack=(*r*,*g*,*b*[,*a*]) or 0

Sets background color for title box.*r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. If not set (or labelBack=0), then background is transparent (not erased).

pos={*leftOrRight*,*top*}

Sets the position in **Control Panel Units** of the top/left or top/right corner of the control at the time it is created. Unless fixedSize=1, a TitleBox control adjusts its size to fit the title text and the position can move in a way that depends on the anchor and align modes. See **TitleBox Positioning** on page V-1040.

pos+={*dx*,*dy*}

Offsets the position of the control in **Control Panel Units**.

size={*w*,*h*}

Sets the width and height of the control in pixels.

If fixedSize=1, sets the width of the control in pixels. If fixedSize=0, a TitleBox control adjusts its size to the title text, resulting in confusion about what the size does. In this case, size={0,0} is recommended. See **TitleBox Positioning** on page V-1040.

title=*titleStr*

Sets the text of the title box to *titleStr*. *titleStr* is limited to 255 bytes.

Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.

variable= *svar*

Specifies an optional global string variable from which to get the TitleBox text.

win=*winName*

Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Flags

/Z

No error reporting.

Details

The text can come from either the title=*titleStr* or variable=*svar* method. Whichever is used last is the current method. The maximum length of text with the title=*titleStr* method is 255 bytes while the variable=*svar* method has no limit.

Using escape codes you can change the font, size, style and color of text, and apply other effects. See **Annotation Escape Codes** on page III-53 for details.

By default, the titlebox automatically resizes itself relative to the anchor point on the rectangle that encloses the text. Therefore you can specify a size of 0,0 along with a pos value in order to place the anchor point at the desired position. When fixedSize=1 is used, the titlebox does not resize itself and instead honors the values specified via the size keyword.

TitleBoxes can be used not only for titles but also as status or results readout areas, especially in conjunction with the variable= *svar* mode. When using a titlebox like this, you may find it useful to use fixedSize=1 so that the titlebox doesn't change size as the text changes.

TitleBox Positioning

For compatibility with other controls, you can use the align=1 to maintain the position of the right end of a TitleBox control. This presents a conflict with the anchor keyword. To resolve the conflict, these rules are applied:

Applying align=1, which sets the alignment mode to right alignment, overrides any horizontal positioning specified using the anchor keyword.

Applying the anchor keyword resets the alignment mode to 0 and the anchor mode controls the horizontal position of the control.

If you specify fixedSize=1, only the align and pos keywords affect horizontal positioning; the anchor keyword has no effect.

For clarity, don't mix the anchor and align keywords unless you are using fixedSize=1. Use anchor in preference to align unless you are trying to keep the right ends of various types of controls aligned, especially on high-resolution displays on Windows.

Examples

```
NewPanel /W=(94,72,459,294)
DrawLine 150,32,150,140
DrawLine 70,100,213,100          // draw crossing lines at 150,100

// illustrate a default box
TitleBox tb1,title="A title box\rwith 2 lines",pos={150,100}

// Move center to 150,100
TitleBox tb1,pos={150,100},size={0,0},anchor=MC

// Set background color and therfore opaque mode
TitleBox tb1,labelBack=(55000,55000,65000)

// Now a few frame styles. Run these one at a time
TitleBox tb1,frame= 0            // no frame
TitleBox tb1,frame= 2            // plain frame
TitleBox tb1,frame= 3            // 3D sunken
TitleBox tb1,frame= 4            // 3D raised
TitleBox tb1,frame= 5            // text well

// Now some fancy text…
TitleBox tb1,frame= 1            // back to default (3D raised)
TitleBox tb1,title= "\Z18\[020 log\\B10\\M|[1 + 2K(jwt) + (jwt)\\S2\\M]|\\S-1"

// Create a string variable and hook up to the TitleBox
String s1= "text from a string variable"
TitleBox tb1,variable=s1

// Change string variable contents & note automatic update of TitleBox
s1= "something new"

// A TitleBox with right end at X=200 because the align keyword takes precedence
TitleBox tb, pos={200,40},size={0,0},align=1,anchor=MT,title="Short Title"

// A TitleBox with right end at X=200 because align by itself overrides
// any preexisting anchor
TitleBox tb, pos={200,40},size={0,0},align=1,title="Short Title"

// A TitleBox with text centered in the frame and right end at X=200,
// with frame always 75 points wide
TitleBox tb, pos={200,40},size={75,20},align=1,fixedSize=1,anchor=MT,title="Short Title"
```

See Also

Annotation Escape Codes on page III-53.

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

The **ControlInfo** operation for information about the control.

ToCommandLine

`ToCommandLine` ***`commandsStr`***

The ToCommandLine operation sends command text to the command line without executing the command(s).

The intended usage is for user-created panel windows with “To Cmd Line” buttons that are mimicking built-in Igor dialogs. You’ll usually want to use Execute, instead.

Parameters

commandsStr	The text of one or more commands.

Details

To send more than one line of commands, separate the commands with “\r” characters.

Note: ToCommandLine does not work when typed on the command line; use it only in a Macro, Proc, or Function.

Examples

```
Macro CmdPanel()
   PauseUpdate; Silent 1
   NewPanel /W=(150,50,430,229)
   Button toCmdLine,pos={39,148},size={103,20},title="To Cmd Line"
   Button toCmdLine,proc=ToCmdLineButtonProc
End

Function ToCmdLineButtonProc(ctrlName) : ButtonControl
   String ctrlName

   String cmd="MyFunction(xin,yin,\"yResult\")"// line 1: generate results
   cmd +="\rDisplay yOutput vs wx as \"results\"" // line 2: display results
   ToCommandLine cmd

   return 0
End
```

See Also

The **Execute** and **DoIgorMenu** operations.

ToolsGrid

ToolsGrid [**/W=*winName***] ***keyword = value*** [, ***keyword = value*** …]

The ToolsGrid operation controls the grid you can use for laying out draw or control objects.

Parameters

ToolsGrid can accept multiple *keyword* = *value* parameters on one line.

snap=*n*	Turns snap to grid on (*n*=1) or off (*n*=0).
visible=*n*	Turns on grid visibility (*n*=1) or hides it (*n*=0).
grid=(*xy0*,*dxy*,*ndiv*)	Defines both X and Y grids where *ndiv* is the number of subdivisions between major grid lines and *xy0* and *dxy* define the origin and spacing. Units are in points.
gridx=(*x0*,*dx*,*ndiv*)	Defines the X grid where *ndiv* is the number of subdivisions between major grid lines and *x0* and *dx* define the origin and spacing. Units are in points.
gridy=(*y0*,*dy*,*ndiv*)	Defines the Y grid where *ndiv* is the number of subdivisions between major grid lines and *y0* and *dy* define the origin and spacing. Units are in points.

Flags

/W=*winName*	Sets the named window or subwindow for drawing. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The default grid is 1 inch with 8 subdivisions. The grid is visible only in draw or selector mode and appears in front of the currently active draw layer.

TraceFromPixel

`TraceFromPixel(`*`xpixel`*`, `*`ypixel`*`, `*`optionsString`*`)`

The TraceFromPixel function returns a string based on an attempt to hit test the provided X and Y coordinates. Used to determine if the mouse was clicked on a trace in a graph.

When a trace is found, TraceFromPixel returns a string containing the following `KEY:value;` pairs:

`TRACE:`*`tracename`*

`HITPOINT:`*`pnt`*

tracename will be quoted if necessary and may contain instance notation. *pnt* is the point number index into the trace's wave when the hit was detected. If a trace is not found near the coordinate point, a zero length string is returned.

Parameters

xpixel and *ypixel* are the X and Y pixel coordinates.

optionsString can contain the following:

`WINDOW:`*`winName`*`;`

`PREF:`*`traceName`*`;`

`ONLY:`*`traceName`*`;`

`DELTAX:`*`dx`*`;DELTAY:`*`dy`*`;`

Use the `WINDOW` option to hit test in a graph other than the top graph. Use the `ONLY` option to search only for a special target trace. If the PREF option is used then the search will start with the specified trace but if no hit is detected, it will go on to the others.

When identifying a subwindow with `WINDOW:`*`winName`*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

The DELTAX and DELTAY values must both be specified to alter the region that Igor searches for traces. The *dx* and *dy* values are in pixels and the region searched is the rectangle from *xpixel-dx* to *xpixel+dx* and *ypixel-dy* to *ypixel+dy*.

If DELTAX or DELTAY are omitted, the search region depends on whether PREF or ONLY are specified. If either are specified then Igor first searches for the trace using *dx* = 3 and *dy* = 3. If the trace is not identified, Igor searches again using *dx* = 6 and *dy* = 6. If the trace is still not identified, Igor gives up and returns a zero-length result string.

If neither PREF nor ONLY are specified then Igor uses tries 3, 6, 12, and 24 for *dx* and *dy* until it finds a trace or gives up and returns a zero-length result string.

See Also

The **NumberByKey**, **StringByKey**, **AxisValFromPixel**, and **PixelFromAxisVal** functions.

ModifyGraph (traces) and **Instance Notation** on page IV-20 for discussions of trace names and instance notation.

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

TraceInfo

`TraceInfo(`*`graphNameStr`*`, `*`ywavenameStr`*`, `*`instance`*`)`

The TraceInfo function returns a string containing a semicolon-separated list of information about the trace in the named graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

yWaveNameStr is either the name of a wave containing data displayed as a trace in the named graph, or a trace name (wave name with "#n" appended to distinguish the nth image of the wave in the graph). You might get a trace name from the **TraceNameList** function.

If *yWaveNameStr* contains a wave name, *instance* identifies which trace of *yWaveNameStr* you want information about. *instance* is usually 0 because there is normally only one instance of a given wave displayed in a graph. Set *instance* to 1 for information about the second trace of the wave named by *yWaveNameStr*, etc. If *yWaveNameStr* is `""`, then information is returned on the *instance*th trace in the graph.

If *yWaveNameStr* is a trace name, and *instance* is zero, the instance is extracted from *yWaveNameStr*. If *instance* is greater than zero, the wave name is extracted from *yWaveNameStr*, and information is returned concerning the *instance*th instance of the wave.

Details

The string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

Keyword	Information Following Keyword
AXISFLAGS	Flags used to specify the axes. Usually blank because /L and /B (left and bottom axes) are the defaults.
AXISZ	Z value of a contour level trace or NaN if the trace is not a contour trace.
ERRORBARS	The ErrorBars command for the trace, as it would appear in the recreation macro (without the beginning tab character).
RECREATION	List of keyword commands as used by ModifyGraph command. The format of these keyword commands is: *keyword*(x)=*modifyParameters*;
TYPE	The type of trace: 0: XY or waveform trace 1: Contour trace 2: Box plot trace 3: Violin plot trace TYPE was added in Igor Pro 8.00.
XAXIS	X axis name.
XRANGE	Point subrange of the trace's X data wave in "[*startPoint*,þ*endPoint* : *increment*]" format. **Note**: Unlike the actual syntax of a trace subrange specification where increment is preceded by a semicolon character, here it is preceded by a colon character to preserve the notion that semicolon is what separates the keyword-value groups. If the entire X wave is displayed (the usual case), the XRANGE value is "[*]". If an X wave is not used to display the trace, then the XRANGE value is `""`.
XWAVE	X wave name if any, else blank.
XWAVEDF	Full path to the data folder containing the X wave or blank if no X wave.
YAXIS	Y axis name.
YRANGE	Point subrange of the trace's Y data wave or "[*]".

The format of the RECREATION information is designed so that you can extract a keyword command from the keyword and colon up to the ";", prepend "`ModifyGraph `", replace the "x" with the name of a trace ("`data#1`" for instance) and then **Execute** the resultant string as a command.

Note: The syntax of any subrange specifications in the RECREATION information are modified in the same way as for XRANGE and YRANGE. Currently only the zColor, zmrkSize, and zmrkNum keywords might have a subrange specification.

Examples

This example shows how to extract a string value from the keyword-value list returned by TraceInfo:

```
String yAxisName= StringByKey("YAXIS", TraceInfo("","",0))
```

This example shows how to extract a subrange and put the semicolon back:

```
String yRange= StringByKey("YRANGE", TraceInfo("","",0))
Print yRange                  // prints "[30,40:2]"
yRange= ReplaceString(":", yRange, ";")
Print yRange                  // prints "[30,40;2]"
```

The next example shows the trace information for the second instance of the wave "data" (which has an instance number of 1) displayed in the top graph:

```
Make/O data=x;Display/L/T data,data     // two instances of data: 0 and 1
Print TraceInfo("","data",1)[0,64]
Print TraceInfo("","data",1)[65,128]
```

Prints the following in the history area:

```
XWAVE:;YAXIS:left;XAXIS:top;AXISFLAGS:/T;AXISZ:NaN;XWAVEDF:;YRANG
E:[*];XRANGE:;TYPE:0;ERRORBARS:;RECREATION:zColor(x)=0;zColorMax
```

Following is a function that returns the marker code from the given instance of a named wave in the top graph. This example uses the convenient GetNumFromModifyStr() function provided by the #include <Readback ModifyStr> procedures, which are useful for parsing strings returned by TraceInfo.

```
#include <Readback ModifyStr>

Function MarkerOfWave(wv,instance)
   Wave wv
   Variable instance

   Variable marker
   String info = TraceInfo("",NameOfWave(wv),instance)

   marker = GetNumFromModifyStr(info,"marker","",0)

   return marker
End
```

See Also

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

The **Execute** operation.

TraceNameList

TraceNameList(*graphNameStr, separatorStr, optionsFlag*)

The TraceNameList function returns a string containing a list of trace names in the graph window or subwindow identified by *graphNameStr*.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

The parameter *separatorStr* should contain a single ASCII character such as "," or ";" to separate the names.

Details

The bits of *optionsFlag* have the following meanings:

Bit Number	Bit Value	Meaning
0	1	Include normal graph traces
1	2	Include contour traces
2	4	Omit hidden traces (the default is to list even hidden traces)
3	8	Include box plot traces
4	16	Include violin plot traces

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

A trace name is defined as the name of the Y wave that defines the trace with an optional #ddd suffix that distinguishes between two or more traces that have the same wave name. It may also be a user-defined trace name. Since the trace name has to be parsed, it is quoted if necessary.

Commands that take a trace name as a parameter or in a keyword can use a string containing a trace name with the $ operator to specify traceName. For instance, to change the display mode of a wave, you might use

```
ModifyGraph mode(myWave#1)=3
```

but

```
String myTraceName="myWave#1"
ModifyGraph mode($myTraceName)=3
```

will also work.

Examples

```
Make/O jack,'jack # 2';Display jack,jack,'jack # 2','jack # 2'
Print TraceNameList("",";",1)
Prints: jack;jack#1;'jack # 2';'jack # 2'#1;
```

```
// Generate a list of hidden traces
Make/O jack,jill,joy;Display jack,jill,joy
ModifyGraph hideTrace(joy)=1// hide joy
// (hidden + visible) - visible = hidden
String visibleTraces=TraceNameList("",";",1+4)// only visible normal traces
String allNormalTraces=TraceNameList("",";",1)// hidden + visible normal traces
String hiddenTraces= RemoveFromList(visibleTraces,allNormalTraces)
Print hiddenTraces
// Prints: joy;
```

See Also

Trace Names on page II-282, **Programming With Trace Names** on page IV-87, **User-defined Trace Names** on page IV-89.

For other commands related to waves and traces: **WaveRefIndexed**, **XWaveRefFromTrace**, **TraceNameToWaveRef**, **CsrWaveRef**, and **CsrXWaveRef**.

For a description of traces: **ModifyGraph**. For a discussion of contour traces: **Contour Traces** on page II-370.

For commands referencing other waves in a graph: **ImageNameList**, **ImageNameToWaveRef**, **ContourNameList**, and **ContourNameToWaveRef**.

ModifyGraph (traces) and **Instance Notation** on page IV-20 for discussions of trace names and instance notation.

TraceNameToWaveRef

TraceNameToWaveRef(*graphNameStr*, *traceNameStr*)

The TraceNameToWaveRef function returns a wave reference to the Y wave corresponding to the given trace in the graph window or subwindow named by *graphNameStr*.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

The trace is identified by the string in *traceNameStr*, which could be a string determined using **TraceNameList**. Note that the same trace name can refer to different waves in different graphs.

Use **Instance Notation** (see page IV-20) to choose from traces in a graph that represent waves of the same name. For example, if *traceNameStr* is “myWave#2”, it refers to the third instance of wave “myWave” in the graph (“myWave#0” or just “myWave” is the first instance).

See Also

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

For other commands related to waves and traces: **WaveRefIndexed**, **XWaveRefFromTrace**, **TraceNameList**, **CsrWaveRef**, and **CsrXWaveRef**.

For a description of traces: **ModifyGraph**. For a discussion of contour traces, see **Contour Traces** on page II-370.

For a discussion of wave references, see **Wave Reference Functions** on page IV-197.

For commands referencing other waves in a graph: **ImageNameList**, **ImageNameToWaveRef**, **ContourNameList**, and **ContourNameToWaveRef**.

Triangulate3D

`Triangulate3D` `[`**`/OUT=`*`format`***`]` ***`srcWave`***

The Triangulate3D operation creates a Delaunay "triangulation" of a 3D scatter wave. The output is a list of tetrahedra that completely span the convex volume defined by *srcWave*. Triangulate3D can also generate the triangulation needed for performing 3D interpolation for the same domain. Normally *srcWave* is a triplet wave (a 2D wave of 3 columns), but can use any 2D wave that has more than 3 columns (the operation ignores all but the first 3 columns).

Flags

/OUT=*format* Specifies how to save the output triangulation data.

format=1: Default; saves the triangulation result in the wave M_3DVertexList, which contains in each row, indices to rows in *srcWave* that describe the X, Y, Z coordinates of a single tetrahedral vertex. Each tetrahedron is described by one row in M_3DVertexList.

format=2: Saves the triangulation result in the wave M_TetraPath, which is a triplet path wave describing the tetrahedra edges. For each tetrahedron, there are four rows (triangles) separated by row of NaNs. The total number of rows in M_TetraPath is 20 times the number of tetrahedra in the triangulation.

format=4: Saves a wave containing internal diagnostic information generated during the triangulation process.

/VOL Computes the volume of the full convex hull by summing the volumes of the tetrahedra generated in the triangulation. The result is stored in the variable V_value. This flag requires Igor Pro 7.00 or later.

Details

Triangulate3D implements Watson's algorithm for tetrahedralization of a set of points in three dimensions. It starts by creating a very large tetrahedron which inscribes all the data points followed by a sequential insertion of one datum at a time. With each new datum the algorithm finds the tetrahedron in which the datum falls. It then proceeds to subdivide the tetrahedron so that the datum becomes a vertex of new tetrahedra.

The algorithm suffers from two known problems. First, it may, due to numerical instabilities, result in tetrahedra that are too thin. You can get around this problem by introducing a slight random perturbation in the input wave. For example:

```
srcWave+=enoise(amp)
```

Here amp is chosen so that it is much smaller than the smallest cartesian distance between two input points.

The second problem has to do with memory allocations which may exhaust available memory for some pathological spatial distributions of data points. The operation reports both problems in the history area.

Examples

```
Make/O/N=(10,3) ddd=gnoise(5)     // create random 10 points in space
Triangulate3d/out=2 ddd

// now display the triangulation in Gizmo:
Window Gizmo0() : GizmoPlot
    PauseUpdate; Silent 1
    if(exists("NewGizmo")!=4)
        DoAlert 0, "Gizmo XOP must be installed"
        return
    endif
    NewGizmo/N=Gizmo0 /W=(309,44,642,373)
    ModifyGizmo startRecMacro
    ModifyGizmo scalingMode=2
    AppendToGizmo Scatter=root:ddd,name=scatter0
    ModifyGizmo ModifyObject=scatter0 property={ scatterColorType,0}
    ModifyGizmo ModifyObject=scatter0 property={ Shape,2}
    ModifyGizmo ModifyObject=scatter0 property={ size,0.2}
    ModifyGizmo ModifyObject=scatter0 property={ color,0,0,0,1}
    AppendToGizmo Path=root:M_TetraPath,name=path0
    ModifyGizmo ModifyObject=path0 property={ pathColor,0,0,1,1}
    ModifyGizmo setDisplayList=0, object=scatter0
    ModifyGizmo setDisplayList=1, object=path0
    ModifyGizmo autoscaling=1
    ModifyGizmo compile
    ModifyGizmo endRecMacro
End
```

References

Watson, D.F., Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes, *The Computer J.*, *24*, 167-172, 1981.

Further information about this algorithm can be found in:

Watson, D.F., *CONTOURING: A guide to the analysis and display of spatial data*, Pergamon Press, 1992.

See Also

The **Interpolate3D** operation and the **Interp3D** function.

TrimString

TrimString(*str* [, *simplifyInternalSpaces*])

The TrimString function returns a string identical to *str* except that leading and trailing whitespace characters are removed. The whitespace characters are space, tab, carriage-return and linefeed.

If the optional second parameter is non-zero, then each run of whitespace characters between words in *str* is "simplified" to a single space character.

TrimString was added in Igor Pro 7.00.

Examples

```
Print TrimString("   spaces   at  ends  ")        // Prints "spaces   at  ends"
Print TrimString("   spaces   at  ends  ", 1)     // Prints "spaces at ends"
```

See Also

SplitWave, **RemoveEnding**, **ReplaceString**

trunc

trunc(*num*)

The trunc function returns the integer closest to *num* in the direction of zero.

The result for INF and NAN is undefined.

See Also

The **round**, **floor**, and **ceil** functions.

try

try

The try flow control keyword marks the beginning of the initial code block in a try-catch-endtry flow control construct.

See Also

The **try-catch-endtry** flow control statement for details.

try-catch-endtry

```
try
   <code>
catch
   <code to handle abort>
endtry
```

A try-catch-endtry flow control statement provides a means for catching and responding to abort conditions in user functions. A programmatic abort is generated when the code executes Abort, AbortOnRTE or

AbortOnValue. A user abort is generated when the user clicks the Abort button or presses the user abort key combination.

When code executes in the try-catch area, a programmatic abort immediately jumps to the code in the catch-endtry area rather than jumping to the end of the user function. A user abort jumps to the catch-endtry area when a flow control keyword such as for or while executes or at the end of the try code. Normal flow (no aborts) skips all code within the catch-endtry area.

Details

During execution of code in the catch-endtry area, user aborts are suppressed. This means that, if the user attempts to abort procedure execution by pressing the **User Abort Key Combinations** or by clicking the Abort button, this will not abort the catch code itself.

When an abort occurs, information about the cause of the abort is returned via the `V_AbortCode` variable as follows:

-4:	Abort triggered by AbortOnRTE.
-3:	Abort caused by Abort operation.
-2:	Stack overflow abort.
-1:	User abort.
>=1:	Abort triggered by AbortOnValue.

See Also

Flow Control for Aborts on page IV-48 and **try-catch-endtry Flow Control** on page IV-49 for further details.

The **AbortOnRTE** and **AbortOnValue** keywords, and the **Abort** operation.

UInt64

`uint64 localName`

Declares a local unsigned 64-bit integer in a user-defined function or structure.

UInt64 is available in Igor Pro 7 and later. See **Integer Expressions** on page IV-38 for details.

See Also

Int, **Int64**

UniqueName

`UniqueName(baseName, objectType, startSuffix [, windowNameStr])`

The UniqueName function returns the concatenation of *baseName* and a number such that the result is not in conflict with any other object name.

windowNameStr is optional. If missing, it is taken to be the top graph, panel, layout, or notebook according to the value of *objectType*.

In Igor Pro 9.00 or later, you can use the **CreateDataObjectName** function as a replacement for some combination of CheckName, CleanupName, and UniqueName to create names of waves, global variables, and data folders.

Details

baseName should be an unquoted name, such as you might receive from the user via a dialog or control panel.

objectType is one of the following:

1	Wave
2	Reserved
3	Numeric variable
4	String variable
5	XOP target window
6	Graph window
7	Table window
8	Layout window
9	Control panel window
10	Notebook window
11	Data folder
12	Symbolic path
13	Picture
14	Annotation in the named or topmost graph or layout
15	Control in the named or topmost graph or panel
16	Notebook action character in the named or topmost notebook
17	Gizmo window (added in Igor Pro 9.00)

startSuffix is the number used as a starting point when generating the numeric suffix that makes the name unique. Normally you should pass zero for startSuffix. If you know that names of the form base0 through baseN are in use, you can make UniqueName run a bit faster by passing `N+1` as the *startSuffix*.

The *windowNameStr* argument is used only with object types 14, 15, and 16. The returned name is unique only to the window (other windows might have objects with the same name). If a named window is given but does not exist, UniqueName returns *baseName startSuffix*. *windowNameStr* is ignored for other object types.

The Main Namespace

Waves, numeric variables, string variables, built-in and external functions, built-in and external operations, user-defined functions, macros and reserved keywords exist in the main namespace. The names of each of those objects must be unique in that namespace.

Window Namespace

Values of *objectType* from 5 through 10 refer to the window namespace.

The expression

```
CheckName("<name>", x)     // where x is 5, 6, 7, 8, 9 or 10
```

returns 0 if the specified name is syntactically legal and unique in the window namespace.

That does not guarantee that the name is allowed as a window name because the DoWindow/C operation, which changes a window's name, requires that the name also not conflict with objects in the main namespace. Consequently, to determine if a name is allowed as a window name, use this:

```
// Window names must be unique in both the window and main namespaces
Variable nameOK = CheckName("<name>",6)==0 && CheckName("<name>",1)==0
```

Window Macro Names

Window macro names are treated different from other main namespace names. DoWindow/C allows you to change the name of a window to the name of an existing window macro but not to any other name in the main namespace. Also this expression:

```
CheckName("<name>", x)     // where x is 1, 3, 4 (main namespace)
```

returns 0 even if the specified name is used as a window macro name.

UniqueName Thread Safety

As of Igor Pro 8.00, you can call UniqueName from an Igor preemptive thread but only if *objectType* is 1 (wave), 3 (global numeric variable), 4, (global string variable), 11 (data folder), or 12 (symbolic path). For any other value of *objectType*, UniqueName returns a runtime error.

Examples

```
String uniqueWaveName = UniqueName(baseWaveName, 1, 0)
String uniqueControlName = UniqueName("ctrl", 15, 0, "Panel0")
```

See Also

Object Names on page III-501, **Programming with Liberal Names** on page IV-168, **CreateDataObjectName**, **CheckName**, **CleanupName**

UnPadString

`UnPadString(`*`str`*`, `*`padValue`*`)`

The UnPadString function undoes the action of PadString. It returns a string identical to *str* except that trailing bytes of *padValue* are removed.

See Also

PadString

UnsetEnvironmentVariable

`UnsetEnvironmentVariable(`*`varName`*`)`

The UnsetEnvironmentVariable function deletes the variable named *varName* from the environment of Igor's process, if it exists

The function returns 0 if it succeeds or a nonzero value if it fails.

The UnsetEnvironmentVariable function was added in Igor Pro 7.00.

Parameters

varName	The name of an environment variable which does not need to actually exist. It must not be an empty string and may not contain an equals sign (=).

Details

The environment of Igor's process is composed of a set of key=value pairs that are known as environment variables.

The environment of Igor's process is composed of a set of key=value pairs that are known as environment variables. Any child process created by calling ExecuteScriptText inherits the environment variables of Igor's process.

UnsetEnvironmentVariable changes the environment variables present in Igor's process and any future process created by ExecuteScriptText but does not affect any other processes already created.

On Windows, environment variable names are case-insensitive. On other platforms, they are case-sensitive.

Examples

```
Variable result
result = SetEnvironmentVariable("SOME_VARIABLE", "15")      // Sets the variable
result = UnsetEnvironmentVariable("SOME_VARIABLE")          // Unsets the variable
```

See Also

GetEnvironmentVariable, **SetEnvironmentVariable**

Unwrap

`Unwrap `*`modulus`*`, `*`waveName`*` [, `*`waveName`*`]`…

The Unwrap operation scans through each named wave trying to undo the effect of a modulus operation.

Parameters

modulus is the value applied to the named waves through the **mod** function as if the command:

```
waveName = mod(waveName,modulus)
```

had been executed. It is this calculation which Unwrap attempts to undo.

Details

The unwrap operation works with 1D waves only. See **ImageUnwrapPhase** for phase unwrapping in two dimensions.

Examples

If you perform an FFT on a wave, the result is a complex wave in rectangular coordinates. You can create a real wave that contains the phase of the result of the FFT with the command:

```
wave2 = imag(r2polar(wave1))
```

However, the rectangular to polar conversion leaves the phase information modulo 2π. You can restore the phase information with the command:

```
Unwrap 2*pi, wave2
```

Because the first point of a wave that has been FFTed has no phase information, in this example you would *precede* the Unwrap command with the command:

```
wave2[0] = wave2[1]
```

See Also

The **ImageUnwrapPhase** operation and **mod** function.

UnzipFile

UnzipFile [/O[=*mode*] /PASS=*passwordStr* /PIN=*inputPathName* /POUT=*outputPathName* /Z[=*z*]] *inputFileStr*, *outputFolderStr*

The UnzipFile operation unzips a file and saves the contents of the file in the specified output directory.

Warning: If you specify /O=2 for an output directory that already exists, all previous contents of the directory are deleted.

The UnzipFile operation was added in Igor Pro 9.00.

Input File Parameter

inputFileStr specifies the zip file to unzip. The file name extension is not important but the file must be a zip file.

inputFileStr can be a full path to the file, in which case /PIN=*inputPathName* is not needed, a partial path relative to the directory associated with *inputPathName*, or the name of a file in the folder associated with *inputPathName*.

If you use a full or partial path for *inputFileStr*, see **Path Separators** on page III-451 for details on forming the path.

Output Folder Parameter

outputFolderStr specifies the directory into which the contents of the zip file will be extracted.

outputFolderStr can be a full path to the directory, in which case /POUT=outputPathName is not needed, a partial path relative to the directory associated with *outputPathName*, or an empty string in which case the directory specified by /POUT=*outputPathName* is used for the output directory.

If *outputFolderStr* is an empty string, the /POUT flag specifies the output directory.

If *outputFolderStr* is not an empty string, it must end with a directory name and not a path separator such as colon or backslash.

If you use a full or partial path for *outputFolderStr*, see **Path Separators** on page III-451 for details on forming the path.

If the output directory does not exist, it is created automatically.

Flags

/O[=*mode*] Controls whether the contents of the output directory are overwritten.

mode=0: Does not overwrite. If the output directory exists and is not empty, the operation generates an error. This is the default behavior if you omit /O.

mode=1: Merges the contents of the zip file with the existing contents of the output directory. Any existing file with the same name as a file within the zip file is overwritten.

Unlike /O=2, the contents of the output directory and any subdirectories are not deleted. If the output directory already contains files whose names do not conflict with files in the zip file, those files remain untouched.

/O=1 is the same as /O.

mode=2: Deletes all contents of the output directory before extracting the contents of the zip file. Use this option if you want the contents of the output directory to exactly reflect the contents of the zip file.

Warning: If you specify /O=2 for an output directory that already exists, all previous contents of the directory are deleted.

/PASS=*passwordStr*

Specifies the password for a password-protected zip file. Only the older ZipCrypto "encryption" algorithm is supported, not the newer and much more secure AES-256 algorithm.

/PIN=*inputPathName*

Contributes to the specification of the input zip file to be extracted. inputPathName is the name of an existing symbolic path. See ***Input File Parameter*** above for details.

/POUT=*outputPathName*

/ Contributes to the specification of the output directory into which the zip file's contents will be extracted. outputPathName is the name of an existing symbolic path. See ***Output Folder Parameter*** above for details.

/Z[=z] Suppress error generation.

/Z=0: Do not suppress errors. If an error occurs, Igor aborts procedure execution. This is the default behavior if you omit /Z.

/Z=1: Suppress errors. Errors do not abort procedure execution. Check the V_Flag output variable to see if an error occurred.

/Z alone has the same effect as /Z=1.

Output Variables

UnzipFile sets the following output variables:

V_flag	V_flag is set to zero if the operation succeeds or to a non-zero Igor error code if it fails. You can use V_flag along with the /Z flag to handle errors and prevent them from halting procedure execution.
S_outputFullPath	A string containing the full path to the output directory. If the operation fails, S_outputFullPath is set to "".

Limitations

File and directory names within a zip file that contain non-ASCII characters may not have the correct names after unzipping.

The created and modified timestamps of a file are reconstructed with limited range and precision. Some zip files store these timestamps in an alternative format that allows for greater precision and range, but the current unzip algorithm does not support this newer format.

Examples

```
// Unzip a file using full paths
String zipFileString = "C:Users:Fred:Documents:Test.zip"
String outputPathString = "C:Users:Fred:Documents:Extracted"
UnzipFile zipFileString, outputPathString
// Unzip a file using symbolic paths
String zipFileName = "Test.zip"
String outputFolderName = ""         // Unzip to folder specified by /POUT
UnzipFile/PIN=home/POUT=home zipFileName, outputFolderName
```

UpperStr

UpperStr(*str*)

The UpperStr function returns a string expression in which all lower-case ASCII characters in *str* are converted to upper-case.

See Also

The **LowerStr** function.

URLDecode

URLDecode(*inputStr*)

The URLDecode function returns a percent-decoded copy of the percent-encoded string *inputStr*. It is unlikely that you will need to use this function; it is provided for completeness.

For an explanation of percent-encoding, see **Percent Encoding** on page IV-268.

Example

```
String theURL = "http://google.com?key1=35%25%20larger"
theURL = URLDecode(theURL)
Print theURL
  http://google.com?key1=35% larger
```

See Also

URLEncode, **URLRequest**, **URLs** on page IV-267.

URLEncode

URLEncode(*inputStr*)

The URLEncode function returns a percent-encoded copy of *inputStr*.

Percent-encoding is useful when encoding the query part of a URL or when the URL contains special characters that might otherwise be misinterpreted by a web server. For an explanation of percent-encoding, see **Percent Encoding** on page IV-268.

Example

```
String baseURL = "http://google.com"
String key1 = "key1"
String value1 = URLEncode("35% larger")
String theURL = ""
sprintf theURL, "%s?%s=%s", baseURL, key1, value1
Print theURL
  http://google.com?key1=35%25%20larger
```

See Also

URLDecode, **URLRequest**, **URLs** on page IV-267.

URLRequest

URLRequest [flags] url=urlStr [method=methodName, headers=headersStr]

The URLRequest operation connects to a URL using the specified method and optionally stores the response from the server. *urlStr* can point to a remote server or to a local file. The server's response is stored in the S_serverResponse output variable or in a file if you use the /FILE flag.

The URLRequest operation was added in Igor Pro 7.00.

Keywords

The url=*urlStr* keyword is require. All others are optional.

url=*urlStr* A string containing the URL to retrieve. See **URLs** on page IV-267 for details.

method=*methodName* Specifies which method to use for the request. The get method is used by default.

This table shows the valid methods for each supported scheme. Not all servers support all of the listed methods.

Scheme	Supported Methods
http, https	get, post, put, head, delete
ftp	get, put
file	get, put

Because *methodName* is a name, not a string, you must not enclose it in quotes.

Before using the post method, you should read The **The HTTP POST Method** on page V-1058 so that you know how to use the optional headers parameter.

If you use the head method, URLRequest sets the S_serverResponse output variable to "" because only the headers are retrieved. As with other methods, the headers are stored in the S_headers output variable.

headers=*headersStr* Specifies a string containing additional or replacement headers to use with the request. This parameter is ignored unless the scheme is http or https.

The headers parameter is provided primarily for use with the post method when making HTTP requests and is ignored for schemes other than http/https. The header consists of a colon-separated key:value pair (though see the next paragraph for an exception). Pairs must be separated by a carriage return (\r) character.

Certain standard headers (such as Content-Type and User-Agent) may automatically be set when making the request. You can override those default values by using this keyword and setting a different value. If you add a header with no content, as in "Accept:" (there is no data on the right side of the colon), the internally used header is disabled. To actually add a header with no content, use the form "MyHeader;" (note the trailing semicolon).

Any headers specified with this keyword are sent only to the http server, not to the proxy server, if one is in use.

See **The HTTP POST Method** on page V-1058 for a detailed explanation and examples.

Flags

/AUTH={*username*, *password* }

Uses the specified *username* and *password* string parameters for authentication. Values provided here override any username and/or password provided as part of the URL. To specify a username but not a password, pass "" for the *password* parameter.

Note: See **Safe Handling of Passwords** on page IV-270 for more information on how to use URLRequest to prevent authentication information, such as passwords, from being accidentally revealed.

/DFIL=*dataFileNameStr* Specifies a file name to use as a source of data. Typically this flag is used only with the put or post methods, but it is accepted with all methods. Unless *dataFileNameStr* is a full path, the /P flag must also be used. When using the post or put methods, one and only one of the /DFIL or /DSTR flags must be used.

When you use /DFIL and the method is anything other than put, the "Content-Type: application/x-www-form-urlencoded" header is automatically added. If necessary, you can override this behavior using the headers keyword. See **The HTTP POST Method** on page V-1058 for more information.

/DSTR=*dataStr* Specifies the string to use as a source of data. Typically you use this flag only with the put or post methods, but it is accepted with all methods. When using the post or put methods, one and only one of the /DFIL or /DSTR flags must be used.

When /DSTR is used and the method is anything other than put, the "Content-Type: application/x-www-form-urlencoded" header is automatically added. If necessary, you can override this behavior using the headers keyword. See **The HTTP POST Method** on page V-1058 for more information.

/FILE=*destFileNameStr* If present, URLRequest saves the server's response in a file instead of in the S_serverResponse output variable.

URLRequest ignores /FILE if you include the /IGN flag and the *ignoreResponse* parameter is not 0.

destFileNameStr can be a full path to the file, in which case /P=*pathName* is not needed, a partial path relative to the folder associated with pathName, or the name of a file in the folder associated with pathName. If the file already exists, URLRequest returns an error unless you include the /O flag.

If you include /O and the file already exists, the existing file is overwritten. This happens even in the event of an empty response or transfer error.

You should consider using the /FILE flag when you are expecting the server to return a large amount of data, such as when downloading a file.

/IGN[=*ignoreResponse*] Ignore and do not store the server's response to the request. /IGN alone has the same effect as /IGN=1.

If ignore is turned on, URLRequest sets the S_serverResponse output variable to "", regardless of whether the server responded to the request or not. If you include the /FILE flag, URLRequest does not create the output file and sets S_fileName is set to "".

This flag is useful only when your goal is to establish a connection with a server and the server's response is not important. All error message codes and strings are still set when ignore is on.

/IGN=0: Same as no /IGN.

/IGN=1: Ignore server response completely. S_headers is set to "".

/IGN=2: Ignore server response but capture the headers of the response. S_serverResponse is set to "" but S_headers will contain the headers.

/NRED=*maxNumRedirects* Specifies the maximum number of redirects that are allowed. A redirect means that when a certain URL is requested, the server responds telling the client to try a different URL. Most web browsers automatically follow server redirects, up to a certain limit. For security purposes, it is sometimes useful to prevent redirects from being followed at all.

maxNumRedirects is a number between -1 and 1000. To allow infinite redirection, set maxNumRedirects to -1. If you omit the /NRED flag, a moderate value (currently 20, but subject to change in the future) is used.

/O
Specifies that the file is to be overwritten when you use the /FILE flag and the output file already exists. If you omit /O and the file exists, URLRequest returns an error.

/P=*pathName*
Specifies the folder to use for the output file, specified by the /FILE flag, and/or the source data file, specified by the /DFIL flag. pathName is the name of an existing Igor symbolic path.

The /P flag affects both the /FILE and /DFIL flags. If you use both flags and want to use different directories, you must provide a full path for one or both of the /FILE and /DFIL flag parameters.

If the /P flag is used without one or both of the /FILE and /DFIL flags, it is ignored.

/PROX[={*proxyURLStr*, *proxyUserNameStr*, *proxyPassStr*, *proxyOptions*}]

NOTE: This flag is experimental and has not been extensively tested. The behavior of this flag may change in the future, or it may be eliminated entirely.

Designates a proxy server to be used when making the connection. *proxyURLStr* is either the host name or IP address of the proxy server, or a full URL. If a full URL is used, the scheme specifies which kind of proxy is used. Typically the scheme for a proxy server should be either http or socks5. If no scheme is provided, http is assumed. See **URLs** on page IV-267 for more information.

If the proxy server does not require authentication, or if the username and password for the proxy server are specified in proxyURLStr, the optional proxyUserNameStr and proxyPassStr parameters do not need to be provided. The following two examples do the same thing:

```
/PROX={"http://proxy.example.com:800"}
```

```
/PROX={"http://proxy.example.com:800", "", ""}
```

As with other URLs, you can also provide the username and password as part of the URL itself. The following two examples do the same thing:

```
/PROX={"http://proxy.example.com:800", "user", "pass"}
```

```
/PROX={"http://user:pass@proxy.example.com:800"}
```

proxyOptions is optional and for future use. It is currently ignored. If provided, you must set it to 0.

If you use the /PROX flag without any parameters, Igor attempts to get proxy information from the operating system's proxy configuration information. If Igor cannot get any proxy server information from the system, no proxy server is used.

See **Safe Handling of Passwords** on page IV-270 for more information on how to use URLRequest to prevent authentication information, such as passwords, from being accidentally revealed.

/TIME=*timeoutSeconds*
Forces the operation to time out after timeoutSeconds seconds if it has not completed by that time. If this flag is not provided, URLRequest runs until the server has finished sending and receiving data. For simple requests a value of a few seconds is appropriate. However, because uploading and/or downloading large amounts of data may take a long time, if *timeoutSeconds* is too small the request might be prematurely terminated. If you omit the /TIME flag, or if *timeoutSeconds* = 0, there is no timeout.

Regardless of whether or not you include /TIME, you can abort the operation by pressing the **User Abort Key Combinations**.

/V=*diagnosticMode* — This flag is useful only when your goal is to establish a connection with a server and the server's response is not important. All error message codes Controls diagnostic messages printed in the history area of the command window.

/V=0: Do not print any diagnostic messages. This is the default if you omit /V.

/V=1: Prints an error message if a run time error occurs.

/V=2: Prints full debugging information.

/Z[=z] — Suppress error generation. Use this if you want to handle errors yourself.

/Z=0: Do not suppress errors. This is the default if /Z is omitted.

/Z=1: Any errors generated by Igor do not stop execution. If there is an error the error code is stored in V_Flag.

/Z alone has the same effect as /Z=1.

Output Variables

URLRequest sets the following output variables:

`V_flag` — V_flag is zero if the operation succeeds without an error or a non-zero Igor error code if it fails.

Note that "succeeds without an error" does not necessarily mean that the operation performed as you intended. For example, if you attempt to connect to a server that requires a username and password for authentication but you do not provide this information, V_flag is likely to be 0, indicating no error. You need to inspect the value of V_responseCode to ensure that it is what you expect.

`V_responseCode` — Contains the status code provided by the server.

V_responseCode is valid only if V_flag is 0, meaning that no error occured. If V_flag is nonzero, an error occured and V_responseCode will be 0.

Different schemes use different sets of status codes.

A list of http/https status codes and their definitions can be found at:

http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

A list of FTP status codes can be found at:

https://en.wikipedia.org/wiki/List_of_FTP_server_return_codes

`S_serverResponse` — Contains the server's response to the request.

S_serverResponse is set to "" if you use the /FILE flag or the /IGN=1 or /IGN=2 flags.

We recommend that you use the /FILE flag when you expect the size of the server response to be large, such as when downloading a large file.

`S_headers` — Contains a list of all of the headers received as part of the response. Header names are separated from their value by a colon and pairs are separated by a carriage return followed by a line feed (`\r\n`). For schemes other than http and https, S_headers is set to "".

`S_fileName` — Contains the full Igor-style path to the output file if the /FILE flag was used. If there was an error saving to the specified file, or another kind of error, or if the /FILE flag was not used, S_fileName is set to "".

Basic Examples

```
// Retrieve the contents of the WaveMetrics home page.
// Output is stored in S_serverResponse
URLRequest url="http://www.wavemetrics.com"

// Download the Windows Igor6 installer and save it as a file on the desktop.
```

```
NewPath/O Desktop, SpecialDirPath("Desktop", 0, 0, 0)
String theURL = "http://www.wavemetrics.net/Downloads/Win/setupIgor6.exe"
URLRequest/FILE="setupIgor6.exe"/P=Desktop url=theURL

// Download a file to the desktop from an FTP server.
NewPath/O Desktop, SpecialDirPath("Desktop", 0, 0, 0)
String theURL = "ftp://ftp.wavemetrics.com/test/test.htm"
URLRequest/O/FILE="test.htm"/P=Desktop url=theURL

// Upload the same file to the FTP server.
theURL = "ftp://user:pass@ftp.wavemetrics.com/test.htm"
URLRequest/DFIL="test.htm"/P=Desktop method=put, url=theURL
```

Using the File Scheme

URLRequest supports the file scheme in URLs. You must provide the full path to the file as a native Windows-style or UNIX-style path, depending on which platform the code is running.

You must specify the schema as "file:///". Note that you must use three front slash characters (/) between the colon and the full file path (for most URLs you would only use two slashes).

```
URLRequest url="file:///C:\\Data\\Trial1\\control.ibw"      // Works on Windows only
URLRequest url="file:///Users/bob/Data/Trial1/control.ibw"  // Works on Macintosh only
URLRequest url="file:///C:\\Data\\Trial1\\control.ibw"      // Doesn't work
URLRequest url="file:///Users/bob/Data/Trial1/control.ibw"  // Doesn't work
```

The following example shows how to convert a full Igor file path to a native path suitable for use as a file URL:

```
String nativeFilePath
#ifdef WINDOWS
    String igorFilePath = "C:Documents:myFile.txt"
    nativeFilePath = ParseFilePath(5, igorFilePath, "*", 0, 0)
#endif
#ifdef MACINTOSH
    String igorFilePath = "MacintoshHD:Documents:myFile.txt"
    nativeFilePath = ParseFilePath(5, igorFilePath, "/", 0, 0)
    // Remove the leading "/" from nativeFilePath. Otherwise,
    // when we prepend "file:///" below, we'd end up with four
    // slashes, which isn't allowed.
    nativeFilePath = nativeFilePath[1,INF]
#endif
URLRequest url="file:///" + nativeFilePath
```

The HTTP POST Method

Like the HTTP GET method, the HTTP POST method can be used to transmit information to a web server. When using the GET method, all information must be contained within the URL itself. As an example, making a GET request to the URL <http://www.google.com/search?q=Igor+Pro+WaveMetrics> searches Google using the keywords "Igor", "Pro", and "WaveMetrics".

Unlike the GET method, the POST method allows the client to send information to the server that is not contained within the URL itself. This is necessary in many situations, such as when uploading a file or transfering a large amount of data. In addition, because the data contained in POST requests is typically not stored in the log files of web servers, it is more appropriate for sending data that is secure, such as login credentials and form submissions which might contain sensitive information.

When using the POST method, the client must encode its data using one of several methods and must tell the server what type of data is being sent and how it is encoded. The client does this by setting the Content-Type header that is part of the request. The most commonly used content type is "application/x-www-form-urlencoded". Unless you provide your own Content-Type header using the optional headers parameter, URLRequest will set this header for you. This is true regardless of which data source (/DSTR for string or /DFIL for file) you use for the POST.

Here is a simple example that uses the POST method with URL encoded data (the default Content-Type).

```
String theURL = "http://www.example.com/process.php"
String nameData = URLEncode("name") + "=" + URLEncode("Dan P. Ikes, Jr.")
String address = "650 E. Chicago Ave."
String addressData = URLEncode("address") + "=" + URLEncode(address)
String postData = nameData + "&" + addressData
URLRequest/DSTR=postData method=post, url=theURL
```

There are two things to note in this example.

First, both the key name and value string are passed through **URLEncode** so that any special characters can be percent-encoded.

Second, keys and values are separated by an equal sign ("=") and key/value pairs are separated by an ampersand ("&"). These characters are not passed through URLEncode because doing so would cause them to lose their meaning as special characters.

For more information on using the application/x-www-form-urlencoded content type, see http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1

It is also possible to transmit more complicated data using the POST method, such as simulating a form that contains regular text fields as well as a file upload field. To do so you must set the Content-Type header using the headers parameter. You must also build your post data using a multi-part header. Here is an example:

```
String theURL = "http://www.example.com/process.php"
String postData = ""
// Note: The boundary is arbitrary. It needs to be
// unique enough that it is not contained within any
// of the actual data being transmitted in the post.
String boundary = "AaBbCcDd0987"

// name
postData += "--" + boundary + "\r\n"// beginning of this part
postData += "Content-Disposition: form-data; name=\"name\"\r\n"
postData += "\r\n"
postData += "Dan P. Ikes, Jr.\r\n"

// address
postData += "--" + boundary + "\r\n"
postData += "Content-Disposition: form-data; name=\"address\"\r\n"
postData += "\r\n"
postData += "650 E. Chicago Ave.\r\n"

// file
// Open the file we plan to send and store the binary
// contents of the file in a string.
Variable refNum
Open/R/Z/P=Igor refNum as "ReadMe.ihf"
FStatus refNum
Variable fileSize = V_logEOF
String fileContents = ""
fileContents = PadString(fileContents, fileSize, 0)
FBinRead refNum, fileContents
Close refNum
postData += "--" + boundary + "\r\n"
postData += "Content-Disposition: form-data; name=\"file\"; filename=\"ReadMe.ihf\"\r\n"
postData += "Content-Type: application/octet-stream\r\n"
postData += "Content-Transfer-Encoding: binary\r\n"
postData += "\r\n"
postData += fileContents + "\r\n"

postData += "--" + boundary + "--\r\n"      // end of this part

String headers
headers = "Content-Type: multipart/form-data; boundary=" + boundary
URLRequest/DSTR=postData method=post, url=theURL, headers=headers
```

For more information on using the multipart/form-data content type, see http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.2

Handling JSON Data

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is commonly used when interacting with websites - see https://www.json.org/json-en.html. For full read and write JSON support in Igor Pro, we recommend using the JSON XOP, developed by byte physics. You can download it from https://docs.byte-physics.de/json-xop.

See Also

URLEncode, **URLDecode**, **Base64Encode**, **Base64Decode**, **FetchURL**, **BrowseURL**

Network Communication on page IV-267, **URLs** on page IV-267

ValDisplay

`ValDisplay` `[`**`/Z`**`]` ***`ctrlName`*** `[`***`keyword = value`*** `[,` ***`keyword = value`*** `…]]`

The ValDisplay operation creates or modifies the named control that displays a numeric value in the target window. The appearance of the control varies; see the **Examples** section.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the ValDisplay control to be created or changed.

The following keyword=value parameters are supported:

align=*alignment*

Sets the alignment mode of the control. The alignment mode controls the interpretation of the *leftOrRight* parameter to the pos keyword. The align keyword was added in Igor Pro 8.00.

If *alignment*=0 (default), *leftOrRight* specifies the position of the left end of the control and the left end position remains fixed if the control size is changed.

If *alignment*=1, *leftOrRight* specifies the position of the right end of the control and the right end position remains fixed if the control size is changed.

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **Button** and **DefaultGUIControls** for more appearance details.

barBackColor=(*r*,*g*,*b*[,*a*])

Sets the background color under the bar (if any). *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

barBackColor=0

Sets the background color under the bar to the default color, the standard document background color used on the current operating system, which is usually white.

barmisc={*lts*, *valwidth*}

Sets the “limits text size” and the size of the type showing the bar limits. If *lts* is zero, the bar limits are not displayed. Otherwise, *lts* must be between 5 and 100.

valwidth is the “value readout width”. It claims the amount of horizontal space for the numeric part of the display.

If *valwidth* equals or exceeds the control width available to it, the numeric readout uses all the room, and prevents display of any bar.

If *valwidth* is zero, there is no numeric readout, and only the bar is displayed.

valwidth can range from zero to 4000, and it defaults to 1000 (which usually leaves no room for the display bar).

bodyWidth=*width*

Specifies an explicit size for the body (nontitle) portion of a ValDisplay control. By default (bodyWidth=0), the body portion is the amount left over from the specified control width after providing space for the current text of the title portion. If the font, font size, or text of the title changes, then the body portion may grow or shrink. If you supply a bodyWidth>0, then the body is fixed at the size you specify regardless of the body text. This makes it easier to keep a set of controls right aligned when experiments are transferred between Macintosh and Windows, or when the default font is changed.

disable=*d*

Sets user editability of the control.

d=0: Normal.

d=1: Hide.

d=2: Disable user input.
Does not change control appearance because it is read-only.

d=3: Hide and disable the control.
This is useful to disable a control that is also hidden because it is in a hidden tab.

fColor=(*r*,*g*,*b*[,*a*])

Sets the initial color of the title. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is opaque black.

To further change the color of the title text, use escape sequences as described for title=*titleStr*.

font="*fontName*"

Sets the font used to display the value of the variable, e.g., `font="Helvetica"`

format=*formatStr*

Sets the numeric format of the displayed value. The default format is "%g". For a description of *formatStr*, see the **printf** operation.

frame=*f*

Sets frame style:

f=0: Value is unframed.

f=1: Default; value is framed (same as *f*=3).

f=2: Simple box.

f=3: 3D sunken frame.

f=4: 3D raised frame.

f=5: Text well.

fsize=*s*

Sets the size of the type used to display the value in the numeric readout. The default is 12 points.

fstyle=*fs*

fs is a bitwise parameter with each bit controlling one aspect of the font style as follows:

Bit 0: Bold

Bit 1: Italic

Bit 2: Underline

Bit 4: Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*}

Sets the help for the control.

helpStr is limited to 1970 bytes (255 in Igor Pro 8 and before).

You can insert a line break by putting "\r" in a quoted string.

highColor=(*r*,*g*,*b*[,*a*])

Specifies the bar color when the value is greater than *base* in the limits keyword. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

labelBack=(*r*,*g*,*b*[,*a*]) or 0

Specifies the background fill color for labels. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. The default is 0, which uses the window's background color.

limits={*low*,*high*,*base*}

Controls how the value is translated into a graphical representation when the display includes a bar (described fully in **Details**). Defaults are {0,0,0}, which aren't too useful.

limitsColor=(*r*,*g*,*b*[,*a*])

Sets the color of the limits text, if any. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

limitsBackColor=(*r*,*g*,*b*[,*a*])

Sets the background color under the limits text. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

limitsBackColor=0

Sets the background color under the limits text to the default color, the standard document background color used on the current operating system, which is usually white.

lowColor=(*r*,*g*,*b*[,*a*])

Specifies the bar color when the value is less than *base* in the limits keyword. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

mode=*m*

Specifies the type of LED display to use, if any.

m=0: Bar mode (default).

m=1: Oval LED.

m=2: Rectangular LED.

m=3: Bar mode with no fractional part.

m=4: Candy-stripe effect for the bar area to support indefinite-style progress windows. The value is taken to be the phase of the candy stripe. When using value= _NUM:n, n is taken as an increment value so you would normally just use 1. Uses the native platform appearance if the high and low colors are left as default. Note native formats may not fill vertical space. See **Progress Windows** on page IV-156 for an example.

pos={*leftOrRight*,*top*}

Sets the position in **Control Panel Units** of the top/left corner of the control if its alignment mode is 0 or the top/right corner of the control if its alignment mode is 1. See the align keyword above for details.

pos+={*dx*,*dy*}

Offsets the position of the display in **Control Panel Units**.

rename=*newName*

Gives the ValDisplay control a new name.

size={*width*,*height*}

Sets width and height of display in **Control Panel Units**. *width* can range from 10 to 200 units, *height* from 5 to 200 units. Default width is 50, default height is determined by the numeric readout font size.

title=*titleStr*

Sets title of display to the specified string expression. The title appears to the left of the display. If this title is too long, it won't leave enough room to display the bar or even the numeric readout! Defaults to "" (no title).

Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.

value=*valExpr*

Displays the numeric expression *valExpr*. It is *not* a string.

As of version 6.1, you can use the syntax _NUM:num to specify a numeric value without using a dependency.

valueColor=(*r*,*g*,*b*[,*a*])

Sets the color of the value readout text, if any. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

valueBackColor=(*r*,*g*,*b*[,*a*])

Sets the background color under the value readout text. *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**.

valueBackColor=0

Sets the background color under the value readout text to the default color, the standard document background color used on the current operating system, which is usually white.

win=*winName*

Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

zeroColor=(*r*,*g*,*b*[,*a*]) Governs the LED color (in LED mode only). *r*, *g*, *b*, and *a* specify the color and optional opacity as **RGBA Values**. Used in conjunction with the limits keyword such that zeroColor determines one endpoint color when base is between *low* and *high*, or LED color when the value is less than *low*.

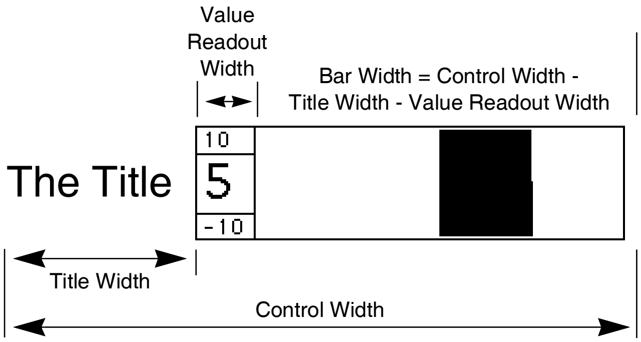
Flags

/Z No error reporting.

Details

The target window must be a graph or panel.

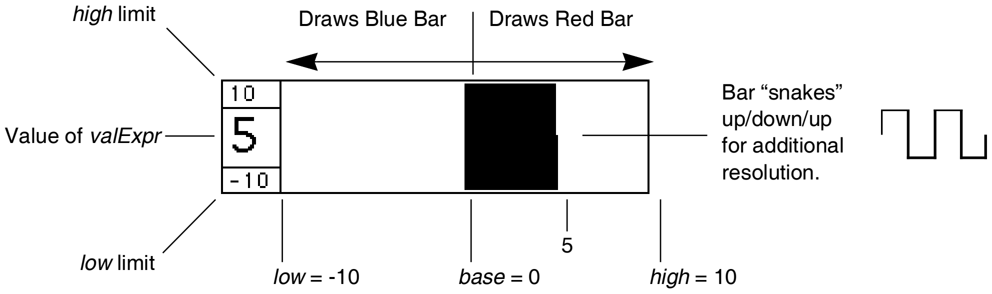
The appearance of the ValDisplay control depends primarily on the *width* and *valwidth* parameters and the width of the title. Space for the individual elements is allocated from left to right, with the title receiving first priority. If the control width hasn't all been used by the title, then the value display gets either *valwidth* **Control Panel Units** of room, or what is left. If the control width hasn't been used up, the bar is displayed in the remaining control width:


Value Readout Width
Bar Width = Control Width - Title Width - Value Readout Width
The Title
10
5
-10
Title Width
Control Width

If you use the bodyWidth keyword, the value readout width and bar width occupy the body width. The total control width is then bodyWidth+title width, and the width from the size keyword is ignored.

The limits values *low*, *high,* and *base* and the value of *valExpr* control how the bar, if any, is drawn. The bar is drawn from a starting position corresponding to the *base* value to an ending position determined by the value of *valExpr*, *low* and *high*. *low* corresponds to the left side of the bar, and *high* corresponds to the right. The position that corresponds to the *base* value is linearly interpolated between *low* and *high*.

For example, with *low*= -10, *high*=10, and *base*= 0, a *valExpr* value of 5 will draw from the center of the bar area (0 is centered between -10 and 10) to the right, halfway from the center to the right of the bar area (5 is halfway from 0 to 10):



The *valExpr* must be executable at any time. The expression is stored and executed when the ValDisplay needs to be updated. However, execution will occur outside the routine that creates the ValDisplay, so you must not use local variables in the expression.

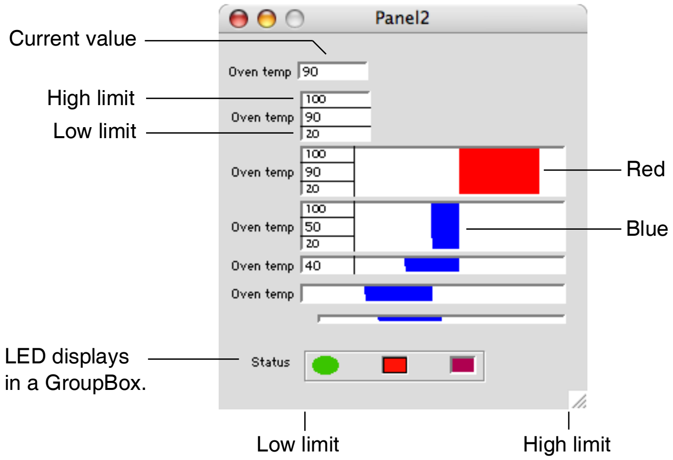
valExpr may be enclosed in quotes and preceded with a # character (see **When Dependencies are Updated** on page IV-233) to defer evaluation of the validity of the numeric expression, which may be needed if the expression references as-yet-nonexistent global variables or user-defined functions:

```
ValDisplay valdisp0 value=notAVar*2      // "unknown name or symbol" error
ValDisplay valdisp0 value=#"notAVar*2"       // still not valid, no error
Variable notAVar=3                       // now valid; ValDisplay works
```

In a ValDisplay, the #"" syntax permits use of a string expression. Normally, the # prefix signifies that the following text must be a literal quoted string. String expressions are evaluated at runtime to obtain the final expression for the ValDisplay. In other words, there is a level of indirection.

Examples

Here is a sampling of the various types of ValDisplay controls available:



You can use a ValDisplay to replace the bar mode with a solid color fill designed to look like an LED. Use the mode keyword with mode=1 to create an oval LED or mode=2 to create a rectangular LED. You can specify different frames with the rectangular LED but only a simple frame is available for the oval mode. Use mode=0 to revert to bar mode.

The color and brightness of the LED depends on the value that the ValDisplay is monitoring combined with the limits={*low*, *high*, *base*) setting, the two color settings used in bar mode along with a third color (zeroColor) that is used only in LED mode. When the value is between *low* and *high*, the color is a linear combination of endpoint colors. If *base* is between *low* and *high*, the endpoint colors are the low color and the zero color, or the zero color and the high color. For values outside the limits, the appropriate limiting color is chosen.

If *base* is less than the *low*, the endpoint colors are the low color and the high color. In this case, if the value is less than *low* the LED takes on the zero color.

You should use the bodyWidth setting in conjunction with LED mode to keep the LED from dramatically changing size or disappearing when the title is changed or if your experiment is moved to a different platform (Macintosh vs PC).

Try the ValDisplay Demo example experiment to see these different modes in action. Choose File→Example Experiments→Feature Demos→ValDisplay Demo.

See Also

See **Creating ValDisplay Controls** on page III-433 for more examples.

printf for an explanation of *formatStr*.

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Control Panel Units on page III-444 for a discussion of the units used for controls.

The **GetUserData** function for retrieving named user data.

The **ControlInfo** operation for information about the control.

Progress Windows on page IV-156 for an example of candy-stripe mode=4.

Variable

`Variable` `[`***`flags`***`]` ***`varName`***`[`**`/N=`*`name`***`][`**`=`*`numExpr`***`][,` ***`varName`***`[`**`/N=`*`name`***`][`**`=`*`numExpr`***`]]`…

The Variable operation creates real or complex variables and gives them the specified name.

Flags

/C	Declares a complex variable.
/D	Obsolete, included only for backward compatibility (see **Details**).
/G	Creates a variable with global scope and overwrites any existing variable.
/N=*name*	Specifies a local name for the global string variable. /N was added in Igor Pro 8.00 and is available in user-defined functions only. See **NVAR Creation** below for details.

Details

The variable is initialized when it is created if you supply the initial value. However, when Variable is used to declare a function parameter, it is an error to attempt to initialize it.

You can create more than one variable at a time by separating the names and optional initializers for multiple variables with a comma.

Numeric variables are double precision. In ancient times, variables could be single or double precision and the /D flag meant double precision. The /D flag is allowed for backward compatibility but is no longer needed and should not be used in new code.

If used in a macro or function the new variable is local to that macro or function unless the /G flag is used. If used on the command line, the new variable is global.

varName can include a data folder path.

NVAR Creation

In a user-defined function, you need a local NVAR reference to access a global string variable. If you use a simple name rather than a path, Igor automatically creates an NVAR:

```
Variable/G nVar1                    // Creates an NVAR named nVar1
```

If you use a path or a $ expression, Igor does not create an automatic NVAR reference. You can explicitly create NVARs like this:

```
Variable/G root:nVar2
NVAR nVar2 = root:nVar2             // Creates an NVAR named nVar2

String path = "root:nVar3"
Variable/G $path
NVAR nVar3 = $path                  // Creates an NVAR named nVar3
```

In Igor Pro 8.00 and later, you can explicitly create an NVAR reference in a user-defined function using the /N flag, like this:

```
Variable/G nVar4/N=nVar4            // Creates an NVAR named nVar4

Variable/G root:nVar5/N=nVar5       // Creates an NVAR named nVar5

String path = "root:nVar6"
Variable/G $path/N=nVar6            // Creates an NVAR named nVar6
```

The name used for the NVAR does not need to be the same as the name of the global variable:

```
Variable/G nVar7/N=nv7              // Creates an NVAR named nv7

Variable/G root:nVar8/N=nv8         // Creates an NVAR named nv8

String path = "root:nVar9"
Variable/G $path/N=nv9              // Creates an NVAR named nv9
```

Examples

To initialize a complex variable, use the **cmplx** function. For example:

```
Variable/C cv1 = cmplx(1,2)
```

This sets the real part of cv1 to 1 and the imaginary part to 2.

See Also

Numeric Variables on page II-104, **Accessing Global Variables and Waves** on page IV-65

Variance

`Variance(`*`inWave`*` [, `*`x1`*`, `*`x2`*` ] )`

Returns the variance of the real-valued *inWave*. The function ignores NaN and INF values in *inWave*.

Parameters

inWave is expected to be a real-valued numeric wave. If *inWave* is a complex or text wave, Variance returns NaN.

x1 and *x2* specify a range in *inWave* over which the variance is to be calculated. They are used only to locate the points nearest to x=*x1* and x=*x2* . The variance is then calculated over that range of points. The order of *x1* and *x2* is immaterial.

If omitted, *x1* and *x2* default to -∞ and +∞ respectively and the variance is calculated for the entire wave.

Details

The variance is defined by

$$\mathrm{var} = \frac{\sum_{i=1}^{n}\left(x_i - \overline{x}\right)^2}{n-1}$$

where

$$\overline{x} = \frac{\sum_{i=1}^{n} X_i}{n}.$$

Examples

```
Make/O/N=5 test = p
SetScale/P x, 0, .1, test

// Print variance of entire wave
Print Variance(test)

// Print variance from x=0 to x=.2
Print Variance(test, 0, .2)

// Print variance for points 1 through 3
Variable x1=pnt2x(test, 1)
Variable x2=pnt2x(test, 3)
Print Variance(test, x1, x2)
```

See Also

mean, **median**, **WaveStats**, **APMath**

VariableList

`VariableList(`*`matchStr`*`, `*`separatorStr`*`, `*`variableTypeCode`*` [, `*`dfr`*` ])`

The VariableList function returns a string containing a list of the names of global variables selected based on the *matchStr* and *variableTypeCode* parameters. The variables listed are all in the current data folder or the data folder specified by *dfr*.

Details

For a variable name to appear in the output string, it must match *matchStr* and also must fit the requirements of *variableTypeCode*. *separatorStr* is appended to each variable name as the output string is generated.

The name of each variable is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all variable names.

"xyz"	Matches variable name xyz only.
"*xyz"	Matches variable names which end with xyz.
"xyz*"	Matches variable names which begin with xyz.
"*xyz*"	Matches variable names which contain xyz.
"abc*xyz"	Matches variable names which begin with abc and end with xyz.

matchStr may begin with the ! character to return items that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches variable names which *do not* end with xyz.

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

variableTypeCode is used to further qualify the variable. The variable name goes into the output string only if it passes the match test and its type is compatible with *variableTypeCode*. *variableTypeCode* is any one of:

2:	System variables (K0, K1 . . .)
4:	Scalar variables
5:	Complex variables

dfr is an optional data folder reference: a data folder name, an absolute or relative data folder path, or a reference returned by, for example, **GetDataFolderDFR**.

Examples

`VariableList("*",";",4)`	Returns a list of all scalar variables.
`VariableList("!V_*", ";",5)`	Returns a list of all complex variables except those whose names begin with “V_”.
`VariableList("*",";",4,root:MyData)`	Returns a list of all scalar variables in the root:MyData data folder.

See Also

See the **StringList** and **WaveList** functions.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

vcsr

`vcsr(`*`cursorName`*` [, `*`graphNameStr`*`])`

The vcsr function returns the Y (vertical) value of the point which the specified cursor (A through J) is attached to in the top (or named) graph.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The result is computed from the coordinate system of the graph’s Y axis. The Y axis used is the one used to display the wave on which the cursor is placed.

See Also

The **hcsr**, **pcsr**, **qcsr**, **xcsr**, and **zcsr** functions.

Programming With Cursors on page II-321.

version

#pragma version = *versNum*

In the File Information dialog, `#pragma version=`*versNum* provides file version information that is displayed next to the file name in the dialog. This line must not be indented and must appear in the first fifty lines of the file. See **Procedure File Version Information** on page IV-166.

See Also

The **The version Pragma** on page IV-54, **Procedure File Version Information** on page IV-166, the **IgorInfo** function, and #**pragma**.

VoigtFunc

VoigtFunc(X,Y)

The VoigtFunc function computes the Voigt function using an approximation that has, as described by the author, "accuracy is typically at at least 13 significant digits".

VoigtFunc returns values from a normalized Voigt peak centered at X=0 for the given value of X. The X input is a normalized distance from the peak center:

$$X = \sqrt{\ln(2)}\,\frac{\nu - \nu_0}{\gamma_g}$$

where γ_g is the Gaussian component half-width, and $\nu - \nu_0$ is the distance from the peak center.

The parameter Y is the shape parameter: when Y is zero, the peak is pure Gaussian. When Y approaches infinity, the shape becomes pure Lorentzian. When Y is sqrt(ln(2)), the mix is half-and-half.

VoigtFunc was added in Igor Pro 7.00. The approximation used to compute it was changed Igor 8.00 for greater accuracy.

Details

The VoigtFunc function returns values from a normalized peak that can be used as the basis for user-defined fitting functions. The function is used as the basis for the built-in Voigt fitting function and the **VoigtPeak** function.

VoigtFunc Curve Fitting Example

Here is an example of a user-defined fitting function built on VoigtFunc:

```
Constant sqrtln2=0.832554611157698          // sqrt(ln(2))
Constant sqrtln2pi=0.469718639349826        // sqrt(ln(2)/pi)

Function MyVoigtFit(w,xx) : FitFunc
    Wave w
    Variable xx

    //CurveFitDialog/ These comments were created by the Curve Fitting dialog.
    //CurveFitDialog/ Equation:
    //CurveFitDialog/ Variable ratio = sqrtln2/gw
    //CurveFitDialog/ Variable xprime = ratio*(xx-x0)
    //CurveFitDialog/ Variable voigtY = ratio*shape
    //CurveFitDialog/ f(xx) = y0 + area*sqrtln2pi*VoigtFunc(xprime, voigtY)
    //CurveFitDialog/ End of Equation
    //CurveFitDialog/ Independent Variables 1
    //CurveFitDialog/ xx
    //CurveFitDialog/ Coefficients 5
    //CurveFitDialog/ w[0] = y0
    //CurveFitDialog/ w[1] = area
    //CurveFitDialog/ w[2] = x0
    //CurveFitDialog/ w[3] = gw (FWHM)
    //CurveFitDialog/ w[4] = shape (Lw/Gw)

    Variable voigtX = 2*sqrtln2*(xx-w[2])/w[3]
    Variable voigtY = sqrtln2*w[4]
    return w[0] + (w[1]/w[3])*2*sqrtln2pi*VoigtFunc(voigtX, voigtY)
End
```

Parameter w[0] sets the vertical offset, w[1] sets the peak area, w[2] sets the location of the peak, w[3] gives the Gaussian component's full width at half max and w[4] is the ratio of the Lorentzian width to the Gaussian width.

After the fit, assuming you used a coefficient wave named voigtCoefs, you can calculate the width of the full Voigt peak as follows:

```
Variable/G wl = voigtCoefs[4]*voigtCoefs[3]
Variable/G wg = voigtCoefs[3]
Variable wv = wl/2 + sqrt( wl^2/4 + wg^2)
```

References

The code used to compute the VoigtFunc was written by Steven G. Johnson of MIT. You can learn more about it at http://ab-initio.mit.edu/Faddeeva.

See Also

VoigtPeak, **Faddeeva**, **Built-in Curve Fitting Functions** on page III-206

VoigtPeak

VoigtPeak(*w*,*x*)

The VoigtPeak function returns a value from a Voigt peak shape defined by coefficients in wave *w* at location *x*. It was added in Igor Pro 8.00.

The Voigt peak shape is defined as a convolution of a Gaussian and a Lorentzian peak. We use an approximation that is described by the author as having "accuracy typically at least 13 significant digits". This function is equivalent to the built-in Voigt fitting function. See **Built-in Curve Fitting Functions** on page III-206.

The coefficients are:

w[0]:	Vertical offset.
w[1]:	Peak area.
w[2]:	Peak center location.
w[3]:	Gaussian component width expressed as Full Width at Half Max (FWHM).
w[4]:	Ratio of Lorentzian component width to the Gaussian component width. For w[4]=0, the peak shape is purely Gaussian, as w[4] → ∞, the peak shape become purely Lorentzian. A value of 1 results in Gaussian and Lorentzian components of equal width.

References

The code used to compute VoigtPeak was written by Steven G. Johnson of MIT. You can learn more about it at http://ab-initio.mit.edu/Faddeeva.

See Also

VoigtFunc, **Faddeeva**, **Built-in Curve Fitting Functions** on page III-206.

WAVE

WAVE [**/C**][**/T**][**/WAVE**][**/DF**][**/Z**][**/ZZ**][**/SDFR**=*dfr*] ***localName*** [**=*pathToWave***][, ***localName1*** [**=*pathToWave1***]]…

WAVE is a declaration that identifies the nature of a user-defined function parameter or creates a local reference to a wave accessed in the body of a user-defined function.

The optional parameter /SDFR flag and *pathToWave* parameter are used only in the body of a function, not in a parameter declaration.

The WAVE declaration is required when you use a wave in an assignment statement in a function. At compile time, the WAVE statement specifies that the local name references a wave. At runtime, it makes the connection between the local name and the actual wave. For this connection to be made, the wave must exist when the WAVE statement is executed.

The WAVE declaration is also required if you use a wave name as a parameter to an operation or function if **rtGlobals**=3 is in effect which is the usual case.

When *localName* is the same as the global wave name and you want to reference a wave in the current data folder, you can omit the *pathToWave*.

pathToWave can be a full literal path (e.g., root:FolderA:wave0), a partial literal path (e.g., :FolderA:wave0) or $ followed by string variable containing a computed path (see **Converting a String into a Reference Using $** on page IV-62).

You can also use a data folder reference or the /SDFR flag to specify the location of the wave if it is not in the current data folder. See **Data Folder References** on page IV-78 and **The /SDFR Flag** on page IV-80 for details.

If the wave may not exist at runtime, use the /Z flag and call **WaveExists** before accessing the wave. The /Z flag prevents Igor from flagging a missing wave as an error and dropping into the debugger. For example:

```
WAVE/Z wv=<pathToPossiblyMissingWave>
if( WaveExists(wv) )
    <do something with wv>
endif
```

In Igor Pro 9.00 and later, you can avoid the runtime lookup of localName in the current data folder by including the /ZZ flag. For example:

```
Function CallingRoutine()
    WAVE/ZZ w
    PassByRefRoutine(w)
    Print w
End

Function PassByRefRoutine(WAVE& wr)
    WAVE wr = NewFreeWave(2,2)
End
```

Without the /ZZ flag, at runtime Igor would attempt to find a wave named w in the current data folder. This is unnecessary in this case since PassByRefRoutine sets the w variable.

Flags

/C	Complex wave
/T	Text wave
/WAVE	Wave reference wave
/DF	Data folder reference wave
/SDFR=*dfr*	Specifies the source data folder. See **The /SDFR Flag** on page IV-80 for details.
/Z	Ignores wave reference checking failures
/ZZ	Ignores wave reference checking failures and prevents wave lookup

See Also

WaveExists function.

WAVE Reference Type Flags on page IV-74 for additional wave type flags and information.

Accessing Global Variables and Waves on page IV-65.

Accessing Waves in Functions on page IV-82.

Converting a String into a Reference Using $ on page IV-62.

WAVEClear

`WAVEClear` ***`localName`*** `[,` ***`localName1`*** …]

The WAVEClear operation clears out a wave reference variable. WAVEClear is equivalent to `WAVE/Z localName= $""`.

Details

Use WAVEClear to avoid unexpected results from certain operations such as **Duplicate** or **Concatenate**, which will reuse the contents of a WAVE reference variable and may not generate the wave in the desired data folder or with the desired name.

WAVEClear ensures that memory is deallocated after waves are killed as in this example:

```
Function foo()
   Make wave1
   FunctionThatKillsWave1()
   WAVEClear wave1
   AnotherFunction()
End
```

Although memory used for wave1 will be deallocated when `foo` returns, that memory will not be automatically released while the function executes because the WAVE variable still contains a reference to the wave. In this example, WAVEClear deallocates that memory before `AnotherFunction` executes.

You can also use WAVEClear before passing a data folder to preemptive threads using **ThreadGroupPutDF**.

See Also

Accessing Waves in Functions on page IV-82, **Wave Reference Counting** on page IV-205, and **ThreadSafe Functions and Multitasking** on page IV-329.

WaveCRC

`WaveCRC(`*`inCRC`*`, `*`waveName`*` [, `*`checkHeader`*`])`

The WaveCRC function returns a 32-bit cyclic redundancy check value of the bytes in the named wave starting with *inCRC*.

Pass 0 for *inCRC* the first time you call WaveCRC for a particular stream of bytes as represented by the wave data.

Pass the last-returned value from WaveCRC for *inCRC* if you are creating a CRC value for a given stream of bytes through multiple calls to WaveCRC.

waveName may be a numeric or text wave.

The optional *checkHeader* parameter determines how much of the wave is checked:

checkHeader	**What It Does**
0	Check only the wave data (default).
1	Check only the internal binary header.
2	Check both.

Details

Polynomial used is:

x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x+1

See crc32.c in the public domain source code for zlib for more information.

See Also

StringCRC, **Hash**, **WaveHash**, **WaveModCount**

WaveDataToString

`WaveDataToString(`*`wave`*`)`

The WaveDataToString function returns a string containing an exact copy of the data from the numeric wave wave. The returned string may contain null bytes.

The WaveDataToString function was added in Igor Pro 9.00.

If *wave* has 0 points, WaveDataToStrings return an empty string. If *wave* is an invalid wave reference or a non-numeric wave, the WaveDataToStrings returns a null string.

If the size of the data in the wave is larger than about 2^31 bytes, the function returns a null string because Igor strings are limited to about 2GB in size. You can determine the size of the wave's data by multiplying the number of points by the size per point (double=8 bytes/point, float=4 bytes/point, etc.). The per-point size is doubled for complex waves.

Parameters

wave is a wave reference to a numeric wave.

Details

Only the wave's data is present in the returned string. Other information, such as scaling, dimension labels, etc., is not included. You may need to use the Redimension operation to change the type of the wave or to perform an endian swap before you use WaveDataToString.

While Igor strings can contain embeded nulls, some parts of Igor are not prepared to handle them. For example printing a string with a null will only print the part of the string before the null. For more information, see **Embedded Nulls in Literal Strings** on page IV-16.

Example

```
Function WaveDataToStringDemo1()
   Make/FREE/B/U w1 = {49, 50, 51}

   // This requires Igor 9.00
   String s1 = WaveDataToString(w1)
   Print s1                // Prints 123

   // This approach works in older versions of Igor
   Variable np = numpnts(w1)
   String s2 = ""
   s2 = PadString(s2, np, 0)
   Variable n
   For (n=0; n < np; n++)
      s2[n,n] = num2char(w1[n])
   EndFor
   Print s2                // Prints 123
End

// Round trip using WaveDataToString and StringToUnsignedByteWave
Function WaveDataToStringDemo2()
   Make/FREE/D w2 = {1}
   String w2Str = WaveDataToString(w2)
   Print w2Str             // Prints nothing because w2Str contains leading null bytes
   Print strlen(w2Str)
   WAVE/B/U w2ByteWave = StringToUnsignedByteWave(w2Str)
   Print w2ByteWave        // Prints {0,0,0,0,0,0,240,63}

   // Redimension the byte wave to a double precision floating point wave
   Redimension/E=1/D/N=1 w2ByteWave
   Print w2ByteWave        // Prints {1}
End

// Generate mixed-case random letters
Function WaveDataToStringDemo3()
   Make/O/FREE/N=(1e3) letters
   MultiThread letters = trunc(abs(enoise(52)))

   // 0-25 uppercase, 26-51 become lowercase
   MultiThread letters = letters[p] < 26 ? letters[p] + 65 : letters[p] + 71
   Redimension/B/U letters

   // Create a string with all the letters.
   String lettersStr = WaveDataToString(letters)
   Print lettersStr[0,100]
End
```

See Also

StringToUnsignedByteWave, **wfprintf**, **Working With Binary String Data** on page IV-175

WaveDims

WaveDims(*wave*)

The WaveDims function returns the number of dimensions used by *wave*.

Returns zero if wave reference is null. See **WaveExists** for a discussion of null wave references.

Also returns zero if wave has zero rows. A matrix will return 2.

WaveExists

`WaveExists(`*`wave`*`)`

The WaveExists function returns one if wave reference is valid, or zero if the wave reference is null. For example if, in a user function, you have:

```
Wave w= $"no such wave"
```

then `WaveExists(w)` will return zero.

Details

If wave is not a valid wave reference, WaveExists also checks whether wave itself is the name of an existing wave when executing the function. WaveExists should be used in functions only. In macros, use the exists function instead.

See Also

WAVE, **exists**, **NVAR_Exists**, **SVAR_Exists**, and **Accessing Global Variables and Waves** on page IV-65.

WaveHash

`WaveHash(`*`wave`*`, `*`method`*`)`

The WaveHash function returns a cryptographic hash of the bytes of the wave's data using the specified method. All contents of the wave's header are ignored.

Parameters

wave may reference a numeric or text wave.

method is a number indicating the hash algorithm to use:

1	SHA-256 (SHA-2)
2	MD4
3	MD5
4	SHA-1
5	SHA-224 (SHA-2)
6	SHA-384 (SHA-2)
7	SHA-512 (SHA-2)

See Also

WaveModCount, **WaveCRC**, **StringCRC**, **Hash**

WaveInfo

`WaveInfo(`*`waveName`*`, 0)`

The WaveInfo function returns a string containing a semicolon-separated list of information about the named wave.

The second parameter is reserved for future use and must be zero.

Details

The result string contains a list of keyword-value pairs. Each pair consists of a keyword, a colon, the value text, and a semicolon.

Here are the keywords:

Always pass 0 as the second input parameter. In future versions of Igor, this parameter may request other kinds of information to be returned.

A null wave reference returns a zero-length string. This might be encountered, for instance, when using **WaveRefIndexedDFR** in a loop to act on all waves in a data folder, and the loop has incremented beyond the highest valid index.

Keyword	Information Following Keyword
DUNITS	The wave’s data units.
FULLSCALE	Three numbers indicating whether the wave has any data full scale information, and the min and max data full scale values. The format of the FULLSCALE description is: `FULLSCALE:`*`validFS`*`,`*`minFS`*`,`*`maxFS`*`;` *validFS* is 1 if *minFS* and *maxFS* have been set via a `SetScale d` command; otherwise it is 0.
LOCK	Reads back the value set by **SetWaveLock**.
MODIFIED	1 if the wave has been modified since the experiment was last saved, else 0.
MODTIME	The date and time that the wave was last modified in seconds since January 1, 1904.
NUMTYPE	A number denoting the data type of the wave. For text waves this is 0. For wave reference waves it is 16384. For data folder reference waves it is 256. For numeric waves it is one of the following: 1: Complex, added to one of the following 2: 32-bit (single precision) floating point 4: 64-bit (double precision) floating point 8: 8-bit signed integer 16: 16-bit signed integer 32: 32-bit signed integer 128: 64-bit signed integer 64: Unsigned, added to 8, 16, 32 or 128 if wave is unsigned For example, the number denoting a complex double precision wave is 5 (i.e., 1+4).
PATH	The name of the symbolic path in which the wave file is stored (e.g., PATH:home;) or nothing if there is no path for the wave (PATH:;).
SIZEINBYTES	The total size of the wave in bytes. This includes the wave’s header, data, note, dimension labels, and unit strings. This keyword was added in Igor Pro 7.00.
XUNITS	The wave’s X units.

Examples

```
Make/O wave1;SetScale x,0,1,"dyn",wave1;SetScale y,3,20,"v",wave1
String info = WaveInfo(wave1,0)
Print NumberByKey("NUMTYPE", info)          // Prints 2
Print StringByKey("DUNITS", info)           // Prints "v"
```

See Also

The functions and operations listed under “About Waves” categories in the Command Help tab of the Igor Help Browser; among them are **CreationDate**, **ModDate**, **WaveModCount**, **WaveType**, **note**, and **numpnts**.

NumberByKey and **StringByKey** functions for parsing the returned keyword list.

WaveInfo lacks information about multidimensional waves. Individual functions are provided to return dimension-related information: **DimDelta**, **DimOffset**, **DimSize**, **WaveUnits**, and **GetDimLabel**.

WaveList

`WaveList(`*`matchStr`*`, `*`separatorStr`*`, `*`optionsStr`*` [, `*`dfr`*` ])`

The WaveList function returns a string containing a list of wave names selected from the current data folder or the data folder specified by *dfr* based on *matchStr* and *optionsStr* parameters. The *dfr* parameter requires Igor Pro 9.00 or later.

See **Details** for information on listing waves in graphs, and for references to newer, data folder-aware functions.

Details

For a wave name to appear in the output string, it must match *matchStr* and also must fit the requirements of *optionsStr* and it must be in the current data folder. *separatorStr* is appended to each wave name as the output string is generated.

The name of each wave is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything.

For example:

`"*"`	Matches all wave names in current data folder.
`"xyz"`	Matches wave name xyz only, if xyz is in the current data folder.
`"*xyz"`	Matches wave names which end with xyz and are in the current data folder.
`"xyz*"`	Matches wave names which begin with xyz and are in the current data folder.
`"*xyz*"`	Matches wave names which contain xyz and are in the current data folder.
`"abc*xyz"`	Matches wave names which begin with abc and end with xyz and are in the current data folder.

matchStr may begin with the ! character to return items that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches wave names which do not end with xyz.

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

optionsStr is used to further qualify the wave.

Use `""` to accept all waves in the current data folder that are permitted by *matchStr*.

Set *optionsStr* to one or more of the following comma-separated keyword-value pairs:

optionsStr	**Selection Criteria**
`"BYTE:0"` *or* `"BYTE:1"`	Waves that are not 8-bit integer (if 0) or only waves that are 8-bit integer (if 1).
`"CMPLX:0"` *or* `"CMPLX:1"`	Waves that are not complex (if 0) or only waves that are complex (if 1).
`"DIMS:`*`numberOfDims`*`"`	All waves in current data folder that have *numberOfDims* dimensions. This is the number of dimensions reported by **WaveDims**. Use `"DIMS:0"` for all waves having no points (`numpnts(w)==0`). Use "DIMS:1" for graph traces (or one of the X, Y, and Z waves of a contour plot). Use "DIMS:2" for false color and indexed color images (see **Indexed Color Details** on page II-400). Use "DIMS:3" for direct color images (see **Direct Color Details** on page II-401).
`"DF:0"` *or* `"DF:1"`	Consider waves that are not data folder reference waves (if 0) or only waves that are data folder reference waves (if 1). You can create waves that contain data folder references using the **Make** /DF flag.

optionsStr	**Selection Criteria**
`"DP:0"` *or* `"DP:1"`	Waves that are not double precision floating point (if 0) or only waves that are double precision floating point (if 1).
`"INT64:0"` *or* `"INT64:1"`	Consider waves that are not 64-bit integer (if 0) or only waves that are 64-bit integer (if 1). 64-bit integer waves are supported in Igor7 and later.
`"INTEGER:0"` *or* `"INTEGER:1"`	Waves that are not 32-bit integer (if 0) or only waves that are 32-bit integer (if 1).
`"MAXCHUNKS:`*`max`*`"`	Waves having no more than *max* chunks.
`"MAXCOLS:`*`max`*`"`	Waves having no more than *max* columns.
`"MAXLAYERS:`*`max`*`"`	Waves having no more than *max* layers.
`"MAXROWS:`*`max`*`"`	Waves having no more than *max* rows.
`"MINCHUNKS:`*`min`*`"`	Waves having at least *min* chunks.
`"MINCOLS:`*`min`*`"`	Waves having at least *min* columns.
`"MINLAYERS:`*`min`*`"`	Waves having at least *min* layers.
`"MINROWS:`*`min`*`"`	Waves having at least *min* rows.
`"SP:0"` *or* `"SP:1"`	Waves that are not single precision floating point (if 0) or only waves that are single precision floating point (if 1).
`"TEXT:0"` *or* `"TEXT:1"`	Waves that are not text (if 0) or only waves that are text (if 1).
`"UNSIGNED:0"` *or* `"UNSIGNED:1"`	Waves that are not unsigned integer (if 0) or only waves that are unsigned integer (if 1).
`"WAVE:0"` *or* `"WAVE:1"`	Consider waves that do not contain wave references (if 0) or only waves that contain wave references (if 1). You can create waves that contain wave references using the **Make** /WAVE flag.
`"WIN:"`	All waves in the current or specified data folder that are displayed in the top graph or table. The WIN option is not threadsafe.
`"WIN:`*`windowName`*`"`	All waves in the current or specified data folder that are displayed in the named table or graph window or subwindow. The WIN option is not threadsafe.
`"WORD:0"` *or* `"WORD:1"`	Waves that are not 16-bit integer (if 0) or only waves that are 16-bit integer (if 1).

You can specify more than one option by separating the options with a comma. See the **Examples**.

Note: Even when *optionsStr* is used to list waves used in a graph or table, the waves must be in the current data folder.

Note: In addition to waves displayed as normal graph traces, WaveList will list matrix waves used with **AppendImage** or NewImage and the X, Y, and Z waves used with **AppendXYZContour**.

Note: Individual contour traces are not listed because they have no corresponding waves. See **Contour Traces** on page II-370.

There are several functions that are more useful for listing waves in graphs and tables.

WaveList with `WIN:`*`windowName`* gives only the names of the waves in the graph or table and does not include the data folder for each wave. If you need to know what data folder the waves are in, use **WaveRefIndexed** to get the wave itself and then if needed use **GetWavesDataFolder** to get the path.

When identifying a subwindow with `WIN:`*`windowName`*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

To list the actual waves used in a graph, or to distinguish two or more instances of the same named wave in a graph, use **TraceNameList**. This function can be used in conjunction with **TraceNameToWaveRef**, and **XWaveRefFromTrace**.

Use **ContourNameList** to list contour plots in a given window and **ContourNameToWaveRef** to access the waves used to generate the contour plot.

To list the contour traces (that is, the contour lines themselves) use **TraceNameList** with the appropriate option.

Use **ImageNameList** to list images in a given window and **ImageNameToWaveRef** to access the waves used to generate the images.

Processing Lists of Waves

Contrary to what you might expect, you can *not* use the output of WaveList directly with operations that have a list of waves as their parameters. See **Processing Lists of Waves** on page IV-198 for ways of dealing with this.

Examples

```
// Returns a list of all waves in the current data folder.
WaveList("*",";","")
```

```
// Returns a list of all waves in the current data folder and displayed in the top table or graph.
WaveList("*", ";","WIN:")
```

```
// Returns a list of waves in the current data folder whose names
// end in "_bkg" and which are displayed in Graph0 as 1D traces.
WaveList("*_bkg", ";", "WIN:Graph0")
```

```
// Returns a list of waves in the current data folder whose names do not
// end in "X" and which are displayed in Graph0 as 1D traces or as one
// of the X, Y, and Z waves of an AppendXYZContour plot.
WaveList("!*X", ";", "WIN:Graph0,DIMS:1")
```

```
// Returns a list of waves in the root:Packages:MyPackage data folder
WaveList("*", ";", "", root:Packages:MyPackage)
```

See Also

Chapter II-6, **Multidimensional Waves**.

Execute, **ContourNameList**, **ImageNameList**, **TraceNameList**, and **WaveRefIndexed**.

WaveMax

WaveMax(*waveName* [, *x1*, *x2*])

The WaveMax function returns the maximum value in the wave for points between x=*x1* to x=*x2*, inclusive.

Details

If *x1* and *x2* are not specified, they default to -inf and +inf, respectively.

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with `pnt2x(`*`waveName`*`,`*`pointNumber1`*`)`, and a similar expression for *x2*.

If the points nearest to *x1* or *x2* are not within the point range of 0 to numpnts(*waveName*)-1, WaveMax limits them to the nearest of point 0 or point numpnts(*waveName*)-1.

NaN values in the wave are ignored.

See Also

WaveMin, **WaveMinAndMax**, **WaveStats**

WaveMeanStdv

WaveMeanStdv *srcWave binSizeWave*

The WaveMeanStdv operation calculates the standard deviation of the means for the specified bin distribution saving the result in the wave W_MeanStdv.

For each entry in *binSizeWave*, *srcWave* is divided into the specified number of bins. Values in each bin are averaged and then the mean and standard deviation of the averages (among all bins) are calculated. The value of the standard deviation of the bin averages divided by the mean is then stored in W_MeanStdv corresponding to the bin size entry in *binSizeWave*.

All entries in *binSizeWave* must be positive integers.

Details

When the number of points in *srcWave* does not divide evenly into the bin size entry from *binSizeWave*, the last bin will have a smaller number of data points. In order not to skew the results the values corresponding to the last bin will be dropped. If your data set is small compared to the bin size you might want to pad *srcWave* with additional values (e.g., duplicate values from the beginning of the wave).

This operation does not support NaNs. If you get a NaN as an entry in the output wave then there is either a NaN in *srcWave* or something is wrong with the calculation for that entry.

WaveMin

WaveMin(*waveName* [, *x1*, *x2*])

The WaveMin function returns the minimum value in the wave for points between x=*x1* to x=*x2*, inclusive.

Details

If *x1* and *x2* are not specified, they default to -inf and +inf, respectively.

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with `pnt2x(`*`waveName`*`,`*`pointNumber1`*`)`, and a similar expression for *x2*.

If the points nearest to *x1* or *x2* are not within the point range of 0 to numpnts(*waveName*)-1, WaveMin limits them to the nearest of point 0 or point numpnts(*waveName*)-1.

NaN values in the wave are ignored.

See Also

WaveMax, **WaveMinAndMax**, **WaveStats**

WaveMinAndMax

WaveMinAndMax(wave [, *x1*, *x2*])

The WaveMinAndMax function returns the minimum and maximum values in the wave for points between x=*x1* to x=*x2*, inclusive.

WaveMinAndMax must be called from a function, not from the command line, because it uses multiple return syntax as shown in the example below.

WaveMinAndMax was added in Igor Pro 9.00.

Details

If x1 and *x2* are omitted, they default to -inf and +inf.

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with "pnt2x(wave,pointNumber1)", and a similar expression for *x2*. The resulting point numbers are clipped to the range 0..n where n is the numpnts(wave)-1.

NaN values in the wave are ignored.

Example

```
Function DemoWaveMinAndMax()
   Make/FREE wave0 = p
   wave0[0] = NaN                       // NaN values are ignored
   SetScale/P x, 0, 0.1, "s", wave0
   double minValue, maxValue

   [minValue, maxValue] = WaveMinAndMax(wave0)
   Printf "Entire wave: min=%g, max=%g\r", minValue, maxValue

   [minValue, maxValue] = WaveMinAndMax(wave0, 5, 10)
   Printf "From x=5 to x=10: min=%g, max=%g\r", minValue, maxValue
End
```

See Also

WaveMin, **WaveMax**, **WaveStats**

WaveModCount

WaveModCount(*wave*)

The WaveModCount function returns a value that can be used to tell if a global wave has been changed between one call to WaveModCount and another.

WaveModCount was added in Igor Pro 8.00.

The exact value returned by WaveModCount has no significance. The only use for it is to compare the values returned by two calls to WaveModCount. If they are the different, the wave was changed in the interim.

The wave mod count for free and thread-local waves is undefined, so WaveModCount should only be used with global waves in the data hierarchy of the main thread.

A wave's mod count changes when the wave's data or properties, such as scaling, note, and dimensionality, are set. The mod count changes even if the new data or property values are the same as the old. For example, executing:

```
wave1 += 0
```

causes the mod count to change even though the data itself was not actually changed.

Examples

```
Make/O wave1 = 5
Variable waveModCount1, waveModCount2
waveModCount1 = WaveModCount(wave1);
wave1 += 1                              // Modify wave1
waveModCount2 = WaveModCount(wave1);
if (waveModCount2 != waveModCount1)
   Print "Wave has changed"
endif
```

See Also

WaveInfo, **ModDate**

WaveName

WaveName(*winNameStr*, *index*, *type*)

The WaveName function returns a string containing the name of the *index*th wave of the specified *type* in the named window or subwindow.

Parameters

winNameStr can be "" to refer to the top graph or table.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

WaveName works on waves displayed in a graph, in a table or on the list of waves in the current data folder. If the window is a table, WaveName returns the column name (e.g., “wave0.d”), rather than the name of the wave itself (e.g., “wave0”).

For most uses, we recommend that you use **WaveRefIndexed** or **WaveRefIndexedDFR** instead of WaveName. WaveName returns a string containing the wave name only, with no data folder path qualifying it. Thus, you may get erroneous results if the wave referred to in the graph has the same name as a different wave in the current data folder. Likewise, if the named wave resides in a data folder that is not the current data folder, you will not be able to refer to the named wave. Use **WaveRefIndexedDFR** instead.

winNameStr is a string expression containing the name of a graph or table or an empty string (""). If the string is empty and *type* is 4 then WaveName works on the list of all waves in the current data folder. If the string is empty and the type parameter is not 4 then WaveName works on the top graph or table.

index starts from zero.

type is a number from 1 to 4. When type is 4 and *winNameStr* is "", WaveName works on the list of all waves in the current data folder.

For graph windows, *type* is 1 for y waves, 2 for x waves, 3 for either y or x waves.

For table windows, *type* is 1 for data columns, 2 for index or dimension label columns, 3 for either data or index or dimension label columns.

WaveName returns an empty string ("") if there is no wave matching the parameters.

Examples

```
WaveName("",0,4)    // Returns name first wave current data folder.
WaveName("",0,1)    // Returns name of first Y wave in the top graph.
WaveName("Graph0",1,2)     // Returns name of second X wave in Graph0.
WaveName("Table0",1,3)     // Returns name of second column in Table0.
```

WaveRefIndexed

`WaveRefIndexed(`*`winNameStr`*`, `*`index`*`, `*`type`*`)`

The WaveRefIndexed function returns a wave reference to the *index*th wave of the specified *type* in the named window or subwindow.

To iterate through the waves in a data folder, use **WaveRefIndexedDFR** instead of WaveRefIndexed.

Parameters

winNameStr can be "" to refer to the top graph or table window or the current data folder.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

WaveRefIndexed is analogous to WaveName but works better with data folders. We recommend that you use it instead of WaveName.

winNameStr is a string expression containing the name of a graph or table or an empty string (""). If the string is empty and *type* is 4 then WaveRefIndexed works on Igor's list of all waves in the current data folder. If the string is empty and the type parameter is not 4 then WaveRefIndexed works on the top graph or table.

index starts from zero.

type is a number from 1 to 4. When type is 4 and *winNameStr* is "", WaveRefIndexed works on the list of all waves in the current data folder.

For graph windows, *type* is 1 for y waves, 2 for x waves, 3 for either y or x waves.

For table windows, *type* is 1 for data columns, 2 for index or dimension label columns, 3 for either data or index or dimension label columns.

WaveRefIndexed returns a null reference (see **WaveExists**) if there is no wave matching the parameters.

Examples

```
WaveRefIndexed("",0,1)           // Returns first Y wave in the top graph.
WaveRefIndexed("Graph0",1,2)     // Returns second X wave in Graph0.
WaveRefIndexed("Table0",1,3)     // wave in second column in Table0.
```

See Also

WaveRefIndexedDFR, **NameOfWave**, **GetWavesDataFolder**

For a discussion of wave references, see **Wave Reference Functions** on page IV-197.

WaveRefIndexedDFR

`WaveRefIndexedDFR(`*`dfr`*`, `*`index`*`)`

The WaveRefIndexedDFR function returns a wave reference to the *index*th wave in the specified data folder.

Parameters

dfr is a data folder reference.

index is the zero-based index of the wave you want to access.

Details

WaveRefIndexedDFR returns a null reference (see **WaveExists**) if there is no wave corresponding to index in the specified data folder.

Example

```
// DemoWaveRefIndexedDFR can be called like this:
//  DemoWaveRefIndexedDFR(root:, 0)                 // Work on root
//  DemoWaveRefIndexedDFR(root:SubDataFolder, 0)    // Work on root:SubDataFolder
//  DemoWaveRefIndexedDFR(:, 0)                     // Work on current data folder
Function DemoWaveRefIndexedDFR(dfr, recurse)
	DFREF dfr
	Variable recurse

	Variable index = 0
	do
		Wave/Z w = WaveRefIndexedDFR(dfr, index)
		if (!WaveExists(w))
			break
		endif
		String path = GetWavesDataFolder(w, 2)
		Print path
		index += 1
	while(1)

	if (recurse)
		Variable numChildDataFolders = CountObjectsDFR(dfr, 4)
		Variable i
		for(i=0; i<numChildDataFolders; i+=1)
			String childDFName = GetIndexedObjNameDFR(dfr, 4, i)
			DFREF childDFR = dfr:$childDFName
			DemoWaveRefIndexedDFR(childDFR, 1)
		endfor
	endif
End
```

See Also

WaveRefIndexed, **NameOfWave**, **GetWavesDataFolder**

For a discussion of wave references, see **Wave Reference Functions** on page IV-197.

WaveRefsEqual

`WaveRefsEqual(`*`w1`*`, `*`w2`*`)`

The WaveRefsEqual function returns the truth the two wave references are the same.

See Also

Wave Reference Functions on page IV-197

WaveRefWaveToList

`WaveRefWaveToList(`*`waveRefWave`*`, `*`option`*`)`

The WaveRefWaveToList function returns a semicolon-separated string list containing data folder paths.

Each element of the returned string list is the full or partial path to the wave referenced by the corresponding element of *waveRefWave*. Entries in *waveRefWave* that are NULL or entries that correspond to free waves result in an empty list element.

The WaveRefWaveToList function was added in Igor Pro 7.00.

Parameters

waveRefWave is a wave reference wave each element of which contains a reference to an existing wave or NULL (0).

option determines if the returned path is a full path or a partial path relative to the current data folder:

option=0:	Full path.
option=1:	Partial path relative to the current data folder.

Other values of *option* are reserved for the future.

Example

```
Function Test()
   SetDataFolder root:
   Make/O/FREE aaa
   Make/O bbb
   Make/O/WAVE/N=3 wr
   Wr[0]=aaa
   // Wr[1] is null by initialization.
   wr[2]=bbb
   Print WaveRefWaveToList(wr,0)
End
```

```
// Executing Test() gives:
  ;;root:bbb;
```

The first empty string corresponds to the free wave 'aaa' and the second empty string corresponds to the null entry in the wave reference wave.

See Also

ListToWaveRefWave, **ListToTextWave**, **Wave References** on page IV-71

WaveStats

`WaveStats` `[`***`flags`***`]` ***`waveName`***

The WaveStats operation computes several statistics on the named wave.

Flags

/ALPH=*val* Sets the significance level for the confidence interval of the mean (default *val*=0.05).

/C=*method* Calculates statistics for complex waves only. Does not affect real waves.

You can use *method* in various combinations to process the real, imaginary, magnitude, and phase of the wave. The result is stored in the wave M_WaveStats (see **Details** for format).

method is defined as follows:

method=0: Default; ignores the imaginary part of *waveName*. Use /W to also store statistics in M_WaveStats.

method=1: Calculates statistics for real part of *waveName* and stores it in M_WaveStats.

method=2: Calculates statistics for imaginary part of *waveName* and stores the result in M_WaveStats.

method=4: Calculates statistics for magnitude of *waveName*, i.e., `sqrt(real^2 +imag^2)`, and stores the result in M_WaveStats.

method=8: Calculate statistics for phase of *waveName* using `atan2(imag,real)`.

/CCL When computing per-column statistics using /PCST, /CCL tells Igor to copy the column dimension labels of the input to the corresponding columns of M_WaveStats. /CCL was added in Igor Pro 9.00.

If you use a single *method* the results are stored both in M_WaveStats and in the standard variables (e.g., V_avg, etc.). If you specify *method* as a combination of more than one binary field then the variables reflect the results for the lowest chosen field and all results are stored in the wave M_WaveStats.

For example, if you use /C=12, the variables will be set for the statistics of the magnitude and M_WaveStats will contain columns corresponding to the magnitude and to the phase.

In this mode V_numInfs will always be zero.

Note: If you invoke this operation and M_WaveStats already exists in the current data folder, it will be either overwritten or initialized to NaN.

/M=*moment* Calculates statistical moments.

moment is defined as follows:

moment=1: Calculates only lower moments: V_avg, V_npnts, V_numInfs, and V_numNaNs. Use it if you do not need the higher moments.

moment=2: Default; calculates both lower moments and higher order quantities: V_sdev, V_rms, V_adev, V_skew, and v_kurt.

/Q Prevents results from being printed in history.

/P Causes WaveStats to set the location output variables in terms of unscaled index values instead of the default scaled index values. The location output variables are:

```
V_minRowLoc, V_maxRowLoc, V_minColLoc, V_maxColLoc
V_minLayerLoc, V_maxLayerLoc, V_minChunkLoc, V_maxChunkLoc
```

For 1D waves, `V_minRowLoc` and `V_maxRowLoc` are always unscaled.

/P requires Igor Pro 8.03 or later.

/PCST Computes the statistics on a per-column basis for a real valued wave of two or more dimensions. The results are saved in the wave M_WaveStats which has the same number of columns, layers and chunks as the input wave and where the rows, designated by dimension labels, contain the standard WaveStats statistics. All the V_ variables are set to NaN. Note that this flag is not compatible with the flags /C, /R, /RMD.

The /PCST flag was added in Igor Pro 7.00.

/R=(*startX*,*endX*) Specifies an X range of the wave to evaluate.

/R=[*startP*,*endP*] Specifies a point range of the wave to evaluate.

If you specify the range as /R=[*startP*] then the end of the range is taken as the end of the wave. If /R is omitted, the entire wave is evaluated.

/RMD=[*firstRow*,*lastRow*][*firstColumn*,*lastColumn*][*firstLayer*,*lastlayer*][*firstChunk*,*lastChunk*]

Designates a contiguous range of data in the source wave to which the operation is to be applied. This flag was added in Igor Pro 7.00.

You can include all higher dimensions by leaving off the corresponding brackets. For example:

```
/RMD=[firstRow,lastRow]
```

includes all available columns, layers and chunks.

You can use empty brackets to include all of a given dimension. For example:

```
/RMD=[][firstColumn,lastColumn]
```

means "all rows from column A to column B".

You can use a * to specify the end of any dimension. For example:

```
/RMD=[firstRow,*]
```

means "from firstRow through the last row".

/W Stores results in the wave M_WaveStats in addition to the various V_ variables when /C=0.

/Z No error reporting.

/ZSCR Computes z scores

$$z_i = \frac{Y_i - \overline{Y}}{\sigma},$$

which are saved in W_ZScores.

Details

WaveStats uses a two-pass algorithm to produce more accurate results than obtained by computing the binomial expansions of the third and fourth order moments.

WaveStats returns the statistics in the automatically created variables:

`V_npnts` Number of points in range excluding points whose value is NaN or INF.

`V_numNans` Number of NaNs.

`V_numINFs` Number of INFs.

`V_avg` Average of data values.

`V_sum` Sum of data values.

V_sdev Standard deviation of data values,

$$\sigma = \sqrt{\frac{\sum \left(Y_i - V\_avg\right)^2}{V\_npnts - 1}}$$

"Variance" is V_sdev^2.

V_sem Standard error of the mean $sem = \frac{\sigma}{\sqrt{V\_npnts}}$

`V_rms` RMS of Y values $= \sqrt{\frac{1}{V\_npnts} \sum Y_i^2}$

V_adev Average deviation $= \frac{1}{V\_npnts} \sum_{i=0}^{V\_npnts-1} \left|Y_i - \overline{Y}\right|$

V_skew Skewness $= \frac{1}{V\_npnts} \sum_{i=0}^{V\_npnts-1} \left(\frac{Y_i - \overline{Y}}{\sigma}\right)^3$

V_kurt Kurtosis $= \left(\frac{1}{V\_npnts} \sum_{i=0}^{V\_npnts-1} \left(\frac{Y_i - \overline{Y}}{\sigma}\right)^4\right) - 3$

`V_minloc` X location of minimum data value.

`V_min` Minimum data value.

`V_maxloc` X location of maximum data value.

`V_max` Maximum data value.

`V_minRowLoc` Row containing minimum data value. See /P above for further information.

`V_maxRowLoc` Row containing maximum data value. See /P above for further information.

`V_minColLoc` Column containing minimum data value (2D or higher waves). See /P above for further information.

`V_maxColLoc` Column containing maximum data value (2D or higher waves). See /P above for further information.

`V_minLayerLoc` Layer containing minimum data value (3D or higher waves). See /P above for further information.

`V_maxLayerLoc` Layer containing maximum data value (3D or higher waves). See /P above for further information.

`V_minChunkLoc` Chunk containing minimum data value (4D waves only). See /P above for further information.

`V_maxChunkLoc` Chunk containing maximum data value (4D waves only). See /P above for further information.

`V_startRow` The unscaled index of the first row included in caculating statistics.

`V_endRow` The unscaled index of the last row included in caculating statistics.

`V_startCol` The unscaled index of the first column included in calculating statistics. Set only when /RMD is used.

`V_endCol` The unscaled index of the last column included in calculating statistics. Set only when /RMD is used.

`V_startLayer` The unscaled index of the first layer included in calculating statistics. Set only when /RMD is used.

`V_endLayer` The unscaled index of the last layer included in calculating statistics. Set only when /RMD is used.

`V_startChunk` The unscaled index of the first chunk included in calculating statistics. Set only when /RMD is used.

`V_endChunk` The unscaled index of the last chunk included in calculating statistics. Set only when /RMD is used.

WaveStats prints the statistics in the history area unless /Q is specified. The various multidimensional min and max location variables will only print to the history area for waves having the appropriate dimensionality.

The format of the M_WaveStats wave is:

Row	Statistic	Row	Statistic	Row	Statistic	Row	Statistic
0	`numPoints`	9	`minLoc`	18	`maxColLoc`	27	`startCol`
1	`numNaNs`	10	`min`	19	`maxLayerLoc`	28	`endCol`
2	`numInfs`	11	`maxLoc`	20	`maxChunkLoc`	29	`startLayer`
3	`avg`	12	`max`	21	`startRow`	30	`endLayer`
4	`sdev`	13	`minRowLoc`	22	`endRow`	31	`startChunk`
5	`rms`	14	`minColLoc`	23	`sum`	32	`endChunk`
6	`adev`	15	`minLayerLoc`	24	`meanL1`		
7	`skew`	16	`minChunkLoc`	25	`meanL2`		
8	`kurt`	17	`maxRowLoc`	26	`sem`		

`meanL1` and `meanL2` are the confidence intervals for the mean

$$MeanL1 = V\_avg - t_{\alpha,v}\frac{V\_sdev}{\sqrt{V\_npnts}},\quad \text{and} \quad MeanL2 = V\_avg + t_{\alpha,v}\frac{V\_sdev}{\sqrt{V\_npnts}}$$

where $t_{a,v}$ is the critical value of the Student T distribution for *alpha* significance and degree of freedom v=*V_npnts*-1.

Use `Edit M_WaveStats.ld` to display the results in a table with dimension labels identifying each of the row statistics.

WaveStats is not entirely multidimensional aware. Even so, much of the information computed by WaveStats is useful. See **Analysis on Multidimensional Waves** on page II-95 for details.

See Also

Chapter III-12, **Statistics** for details on other statistics.

See the **ImageStats** operation for calculating wave statistics for specified regions of interest in 2D matrix waves.

See the **APMath** operation if you require higher precision than provided by double-precision floating point.

WaveMax, **WaveMin**, **WaveMinAndMax**, **mean**, **median**, **Variance**

WaveTextEncoding

WaveTextEncoding(*wave*, *element*, *getEffectiveTextEncoding*)

The WaveTextEncoding function returns the text encoding code for the specified element of a wave. See **Wave Text Encodings** on page III-472 for background information.

This function is used to deal with text encoding issues that sometimes arise in when you load pre-Igor Pro 7 experiments. Most users will have no need to use it.

The WaveTextEncoding function was added in Igor Pro 6.30. The *getEffectiveTextEncoding* parameter was added in Igor Pro 7.00.

Parameters

wave specifies the wave of interest.

element specifies a part of the wave, as follows:

Value	Meaning
1	Wave name
2	Wave units
4	Wave note
8	Wave dimension labels
16	Text wave content

getEffectiveTextEncoding determines if WaveTextEncoding returns a raw text encoding code or an effective text encoding code as explained below.

Details

WaveTextEncoding returns a integer text encoding code. See **Text Encoding Names and Codes** on page III-490 for details.

As explained under **Wave Text Encodings** on page III-472, each of the wave elements has a corresponding text encoding setting. Because the notion of text encoding settings was added in Igor Pro 6.30, waves created by earlier versions have their text encoding settings set to unknown (0).

The text encoding setting stored for a given element is the "raw" text encoding. If it is unknown, then Igor applies some rules when the wave is accessed to determine an "effective" text encoding for the element being accessed. The rules are explained under **Determining the Text Encoding for a Plain Text File** on page III-467.

If *getEffectiveTextEncoding* is non-zero then WaveTextEncoding returns the effective text encoding. If *getEffectiveTextEncoding* is zero it returns the raw text encoding.

See Also

Wave Text Encodings on page III-472, **Text Encoding Names and Codes** on page III-490, **Determining the Text Encoding for a Plain Text File** on page III-467

WaveTracking

`WaveTracking [/FREE /GLBL /LOCL] keyword`

The WaveTracking operation is a debugging aid that can help you determine if your code creates waves and fails to kill them. This is especially helpful for finding free wave leaks. For background information, see **Wave Reference Counting** on page IV-205, **Free Wave Leaks** on page IV-94 and **Wave Tracking** on page IV-207.

WaveTracking was added in Igor Pro 9.00.

A fast and lightweight complement to wave tracking is **IgorInfo**(16). For details, see **Detecting Wave Leaks** on page IV-206.

Flags

There are three flags that tell the WaveTracking operation what category of waves an invocation of the WaveTracking operation is aimed at. You can simultaneously track or count all three categories of waves, but you must use separate invocations of the operation to control or query the tracking of each category. See **Wave Tracking** on page IV-207 for a discussion of the wave categories.

You must include one of these flags.

/FREE	Specifies that the current command applies to counting or tracking free waves.
/GLBL	Specifies that the current command applies to counting or tracking global waves (waves in the main data hierarchy starting with the root data folder).
/LOCL	Specifies that the current command applies to counting or tracking local waves (waves contained by a free data folder).
/Q	Used with the dump keyword to suppress the information printed to the history.

Keywords

counter — Turns on wave tracking in *counter* mode for the category specified by /FREE, /GLBL, or /LOCL. Clears any existing count or tracking information for that specified category.

tracker — Turns on wave tracking in *tracker* mode for the category specified by /FREE, /GLBL, or /LOCL. In this mode, a list of waves of the specified category is kept. Clears any existing count or tracking information for that category.

count — Stores the count of waves in the variable V_numWaves for the category specified by /FREE, /GLBL, or /LOCL. The count is the number of waves created and not killed since *counter* or *tracker* mode was turned on for that category.

dump[=*n*] — Prints information into the history for the category specified by /FREE, /GLBL, or /LOCL. In *counter* mode, it prints just the number of waves. In *tracker* mode, it prints a line showing the count, and a number of lines showing the name of each wave that was created but not killed since tracking began, and the wave's reference count. If waves are global or local waves, the name of the containing data folder is also printed.

If you omit *n*, the list is limited to 10 lines. Otherwise *n* sets the maximum number of lines to print. Due to the method used for tracking, the list is in random order.

In addition, in *tracker* mode it creates a string variable *S_waveTracker* containing the same information. See below.

Use the /Q flag to suppress the history printout.

status — Stores a number for the category specified by /FREE, /GLBL, or /LOCL indicating the type of tracking or zero into the variable *V_Flag*. 0 means no counting or tracking, 1 means counting, 2 means tracking.

stop — Stops wave tracking for the category specified by /FREE, /GLBL, or /LOCL and clears the count and list of waves of that category.

Details

In *counter* mode, each time a wave in the specified category is created the counter is incremented. Each time a wave is killed, the counter is decremented. If you start counting after some waves have been created, and get the count after killing those waves, it is possible for the count to become negative.

In *tracker* mode, the count is the number of waves created and not killed since you started tracking; it cannot be negative.

Creation of short waves takes approximately twice as long when *tracker* mode is turned on. Using *counter* mode has negligible effect on performance.

By default, all free waves have the name _free_, which limits the usefulness of the tracker dump. Starting with Igor Pro 9.00, to aid wave leak investigation, both **NewFreeWave** and **Make**/FREE have options for giving names to free waves. See **Free Wave Names** on page IV-95.

Output Variables

These variables are created and set by all keywords.

V_Flag — Indicates the type of tracking currently being used. The values are:

0: Not tracking or counting

1: Counter mode

2: Tracker mode

V_numWaves — The number of waves of the specified category created and not killed since the last time the counter or tracker keywords were used. If no tracking is enabled, this value will be zero. Set to zero if no tracking is currently enabled.

S_waveTracker — When you use the dump keyword in tracker mode, this string variable is created with a list of waves created since tracking started. The contents are a list of keyword-value strings separated by a carriage return. Each line of the contents is a keyword-value string containing the name, reference count and data folder for one of the list waves.

Examples

```
WaveTracking/GLBL counter          // start global tracker in counter mode
Make/O/N=1 jack, jill              // make two waves
WaveTracking/GLBL count            // ask for the count of waves in V_numWaves
print V_numWaves                   // print "2" in the history
WaveTracking/GLBL stop             // stops counting waves
KillWaves jack, jill               // so that we can count them all over again
WaveTracking/GLBL tracker          // start global tracker in tracker mode
Make/O/N=1 jack, jill              // make two waves
WaveTracking/GLBL count            // ask for the count of waves in V_numWaves
print V_numWaves                   // print "2" in the history
WaveTracking/GLBL dump             // ask for the history report on waves created
print S_waveTracker                // print the info string to the history
WaveTracking/GLBL stop             // stops counting waves
```

The dump keyword above prints this in the history:

```
Since tracking began, 2 global waves have been created and not killed.
Wave 'jill'; data folder: 'root'; refcount: 1
Wave 'jack'; data folder: 'root'; refcount: 1
```

The print command prints this:

```
WAVE:jill;REFCOUNT:1;DF:root;
WAVE:jack;REFCOUNT:1;DF:root;
```

To extract information from S_waveTracker:

```
print StringFromList(1, S_waveTracker, "\r")// extract and print second line
String str = StringFromList(0, S_waveTracker, "\r")// extract first line of information into string variable
print StringByKey("WAVE", str)// print the name of the wave from the first line
print StringByKey("DF", str)// print the wave's data folder from the first line
```

The first line prints the entire second line from S_waveTracking:

```
WAVE:jack;REFCOUNT:1;DF:root;
```

The third line extracts the wave name from the extracted second line and prints it to the history; the fourth line prints the data folder containing that wave:

```
jill
root
```

One more example showing both global and free trackers in use together:

```
WaveTracking/GLBL tracker
WaveTracking/FREEtracker
Make/N=3/WAVE wavewave             // make a global wave reference wave
// make three named free waves with references in the wave wave
wavewave = NewFreeWave(2, 1, "free_"+num2str(p))
WaveTracking/GLBL/Q dump           // /Q: we only want the S_waveTracker info
print S_waveTracker
WaveTracking/FREE/Q dump           // /Q: we only want the S_waveTracker info
print S_waveTracker
WaveTracking/GLBL stop             // stops counting waves
WaveTracking/FREE stop             // stops counting waves
```

The first print statement prints this:

```
WAVE:wavewave;REFCOUNT:1;DF:root;
```

The second print statement prints this, though the ordering of the lines will be different each time the code above is run again. These are free waves, which do not have a data folder:

```
WAVE:free_2;REFCOUNT:1;DF:;
WAVE:free_0;REFCOUNT:1;DF:;
WAVE:free_1;REFCOUNT:1;DF:;
```

WaveTransform

`WaveTransform` `[`***`flags`***`]` ***`keyword srcWave`***

The WaveTransform operation transforms *srcWave* in various ways. If the /O flag is not specified then unless otherwise indicated the output is stored in the wave W_WaveTransform, which will be of the same data type as *srcWave* and saved in the current data folder.

Parameters

keyword is one of the following:

abs

Calculates the absolute value of the entries in *srcWave*. It stores results in W_Abs if *srcWave* is 1D or M_Abs otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.

acos

Calculates the inverse cosine of the entries in *srcWave*. It stores results in W_Acos if *srcWave* is 1D or M_Acos otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.

asin

Calculates the inverse sine of the entries in *srcWave*. It stores results in W_Asin if *srcWave* is 1D or M_Asin otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.

atan

Calculates the inverse tangent of the entries in *srcWave*. It stores results in W_Atan if *srcWave* is 1D or M_Atan otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.

cconjugate

Calculates the complex conjugate of *srcWave*. Stores results in W_CConjugate or M_CConjugate, depending on wave dimensionality, or overwrites *srcWave* if /O is used.

cos

Calculates the cosine of the entries in *srcWave*. It stores results in W_Cos if *srcWave* is 1D or M_Cos otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.

crystalToRect

Converts triplet (three column {x,y,z}) waves from nonorthogonal crystallographic coordinates to rectangular cartesian system. The parameters provided in the /P flag are the crystallographic definition of the coordinate system given by {a, b, c, alpha, beta, gamma}. The three angles are assumed to be expressed in radians unless the /D flag is specified. The transformation sets the first component parallel to the vector a and the third component parallel to c*. The output is stored in the current data folder in the wave M_CrystalToRect which has the same data type. If the /O flag is specified, the output overwrites the original data.

flip

Flips the data in *srcWave* about its center. If /O flag is used, *srcWave* is overwritten. Otherwise a new wave is created in the current data folder. The wave is named W_flipped or M_flipped according to the dimensionality of *srcWave*.

index

Fills *srcWave* as in `jack=p`.

If /P is specified then `jack=p+`*`param1`*.

The /O flag does not apply here.

inverse

Computes `1/srcWave[i]` for each point in *srcWave* and stores it in W_Inverse or M_Inverse depending on the dimensionality of *srcWave*.

inverseIndex

Fills *srcWave* as in `jack=numPnts-1-p`.

If /P is specified the `jack=numPnts-1-p+`*`param1`*.

magnitude

Creates a real-valued wave that is the magnitude of *srcWave*. If you do not specify the /O flag, the output is stored in W_Magnitude or M_Magnitude depending on the dimensionality of *srcWave*; the output precision will be the same as *srcWave*.

magsqr

Creates a real-valued wave that is the magnitude squared of *srcWave*. If *srcWave* is a double precision complex wave, the output is also double precision, otherwise the output is a single precision wave. Stores the result in wave W_MagSqr or M_MagSqr, depending on the dimensionality of *srcWave*, or overwrites *srcWave* if /O is used.

max

Calculates the maximum of a point in *srcWave* and a fixed number specified as a single parameter with the /P flag. It stores results in W_max if *srcWave* is 1D or M_max otherwise. It will overwrite *srcWave* when used with the /O flag. See also the min keyword and the example below.

min

Calculates the minimum of a point in *srcWave* and a fixed number specified as a single parameter with the /P flag. It stores results in W_min if *srcWave* is 1D or M_min otherwise. It will overwrite *srcWave* when used with the /O flag. See also the max keyword and the example below.

normalizeArea

Calculates the area under the curve and rescales the wave so that the area is 1. Note that waves with negative areas will be rescaled to positive values. Applies to 1D real-valued waves. It does not affect wave scaling. Stores the result in the wave W_normalizedArea or overwrites *srcWave* if /O is used.

phase

Creates a real-valued wave containing the phase of the complex input wave. If the /O flag is not used, the output is stored in W_Phase or M_Phase depending on the dimensionality of *imageMatrix*. You can also use /P={*norm*} to divide the output wave by the value of *norm*.

rectToCrystal

Converts triplet (three column {x,y,z}) waves from cartesian coordinates to nonorthogonal crystallographic coordinate system. The parameters provided in the /P flag are the crystallographic definition of the coordinate system given by {a, b, c, alpha, beta, gamma}. The three angles are assumed to be expressed in radians unless the /D flag is specified. The transformation assumes the first component parallel to the vector a and the third component parallel to c*. The output is stored in the current data folder in the wave M_RectToCrystal which has the same data type. If the /O flag is specified, the output overwrites the original data.

setConstant

Sets *srcWave* points to a constant value specified by the /V flag. This keyword applies to real, numeric waves only.

You can use /R with setConstant to set a subset of a wave.

setConstant was added in Igor Pro 7.00.

setZero

Sets all *srcWave* points to zero. setZero was added in Igor Pro 7.00.

sgn

Sets the value to -1 if the entry is negative, 1 otherwise. Stores the results in W_Sgn or overwrites *srcWave* if /O is used. This operation will not work on UNSIGNED waves.

shift

Shifts the position of data in *srcWave* by the specified number of points.

Unlike Rotate, WaveTransform discards data points that shift outside existing wave boundaries. After the shift, vacated wave points are set to the specified *fillValue*. The shift and the *fillValue* are specified with the /P flag using the syntax: /P={*numPoints*, *fillValue*}. If you do not provide a fill value, it will be 0 for integer waves and NaN for SP and DP.

sin

Calculates the sine of the entries in srcWave. Stores results in W_Sin if *srcWave* is 1D or M_Sin otherwise. Overwrites *srcWave* when used with the /O flag. *srcWave* must be a real single or double precision floating point wave.

sqrt

Calculates the square root of the entries in *srcWave*. It stores results in W_sqrt if *srcWave* is 1D or M_sqrt otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single- or double-precision real wave.

tan

Calculates the tangent of the entries in *srcWave*. The results are stored in W_tan if *srcWave* is 1D or M_tan otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single- or double-precision real wave.

zapINFs

Deletes elements whose value is infinity or -infinity. This is relevant for 1D single-precision and double-precision floating point waves only and does nothing for other types of 1D waves. It is not suitable for multidimensional waves and returns an error if *srcWave* is multidimensional. Use **MatrixOp** `replace` for multidimensional waves.

zapNaNs	Deletes elements whose value is NaN. This is relevant for 1D single-precision and double-precision floating point waves only and does nothing for other types of 1D waves. It is not suitable for multidimensional waves and returns an error if *srcWave* is multidimensional. Use **MatrixOp** `replaceNaNs` for multidimensional waves.
zapZeros	Deletes wave elements whose value is zero. zapZeros works only with 1D 8-bit, 16-bit, and 32-bit integer waves and returns an error if srcWave is multidimensional or another data type. zapZeros was added in Igor Pro 9.00.

Flags

/D If present, angles in wave data are interpreted as in degrees. Otherwise they are interpreted as in radians.

/O Overwrites input wave.

/P={*param1*…} Specifies parameters as appropriate for the keyword that you are using. The number of parameters and their order depends on the keyword.

/R=[*startRow*,*endRow*][*startCol*,*endCol*][*startLayer*,*endLayer*][*startChunk*,*endChunk*]

Specifies the range of elements to set for the setConstant keyword.

You can omit parameters for dimensions that don't exist in *srcWave*. For example, if *srcWave* is 1D, specify just /R=[*startRow*,*endRow*].

/R was added in Igor Pro 7.00.

/V=*value* Specifies the value to use for the setConstant keyword. /V was added in Igor Pro 7.00.

Examples

```
// Produce output values in the range [-1,1]:
WaveTransform /P={(pi)} phase complexWave
```

```
// Faster than myWave=myWave>1 ? 1 : myWave
WaveTransform /P={1}/O min myWave
```

See Also

The **Rotate** operation.

References

Shmueli, U. (Ed.), International Tables for Crystallography, Volume B: 3.3, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

WaveType

`WaveType(`*`waveName`*` [,`*`selector`*` ])`

The WaveType function returns the type of data stored in the wave.

If *selector* = 1, WaveType returns 0 for a null wave, 1 if numeric, 2 if text, 3 if the wave holds data folder references or 4 if the wave holds wave references.

If *selector* = 2, WaveType returns 0 for a null wave, 1 for a normal global wave or 2 for a free wave or a wave that is stored in a free data folder.

If selector = 3, WaveType returns 0 for a null wave reference or a global wave, 1 for a free wave (a wave that is not stored in any data folder) or 2 for a local wave (a wave that is stored in a free data folder hierarchy).

If *selector* is omitted or zero, the returned value for numeric waves is a combination of bit values shown in the following table:

Type	Bit Number	Decimal Value	Hexadecimal Value
complex	0	1	1
32-bit float	1	2	2
64-bit float	2	4	4

Type	Bit Number	Decimal Value	Hexadecimal Value	
8-bit integer	3	8	8	
16-bit integer	4	16	10	
32-bit integer	5	32	20	
64-bit integer	7	128	80	Requires Igor Pro 7 or later
unsigned	6	64	40	

The unsigned bit is used only with the integer types while the complex bit can be used with any numeric type. Set only one of bits 1-5 or bit 7 as they are mutually exclusive. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The returned value for non-numeric waves (text waves, wave-reference waves and data folder-reference waves) is 0.

Examples

```
Variable waveIsComplex = WaveType(wave) & 0x01
Variable waveIs32BitFloat = WaveType(wave) & 0x02
Variable waveIs64BitFloat = WaveType(wave) & 0x04
Variable waveIs8BitInteger = WaveType(wave) & 0x08
Variable waveIs16BitInteger = WaveType(wave) & 0x10
Variable waveIs32BitInteger = WaveType(wave) & 0x20
Variable waveIs64BitInteger = WaveType(wave) & 0x80
Variable waveIsUnsigned = WaveType(wave) & 0x40
```

See Also

For concepts related to selector = 1 or 2, see **Free Waves** on page IV-91, **Wave Reference Waves** on page IV-77 and **Data Folder Reference Waves** on page IV-82.

WaveUnits

WaveUnits(*waveName*, *dimNumber*)

The WaveUnits function returns a string containing the units for the given dimension.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers, and 3 for chunks. Use -1 to get the data units. If the wave is just 1D, *dimNumber*=0 returns X units and 1 returns data units. This behavior is just like the WaveMetrics procedure WaveUnits found in the WaveMetrics Procedures folder in previous versions of Igor Pro.

See Also

DimDelta, **DimOffset**, **DimSize**, **SetScale**

wfprintf

wfprintf *refNumOrStr*, *formatStr* [*flags*] *waveName* [, *waveName*]…

The wfprintf operation is like the **printf** operation except that it prints the contents of the named waves to a file whose file reference number is in *refNum*.

The **Save** operation also outputs wave data to a text file. Use Save unless you need the added flexibility provided by wfprintf.

Parameters

refNumOrStr is a numeric expression, a string variable or an SVAR pointing to a global string variable.

If a numeric expression, then it is a file reference number returned by the **Open** operation or an expression that evaluates to 1.

If *refNumOrStr* is 1, Igor prints to the history area instead of to a file.

If *refNumOrStr* is the name of a string variable, the wave contents are “printed” to the named string variable. *refNumOrStr* can also be the name of an SVAR to print to a global string:

```
SVAR sv = root:globalString
wfprintf sv, "", wave0
```

refNumOrStr can not be an element of a text wave.

The value of each named wave is printed to the file according to the conversion specified in *formatStr*.

formatStr contains one numeric conversion specification per column. See **printf**. If *formatStr* is `""`, wfprintf uses a default format which gives tab-delimited columns.

Flags

Note: /R must follow the *formatStr* parameter directly without an intervening comma.

/R=(*startX*,*endX*)	Specifies an X range in the wave(s) to print.
/R=[*startP*,*endP*]	Specifies a point range in the wave(s) to print.

Details

As of Igor7, wfprintf supports 1D and 2D waves. Previously it supported 1D waves only.

The number of conversion characters in *formatStr* must exactly match the number of wave columns in all input waves. With real waves, the total number of columns is limited to 100. With complex waves, the real column and imaginary column each count as a column and the total number of columns is limited to 200.

The only conversion characters allowed are fFeEgdouxXcs (the floating point, integer and string conversion characters). You cannot use an asterisk to specify field width or precision. If any of these restrictions is intolerable, you can use fprintf in a loop.

With integer conversion characters d, o, u, x, and X, applied to floating point waves, wfprintf rounds the fractional part.

In Igor Pro 9.00 and later, there is no limit to the length of an element of a text wave parameter passed to the wfprintf operation. Previously text wave element contents were clipped to about 500 bytes. There is also no limit to the length of formatStr. Previously it was limited to 800 bytes.

Examples

```
Function Example1()
   Make/O/N=10 wave0=sin(p*pi/10)                 // test numeric wave
   Make/O/N=10/T textWave= "row "+num2istr(p)     // test text wave
   Variable refNum
   Open/P=home refNum as "output.txt"// open file for write
   wfprintf refNum, "%s = %g\r"/R=[0,5], textWave, wave0    // print 6 values each
   Close refNum
End
```

The resulting output.txt file contains:

```
row 0 = 0
row 1 = 0.309017
row 2 = 0.587785
row 3 = 0.809017
row 4 = 0.951057
row 5 = 1
```

```
Function/S NumericWaveToStringList(w)
   Wave w                        // numeric wave (if text, use /T here and %s below)
   String list
   wfprintf list, "%g;" w        // semicolon-separated list
   return list
End
```

```
Print NumericWaveToStringList(wave0)
  0;0.309017;0.587785;0.809017;0.951057;1;0.951057;0.809017;0.587785;0.309017;
```

See Also

The **printf** operation for complete format and parameter descriptions and **Creating Formatted Text** on page IV-259. The **Open** operation about *refNum* and for another way of writing wave files.

The **Save** operation.

WhichListItem

WhichListItem(*itemStr*, *listStr* [, *listSepStr* [, *startIndex* [, *matchCase*]]])

The WhichListItem function returns the index of the first item of *listStr* that matches *itemStr*. *listStr* should contain items separated by *listSepStr* which typically is ";". If the item is not found in the list, -1 is returned.

Use WhichListItem to locate an item in a string containing a list of items separated by a string (usually a single ASCII character), such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

listSepStr, *startIndex*, and *matchCase* are optional; their defaults are ";", 0, and 1 respectively.

Details

WhichListItem differs from **FindListItem** in that WhichListItem returns a list index, while FindListItem returns a character offset into a string.

listStr is searched for *itemStr* bound by *listSepStr* on the left and right.

listStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *listSepStr* are always case-sensitive. The comparison of *itemStr* to the contents of *listStr* is usually case-sensitive. Setting the optional *matchCase* parameter to 0 makes the comparison case insensitive.

If *itemStr* is not found, if *listStr* is `""`, or if *startIndex* is not within the range of 0 to ItemsInList(*listStr*)-1, then -1 is returned.

In Igor6, only the first byte of *listSepStr* was used. In Igor7 and later, all bytes are used.

Items can be empty. In `"abc;def;;ghi"`, the third item, whose zero-based index is 2, is empty. In `";def;;ghi;"` the first and third items, whose zero-based indices are 0 and 2, are empty.

If *startIndex* is specified, then *listSepStr* must also be specified. If *matchCase* is specified, *startIndex* and *listSepStr* must be specified.

Examples

```
Print WhichListItem("wave0", "wave0;wave1;")          // prints 0
Print WhichListItem("c", "a;b;")                      // prints -1
Print WhichListItem("", "a;;b;")                      // prints 1
Print WhichListItem("c", "a,b,c,x,c", ",")            // prints 2
Print WhichListItem("c", "a,b,c,x,c", ",", 3)         // prints 4
Print WhichListItem("C", "x;c;C;")                    // prints 2
Print WhichListItem("C", "x;c;C;", ";", 0, 0)         // prints 1
```

See Also

The **AddListItem**, **FindListItem**, **FunctionList**, **ItemsInList**, **RemoveListItem**, **RemoveFromList**, **StringFromList**, **StringList**, **TraceNameList**, **VariableList**, and **WaveList** functions.

WignerTransform

WignerTransform [/Z][/WIDE=*wSize*][/GAUS=*gaussianWidth*][/DEST=*destWave*] *srcWave*

The WignerTransform operation computes the Wigner transformation of a 1D signal in *srcWave*, which is the name of a real or complex wave. The result of the WignerTransform is stored in *destWave* or in the wave M_Wigner in the current data folder.

Flags

/DEST=*destWave* Creates by default a real wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-72 for details.

/GAUS=*gWidth* Computes the Gaussian Wigner Transform, which is a convolution of the Wigner Transform with a two-dimensional Gaussian (in the two parameters of the transform). The computation of the transform simplifies significantly when the product of the widths of the two Gaussians is unity (minimum uncertainty ellipse).

gWidth uses the same units as the *srcWave* scaling.

/OUT=*type* Sets the output data type of the standard (not Gaussian) Wigner transform. The following data types are supported:

1: Complex

2: Real (default)

3: Magnitude

4: Squared magnitude

/OUT is not allowed with the Gaussian Wigner transform (/GAUS) in which the output is always real.

The /OUT flag was added in Igor Pro 8.00.

/WIDE=*wSize* Computes Wigner Transform and sets the transform width to *wSize*. This is the default transformation with *wSize* set to the size of *srcWave*.

/Z No error reporting.

Details

The Wigner transform maps a time signal *U*(*t*) into a 2D time-frequency representation:

$$W(t,\nu)=\int_{-\infty}^{\infty} U\left(t+\frac{x}{2}\right)U^{*}\left(t-\frac{x}{2}\right)e^{-2\pi i x\nu}\,dx.$$

The computation of the Wigner transform evaluates the offset product

$$U\left(t+\frac{x}{2}\right)U^{*}\left(t-\frac{x}{2}\right)$$

over a finite window and then Fourier transforms the result. The offset product can be evaluated over a finite window width, which can vary from a few elements of the input wave to the full length of the wave. You can control the width of this window using the /WIDE flag. If you do not specify the output destination, WignerTransform saves the results in the wave M_Wigner in the current data folder.

Although the Wigner transform is real, the output will be complex when *srcWave* is complex. By inspecting the complex wave you can gain some insight into the numerical stability of the algorithm. The X-scaling of the output wave is identical to the scaling of *srcWave*. The Y-scaling of the input wave is taken from the Fourier Transform of the offset product, which in turn is determined by the X-scaling of *srcWave*. Specifically, if `dx=DimDelta(srcWave,0)` and *srcWave* has N points then `dy=DimDelta(M_Wigner,1)=1/(dx*N)`. WignerTransform does not set the units of the output wave.

The Ambiguity Function is related to the Wigner Transform by a Fourier Transform, and is defined by

$$A(\tau,\nu)=\int_{-\infty}^{\infty} U\left(t+\frac{\tau}{2}\right)U^{*}\left(t-\frac{\tau}{2}\right)e^{-2\pi i t\nu}dt.$$

Convolving the Wigner Transform with a 2D Gaussian leads to what is sometimes called the Gaussian Wigner Transform or GWT. Formally the GWT is given by the equation:

$$GWT\left(t,\nu;\delta_t,\delta_\nu\right)=\frac{1}{\delta_t\delta_\nu}\iint dt'd\nu'W(t',\nu')\exp\left\{-2\pi\left[\left(\frac{t-t'}{\delta_t}\right)^2+\left(\frac{\nu-\nu'}{\delta_\nu}\right)^2\right]\right\}.$$

Computationally this equation simplifies if the respective widths of the two Gaussians satisfy the minimum uncertainty condition $\delta_t*\delta_\nu=1$. The /GAUS flag calculates the Gaussian Wigner Transform using your specified width, δ_t, and it selects a δ_ν such that it satisfies the minimum uncertainty condition.

See Also

CWT, **FFT**, and **WaveTransform** operations.

For further discussion and examples see **Wigner Transform** on page III-281.

References

Wigner, E. P., On the quantum correction for thermo-dynamic equilibrium, *Physics Review*, *40*, 749-759, 1932.

Bartelt, H.O., K.-H. Brenner, and A.W. Lohman, The Wigner distribution function and its optical production, *Optics Communications*, *32*, 32-38, 1980.

Window

`Window` ***`macroName`*****`(`**`[`***`parameters`***`]`**`)`** `[`***`:macro type`***`]`

The Window keyword introduces a macro that recreates a graph, table, layout, or control panel window. The macro appears in the appropriate submenu of the Windows menu. Window macros are automatically created when you close a graph, table, layout, control panel, or XOP target window. You should use **Macro**, **Proc**, or **Function** instead of Window for your own window macros. Otherwise, it works the same as **Macro**.

See Also

The **Macro**, **Proc**, and **Function** keywords. **Data Folders and Window Recreation Macros** on page II-111 for details.

Macro Syntax on page IV-118 for further information.

WindowFunction

`WindowFunction` `[`**`/FFT`**`[`***`=f`***`]` **`/DEST=`*****`destWave`***`]` ***`windowKind`*****`,`** ***`srcWave`***

The WindowFunction operation multiplies a one-dimensional (real or complex) *srcWave* by the named window function.

By default the result overwrites *srcWave*.

Parameters

srcWave A one-dimensional wave of any numerical type. See **ImageWindow** for windowing two-dimensional data.

windowKind Specifies the windowing function. Choices for *windowKind* are:

Bartlett, Blackman367, Blackman361, Blackman492, Blackman474, Cos1, Cos2, Cos3, Cos4, Hamming, Hanning, KaiserBessel20, KaiserBessel25, KaiserBessel30, Parzen, Poisson2, Poisson3, Poisson4, Riemann, and an assortment of flat-top windows listed under **FFT**.

See **FFT** for window equations and details. The equations assume that /FFT=1.

Flags

/DEST=*destWave* Creates or overwrites *destWave* with the result of the multiplication of *srcWave* and the window function.

When used in a function, the WindowFunction operation by default creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-72 for details.

/FFT [=1] The window interval is `0…N=numpnts(`*`srcWave`*`)`. This sets the first value of *srcWave* to zero, but not the last value. This is appropriate for windowing data in preparation for Fourier Transforms, and is the same algorithm used by **FFT**.

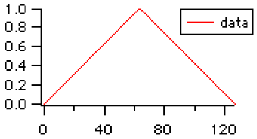
The window interval is `0…N=numpnts(`*`srcWave`*`)-1` if /FFT is missing or /FFT=0. This sets the first and last value of *srcWave* to 0. This is the (only) algorithm that the Hanning operation uses.

Details

A “window function” alters the input data by decreasing values near the start and end of the data smoothly towards zero, so that when the FFT of the data is computed the effects of nonintegral-periodic signals are diminished. This improves the ability of the FFT to distinguish among closely-spaced frequencies. Each window function has advantages and disadvantages, usually trading off rejection of “leakage” against the ability to discriminate adjacent frequencies. For more details, see the **References**.

WindowFunction stores the window function’s normalization value (the average squared window value) in V_value. This is the value you would get from WaveStats’s `V_rms*V_rms` for a wave of *srcWave*’s length whose values were all equal to 1:

```
Make/O data = 1
WindowFunction Bartlet, data      // Bartlet allowed as synonym for Bartlett
Print V_value                // Prints 0.330709, mean of squared window values
```



```
WaveStats/Q data
Print V_rms*V_rms             // Prints 0.330709
```

See Also

FFT, **ImageWindow**, **DPSS**

References

For more information about the use of window functions see:

Harris, F.J., On the use of windows for harmonic analysis with the discrete Fourier Transform, *Proc, IEEE*, *66*, 51-83, 1978.

Wikipedia entry: <`http://en.wikipedia.org/wiki/Window_function`>.

WinList

`WinList(`*`matchStr`*`, `*`separatorStr`*`, `*`optionsStr`*`)`

The WinList function returns a string containing a list of windows selected based on the *matchStr* and *optionsStr* parameters.

Details

For a window name to appear in the output string, it must match *matchStr* and also must fit the requirements of *optionsStr*. *separatorStr* is appended to each window name as the output string is generated.

The name of each window is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all window names
"xyz"	Matches window name xyz only
"*xyz"	Matches window names which end with xyz
"xyz*"	Matches window names which begin with xyz
"*xyz*"	Matches window names which contain xyz
"abc*xyz"	Matches window names which begin with abc and end with xyz

matchStr may begin with the ! character to return items that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches window names which *do not* end with xyz

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

optionsStr is used to further qualify the window. The acceptable values for *optionsStr* are:

`""` Consider all windows.

`"WIN:"` The target window.

`"WIN:`*`windowTypes`*`"` Consider windows that match *windowTypes*.

`"INCLUDE:`*`includeTypes`*`"` Consider procedure windows that match *includeTypes*.

Using INCLUDE: implies WIN:128.

`"INDEPENDENTMODULE:1"` Consider procedure windows that are part of any independent module as well as those that are not. Matching windows names are actually the window titles followed by `" [<independent module name>]"`.

Using INDEPENDENTMODULE: implies WIN:128.

`"INDEPENDENTMODULE:0"` Consider procedure windows only if they are not part of any independent module. Matching windows names are actually the window titles, which for an external file includes the file extension, such as "WMMenus.ipf".

Using INDEPENDENTMODULE: implies WIN:128.

"FLT:1" Return only panels that were created with NewPanel/FLT=1. Specifying "FLT" also implies "WIN:64".

Omit FLT or use "FLT:0" to return windows that do not float (and most do not).

"FLT:2" Return only panels that were created with NewPanel/FLT=2. Specifying "FLT" also implies "WIN:64".

"VISIBLE:1" Return only visible windows (ignore hidden windows).

windowTypes is a literal number. The window name goes into the output string only if it passes the match test and its type is compatible with *windowTypes*. *windowTypes* is a bitwise parameter:

1:	Graphs
2:	Tables
4:	Layouts
16:	Notebooks
64:	Panels
128:	Procedure windows
512:	Help windows
4096:	XOP target windows
16384:	Camera windows in Igor Pro 7.00 or later
65536:	Gizmo windows in Igor Pro 7.00 or later

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Procedure windows and help windows don't have names. WinList returns the window title instead.

includeTypes is also a literal number. The window name goes into the output string only if it passes the match test and its type is compatible with *includeTypes*. *includeTypes* is one of:

1: Procedure windows that are not #included.

2: Procedure windows included by `#include "`*`someFileName`*`"`.

4:	Procedure windows included by `#include <someFileName>`.

or a bitwise combination of the above for more than one type of inclusion.

You can combine the `WIN`, `INCLUDE` and `INDEPENDENTMODULE` options by separating them with a comma.

When the INDEPENDENTMODULE option is used, the title of any procedure window that is part of an independent module will be followed by `" [<independent module name>]"`.

For example, if a procedure file contains:

```
#pragma IndependentModule=myIndependentModule
#include <Axis Utilities>
```

A call to WinList like this:

```
String list = WinList("* [myIndependentModule]", ";", "INDEPENDENTMODULE:1")
```

will store "Axis Utilities.ipf [myIndependentModule];" in the `list` string, along with any other procedure windows that are part of that independent module.

When the INDEPENDENTMODULE option is omitted, the returned procedure window titles do not include any independent module name suffix, and the procedure files "visible" to WinList depend on the setting of `SetIgorOption independentModuleDev` (which must be done after opening the experiment):

`SetIgorOption independentModuleDev=0`	Consider procedure windows only if they are not part of any independent module and if they are not hidden (using `#pragma hide`, for example).
`SetIgorOption independentModuleDev=1`	Consider all procedure windows including those in independent modules or hidden.

Examples

Command	**Returned List**
`WinList("*",";","")`	All existing non-floating windows.
`WinList("*", ";","WIN:3")`	All graph and table windows.
`WinList("Result_*", ";", "WIN:1")`	Graphs whose names start with “Result_”.
`WinList("*", ";","WIN:64,FLT:1,FLT:2")`	All floating panel windows.
`WinList("*", ";","INCLUDE:6")`	All #included procedure windows.
`WinList("*", ";","WIN:1,INCLUDE:6")`	All graphs and #included procedure windows.

See Also

Independent Modules on page IV-238. The **ChildWindowList** and **WinType** functions.

WinName

`WinName(`*`index`*`, `*`windowTypes`*`** `[,` ***`visibleWindowsOnly`*** `[,` ***`floatKind`***`]])`

The WinName function returns a string containing the name of the *index*th window of the specified *type*, or an empty string (`""`) if no window fits the parameters.

If the optional *visibleWindowsOnly* parameter is nonzero, only visible windows are considered. Otherwise both visible and hidden windows are considered.

If the optional *floatKind* parameter is 1, only floating windows created with NewPanel/FLT=1 are considered. If *floatKind* is 2, only NewPanel/FLT=2 windows are considered. *windowTypes* must contain at least 64 (panels).

If *floatKind* is omitted or is 0 only non-floating ("normal") windows are considered.

Procedure windows don’t have names. WinName returns the procedure window title instead.

Details

index starts from zero, and returns the top-most window matching the parameters.

The window names are ordered in window-stacking order, as returned by WinList.

`DoWindow/B` moves the window to the back and changes the index needed to retrieve its name to the greatest index that returns any name.

Hiding or showing a window (with `SetWindow hide=1` or `Notebook visible=0` or by manual means) does not affect the index associated with the window.

windowTypes is a bitwise parameter:

1:	Graphs.
2:	Tables.
4:	Layouts.
16:	Notebooks.
64:	Panels.
128:	Procedure windows.
4096:	XOP target windows.
16384:	Camera windows in Igor Pro 7.00 or later
65536:	Gizmo windows in Igor Pro 7.00 or later

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Examples

```
Print WinName(0,1)          // Prints the name of the top graph.
Print WinName(0,3)          // Prints the name of the top graph or table.
String win=WinName(0,1)     // The name of the top visible graph.
SetWindow $win hide=1       // Hide the graph (it may already be hidden).
Print WinName(0,1)          // Prints the name of the now-hidden graph.
Print WinName(0,1,1)        // Prints the name of the top visible graph.
Print WinName(0,64,1,1)     // Name of the top visible NewPanel/FLT=1 window.
```

See Also

WinList, **DoWindow** (/F and /B flags), **SetWindow** (hide keyword), **Notebook (Miscellaneous)** (visible keyword), **NewPanel** (/FLT flag).

WinRecreation

`WinRecreation(`*`winStr`*`, `*`options`*`)`

The WinRecreation function returns a string containing the window recreation macro (or style macro) for the named window.

Parameters

winStr is the name of a graph, table, page layout, panel, notebook, Gizmo, camera, or XOP target window or the title of a procedure window or help file. If *winStr* is "" and options is 0 or 1, information for the top graph, table, page layout, panel, notebook, or XOP target window is returned.

As of Igor Pro 7.00, *winStr* may be a subwindow path. The returned recreation macro is generated as if the subwindow were extracted from its host as a standalone window. See **Subwindow Syntax** on page III-92 for details on forming the subwindow path.

The meaning of *options* depends on the type of window as described in the following sections.

Target Window Details

Target windows include graphs, tables, page layouts, panels, notebooks, and XOP target windows.

If *options* is 0, WinRecreation returns the window recreation macro.

If *options* is 1, WinRecreation returns the style macro or an empty string if the window does not support style macros.

Graphs Details

If *options* is 2, WinRecreation returns a recreation macro in which all occurrences of wave names are replaced with an ID number having the form ##<number>## (for instance, ##25##). These ID numbers can be found easily using the **strsearch** function. This is intended for applications that need to alter the recreation macro by replacing wave names with something else, usually other wave names. The ID numbers are the same as those returned by the **GetWindow** operation with the wavelist keyword.

Graphs and Panels Details

If *options* is 4, WinRecreation returns the window recreation macro without the default behavior of causing the graph to revert to “normal” mode (as if the GraphNormal operation had been called). This allows the use of WinRecreation when a graph or panel is in drawing tools mode without exiting that mode. For windows other than graphs or panels, this is equivalent to an *options* value of 0.

Notebooks Details

If *options* is -1, WinRecreation returns the same text that the Generate Commands menu item would generate with the Selected paragraphs radio button selected and all the checkboxes selected (includes text commands).

If *options* is 0, WinRecreation returns the same text that the Generate Commands menu item would generate with the Entire document radio button selected and all the checkboxes *except* “Generate text commands” selected).

If *options* is 1, WinRecreation returns the same text that the Generate Commands menu item would generate with the Entire document radio button selected and all the checkboxes selected (includes text commands).

Regardless of the value of *options* the text returned by WinRecreation for notebook always ends with 5 lines of file-related information formatted as comments:

```
// File Name: MyNotebook.txt
// Path: "Macintosh HD:Desktop Folder:"
// Symbolic Path: home
// Selection Start: paragraph 100, position 31
// Selection End: paragraph 100, position 31
```

Help Windows Details

WinRecreation returns the same 5 lines of file-related information as described above for notebooks.

Set *options* to -3 to ensure that *winStr* is interpreted as a help window title (help windows have only titles, not window names).

Procedures Details

WinRecreation returns the same 5 lines of file-related information as described above for notebooks.

Set *options* to -2 to ensure that *winStr* is interpreted as a procedure window title (procedure windows have only titles, not window names).

If `SetIgorOption IndependentModuleDev=1` is in effect, *winStr* can also be a procedure window title followed by a space and, in brackets, an independent module name. In such cases WinRecreation returns text from or information about the specified procedure file which is part of that independent module. (See **Independent Modules** on page IV-238 for independent module details.)

For example, in an experiment containing:

```
#pragma IndependentModule=myIM
#include <Axis Utilities>
```

code like this:

```
String text=WinRecreation("Axis Utilities.ipf [myIM]",-2)
```

will return the file-related information for the Axis Utilities.ipf procedure window, which is normally a hidden part of the myIM independent module.

To get the text content of a procedure window, use the **ProcedureText** function.

Examples

```
WinRecreation("Graph0",0)      // Returns recreation macro for Graph0.
WinRecreation("",1)                      // Style macro for top window.
String win= WinName(0,16,1)                   // top visible notebook
String str= WinRecreation(str,-1)             // Selected Text commands
Variable line= itemsInList(str,"\r")-5        // First file info line
Print StringFromList(line, str,"\r")          // Print File Name:
```

```
Print StringFromList(line+1, str,"\r")      // Print Path:
Print StringFromList(line+2, str,"\r")      // Print Symbolic Path:
Print StringFromList(line+3, str,"\r")      // Selection Start:
Print StringFromList(line+4, str,"\r")      // Selection End:
```

See Also

Saving a Window as a Recreation Macro on page II-47.

WinType

`WinType(`*`winNameStr`*`)`

The WinType function returns a value indicating the type of the named window.

Details

winNameStr is a string or string expression containing the name of a window or subwindow, or `""` to signify the target window. When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

WinType returns the following values:
0: No window by that name.

1:	Graph
2:	Table
3:	Layout
5:	Notebook
7:	Panel
13:	XOP target window
15:	Camera window in Igor Pro 7.00 or later
17:	Gizmo window in Igor Pro 7.00 or later

Because command and procedure windows do not have *names* (they only have *titles*), WinType can not even be asked about those windows.

See Also

The **WinName**, **ChildWindowList**, and **WinList** functions.

WMAxisHookStruct

See **NewFreeAxis** for further explanation of WMAxisHookStruct.

```
Structure WMAxisHookStruct
   char win[200]       // Host window or subwindow name
   char axName[32]     // Name of axis
   char mastName[32]   // Name of controlling axis or ""
   char units[50]      // Axis units.
   Variable min        // Current axis range minimum value
   Variable max        // Current axis range maximum value
EndStructure
```

WMBackgroundStruct

See **CtrlNamedBackground**, **Background Tasks** on page IV-319, and **Preemptive Background Task** on page IV-335 for further explanation of WMBackgroundStruct.

```
Structure WMBackgroundStruct
   char name[32]          // Background task name
   UInt32 curRunTicks     // Tick count when task was called
   Int32 started          // TRUE when CtrlNamedBackground start is issued
   UInt32 nextRunTicks    // Precomputed value for next run
                          // but user functions may change this
EndStructure
```

WMButtonAction

This structure is passed to action procedures for button controls created using the **Button** operation.

```
Structure WMButtonAction
   char ctrlName[32]       // Control name
   char win[200]           // Host window or subwindow name
   STRUCT Rect winRect     // Local coordinates of host window
   STRUCT Rect ctrlRect    // Enclosing rectangle of the control
   STRUCT Point mouseLoc   // Mouse location
   Int32 eventCode         // See details below
   Int32 eventMod          // See Control Structure eventMod Field on page III-438
   String userData         // Primary unnamed user data.
   Int32 blockReentry      // Obsolete, see Control Structure blockReentry Field on page
   III-439
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMButtonAction eventCode Field

Your action function should test the eventCode field and respond only to documented eventCode values because other event codes may be added in the future.

The event code passed to the button action procedure has the following meaning:

Event Code	Event
-3	Control received keyboard focus (Igor8 or later)
-2	Control received keyboard focus (Igor8 or later)
-1	Control being killed
1	Mouse down
2	Mouse up
3	Mouse up outside control
4	Mouse moved
5	Mouse enter
6	Mouse leave
7	Mouse dragged while outside the control

Events 2 and 3 happen only after event 1.

Events 4, 5, and 6 happen only when the mouse is over the control but happen regardless of the mouse button state.

Event 7 happens only when the mouse is pressed inside the control and then dragged outside.

WMCheckboxAction

This structure is passed to action procedures for checkbox controls created using the **CheckBox** operation.

```
Structure WMCheckboxAction
   char ctrlName[MAX_OBJ_NAME+1]     // Control name
   char win[MAX_WIN_PATH+1]          // Host window or subwindow name
   STRUCT Rect winRect               // Local coordinates of host window
   STRUCT Rect ctrlRect              // Enclosing rectangle of the control
   STRUCT Point mouseLoc             // Mouse location
   Int32 eventCode                   // See details below
   Int32 eventMod                    // See Control Structure eventMod Field on page
   III-438
   String userData                   // Primary unnamed user data
   Int32 blockReentry                // Obsolete, see Control Structure blockReentry
   Field on page III-439
   Int32 checked                     // Checkbox state
   char vName[MAX_OBJ_NAME+2 + (MAXDIMS * (MAX_OBJ_NAME+5)) + 1]   // Name of variable
   WAVE ckWave;                      // Valid if using wave
   Int32 rowIndex                    // Row index for a wave, if rowLabel is empty
   char rowLabel[MAX_OBJ_NAME+1]     // Wave row dimension label
   Int32 colIndex                    // Column index for a wave if colLabel is empty
   char colLabel[MAX_OBJ_NAME+1]     // Wave column dimension label
   Int32 layerIndex                  // Layer index for a wave if layerLabel is empty
   char layerLabel[MAX_OBJ_NAME+1]   // Wave layer dimension label
   Int32 chunkIndex                 // Chunk index for a wave if chunkLabel is empty
   char chunkLabel[MAX_OBJ_NAME+1]  // Wave chunk dimension label
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMCheckboxAction eventCode Field

Your action function should test the eventCode field and respond only to documented eventCode values because other event codes may be added in the future.

The event code passed to the checkbox action procedure has the following meaning:

Event Code	Event
-3	Control received keyboard focus (Igor8 or later)
-2	Control received keyboard focus (Igor8 or later)
-1	Control being killed
2	Mouse up, checkbox toggled

WMCustomControlAction

This structure is passed to action procedures for custom controls created using the **CustomControl** operation.

```
Structure WMCustomControlAction
   char ctrlName[32]      // Control name
   char win[200]          // Host window or subwindow name
   STRUCT Rect winRect    // Local coordinates of host window
   STRUCT Rect ctrlRect   // Enclosing rectangle of the control
   STRUCT Point mouseLoc  // Mouse location
   Int32 eventCode        // See details below
   Int32 eventMod         // See Control Structure eventMod Field on page III-438
   String userData        // Primary unnamed user data
   Int32 blockReentry     // Obsolete, see Control Structure blockReentry Field on page
   III-439
   Int32 missedEvents     // TRUE when events occurred but the user
                          // function was not available for action
   Int32 mode             // General purpose

   // Used only when eventCode==kCCE_frame
   Int32 curFrame         // Input and output

   // Used when eventCode is kCCE_mousemoved, kCCE_mouseenter or kCCE_mouseleave
   Int32 needAction       // See below for details

   // These fields are valid only with value=varName
   Int32 isVariable       // TRUE if varName is a variable
   Int32 isWave           // TRUE if varName referenced a wave
   Int32 isString         // TRUE if varName is a String type
   NVAR nVal              // Valid if isVariable and not isString
   SVAR sVal              // Valid if isVariable and isString
   WAVE nWave             // Valid if isWave and not isString
   WAVE/T sWave           // Valid if isWave and isString
   Int32 rowIndex         // If isWave, this is the row index
                          // unless rowLabel is not empty
   char rowLabel[32]      // Wave row label

   // These fields are valid only when eventCode==kCCE_char
   Int32 kbChar           // Keyboard key character code
   Int32 specialKeyCode   // See Keyboard Events on page IV-300 - Added in Igor Pro 7
   char keyText[16]       // UTF-8 string representing key struck - Added in Igor Pro 7
   Int32 kbMods           // Keyboard key modifiers bit field. See details below.
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMCustomControlAction eventCode Field

When determining the state of the eventCode member in the WMCustomControlAction structure, the various values you use are listed in the following table. You can define the kCCE symbolic constants by adding this to your procedure file:

```
#include <CustomControl Definitions>
```

Event Code	Description
`kCCE_mousedown = 1`	Mouse down in control.
`kCCE_mouseup = 2`	Mouse up in control.
`kCCE_mouseup_out = 3`	Mouse up outside control.
`kCCE_mousemoved = 4`	Mouse moved (happens only when mouse is over the control).
`kCCE_enter = 5`	Mouse entered control.
`kCCE_leave = 6`	Mouse left control.
`kCCE_mouseDraggedOutside = 7`	The mouse moved while it was outside the control. This event is delivered only after the mouse is pressed inside the control and dragged outside. While the mouse is inside the control, kCCE_mousemoved is delivered whether the mouse button is up or down.
`kCCE_draw = 10`	Time to draw custom content.
`kCCE_mode = 11`	Sent when executing `CustomControl` *`name`*`, mode=`*m*.
`kCCE_frame = 12`	Sent before drawing a subframe of a custom picture.
`kCCE_dispose = 13`	Sent as the control is killed.
`kCCE_modernize = 14`	Sent when dependency (variable or wave set by value=*varName* parameter) fires. It will also get draw events, which probably don't need a response.
`kCCE_tab = 15`	Sent when user tabs into the control. If you want keystrokes (`kCCE_char`), then set `needAction`.
`kCCE_char = 16`	Sent on keyboard events. Stores the keyboard character in `kbChar` and modifiers bit field is stored in `kbMods`. Sets `needAction` if key event was used and requires a redraw.
`kCCE_drawOSBM = 17`	Called after drawing *pict* from picture parameter into an offscreen bitmap. You can draw custom content here.
`kCCE_idle = 18`	Idle event typically used to blink insertion points etc. Set `needAction` to force the control to redraw. Sent only when the host window is topmost.

WMCustomControl needAction Field

The meaning of needAction depends on the event.

Events kCCE_mousemoved, kCCE_enter, kCCE_leave, and kCCE_mouseDraggedOutside set needAction to TRUE to force redraw, which is normally not done for these events.

Events kCCE_tab and kCCE_mousedown set needAction to TRUE to request keyboard focus (and get kCCE_char events).

Event kCCE_idle sets needAction to TRUE to request redraw.

WMCustomControl kbMods Field

Bit 0:	Command (*Macintosh*)
Bit 1:	Shift
Bit 2:	Alpha Lock. Not supported in Igor7 or later.
Bit 3:	Option (*Macintosh*) or Alt (*Windows*)
Bit 4:	Control (*Macintosh*) or Windows key (*Windows*).

WMDrawUserShapeStruct

See **DrawUserShape** for further explanation of WMDrawUserShapeStruct.

```
Structure WMDrawUserShapeStruct
   char action[32]         // Input: Specifies what action is requested.

   Int32 options           // Input: Value from /MO flag.
                           // Output: When action is getInfo, set bits as follows:
                           // Set bit 0 if the shape should behave like a simple line.
                           //     When resizing end-points, you will get live updates.
                           // Set bit 1 if the shape is to act like a button;
                           //     You will get mouse down in normal operate mode.
                           // Set bit 2 to get roll-over action.
                           //     You will get hitTest action and
                           //     if 1 is returned, the mouse will be captured.

   Int32 operateMode       // Input: If 0, the shape is being edited;
                           // if 1, normal operate mode
                           // (only if options bit 1 or 2 was set during getInfo).

   PointF mouseLoc         // Input: The location of the mouse in normalized coordinates.

   Int32 doSetCursor       // Output: If action is hitTest, set true
                           // to use the following cursor number.
                           // Also used for mouseMoved in rollover mode.

   Int32 cursorCode        // Output: If action is hitTest and doSetCursor is set,
                           // then set this to the desired Igor cursor number.

   double x0,y0,x1,y1      // Input: Coordinates of the enclosing rectangle of the shape.

   RectF objectR           // Input: Coordinates of the enclosing rectangle of the shape
                           // in device units.

   char winName[MAX_HostChildSpec+1] // Input: Full path to host subwindow

   // Information about the coordinate system
   Rect drawRect           // Draw rect in device coordinates
   Rect plotRect           // In a graph, this is the plot area
   Rect axRect             // In a graph, this is the plot area including axis standoff
   char xcName[MAX_OBJ_NAME+1]   // Name of X coordinate system, may be axis name
   char ycName[MAX_OBJ_NAME+1]   // Name of Y coordinate system, may be axis name

   double angle            // Input: Rotation angle, use when displaying text
   String textString       // Input: Use or ignore; special output for "getInfo"
   String privateString    // Input and output: Maintained by Igor
                           // but defined by user function;
                           // may be binary; special output for "getInfo"
EndStructure
```

WMFitInfoStruct

See **The WMFitInfoStruct Structure** on page III-263 for further explanation of WMFitInfoStruct.

```
Structure WMFitInfoStruct
   char IterStarted        // Nonzero on the first call of an iteration
   char DoingDestWave      // Nonzero when called to evaluate autodest wave
   char StopNow            // Fit function sets this to nonzero to
                           // indicate that a problem has occurred
                           // and fitting should stop
   Int32 IterNumber        // Number of iterations completed
   Int32 ParamPerturbed    // See The WMFitInfoStruct Structure on page III-263
EndStructure
```

WMGizmoHookStruct

See **Gizmo Named Hook Functions** on page II-472 for further explanation of WMGizmoHookStruct.

```
Structure WMGizmoHookStruct
   Int32 version
   char winName[MAX_HostChildSpec+1]    // Full path to host window or subwindow
   char eventName[32]
   Int32 width
   Int32 height
   Int32 mouseX
   Int32 mouseY
   Variable xmin
   Variable xmax
   Variable ymin
   Variable ymax
   Variable zmin
   Variable zmax
   Variable eulerA
   Variable eulerB
   Variable eulerC
   Variable wheelDx
   Variable wheelDy
EndStructure
```

WMListboxAction

This structure is passed to action procedures for listbox controls created using the **ListBox** operation.

```
Structure WMListboxAction
   char ctrlName[32]       // Control name
   char win[200]           // Host window or subwindow name
   STRUCT Rect winRect     // Local coordinates of host window
   STRUCT Rect ctrlRect    // Enclosing rectangle of the control
   STRUCT Point mouseLoc   // Mouse location
   Int32 eventCode         // See details below
   Int32 eventMod          // See Control Structure eventMod Field on page III-438
   String userData         // Primary unnamed user data
   Int32 blockReentry      // Obsolete, see Control Structure blockReentry Field on page
   III-439
   Int32 eventCode2        // Obsolete
   Int32 row               // Selection row. See ListBox for details.
   Int32 col               // Selection column. See ListBox for details.
   WAVE/T listWave         // List wave specified by ListBox command
   WAVE selWave            // Selection wave specified by ListBox command
   WAVE colorWave          // Color wave specified by ListBox command
   WAVE/T titleWave        // Title wave specified by ListBox command
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMListboxAction eventCode Field

Your action function should test the eventCode field and respond only to documented eventCode values because other event codes may be added in the future.

The event code passed to the listbox action procedure has the following meaning:

Event Code	Event
-3	Control received keyboard focus (Igor8 or later)
-2	Control received keyboard focus (Igor8 or later)
-1	Control being killed.
1	Mouse down.
2	Mouse up.
3	Double click.
4	Cell selection (mouse or arrow keys).
5	Cell selection plus Shift key.
6	Begin edit.
7	Finish edit.
8	Vertical scroll. See **Scroll Event Warnings** on page V-495.
9	Horizontal scroll by user or by the hScroll=*h* keyword.
10	Top row set by row=*r* or first column set by col=*c* keywords.

Event Code	Event
11	Column divider resized.
12	Keystroke, character code is place in row field. See **Note on Keystroke Event** on page V-495.
13	Checkbox was clicked. This event is sent after selWave is updated.

WMListboxAction row and col Fields

The row field is the zero-based row number of the first selected row in the list or -1 if the selection is in title area. If an event occurs in the empty space below the last row, the row field is set to the number of rows in the list which is one greater than the row number of the last row. A mouse down event in an empty list reports row=0.

The col field is the column number of the selection.

The meanings of row and col are different for eventCodes 8 through 11:

Code	row	col
8	top visible row	horiz shift in pixels.
9	top visible row	horiz shift (user scroll).
9	-1	horiz shift (hScroll keyword).
10	top visible row	-1 (row keyword).
10	-1	first visible col (col keyword).
11	column shift	column resized by user.

If eventCode is 11, row is the horizontal shift in pixels of the column col that was resized, not the total horizontal shift of the list as reported in V_horizScroll by **ControlInfo**. If row is negative, the divider was moved to the left. col=0 corresponds to adjusting the divider on the right side of the first column. Use ControlInfo to get a list of all column widths.

Selection Events 4 and 5

These events are sent when a click on the Listbox could result in a change in selection. If it is important to you to respond only when the selection actually changes, you will need to keep track of the selection yourself.

These events are not sent if the list is empty.

WMMarkerHookStruct

See **Custom Marker Hook Functions** on page IV-308 for further explanation of WMMarkerHookStruct.

```
Structure WMMarkerHookStruct
   Int32 usage                // 0 = normal draw, 1 = legend draw
   Int32 marker               // Marker number minus start
   float x, y                 // Location of desired center of marker
   float size                 // Half width/height of marker
   Int32 opaque               // 1 if marker should be opaque
   float penThick             // Stroke width
   STRUCT RGBColor mrkRGB     // Fill color
   STRUCT RGBColor eraseRGB   // Background color
   STRUCT RGBColor penRGB     // Stroke color
   WAVE ywave                 // Trace's y wave
   double ywIndex             // Point number on ywave where marker is being drawn
   char winName[MAX_HostChildSpec+1] // Full path to window or subwindow
   char traceName[MAX_OBJ_INST+1]    // Full name of trace or "" if no trace
EndStructure
```

WMPopupAction

This structure is passed to action procedures for popup menu controls created using the **PopupMenu** operation.

```
Structure WMPopupAction
   char ctrlName[32]      // Control name
   char win[200]          // Host window or subwindow name
   STRUCT Rect winRect    // Local coordinates of host window
   STRUCT Rect ctrlRect   // Enclosing rectangle of the control
   STRUCT Point mouseLoc  // Mouse location
   Int32 eventCode        // See details below
   Int32 eventMod         // See Control Structure eventMod Field on page III-438
   String userData        // Primary unnamed user data
   Int32 blockReentry     // Obsolete, see Control Structure blockReentry Field on page
   III-439
   Int32 popNum           // Item number currently selected or hovered over (1-based)
   char popStr[MAXCMDLEN] // Contents of current popup item or item hovered over
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMPopupAction eventCode Field

Your action function should test the eventCode field and respond only to documented eventCode values because other event codes may be added in the future.

The event code passed to the pop-up menu action procedure has the following meaning:

Event Code	Event
-3	Control received keyboard focus (Igor8 or later)
-2	Control received keyboard focus (Igor8 or later)
-1	Control being killed
2	Mouse up
3	Hovering - sent when the user highlights a menu item by moving the mouse cursor over it but hasn’t selected it
4	Dismissed - sent when the user closes the menu without making a selection. This is primarily of use in conjunction with the hover event; it allows you to undo any changes made during a hover event when the menu is dismissed. This event code was added in Igor Pro 9.00.

WMSetVariableAction

This structure is passed to action procedures for SetVariable controls created using the **SetVariable** operation.

```
Structure WMSetVariableAction
   char ctrlName[32]      // Control name
   char win[200]          // Host window or subwindow name
   STRUCT Rect winRect    // Local coordinates of host window
   STRUCT Rect ctrlRect   // Enclosing rectangle of the control
   STRUCT Point mouseLoc  // Mouse location
   Int32 eventCode        // See details below
   Int32 eventMod         // See Control Structure eventMod Field on page III-438
   String userData        // Primary unnamed user data
   Int32 blockReentry     // Obsolete, see Control Structure blockReentry Field on page
   III-439
   Int32 isStr            // TRUE for a string variable
   Variable dval          // Numeric value of variable
   char sval[MAXCMDLEN]   // Value of variable as a string
   char vName[MAX_OBJ_NAME+2 + (MAXDIMS * (MAX_OBJ_NAME+5)) + 1]
   WAVE svWave            // Valid if using wave
   Int32 rowIndex                   // Row index for a wave if rowLabel is empty
   char rowLabel[MAX_OBJ_NAME+1]    // Wave row dimension label
   Int32 colIndex                   // Column index for a wave if colLabel is empty
   char colLabel[MAX_OBJ_NAME+1]    // Wave column dimension label
   Int32 layerIndex                 // Layer index for a wave if layerLabel is empty
   char layerLabel[MAX_OBJ_NAME+1]  // Wave layer label
   Int32 chunkIndex                 // Chunk index for a wave if chunkLabel is empty
   char chunkLabel[MAX_OBJ_NAME+1]  // Wave chunk label
   Int32 mousePart                  // Part of the control where mouse down occurred
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMSetVariableAction eventCode Field

Your action function should test the eventCode field and respond only to documented eventCode values because other event codes may be added in the future.

The event code passed to the SetVariable action procedure has the following meaning:

Event Code	Event
-3	Control received keyboard focus (Igor8 or later)
-2	Control received keyboard focus (Igor8 or later)
-1	Control being killed
1	Mouse up
2	Enter key
3	Live update
4	Mouse scroll wheel up
5	Mouse scroll wheel down
6	Value changed by dependency update
7	Begin edit (Igor7 or later)
8	End edit (Igor7 or later)
9	Mouse down (Igor8 or later)

Event code -1 is never sent to an old-style (non-structure parameter) action procedure.

Event code 1 is sent when the mouse is released after clicking the up-arrow or down-arrow buttons. It is also sent for value changes caused by the mouse scroll wheel for a non-live mode control.

Event codes 4 and 5 are sent only for string SetVariables or numeric SetVariables whose increment setting is zero. Otherwise the value change is signaled by event code 1.

For numeric SetVariables whose increment is non-zero, the mouse scroll wheel acts like a mouse click on the up-arrow button or down-arrow button. That is, event code 1, mouse up, is more like "value changed".

Event code 6 is by default sent to only structure-based action procedures.

Use SetIgorOption EnableSVE6=0 to disable sending this event at all and EnableSVE6=2 to send the event to both structure-based and old-style SetVariable action procedures. The default for EnableSVE6 is =1.

The mousePart field is valid for mouse down, mouse scroll wheel, live update, and Enter key events only:

Value	Mouse Part
0	A part other than 1, 2, or 3 - most likely the title
1	The up-arrow button on a numeric SetVariable via a mouse click on the button or the use of the up-arrow key
2	The down-arrow button on a numeric SetVariable via a mouse click on the button or the use of the down-arrow key
3	The value field

WMSliderAction

This structure is passed to action procedures for slider controls created using the **Slider** operation.

```
Structure WMSliderAction
   char ctrlName[32]       // Control name
   char win[200]           // Host window or subwindow name
   STRUCT Rect winRect     // Local coordinates of host window
   STRUCT Rect ctrlRect    // Enclosing rectangle of the control
   STRUCT Point mouseLoc   // Mouse location
   Int32 eventCode        // See details below
   Int32 eventMod         // See Control Structure eventMod Field on page III-438
   String userData        // Primary unnamed user data
   Int32 blockReentry     // Obsolete, see Control Structure blockReentry Field on page
   III-439
   Variable curval        // Value of slider
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMSliderAction eventCode Field

Your action function should test the eventCode field and respond only to documented eventCode values because other event codes may be added in the future.

The event code passed to slider action procedure has the following meaning:

Event Code	Event
-3	Control received keyboard focus (Igor8 or later)
-2	Control received keyboard focus (Igor8 or later)
-1	Control being killed
1	Value set
2	Mouse down
4	Mouse up
8	Mouse moved or arrow key moved the slider
16	Repeat timer fired (see **Repeating Sliders** on page III-418)

The event codes greater than zero are bits that may be combined in certain cases. For instance, if your slider does not have the live mode set, you may receive event code 5 = 1+4. This indicates that a mouse up event was detected, and that mouse up set the slider's value.

For a demo of many of these events, see **Handling Slider Events** on page III-430.

WMTabControlAction

This structure is passed to action procedures for tab controls created using the **TabControl** operation.

```
Structure WMTabControlAction
   char ctrlName[32]      // Control name
   char win[200]          // Host window or subwindow name
   STRUCT Rect winRect    // Local coordinates of host window
   STRUCT Rect ctrlRect   // Enclosing rectangle of the control
   STRUCT Point mouseLoc  // Mouse location
   Int32 eventCode        // See details below
   Int32 eventMod         // See Control Structure eventMod Field on page III-438
   String userData        // Primary unnamed user data
   Int32 blockReentry     // Obsolete, see Control Structure blockReentry Field on page
   III-439
   Int32 tab              // Tab number
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMTabControlAction eventCode Field

Your action function should test the eventCode field and respond only to documented eventCode values because other event codes may be added in the future.

The event code passed to tab control action procedure has the following meaning:

Event Code	Event
-3	Control received keyboard focus (Igor8 or later)
-2	Control received keyboard focus (Igor8 or later)
-1	Control being killed
2	Mouse up

WMTooltipHookStruct

See **Tooltip Hook Functions** on page IV-310 for further explanation of WMTooltipHookStruct.

```
Structure WMTooltipHookStruct
   char winName[MAX_WIN_PATH+1]      // Host window name or subwindow path
   double ticks                      // Tick count when event happened
   STRUCT Rect winRect               // Local coordinates of the window or subwindow
   STRUCT Point mouseLoc             // Mouse location
   STRUCT Rect trackRect             // Tooltip tracking rect
   double duration_ms                // Time to display the tooltip, in milliseconds
   char traceName[MAX_OBJ_NAME+1]    // If in a graph window, name of the trace
   char imageName[MAX_OBJ_NAME+1]    // If in a graph window, name of the image
   waveHndl yWave                    // Y wave for trace, image, or table column
   double row                        // Row in trace, image or wave
   double column                     // Column in trace, image or wave
   double layer                      // Layer in trace, image or wave
   double chunk                      // Chunk in trace, image or wave
   char ctrlName[MAX_OBJ_NAME+1]     // Name of control during hover event
   Int32 isHtml                      // Set to indicate tooltip contains HTML tags
   String tooltip                    // Set this to your tooltip text
EndStructure
```

WMWinHookStruct

See **Named Window Hook Functions** on page IV-295 for further explanation of WMWinHookStruct.

```
Structure WMWinHookStruct
   char winName[200]       // Host window or subwindow name
   STRUCT Rect winRect     // Local coordinates of the affected (sub)window
   STRUCT Point mouseLoc   // Mouse location
   Variable ticks          // Tick count when event happened
   Int32 eventCode         // See Named Window Hook Events on page IV-295
   char eventName[32]      // See Named Window Hook Events on page IV-295
   Int32 eventMod          // See Control Structure eventMod Field on page III-438
   char menuName[256]      // Name of the menu item as for SetIgorMenuMode
   char menuItem[256]      // Text of the menu item as for SetIgorMenuMode
   char traceName[32]      // See Named Window Hook Functions on page IV-295
   char cursorName[2]      // Cursor name A through J
   Variable pointNumber    // See Named Window Hook Functions on page IV-295
   Variable yPointNumber   // See Named Window Hook Functions
   Int32 isFree            // 1 if the cursor is not attached to anything
   Int32 keycode           // ASCII value of key struck
   Int32 specialKeyCode    // See Keyboard Events on page IV-300 - Igor Pro 7 or later
   char keyText[16]        // UTF-8 string representing key struck - Igor Pro 7 or later
   char oldWinName[32]     // Simple name of the window or subwindow
   Int32 doSetCursor       // Set to 1 to change cursor to cursorCode
   Int32 cursorCode        // See Setting the Mouse Cursor on page IV-302
   Variable wheelDx        // Vertical lines to scroll
   Variable wheelDy        // Horizontal lines to scroll
   char focusCtrl[MAX_WIN_PATH+1]    // Added in Igor Pro 9.00. See EarlyKeyboard Events.
EndStructure
```

wnoise

wnoise(*shape*, *scale*)

The wnoise function returns a pseudo-random value from the two-parameter Weibull distribution characterized by the *shape* and *scale,* the respective *gamma* and *alpha* parameters. The two-parameter Weibull probability distribution function is

$$f(x;\alpha,\gamma)=\frac{\gamma}{\alpha}x^{\gamma-1}\exp\left[-\frac{1}{\alpha}x^{\gamma}\right]\qquad \begin{array}{l} x\ge 0 \\ \alpha>0 \\ \gamma>0 \end{array}$$

The mean of the Weibull distribution is

$$\alpha^{\frac{1}{\gamma}}\Gamma\left(1+\frac{1}{\gamma}\right),$$

and the variance is

$$\alpha^{\frac{2}{\gamma}}\Gamma\left(1+\frac{2}{\gamma}\right)-\alpha^{\frac{2}{\gamma}}\left[\Gamma\left(1+\frac{1}{\gamma}\right)\right]^2.$$

Note that this definition of the PDF uses different scaling than the one used in StatsWeibullPDF. To match the scaling of StatsWeibullPDF multiply the result from Wnoise by the factor scale^(1-1/shape).

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-390.

Chapter III-12, **Statistics** for a function and operation overview.

x

`x`

The x function returns the scaled row index for the current point of the destination wave in a wave assignment statement. This is the same as the X value if the destination wave is a vector (1D wave).

Details

Outside of a wave assignment statement, x acts like a normal variable. That is, you can assign a value to it and use it in an expression.

See Also

The **p** function and **Waveform Arithmetic and Assignments** on page II-74.

x2pnt

`x2pnt(`*`waveName`*`, `*`x1`*`)`

The x2pnt function returns the integer point number on the wave whose X value is closest to *x1*.

For higher dimensions, use **ScaleToIndex**.

See Also

DimDelta, **DimOffset**, **pnt2x**, **ScaleToIndex**

For an explanation of waves and X scaling, see **Changing Dimension and Data Scaling** on page II-68.

xcsr

`xcsr(`*`cursorName`*` [, `*`graphNameStr`*`])`

The xcsr function returns the X value of the point which the named cursor (A through J) is on in the top or named graph.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

The result is derived from the wave that the cursor is on, not from the X axis of the graph. If the wave is displayed as an XY pair, the X axis and the wave's X scaling will usually be different.

See Also

The **hcsr**, **pcsr**, **qcsr**, **vcsr**, and **zcsr** functions.

Programming With Cursors on page II-321.

XLLoadWave

`XLLoadWave [`*`flags`*`] [`*`fileNameStr`*`]`

The XLLoadWave operation loads data from the named Excel .xls, .xlsx or .xlsm file into waves.

XLLoadWave does not support .xlsb files and can not load password-protected Excel files.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If XLLoadWave can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-451 for details on forming the path.

If *fileNameStr* is omitted or is "", or if the /I flag is used, XLLoadWave presents an Open File dialog from which you can choose the file to load.

Flags

/A — Automatically assigns arbitrary wave names using "wave" as the base name. Skips names already in use.

/A=*baseName* — Same as /A but it automatically assigns wave names of the form *baseName*0, *baseName*1.

/C=*columnType* — XLLoadWave will use the Deduce from Row method of determining the type of the Excel file columns, using the row specified by *columnType* to deduce column types. See **Deduce from row** on page II-160.

/COLT=*columnTypeStr*

columnTypeStr specifies how XLLoadWave should treat each column. For example, "1T3N" means 1 text column followed by 3 numeric columns. See **Determining Wave Types** on page V-1117.

/D — Creates double-precision floating point waves. If omitted, XLLoadWave creates single-precision floating point waves.

/F=f — New programming should use the /T flag instead of the /D, /L and /F flags.

f specifies the data format of the file:

f=1: Signed integer (8, 16, 32 bits allowed)

f=2: Creates double-precision waves

f=3: Floating point (default, 32, 64 bits allowed)

/I — Forces XLLoadWave to display an Open File dialog even if the file is fully specified via /P and *fileNameStr*.

/J=*infoMode* — If infoMode is 1, 2 or 3, XLLoadWave does not load the file but instead returns information about the worksheets within the workbook via the string variable S_value. See **Getting Information About the Excel File** on page V-1117.

/K=*k* — Discards waves with fewer than *k* points. For historical reasons, *k* defaults to 2.

/N Same as /A except that, instead of choosing names that are not in use, it overwrites existing waves.

/N=*baseName* Same as /N except that it automatically assigns wave names of the form *baseName0*, *baseName1*.

/NAME=*nameList* *nameList* is a semicolon-separated list of wave names to be used for the loaded waves. See **Wave Names** on page V-1116 for details.

/O Overwrites existing waves in case of a name conflict.

/P=*pathName* Specifies the folder to look in for *fileNameStr*. *pathName* is the name of an existing symbolic path.

/Q Suppresses the normal messages in the history area.

/R=(*cell1*,*cell2*) Restricts loading to the specified cells, e.g. /R=(A3,D21). Row and column numbers start from 1.

The /R flag supports an optional extra parameter that should be used only in very rare cases. XLLoadWave reads the range of defined cells from the file itself and clips *cell1* and *cell2* to that range.

In very rare cases the file does not accurately identify the range of defined cells so the clipping prevents loading cells that exist in the file. In this rare case, use /R=(*cell1*, *cell2*,1). The last parameter tells XLLoadWave to skip the clipping. If you specify incorrect values for cell1 or *cell2* you may get errors or garbage results.

/S=*sheetNameStr* Specifies which worksheet to load from a workbook file. If you omit /S=*sheetNameStr*, or if *sheetNameStr* is "", XLLoadWave loads the first worksheet in the workbook.

/T Automatically creates a table of loaded waves.

/V=*v* Controls the handling of blanks at the end of a column.

v=0: XLLoadWave leaves blanks at the end of a column in the Igor wave.

v=1: XLLoadWave removes blanks at the end of a column from the Igor wave. If the column has fewer than two remaining points, it is not loaded into a wave. This is the default mode that is used if you omit /V.

/W=*w* *w* specifies the row in which XLLoadWave will look for wave names. The first row is row number 1.

Wave Names

The names of the loaded waves are determined by the /A, /N, /W and /NAME flags. If all of the flags are omitted, default names, like ColumnA and ColumnB, are used.

If /W=w is present, names are loaded from row *w* of the worksheet and then converted to standard Igor names by replacing spaces and punctuation characters with underscores.

If /NAME=*nameList* is present, the wave names come from *nameList*, a semicolon-separated list of names. For example:

```
/NAME="StartTime;UnitA;UnitB;"
```

The names in *nameList* can be standard or liberal names. For example, this specifies names two standard names and one liberal name which contains a space:

```
/NAME="Signal;Ambient Temp;Response;"
```

If a name in the list is _skip_, the corresponding Excel column is skipped. For example, this would load the first and third columns and skip the second:

```
/NAME="Signal;_skip_;Response;"
```

If a name in the list is empty, the name used for the corresponding wave is as it would be if /NAME were omitted. This can be used to skip columns while taking wave names from the spreadsheet for loaded columns. In this example, the names of the first and third waves would be determined by row 1 of the spreadsheet while the second column would be skipped:

```
/W=1 /NAME=";_skip_;;"
```

The /N flag instructs Igor to automatically name new waves "wave", or *baseName* if /N=*baseName* is used, plus a number. The number starts from zero and increments by one for each wave loaded from the file. If the resulting name conflicts with an existing wave, the existing wave is overwritten.

The /A flag is like /N except that it skips names already in use.

/NAME overrides /W. /A or /N overrides both /NAME and /W.

No matter how the wave names are generated, if there is a name conflict and overwrite is off (/O is omitted), a unique name is generated. See **XLLoadWave and Wave Names** on page II-161 for further details.

Determining Wave Types

The /C or /COLT flag tells XLLoadWave how to decide what kind of wave, numeric, text, or date/time, to make for each Excel column.

Using /C=*columnType* causes XLLoadWave to use the Deduce from Row method of determining the type of the Excel file columns. *columnType* is the Excel row number that XLLoadWave should use to make the deduction.

Using /COLT=*columnTypeStr* causes XLLoadWave treat the columns based on the *columnTypeStr* parameter. If *columnTypeStr* is "N", XLLoadWave uses the Treat all Columns as Numeric method. If *columnTypeStr* is "T", XLLoadWave uses the Treat all Columns as Text method. If *columnTypeStr* is "D", XLLoadWave uses the Treat all Columns as Date method.

For any other value of columnTypeStr , XLLoadWave uses the Use Column Type String method. For example, "1T5N" tells XLLoadWave to create a text wave for the first column and numeric waves for the next 5 or more columns.

If you omit /C and /COLT, XLLoadWave uses the Treat all Columns as Numeric method.

See **What XLLoadWave Loads** on page II-159 for further details.

Output Variables

XLLoadWave sets the followin output variables:

`V_flag`	Number of waves loaded.
`S_fileName`	Name of the file being loaded.
`S_path`	File system path to the folder containing the file.
`S_waveNames`	Semicolon-separated list of the names of loaded waves.
`S_worksheetName`	Name of the loaded worksheet within the workbook file.
`S_value`	Set only if you use the /J flag. See **Getting Information About the Excel File** below.

S_path uses Macintosh path syntax (e.g., “`hd:FolderA:FolderB:`”), even on Windows. It includes a trailing colon.

When XLLoadWave presents an Open File dialog and the user cancels, V_flag is set to 0 and S_fileName is set to `""`.

Getting Information About the Excel File

The /J flag allows you to get information about an Excel file without actually loading it.

If *infoMode* is 1, XLLoadWave does not load the file but instead returns a semicolon-separated list of the names of the worksheets within the workbook via the string variable S_value.

If *infoMode* is 2, XLLoadWave does not load the file but instead returns information about the first worksheet or the worksheet specified by /S via the string variable S_value. The format of the returned information is:

```
NAME:<worksheet name>;FIRSTROW:<first row>;FIRSTCOL:<first col>;LASTROW:<last
   row>;LASTCOL:<last col>;
```

<first row> and <last row> are 1-based row numbers. <first col> and <last col> are 1-based column numbers; 1 refers to Column A. These refer to the defined rows and columns in the worksheet even if some or all cells are blank. If <last col> is zero, this means that there are no defined cells in the worksheet.

If *infoMode* is 3, XLLoadWave does not load the file but instead returns information about the first worksheet or the worksheet specified by /S via the string variable S_value. The format of the returned information is:

```
NAME:<worksheet name>;FIRST:<first cell>;LAST:<last cell>;
```

<first cell> and <last cell> are expressed in standard Excel notation (A1, B24, etc.). These refer to the defined rows and columns in the worksheet even if some or all cells are blank. If <last cell> is "@0", this means that there are no defined cells in the worksheet.

Use the **StringByKey**, **NumberByKey** functions to extract the information from S_value. If you use these functions, your code won't break if we later add a keyword/value pair to the returned information.

Examples

Old versions of Excel came with a number of sample files. One of them was called “Instrument Data”. The following procedure loads an area of this file, makes a table and then makes a graph of the loaded waves.

This example assumes that you have the "Instrument Data.xls" file and a symbolic path named Science that points to the folder containing the file.

```
Function InstrumentData()
    // Load Instrument Data file from the Scientific Analysis folder
    XLLoadWave/O/T/R=(C9,M27)/W=8/C=9/P=Science "Instrument Data.xls"

    // Make graph.
    Display M1, M2, M3 vs X_Time
    Label bottom, "Time"; Label left, "Mass"
    ModifyGraph dateInfo(bottom)={1,0,0}
End
```

See also **Loading Excel Data Into a 2D Wave** on page II-162.

XWaveName

`XWaveName(`*`graphNameStr`*`, `*`traceNameStr`*`)`

The XWaveName function returns a string containing the name of the wave supplying the X coordinates for the named trace in the named graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

traceNameStr is the name of the trace in question.

Details

XWaveName returns an empty string (`""`) if the trace is not plotted versus an X wave.

For most uses, we recommend that you use **XWaveRefFromTrace** instead of WaveName. XWaveName returns a string containing the wave name only, with no data folder path qualifying it. Thus, you may get erroneous results if the X wave referred to in the graph has the same name as a different wave in the current data folder. Likewise, if the named wave resides in a folder that is not the current data folder, you will not be able to refer to the named wave.

graphNameStr and *traceNameStr* are strings, *not* names.

Examples

```
Display ywave vs xwave              // XY graph
Print XWaveName("","ywave")         // prints xwave
```

See also

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

XWaveRefFromTrace

XWaveRefFromTrace(*graphNameStr*, *traceNameStr*)

The XWaveRefFromTrace function returns a wave reference to the wave supplying the X coordinates against which the named trace is displayed in the named graph window or subwindow.

Parameters

graphNameStr can be "" to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Details

XWaveRefFromTrace returns a null reference (see **WaveExists**) if the wave is not plotted versus an X wave.

graphNameStr and *traceNameStr* are strings, not names.

Examples

```
Display ywave vs xwave                  // XY graph
Print XWaveRefFromTrace("","ywave")[50] // prints value of xwave at point 50
```

See Also

For other commands related to waves and traces: **WaveRefIndexed**, **TraceNameToWaveRef**, **TraceNameList**, **CsrWaveRef**, and **CsrXWaveRef**.

For a description of traces: **ModifyGraph**.

For a discussion of contour traces see **Contour Traces** on page II-370.

For commands referencing other waves in a graph: **ImageNameList**, **ImageNameToWaveRef**, **ContourNameList**, and **ContourNameToWaveRef**.

For a discussion of wave references, see **Wave Reference Functions** on page IV-197.

See Also

Trace Names on page II-282, **Programming With Trace Names** on page IV-87.

y

y

The y function returns the Y value for the current column of the destination wave when used in a multidimensional wave assignment statement. Y is the scaled column index whereas **q** is the column index itself.

Details

Unlike **x**, outside of a wave assignment statement, y does not act like a normal variable.

See Also

x, **z**, and **t** functions for other dimensions.

p, **q**, **r**, and **s** functions for the scaled indices.

z

z

The z function returns the Z value for the current layer of the destination wave when used in a multidimensional wave assignment statement. z is the scaled layer index whereas **r** is the layer index itself.

Details

Unlike **x**, outside of a wave assignment statement, z does not act like a normal variable.

See Also

x, **y**, and **t** functions for other dimensions.

p, **q**, **r**, and **s** functions for the scaled indices.

zcsr

zcsr(*cursorName* [, *graphNameStr*])

The zcsr function returns a Z value when the specified cursor is on a contour, image, or waterfall plot. Otherwise, it returns NaN.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-92 for details on forming the window hierarchy.

Examples

```
Print zcsr(A)              // not zcsr("A")
Print zcsr(A,"Graph0")     // specifies the graph
```

See Also

The **hcsr**, **pcsr**, **qcsr**, **vcsr**, and **xcsr** functions.

Programming With Cursors on page II-321.

zeta

zeta(*a*, *b* [, *terms*])

The zeta function returns the Hurwitz Zeta function for real or complex arguments *a* and *b*

$$\zeta(a,b)=\sum_{k=0}^{\infty}\frac{1}{(k+b)^a},$$

$$\Re(a)>1,$$

$$b\neq 0,-1,-2,...$$

The Riemann zeta function is the special case:

$$\zeta(a)=\zeta(a,1).$$

The zeta function was added in Igor Pro 7.00.

Parameters

The *terms* parameter defaults to 40. In practice evaluation may terminate before the specified number of terms when convergence is achieved.

References

Olver, Frank W. J.; Lozier, Daniel W.; Boisvert, Ronald F.; Clark, Charles W., eds., "NIST Handbook of Mathematical Functions", 607 pp., Cambridge University Press, 2010.

See Also

Dilogarithm

ZernikeR

ZernikeR(*n*,*m*,*r*)

The ZernikeR function returns the Zernike radial polynomials of degree *n* that contains no power of *r* that is less than *m*. Here *m* is even or odd according to whether *n* is even or odd, and *r* is in the range 0 to 1.

Note that the full circle polynomials are complex. For any angle *t* (theta), they are given by: `ZernikeR(n,m,r)*exp(imt).`

Index

Symbols

(character in menus IV-134
(left parenthesis
  multidimensional wave indexing II-95
  wave indexing II-76
! character in menus IV-134
! operator IV-6, IV-41
!= operator IV-6, IV-41
# operator
  deferred assignment V-1063–V-1064
  deferred assignments IV-233
  in instance names IV-20
  instance names and $ IV-20
  string expressions V-1064
#define IV-108
  example IV-110
  predefined symbols IV-110
#define statement V-16
#if-#elif-#endif statement V-16
#if-#endif statement V-16
#ifdef-#endif statement V-17
  example IV-110
#ifndef-#endif statement V-17
#pragma hide III-402
#undefine statement V-18
$ operator IV-6
  convert string to name IV-6, IV-18
  in dependency formulas IV-234
  in macros IV-121
  in user functions IV-62–IV-63, IV-83
  instance notation IV-20
  load waves (example) II-171
  page layout object names V-477
  precedence IV-19
    tricky case IV-19
  quoting liberal data folder path names II-110
  quoting liberal data folder paths IV-169
  with # operator IV-20
  with data folder paths IV-63
  with NVAR, SVAR, WAVE IV-62
  with window names IV-63
$operator
  examples IV-63
%& operator
  obsolete IV-8
%^ operator IV-6
%~ operator
  obsolete IV-8
%| operator
  obsolete IV-8
%d, %e, %f, %g, %s, etc.
  conversion specifications for printf IV-260, V-770–V-772
%W V-771
& character in menus IV-135
& operator IV-6, IV-41
  pass-by-reference IV-59
&& operator IV-6–IV-7, IV-41
&~ operator IV-6
* (asterisk)
  operator IV-6
  wave indexing II-76
*= operator IV-5
+ operator IV-6
++ operator IV-6, IV-43
+= operator
  accumulate operator IV-5
  string concatenation operator IV-6
- character in menus IV-134
- operator IV-6
-- operator IV-6, IV-43
-= operator IV-5
/ character in menus IV-134
/ operator IV-6
// (comment symbol) IV-2
/= operator IV-5
/C flag
  in user functions IV-31, IV-68
  WAVE reference IV-83, V-1069
/D flag
  in user functions IV-31
/S flag
  in user functions IV-31
/SDFR flag IV-80
/T flag
  in user functions IV-68, IV-83
  WAVE reference V-1069
/Z flag IV-21
^ operator IV-6
:
  conditional operator IV-6–IV-7
  indicating current data folder II-109
; character in menus IV-134
; command separator IV-2
< operator IV-6, IV-41
<<
  shift left operator IV-41
<< operator IV-6, IV-43
<= operator IV-6, IV-41
= operator
  assignment operator IV-5
  string assignment operator IV-6
== operator IV-6, IV-8, IV-41
  and roundoff error IV-8

> operator IV-6, IV-41
>= operator IV-6, IV-41
>>
    shift right operator IV-41
>> operator IV-6, IV-43
?: conditional operator IV-6–IV-7
[left bracket
    wave indexing II-76
[left square bracket
    multidimensional wave indexing II-95
\\ (double backslash)
    insert \ in annotation III-37
~ operator IV-6, IV-41
auto trace
    curve fit residuals III-186, III-219
auto wave
    curve fit residuals III-186, III-220
default
    curve fitting coefficient wave menu III-195
    curve fitting coefficients III-189
free
    default free wave name IV-95
New Wave
    curve fit destination III-197
    curve fit residuals III-220
    curve fitting coefficients wave III-195
none
    curve fit destination III-196
    curve fit epsilon III-193
PictGallery V-827
| (vertical bar character)
    bitwise operator IV-6, IV-41
    obsolete for comments IV-113
|| operator IV-6–IV-7, IV-41
.autosave files II-40
    opening explicitly II-41
.noindex files II-29

Numbers

32-bit version I-2
3D bar plots
    Gizmo II-457
3D scatter plot
    data format II-451
    Gizmo tour II-407, II-411
3D scatter plots
    Gizmo II-455
3D waves
    displaying II-95
    Gizmo II-407
4K displays III-507
    resolution of panels III-456
64-bit integers IV-39
    complex IV-39
64-bit version I-2

A

abbreviated date
    in notebooks III-13
Abort Experiment Load button II-27–II-28
abort key combinations IV-51
Abort operation V-18
aborting
    curve fitting III-180
    debug on abort IV-214, V-146
    from control panel IV-160
    macros IV-124
    network operations IV-270
    procedures IV-49, IV-51, V-18–V-19, V-1047
    progress windows IV-156
    threads IV-337
    user functions IV-112
AbortOnRTE keyword IV-49, V-18
AbortOnValue keyword IV-49, V-19
abs (absolute value) function V-19
abs of a wave V-1090
absolute value V-1090
    abs (real) function V-19
    cabs (complex) function V-58
acausal convolution III-285
accelerators
    in menus IV-135
Accumulate into dest III-128
accumulating histogram III-128
acos function V-19
acos of a wave V-1090
acosh function V-19
active page
    in page layouts II-478
ActiveX Automation IV-265
    as server IV-265
    example experiment IV-265
Adams-Moulton method V-454
adaptive thresholding III-355
Add Annotation dialog III-35–III-47
    dynamic pop-up menu III-38
    for page layouts II-494
    Insert: group III-35
    Special pop-up menu III-36
Add Controls submenu III-419
AddDropLine macro III-77
AddFIFOData operation V-20
AddFIFOVectData operation V-20
addition IV-6
AddListItem function V-20
AddMovieAudio operation V-21
AddMovieFrame operation V-21
AddWavesToBoxPlot operation V-22
AddWavesToViolinPlot operation V-22
ADInstruments data acquisition package IV-312
Adjust Indentation IV-27

- AdoptFiles operation V-23
- adopting
  - all procedure files II-25
  - files II-24
  - files programmatically V-23
  - notebooks II-25, V-696
  - procedure files II-25, III-403
  - unadopting II-25
  - waves II-25
  - waves in Data Browser II-119
- AfterFileOpenHook hook function IV-281–IV-282
  - example IV-284
- AfterMDIFrameSizedHook hook function IV-285
- AfterWindowCreatedHook hook function IV-286
- AIFF files V-892
- airy functions V-24–V-25
- airyA function V-24
- airyAD function V-24
- airyB function V-25
- airyBD function V-25
- alerts
  - from procedures V-165
- aliases
  - CreateAliasShortcut operation V-111
  - creating V-111
- aligning
  - controls III-420
  - drawing objects III-65
  - graphs in page layouts II-490
  - pictures in notebooks III-9
- aligning comments III-409
- alignment
  - in annotation text III-54
- All Files IV-149, IV-151
- all-at-once fit functions III-256
  - MPFXEMGPeak function V-667
  - MPFXExpConvExpPeak function V-668
  - MPFXGaussPeak function V-669
  - MPFXLorentzianPeak function V-670
  - MPFXVoigtPeak function V-671
  - multivariate III-260
- all-at-once functions
  - complex fitting III-249
- alog function V-25
- alpha
  - color blending III-498
  - color table waves II-400
  - Gizmo II-432
  - graphics technology III-507
  - image plots II-400–II-401
  - in Gizmo II-428
  - in graph traces II-298
  - RGBA values IV-13
- alternation
  - in regular expressions IV-183
- ambient light
  - Gizmo II-432
- Ambiguity function III-281
- amplitude compensation in windowing III-278
- analysis III-108–III-175
  - area III-120–III-122
  - convolution III-284–III-286
  - correlation III-286
  - decimation III-135–III-320
    - by omission III-135
    - by smoothing III-136
  - differential equations III-322–III-336
  - differentiation III-120, V-158
  - Fourier Transforms III-270–III-279
  - function plotting III-320–III-322
  - functions
    - plotting III-320–III-322
  - histogram III-125–III-132
  - image operations V-4
  - integration III-120
  - interpolation III-109–III-112
  - level detection III-287–III-290
  - matrix operations V-3, V-7
  - missing values (NaNs) III-112
  - of XY data III-108
  - peak detection III-290–III-292
  - principle component analysis V-739
  - programming III-170–III-175
  - related functions V-8
  - related operations V-3
  - root finding III-338–III-343
  - rotate wave III-316
  - smoothing III-292–III-299
  - sorting III-132–III-135
  - statistics operations V-4
  - unwrap wave III-316
  - wave statistics III-122–III-124
  - windowing III-275–III-279
- anchors
  - in page layouts II-494
  - legends III-40–III-41
  - Simple Text drawing tool III-63
  - tags III-46
  - textboxes III-40–III-41
- AND
  - bitwise operator IV-6–IV-7, IV-41
  - logical operator IV-6–IV-7, IV-41
- Annotation Tweaks dialog III-39, III-41, III-46
- AnnotationInfo function V-25
- AnnotationList function V-26
- annotations III-34–III-53
  - Add Annotation dialog III-35
  - alignment III-54
  - anchors III-40–III-41, III-46
  - AnnotationInfo function V-25
  - AnnotationList function V-26
  - AppendText operation V-35

  as input mechanisms V-25
  axis escape codes III-57
  background color III-39
  backslash characters III-37, III-58
  centering text III-37
  color III-54
  color scale bars III-34, III-47–III-51
  creating III-34, III-52
  deleting III-34, III-45, III-53, V-154
  drawing layer III-68
  duplicating III-34
  dynamic text III-56
    in tags III-38
  elaborate III-51–III-52
  escape codes III-35, III-42, III-53–III-58
  exterior III-40
    affects graph plot area III-41
  font styles III-54
  fonts III-36, III-54
    default III-36
    relative size III-36
    size III-36, III-43
      default III-36
    style (bold, italic, etc.) III-37
  frame III-39
  Gizmo II-444–II-445, II-472
  halo III-39, IV-292
  in graphs III-34–III-53
  in page layouts II-483, II-494–II-496, III-34–III-53
  in style macros II-352
  interior III-40
  LaTeX III-54
  Legend operation V-483–V-484
  legend symbols in III-57
  legends III-42–III-43
    definition III-34
    position III-40–III-41
  markers III-43, III-55
  Modify Annotation dialog III-35
  modifying III-34–III-35, III-53
  names III-39, III-53
  naming rules III-501
  Normal escape code, \M III-37
  opaque III-39
  pictures III-54
  position III-40–III-42, III-46–III-47
    anchors III-40–III-41
    exterior III-40
    frozen III-41, III-46
    in graphs III-40–III-41, III-46–III-47
    in page layouts II-494, III-41
    interior III-40
    offset III-41
  programming with III-52
  quick start III-34
  renaming III-39, III-53
  ReplaceText operation V-801
  rotating III-40
  subscripts III-37, III-54
  superscripts III-37, III-54
  tab characters III-39
  Tag operation V-1017–V-1022
  tags III-43–III-47, III-55
    definition III-34
  TagVal function III-38
  TagWaveRef function III-38
  TeX III-54
  text
    color III-37, III-39
    content III-35, V-35, V-801
    justification III-37
  text info variables III-55–III-56
  TextBox operation V-1024–V-1027
  textboxes
    definition III-34
    position III-40–III-41
  transparency III-39
  wave symbols III-37, III-42–III-43
APMath operation V-27
Append Columns dialog II-242
Append from Layout dialog II-486
Append Objects dialog II-486
Append operation V-29
Append to Layout dialog II-486
Append Traces to Graph dialog II-280, II-349
AppendBoxPlot operation V-30
AppendImage operation V-32
AppendLayoutObject operation V-33
AppendMatrixContour operation V-33–V-40
  example II-367
AppendText operation V-35
AppendToGizmo operation V-35
AppendToGraph operation V-35
AppendToLayout operation V-36
AppendToTable operation V-37
AppendViolinPlot operation V-37
AppendXYZContour operation V-39–V-40
Apple events IV-261–IV-263
  supported events IV-262
    details IV-262
AppleScript IV-263–IV-264
  executing scripts V-205
AppleScripts
  executing Unix shell scripts IV-264
Apply to All
  Copy or Share Wave dialog II-157
arbitrary numeric precision V-27
arc functions
  inverse cosine V-19, V-1090
  inverse hyperbolic cosine V-19
  inverse hyperbolic sine V-42

inverse hyperbolic tangent V-42
inverse sine V-41, V-1090
inverse tangent V-42, V-1090
area
faverage function V-218
faverageXY function V-218
functions compared V-218
Integrate operation V-446
integrate1D function V-448
normalized V-1091
of polygons V-754
PolygonArea function V-750
waveform data III-120
XY area procedures III-122
area function III-121, V-40
subranges III-122
areaXY function III-122, V-41
arithmetic-geometric mean V-295
Arrange Objects dialog II-487
array data
HDF5 II-207
arrow keys
in Gizmo II-424
in tables II-243
moving graph cursors II-320
arrow markers in graphs V-614
arrow tool
Gizmo II-414
in page layouts II-481
selecting, deselecting drawing objects III-63
arrows
double arrow cursor II-285, II-288, II-308
drawing V-838
arrow head shape III-64, V-838
four-arrow cursor II-302
markers in graphs II-294
examples V-625
wind barbs V-614
on tags III-45–III-46
ASCII codes
char2num function V-60
escape codes for IV-14
num2char function V-713
ASCII text encoding III-460
Asian text encodings III-462
asin function V-41
asin of a wave V-1090
asinh function V-42
aspect ratio tool
Gizmo II-414
assignment operators IV-5
assignment statements IV-4–IV-10
data folders IV-3
deferred IV-233, V-1063–V-1064
associativity
of operators IV-6
atan function V-42
atan of a wave V-1090
atan2 function V-42
atanh function V-42
attribute list
Gizmo II-416–II-417
attributes
Gizmo II-419
global II-420
internal II-419
HDF5 II-198
HDF5 Browser II-195
authentication
in URLs IV-268
security IV-270
URLRequest operation V-1054
auto
general text tweaks II-141
auto-compiling procedures III-404, IV-26
auto-correlation III-287, V-107
auto-identify column type II-129
auto-save
in notebooks V-698
auto-trace destination waves in curve fitting III-196, III-234
Auto/Man Ticks tab II-308
automatic Igor update II-4
automatic multithreading
MultiThreadControl operation V-674
automatically multithreaded
MultiThreadingControl operation V-674
automation
Apple events IV-261
AppleScript IV-263
generating notebook commands III-29
loading files in folder (example) II-175
loading waves II-170
example II-171, II-173
notebooks III-25–III-30
examples III-26
overview I-8
retrieving text from notebooks III-28
Unix shell scripts IV-264
updating in notebooks III-28
AutoPositionWindow operation V-42
autosave II-36–II-42
.autosave files II-40
backing up II-37
data folders II-37
direct mode II-38
experiment files II-37
IgorInfo function V-361
indirect mode II-38–II-39
notebook files II-37
packed experiments II-37
procedure files II-37

  - Run Autosave Now II-39
  - unpacked experiments II-37, II-41
  - waves II-37
- autosave modes II-38
- Autoscale Axes in Graph Menu II-285
- autoscaling II-285, II-287, II-302, II-348
  - Gizmo II-421
  - log axes II-302
- Autosizing Columns By Double-Clicking II-252
- Autosizing Columns Using Menus II-253
- auxiliary procedure files III-396, III-398
- average deviation of wave V-415, V-1084
  - 2D V-415
- average functions
  - compared V-218
  - faverage V-218
  - faverageXY V-218
  - FindSegmentMeans user-defined function III-174
  - mean V-586
- average of several waves
  - WavesAverage user-defined function III-173
- average value of a wave III-120
- avg3D filter V-372
- AVI movies
  - NewMovie operation V-683
  - related operations V-6
- axes II-278–II-279, II-284–II-288, II-306–II-319, II-324–II-347
  - adding to graph II-280
  - AppendToGraph operation V-35
  - AppendWithCapturedAxis macro II-349
  - auto ticks II-308
  - autoscaling modes II-287
  - AxisInfo function V-43
  - AxisLabel function V-44
  - AxisList function V-44
  - AxisValFromPixel function V-45
  - bottom axis II-279
  - category axis V-43
  - category plots
    - bar gaps V-627
    - category gaps V-627
  - color II-308, V-634
  - computed manual ticks II-308
  - contour plot V-85
  - controlling wave II-279, II-318, II-357, V-43
  - coordinates for drawing III-68
  - crossing II-308
  - curve fit residuals III-219
  - date/time II-307, II-315–II-317
    - custom formats II-316
    - manual ticks II-317
    - suppress date II-316
  - drag and drop II-280
  - dragging II-288, II-307–II-308
  - draw between (examples) II-307, II-324, II-326
  - drawing layer III-68
  - exponential labels II-308
    - mode II-309
    - tick marks II-308
  - fake (custom axes) II-317
  - fills V-632
  - flags in Display operation V-161
  - free axis II-278–II-279, II-311
    - creating V-470, V-679
    - crossing at II-308
    - distance II-308
    - examples I-36, II-279, II-324, II-326
    - killing V-470
    - mirror axis II-307
    - modifying V-609
  - GetAxis operation V-295
  - hiding II-308
  - HorizCrossing free axis II-278–II-279
  - image plot V-381
  - KillFreeAxis operation V-470
  - labels
    - margin II-311
    - on mirror axes II-307
    - position II-311
    - scaling II-308
    - units II-279
  - left axis II-279
  - log axes II-308–II-309, II-315, II-319
    - curve fitting auto-trace III-196
    - minor ticks II-315
      - labels II-315
    - unexpected behavior II-315
  - log or linear axes II-307
  - LogLin mode II-315
  - low trip, high trip II-308
  - manual ticks II-312–II-314
  - minor ticks II-308
    - on log axes II-315
  - mirror axis II-307–II-308
  - ModifyFreeAxis operation V-609
  - ModifyGraph operation V-626, V-634
  - modifying II-306–II-319, V-626
  - moving II-288, II-307–II-308
  - multiple axes (example) I-36, II-324–II-325
  - NewFreeAxis operation V-470, V-679
  - not listed in dialogs II-357
  - offset II-288, II-307
  - PixelFromAxisVal function V-743
  - preferences II-349, II-363, II-381, II-404
  - probability II-317
  - range II-286–II-288
    - GetAxis operation V-295
    - SetAxis operation V-835
  - reciprocal II-317
  - removed automatically II-279

reversed II-287, V-835
right axis II-279
scaling II-285–II-288, II-319
scaling using mouse wheel II-286
SetAxis operation V-835
shortcuts II-353–II-354
split axes II-347
stacked axes V-626
example I-39
standard axes II-279
standoff II-289, II-307, II-326
ignored II-307
styles II-350–II-352
swap X & Y axes II-288
tags V-1017–V-1022
thickness II-308
thousands separators V-633
TickWavesFromAxis V-1034
top axis II-279
transforming II-317
units II-279, II-318–II-319, V-43
user ticks from waves II-308
VertCrossing free axis II-278–II-279
WMAxisHookStruct V-609
zero line II-310
Axes Properties dialog
Gizmo II-450
axis coordinate system
Gizmo II-421
axis cue
Gizmo II-415
axis cue objects
Gizmo II-448
Axis Label tab in Modify Graph dialog II-318
axis labels II-318–II-319
AxisLabel function V-44
escape codes II-318, III-57
exponential notation II-308
fonts II-308, II-318, III-54
size II-318
formatted text III-53
Gizmo II-471
hiding II-312
Label operation V-474
marker symbols II-319, III-37
parentheses omitted II-319
position II-311
preferences II-349
special characters II-319, III-37
subscripts II-318
superscripts II-318
text color II-308
trace symbols II-318
units II-318–II-319
Manual Override escape code II-319
prefix II-319
scaling II-319
x10^n prevention II-319
axis objects
Gizmo II-449
Axis Range tab II-286–II-288
Axis Range tab in Modify Axis dialog
Quick Set buttons II-287
axis relative drawing III-67
Axis tab II-307–II-308
axis tool
Gizmo II-414
AxisInfo function IV-173, V-43
AxisLabel function V-44
AxisList function V-44
AxisValFromPixel function V-45
axsize V-318
axsizeDC V-318

B

B-spline surfaces V-418
back projection V-418
background color
in graphs III-442
in notebooks III-6
in panels III-443
in procedure windows III-405
background removal
DSPDetrend operation V-182
background tasks IV-319–IV-323
BackgroundInfo operation V-45
caveats IV-321
CtrlBackground operation V-118
CtrlNamedBackground operation V-119
debugger IV-321
demo IV-323
dialogs IV-321
errors IV-321
example IV-319, IV-321, IV-323
exit code IV-319
FakeData function V-215
KillBackground operation V-468
limitations IV-320
multitasking
example IV-335
old techniques IV-323
period IV-320
preemptive
example IV-335
SetBackground operation V-836
SetIdlePeriod operation V-848
SetProcessSleep operation V-852
terminating IV-319
tips IV-321
BackgroundInfo operation IV-323, V-45
backing up

  - autosave II-37
- backslash
  - escape sequences IV-14
  - Unicode escape sequences IV-15
- backslash characters
  - in annotations III-37, III-58
  - in file paths III-451
    - procedures III-455
  - in strings IV-14
  - LoadWave operation II-150
  - Shift JIS III-484
- Backwards Differentiation Formula V-454
- band pass filters III-312
- band stop filters III-312
- bar graph
  - with waveforms II-295
  - with XY pairs II-295
- bar limits for ValDisplays III-419, III-435
- Bartlet window function V-225
  - for images V-436
- Bartlett window function V-183, V-225, V-1097
  - for images V-436
- Base64Decode function V-45
- Base64Encode function V-46
- batch files
  - calling Igor from IV-265
- Battle-Lemarie wavelet transform V-189
- beams
  - definition V-420
  - extracting V-420
- Beep operation V-46
- BeforeDebuggerOpensHook hook function IV-284
- BeforeExperimentSaveHook hook function IV-286
- BeforeFileOpenHook hook function IV-287
  - example IV-288
- BeforeUncompiledHook hook function IV-289
- BEGIN keyword in Igor Text files II-151
- Bessel functions V-46–V-47, V-897–V-898
- besseli function V-46
- besselj function V-46
- besselk function V-47
- bessely function V-47
- bessI function V-47
- beta function V-49
- betai function V-49
- bevels
  - for lines and traces III-496
- Bezier curves
  - BezierToPolygon operation V-50
  - editing III-70–III-71
  - even-odd rule III-72
  - fill rules III-72
  - Text2Bezier operation V-1023
  - winding rule III-72
- BezierToPolygon operation V-50
- BigWaveLoadMode II-35
- bilevel
  - TIFF files
    - II-158
- bilinear interpolation V-384
- bimodal thresholding III-355
- binary data
  - in strings IV-175, V-1000, V-1071
  - in text waves III-475–III-477
- binary files
  - description of II-126
  - FBinRead operation V-219
  - FBinWrite operation V-220
  - FReadLine operation V-261
  - GBLoadWave operation II-166, V-291
  - Igor binary wave files II-154–II-157
  - PadString function V-732
- binary operators IV-6
  - AND IV-7
  - OR IV-7
  - XOR IV-7
- BinarySearch function V-51
- BinarySearchInterp function V-51
- binomial function V-52
- binomial smoothing III-294
- binomialln function V-52
- binomialNoise function V-52
- bit streams
  - random V-485
- bitfields
  - HDF5 II-211
- bitmaps
  - exporting graphics III-97, III-104
  - PNGs III-97
- bits
  - setting IV-12
- bitwise operators IV-6, IV-41
  - << IV-41
  - >> IV-41
  - AND IV-7, IV-41
  - complement IV-6, IV-41
  - OR IV-7, IV-41
  - XOR IV-7
- Blackman window function V-183, V-225, V-1097
  - for images V-436
- blanks
  - entering in tables II-247
  - in delimited text files II-129, II-137
- blocks
  - in general text files II-138
  - in Igor Text files II-151
- blurry images in PDFs III-96, III-103
- BMP files
  - exporting V-405
  - importing V-390
- BMPs
  - exporting graphics III-103

  - importing III-509
  - loading II-158
- bottlenecks IV-340
  - GetRTLocation V-313
  - GetRTLocInfo V-314
- bounce end effect method III-299
- bounding box
  - page layouts II-489
- bounding sphere V-53
- boundingBall operation V-53
- box axes
  - Gizmo II-449
- box objects
  - Gizmo II-440
- box plots II-330–II-337, II-345
  - AddWavesToBoxPlot operation V-22
  - AddWavesToViolinPlot operation V-22
  - appearance II-335
  - AppendBoxPlot operation V-30
  - creating V-30
  - fences II-332
  - hinges II-332
  - markers V-596
  - median II-331
  - ModifyBoxPlot operation V-593
  - outliers II-334, V-594
  - per dataset settings II-342
  - per-point settings II-343
  - quartiles II-331
  - trace names II-330
  - Tukey II-331
  - whiskers II-331–II-332, V-595
- box smoothing III-295, V-878
- BoxSmooth operation V-53
- break
  - in for loops IV-48
  - in switch statements IV-44
- break keyword V-54
- breakpoints
  - setting in debugger IV-213
- Brent’s method V-725
- Bring to Front II-481
- broken dependency objects IV-233
- Browse Waves dialog II-72
- Browser XOP II-72
- BrowseURL operation V-54
- browsing
  - experiments II-21
  - waves II-72, II-228
- BuildMenu operation IV-130, V-54
- built-in fit functions
  - notes III-206–III-212
- Bulirsch–Stoer method V-454
- bulldozer icon (what it is) III-65
- Burt-Adelson wavelet transform V-189
- bush icon (it’s a tree, really) III-65
- Butterworth filters III-312
- Button Control dialog III-420
- Button controls
  - custom III-422
  - custom examples III-422
  - examples III-421
  - programming III-420–III-423
  - using III-415
- button controls
  - aborts IV-160
- Button operation III-420–III-423, V-55
- ButtonControl subtype IV-205
  - keyword V-57
- .bwav extension II-155
- byte order marks III-471
  - FReadLine operation V-263
  - in notebooks V-697

C

- C language IV-208
  - for external functions and operations IV-209
- cabs (complex absolute value) function V-58
- Calculate Normals
  - Gizmo II-432
- cameras
  - data acquisition IV-312
  - GetCamera operation V-296
  - ModifyCamera operation V-600
  - NewCamera operation V-678
- CameraWindow subtype IV-205
  - keyword V-58
- Canny edge detector V-370
- canonic tick defined II-312
- canonical equivalence
  - Unicode V-693
- Capture Graph Preferences dialog II-348
- Capture Layout Preferences dialog II-498
- Capture Notebook Preferences dialogs III-30
- Capture Procedure Preferences dialog III-407
- Capture Table Preferences dialog II-272
- CaptureHistory function V-58
- CaptureHistoryStart function V-58
- carriage return
  - in data files II-129
  - in delimited text files II-126, II-135
  - in Igor Text files II-154
  - in tables II-258
  - saving text waves II-180
- carriage returns
  - cross-platform issues III-453
  - escape code for IV-14
- carriage-return/linefeed
  - in data files II-129
  - in delimited text files II-126
- cartesian coordinates

conversion from crystallographic V-1090
conversion to crystallographic V-1091
Case Sensitive III-24
case statements IV-43
Constant keyword V-85
default keyword V-146
Strconstant keyword V-996
catch keyword V-58
category plots II-356–II-364
Add to next II-360–II-361
axis preferences II-363
axis range II-359, II-362
bar gaps II-358, V-627
bars disappear II-363
bars don’t line up II-361–II-362
bars don’t stack correctly II-363
bars on left or right II-359
category gaps II-358, V-627
combining with numeric II-362
creating II-356–II-357, V-161
example I-30
Draw to next II-360, II-363
drawing order II-359
horizontal bars II-359
Keep with next II-361
modifying II-357–II-360
New Category Plot dialog II-364
numeric categories II-362
options I-31
pitfalls II-362–II-363
preferences II-363–II-364
axes II-363
wave styles II-364
programming (example) II-356
Stack on next II-363
stacked bar charts II-360
tick in center II-358
tick mark labels II-358
using dimension labels II-357
vs numeric categories II-362
wave style preferences II-364
X scaling II-359, II-362
Catmull-Clark surfaces V-418
cd operation V-58
CDFFunc keyword V-948
CDFFunc subtype IV-205
ceil function V-59
centerOfMass function V-59
centerOfMassXY function V-59
cequal function V-60
Change Wave Scaling dialog II-68–II-69
Delta II-69
End II-69
Right II-69
SetScale Mode II-69
Start II-69
changeableByCommandOnly
in notebooks III-11
channels
in FIFO buffers IV-313
char2num function V-60
character classes IV-178
character encoding schemes III-462
character properties
in notebooks III-8
character set
cross-platform issues III-453
character-by-character operations IV-173
characters versus bytes III-483
chart controls III-415, IV-317–IV-319
creating III-423
live mode IV-314
programming IV-313–IV-319
review mode IV-314
scrolling IV-314
updates IV-316
using IV-313
Chart operation V-61–V-63
chebyshev function V-63
chebyshevU function V-64
Checkbox controls
custom III-424
programming III-424
using III-416
CheckBox operation III-424, V-64
CheckBoxControl subtype IV-205, V-66
keyword V-67
CheckDisplayed operation V-68
CheckName function V-68
chi-square
weighting wave III-199
ChildWindowList function V-69
Chinese
text encodings III-462
Cholesky decomposition V-555
Choose Dimensions dialog II-263
ChooseColor operation V-69
chunking II-214
HDF5 default compression II-215
chunks
in multidimensional waves II-92, II-95
in tables II-263
indexing
chunk numbers, s function V-811
scaled index, t function V-1011
maximum location V-1085
minimum location V-1085
Ci function V-109
circular convolution III-285
circular correlation V-107
CleanupName function V-70
Clear Cmd Buffer II-9

- clearing
  - drawing objects III-73
  - in tables II-247
- Clipboard
  - Export Graphics dialog III-98, III-104
  - importing data II-248
  - in SavePICT operation V-827
  - loading PICTs V-506–V-508
  - playing sound V-746
  - programming
    - GetScrapText operation V-315
    - PutScrapText operation V-785
  - searching V-327
- clipboard
  - exporting data II-252
- clipping
  - Gizmo II-421
  - troubleshooting in Gizmo II-470
- Close Notebook Window dialog III-4
- Close operation IV-195, V-71
- Close Procedure Window dialog III-399
- Close Window dialogs II-47, II-350
- CloseHelp operation V-71
- CloseMovie operation V-71
- CloseProc operation V-72
- closest integer function V-255
- closing
  - graphs II-350
  - help windows II-3
  - page layouts II-477
  - procedure windows III-399
  - tables II-241
  - windows II-46
- clustering
  - distance map V-259
  - FPClustering operation V-259
  - HCluster operation V-338
- CM_Kn: covariance matrix waves V-125
- cmap conversions
  - cmap2rgb V-418
- cmplx function V-73
- CmpStr function V-73
- CMYK III-99, III-105
- CMYK conversions
  - CMYK2RGB V-418
- CMYK images
  - TIFF files II-158
- co-occurrence matrix V-375
- Cochran’s (Q) test V-923
- code marker comments III-408
- code point III-462
- codecs
  - NewMovie operation V-683
- coefficient of correlation III-226, V-125
- coefficients
  - coefficients wave in curve fitting III-195
  - curve fitting III-179
  - filtering III-300–III-301, III-306, III-308, III-311
- coercion
  - in user functions IV-111
- Coifman wavelet transform V-189
- color
  - control panel background V-640
  - dialog V-69
  - direct color mode V-621
  - drawing environment V-839
  - Export Graphics dialog III-98, III-104
  - Gizmo II-428
  - in annotations III-54
  - in contour plots II-371–II-373, V-602–V-605
  - in graphs V-634
    - axes II-308
    - axis label text II-308
    - control bar V-634
    - cursors II-320
    - traces V-618–V-619, V-621
      - direct color mode V-621
  - in image plots II-392–II-402, V-636
  - in notebooks III-8
    - background V-696
    - text V-705
  - in procedure windows III-406
  - legends II-301
  - of annotation background III-39, V-1025
  - of annotation text III-37, III-39, V-1026
  - of markers V-621
  - of text in tables V-648
  - RGBA values IV-13
  - Save EPS File dialog III-98, III-105
  - Save PICT File dialog III-98, III-105
  - selecting V-69
- color blending III-498
  - RGBA values IV-13
- color index wave II-372–II-373, II-400–II-401
  - example II-372, II-401
  - from color table V-82
  - in contour plots II-371–II-373
  - in image plots II-400–II-401, V-636
- color material
  - Gizmo II-429
- color palette III-497–III-498
  - in pop-up menu controls V-766
  - Other... III-497
  - Recent Colors III-497
- color scale bars III-34, III-47–III-51
  - axis ticks III-51
  - ColorScale operation V-74
  - dialog III-48
  - dimensions III-50
  - Gizmo II-472
  - height III-49
  - lateral offset III-50

margins III-50
orientation III-48
pasting in layouts II-494
position III-48
rotation III-50
settings III-48
size III-48
units III-50
user-defined ticks III-51
width III-49
color space conversion
cmap
to RGB V-418
CMYK
to RGB V-418
grayscale V-418
HSL
to RGB V-421
RGB
to grayscale V-424
to HSL V-424
to i123 V-424
to XYZ V-424
XYZ
to RGB V-428
color table waves II-399
alpha II-400
opacity II-400
translucency II-400
transparency II-400
color tables II-393–II-400
automatic color mapping II-372
color effects II-395
ColorTab2Wave operation V-82
compatibility II-395
first half, using II-372
gamma II-395
Gizmo II-431
gradients II-396
Igor and IgorRecent V-82
Igor Pro 4 II-396
Igor Pro 5 II-396
Igor Pro 6 II-398
Igor Pro 6.2 II-399
in contour plots II-371
in image plots II-393, V-636
listing V-118
manual color mapping II-372
names of built-in tables II-393
number of colors II-393
overlays II-393
reversing the color order II-392
RGB waves II-400
RGBA waves II-400
specialized II-395, II-397
color tables contrast effects II-395
color waves
Gizmo II-430
COLORPOP V-766
ColorScale operation V-74
colorscales
deleting V-154
Gizmo II-445
ColorTab2Wave
Gizmo II-429
ColorTab2Wave operation V-82
COLORTABLEPOP V-766
COLORTABLEPOPNONAMES V-766
column formats
in delimited text files II-129
column indices
in tables II-261, V-645
column labels
in delimited text files II-131, II-134, II-136
in general text files II-138, II-140–II-141
troubleshooting II-137
column major II-204
column names
can use index number instead V-645
for multidimensional waves II-261
pasting in tables II-240
column position waves
in delimited text files II-134
column styles preference II-272
column types
in delimited text files II-134, II-136
column-major order II-99, V-294
columns
.i, .d, .id, .real, .imag V-192
.x, .y, .xy V-192
appending to tables II-242
AppendToTable operation V-37
changing
positions II-252
styles II-253
widths II-252
extracting V-420
flipping V-419
formats II-249, II-255
in multidimensional waves II-92, II-95
column numbers, q function V-785
scaled indices, y function V-1119
index columns II-241, II-249
list of V-1075
maximum location V-1084
minimum location V-1084
names II-241
pasting column formats II-249
pasting index columns II-249
removing from tables II-242, V-794
showing X values II-235
specifying by index number V-645

titles in tables II-254
tooltips II-3
X columns II-241
comma-separated values
in delimited text files II-135
command buffer II-8–II-13, IV-2
searching II-11
command line II-8–II-13
command line completion IV-3
copying commands to V-1040
length limit II-8, IV-2
magnification II-53
searching II-11
Silent operation V-871
zooming II-53
command line completion IV-3
command line interface I-6
Command Settings III-4
command window II-8–II-13
background color II-12
Clear Cmd Buffer II-9
getting help II-12
magnification II-12
overview I-6
preferences II-12
shortcuts II-13
text format II-12
title II-9
Command-dot
aborting procedures IV-51
Command/History Settings II-12
commands IV-2–IV-21
assignment operators IV-5
assignment statements IV-4–IV-10
command line completion IV-3
comments IV-2
executing from
help windows II-3, III-4
notebooks III-4
procedure windows III-4
executing from Data Browser II-118
functions IV-11
line continuation IV-2
max characters per line II-8, IV-2
multiple IV-2
notebooks actions III-14–III-18, V-711, V-893, V-895
operands IV-8
operations IV-10
operators IV-6
overview II-8
parameters IV-2, IV-11–IV-21
relation to menus I-7
ToCommandLine operation V-1040
tooltips for IV-3
types IV-4–IV-11
use of commas IV-11
commas
as decimal separator in graphs II-288
in commands IV-11
in delimited text files II-131
in general text files II-138
commentizing III-408
Comments
using obsolete style V-871
comments IV-2
aligning III-409
code marker comments III-408
obsolete style V-871
vertical bar character obsolete IV-113
comparison functions
CmpStr V-73
equalWave V-196
limit V-485
max V-586
min V-589
SelectNumber V-834
SelectString V-834
comparison operations
Extract V-213
comparison operators IV-7
roundoff error IV-8
compatibility
Gizmo II-470
Igor6 and Igor7 III-480
long object names III-503
compatibility mode
Silent operation V-871
turning off V-811
compatible characters
Igor6 and Igor7 III-480
Unicode V-693
Compile button III-397, IV-26
compile time vs runtime IV-64
COMPILEPROCEDURES IV-279
compiling IV-26
auto-compiling IV-26
help files IV-257
procedures III-397, IV-26
complement operator IV-6, IV-41
complementary error function V-198
complex
64-bit integers IV-39
columns in tables II-241
curve fitting III-248
variable in user functions IV-112
waves in graphs III-273
complex conjugate V-1090
complex conjugate function V-84
complex declaration V-82
local variables in user functions IV-112
variables in user functions IV-112

complex functions V-7
all-at-once functions curve fitting III-249
cabs V-58
cequal V-60
cmplx V-73
conj V-84
cpowi V-111
curve fitting III-248
fresnelCS V-264
imag V-362
magsqr V-525
multivariate curve fitting III-248
p2rect V-732
r2polar function V-786
real V-788
user functions IV-31
complex keyword V-82
complex number type II-66
complex variables II-104, V-82, V-1065
complex waves II-81
accessing IV-68, IV-83, V-1069
assignments IV-68, IV-83, V-1069
curve fitting III-248
in graphs II-303
in Igor Text files II-151
in user functions IV-64, IV-68, IV-83, IV-111, V-1069
inverse V-541
statistics V-1082
Compose Expression dialog III-137
table selection III-138
compositing
ImageComposite operation V-368
compound datatypes
HDF5 II-212
compressing images V-418
compression factor
movies V-683
Concatenate operation V-82
concatenating
strings IV-8
waves II-80, V-82
condition
matrix V-531
conditional compilation IV-108
#define statement V-16
#if-#elif-#endif statement V-16
#if-#endif statement V-16
#ifdef-#endif statement V-17
#ifndef-#endif statement V-17
#undefine statement V-18
IgorVersion function V-362
predefined symbols IV-110
Conditional Compilation Examples IV-110
conditional operator II-82, IV-6–IV-7
conditional statements
if-else-endif IV-40
if-elseif-endif IV-40
in macros IV-120
in user functions IV-40
precedence of operators IV-41
conditionals in procedures
conditional operator IV-7
confidence bands
for curve fitting III-223, III-226
confidence intervals
curve fit coefficients III-223
conj function V-84
constant
numeric declaration IV-51
Constant keyword V-85
constant keyword
in switch statements IV-44
constants IV-51
Constant keyword V-85
GL_ constants II-471
in curve fitting III-189
in switch statements IV-44
in user functions IV-51
static IV-51
Static keyword V-906
Strconstant keyword V-996
syntax IV-51
constraints
sum of fit functions III-245
constraints in curve fitting III-227, III-232
context
loading text files II-133
context-sensitive help
for controls III-436
contextual menus
contextualmenu IV-127
controls V-756
DataBrowserObjectsPopup menu IV-138
dynamic IV-127
example IV-162
GraphPopup menu IV-138
TablePopup menu IV-138
WindowBrowserWindowsPopup menu IV-140
contextualmenu IV-127
continuation characters IV-2
in user functions IV-35, IV-113
continue keyword V-85
continue statements
for loops IV-48
continuous phase III-316
contour fills II-373
Contour Labels dialog II-378
contour level not closeable II-374
contour plots II-366–II-383
appearance preferences II-380
Append Contour Plot dialog II-381

AppendMatrixContour operation V-33–V-40
AppendXYZContour operation V-39–V-40
axes
GetAxis operation V-295
SetAxis operation V-835
axis preferences II-381
boundary II-370
change updating V-606
color II-369, II-371–II-373, V-602–V-605
color index wave II-372–II-373
ColorTab2Wave operation V-82
programming (example) II-372
color scale bars II-377, III-34, III-47–III-51, V-74
color tables II-372
ColorScale operation V-74
combining with image plots II-367
contour fills II-373
contour instance names II-376–II-377
Contour Labels dialog II-378
contour lines (see contour traces)
contour traces II-370–II-376
appearance II-370
color II-371–II-373
cursors II-375
drawing order II-376
extracting XY data II-376
instance names II-371
list of V-1044
names II-371
removing II-374
updating II-375
wave reference V-1045
Z value II-375
ContourInfo operation V-85
ContourNameList function V-86
ContourNameToWaveRef function V-87
creating II-367–II-368, V-33, V-39, V-162
crossing lines II-380
cursors II-375
z value V-1120
data
gridded II-366
matrix II-366
nonlinear grid II-366
sparse II-367
XYZ II-367
Delaunay triangulation II-370
DelayUpdate and DoUpdate II-375
drawing order II-376, II-379
FindContour operation V-239
gridded data II-366
interpolation II-370, II-379, V-87
labels II-377–II-379
color II-378
content II-378
coping with II-378
drawing order II-379
format II-378
ModifyContour operation V-603–V-604
modifying II-378–II-379
positioning II-378
removing II-378
rotation II-377–II-378
Tag/Q purpose II-379
turning off II-369
updating II-369
layers II-376, II-379
legends II-377, III-34, III-47–III-51
levels II-368, II-374
arbitrary II-369
automatic II-368
manual II-369
ModifyContour operation V-604
More Levels dialog II-369, II-374
stored in a wave II-369
Line Colors dialog II-369, II-371–II-372
lines (see contour traces)
log colors II-373, V-604
manual color mapping II-372
matrix data II-366
Modify Contour Appearance dialog II-368–II-370
ModifyContour operation V-600–V-606
modifying II-368–II-370
More Levels dialog II-369
multiple plots in graph II-376
names II-371, II-376–II-377
in legends II-371
NaNs V-605
New Contour Plot dialog II-381
pitfalls II-379–II-380
preferences II-380–II-381
programming notes II-371–II-372, II-375–II-377
references II-382
related operations V-1
removing V-791
ReplaceWave operation V-801
resolution II-370, II-379
shortcuts II-382–II-383
sparse data II-367
TraceInfo function V-1042
traces (see contour traces)
treatment of NaNs V-605
triangulation II-370
updating II-369, II-375
Voronoi triangulation V-602
XY markers II-370
XYZ data II-367
Z level (see levels)
Z value II-375
ContourInfo operation V-85
ContourNameList function V-86

- ContourNameToWaveRef function IV-198, V-87
- contours
  - FindContour operation V-239
- ContourZ function V-87
- control bar in graphs III-441
  - color V-634
  - ControlBar operation V-88
- control bars
  - expansion factor III-444
- control characters
  - in tables II-260
- control panel units III-444
  - NewPanel operation III-444
- control panels
  - scrolling mode V-319, V-866
- Control Procedure dialog III-420
- control structures III-437
  - eventMod field III-438
- ControlBar operation V-88
- ControlInfo operation V-89–V-94
- ControlNameList function V-94
- controls III-414–III-446
  - aborts IV-160
  - action procedures III-415, III-420–III-429, III-441–III-442, V-57, V-66, V-137, V-763
    - independent modules IV-240
    - regular modules IV-237
    - sliders III-430
  - background color III-437
  - blue bar III-419, III-435
  - Buttons III-415, III-420–III-423
    - Button operation V-55
    - user data V-57
    - WMButtonAction structure V-57
  - charts III-415, IV-313–IV-319
    - Chart operation V-61–V-63
  - CheckBoxes
    - user data V-66, V-136
    - WMCheckboxAction structure V-66
  - Checkboxes III-416, III-424
    - CheckBox operation V-64
  - color III-437
  - Color pop-up menu V-766
  - context-sensitive help III-436
  - control bar III-441
    - example I-41
  - control panel units III-444–III-445
  - control structures III-437
  - ControlBar operation V-88
  - ControlInfo operation V-89–V-94
  - ControlNameList function V-94
  - ControlUpdate operation V-94
  - copying III-420
  - custom
    - WMCustomAction structure V-136
  - CustomControl III-416, III-424
    - examples III-424
  - CustomControl operation V-134
  - CustomControls
    - WMCustomControAction structure V-137
  - data folders II-111
  - default appearance V-147
  - default font V-149
  - deferred compilation III-428, V-1064
  - disabling III-436
  - duplicating III-419
  - eventMod field III-438
  - Fill Pattern pop-up menu V-766
  - format strings III-420, V-855
  - formatted text in titles III-53
  - global variables
    - deferred evaluation III-428, V-1064
  - graphic elements
    - GroupBox III-416, III-426
    - TitleBox III-418, III-433
  - group boxes V-334
  - GroupBox III-416, III-426
    - user data V-336
  - hiding III-436
  - Igor Tips III-436
  - in graphs III-441
    - drawing limitations III-442
  - killing III-435
    - KillControl operation V-468
  - labelBack keyword III-437
  - Line Style pop-up menu V-766
  - list boxes V-488
  - ListBox III-416, III-426
    - help V-493
    - user data V-492
    - WMListboxAction structure V-494
  - Marker Style pop-up menu V-766
  - ModifyControl operation V-606
  - ModifyControlList operation V-608
  - modifying III-414, III-419, V-606, V-608
  - moving and resizing III-63
  - naming rules III-501
  - points versus pixels III-445
  - pop-up menus III-416, III-426–III-428
    - contextual menus V-756
    - PopupContextualMenu operation V-756
    - PopupMenu operation V-761
    - user data V-763
  - programming III-419–III-442
    - syntax III-420
  - radio buttons III-416, III-424
  - red bar III-419, III-435
  - related operations V-6
  - resolution of III-456
  - selecting III-419–III-420
  - Set Variable III-417, III-428–III-429
    - SetVariable operation V-854–V-1064

WMSetVariableAction structure V-858
SetVariable
user data V-857
shortcuts III-446
size of III-443
Slider operation V-874
Sliders III-418, III-429
user data V-876
WMSliderAction structure V-876
sliders
repeating III-418, V-875, V-877
status line help III-436
structures for III-437
tab controls V-1011
tabbed panes III-418, III-431
TabControl III-418, III-431
creating III-431
user data V-1013
WMTabControlAction structure V-1013
tabs V-1011
TitleBox III-418, III-433
titles III-418, III-433, V-1038
updating III-436
ControlUpdate operation V-94
updating problems III-442
user data III-440, V-90, V-320
examples III-440
using III-321, III-415
example I-41
ValDisplays III-418
Value Displays III-433–III-435, III-442
ValDisplay operation V-1060–V-1064
ControlUpdate operation V-94
conversion functions V-7
char2num V-60
cmplx V-73
date2secs V-145
dateToJulian V-144
imag V-362
IndexToScale function V-442
JulianToDate V-468
LowerStr V-523
magsqr V-525
num2char V-713
num2istr V-713
num2str V-714
number-to-string V-902
p2rect V-732
pnt2x V-748
r2polar V-786
real V-788
ScaleToIndex function V-832
Secs2Date V-833
Secs2Time V-833
str2num V-996
UpperStr V-1053
x2pnt V-1114
conversion specifications in printf IV-260, V-770–V-772
Convert to UTF-8 Text Encoding dialog III-485
ConvertGlobalStringTextEncoding operation V-95
ConvertTextEncoding function
example III-479
ConvertTextEncoding operation V-98
ConvexHull operation V-101
convolution III-284–III-286, V-101–V-103
acausal III-285
circular III-285
curve fitting III-256
delay III-285
end effects III-285
FilterFIR operation V-230
for smoothing III-298
linear III-285
number of points in result III-285
Resample operation V-803
SmoothCustom operation V-883
Convolve dialog III-285
Convolve operation V-101–V-103
COO format sparse matrix III-152
cool graphs II-5
coordinate system
Gizmo II-420
coordinate systems for drawing III-66–III-68
SetDrawEnv operation V-842
coordinate transformation
Gizmo II-436
coordinates
cartesian to crystallographic conversion V-1091
crystallographic to cartesian conversion V-1090
window III-455
Copy or Share Wave dialog II-156
Apply to All II-157
copy to home II-24, II-155
CopyDimLabels operation V-103
CopyFile operation V-104
CopyFolder operation V-105
warning V-105
copying
data folders V-187
data from tables II-240, II-247, II-252, II-269
data objects with Data Browser II-115
from Clipboard V-315
from page layouts II-493
Igor binary wave files II-24
in page layouts II-482
rulers III-12
to Clipboard V-785
waves V-185–V-186
CopyScales operation V-107
Core Graphics
graphics technology III-507, V-866

Correlate dialog III-287
Correlate operation V-107–V-109
correlation III-286, V-107–V-109, V-532
  auto-correlation III-287, V-107
  circular correlation V-107
  cross-correlation III-287
correlation matrix for curve fit III-226
cos function V-109
cos of a wave V-1090
Cos window function V-225
cosh function V-109
cot function V-110
coth function V-110
CountObjects operation V-110
CountObjectsDFR operation V-111
covariance V-532
  cross-covariance V-556
covariance matrix
  curve fitting III-226, V-125
cpowi function V-111
crashes IV-340–IV-341
  crash log file IV-341
  Diagnostics folder IV-341
  minidump files IV-341
Create Line dialog
  Arrow Fat III-64
Create Rectangle dialog
  Fill Mode III-64
Create Text dialog III-63
create-paste II-248
CreateAliasShortcut operation V-111
CreateBrowser operation V-113
  modal Data Browser II-120
CreateDataObjectName function V-113
creating
  control panel V-687
  data
    in tables II-234, II-239, II-248
  data folders V-678
  derived rulers III-12
  experiments II-21
  files V-716
  free data folders V-680
  help files IV-255
  new rulers III-11
  notebook V-684
  procedure files III-398
  procedures III-397
  symbolic paths II-22
  waves V-526
CreationDate function V-114
creator code
  for Igor files III-449
  Open operation V-718
Crop to Page Contents
  page layouts II-498
Cross operation V-115
cross product V-115
cross-correlation III-287
cross-covariance V-556
cross-platform issues
  carriage return III-453
  character set III-453
  file extensions
    in procedures III-455
  file transfers III-448
  file types
    in procedures III-455
  fonts III-453
  FTP III-448
  keyboard III-452
  linefeed characters III-453
  mouse III-452
  notebook pictures III-19
  procedure files III-454
  text III-453
crystallographic coordinates
  conversion from cartesian V-1091
  conversion to cartesian V-1090
crystals
  Gizmo II-457
CSC format sparse matrix III-153
csc function V-115
csch function V-115
CSR format sparse matrix III-152
CsrInfo function V-116
CsrWave function V-116
  use CsrWaveRef instead IV-197
CsrWaveRef function IV-197, V-117
  example IV-163, IV-197
CsrXWave function V-117
CsrXWaveRef function IV-197, V-117
CTabList function V-118
Ctrl-Break
  aborting procedures IV-51
CtrlBackground operation IV-323, V-118
CtrlFIFO operation V-120
CtrlNamedBackground operation IV-319, V-119
cubic spline interpolation III-115
  example III-111, III-115
curly braces
  in wave assignments II-78
current data folder II-108
  Data Browser II-115
current experiment II-16
Cursor operation V-121
CursorMovedHook function IV-339
cursors
  activating II-320
  as input for procedures IV-163
  attached to wave II-320
  Click cursor V-872

  color II-320, V-121
  cross hair style II-320
  CsrInfo function V-116
  CsrWave function V-116
  CsrWaveRef function V-117
  CsrXWave function V-117
  CsrXWaveRef function V-117
  Cursor operation V-121
  dashed lines II-320
  deactivating II-320
  detecting movement IV-339–IV-340
  double arrow cursor II-285, II-288, II-308
  four-arrow cursor II-302
  free II-320
  functions II-321
  hcsr function V-340
  hook functions IV-339
  horizontal coordinate V-340
  hour glass V-872
  in contour plots II-375
  in graphs (example) I-38, II-285
  information about V-116
  moving II-320
  moving cursor calls function IV-339–IV-340
  operations II-321
  PauseForUser operation IV-152, IV-154
  pcsr function V-741
  putting in graphs II-319–II-321
  qcsr function V-785
  ranges of interest in curve fit III-197
   example III-197
   multivariate fit III-201
  related functions V-10
  related operations V-6
  removing II-320
  ShowInfo operation V-869
  Sleep operation V-872
  specifying range of interest II-320
  style functions II-320
  styles II-320
  vcsr function V-1067
  vertical coordinate V-1067
  watch V-872
  xcsr function V-1114
  z value from contour trace V-1120
  zcsr function V-1120
Cursors button
  in Duplicate Waves dialog II-70
CursorStyle subtype IV-204
  keyword V-124
Curve Fit progress window III-179, III-234
curve fitting III-178–III-268
  auto trace (examples) I-47, I-51
  aborting III-180
  algorithms III-179–III-180
  all-at-once functions III-256
  Curve Fitting dialog and III-256
  restrictions III-256
auto-trace destination waves III-196, III-234
  length III-196
built-in functions
  dblexp_XOffset III-207
  double exponential III-208–III-209
  exp_XOffset III-207
  exponential III-207
  gauss III-206
  gauss2D III-212
  Hill equation III-210
  line III-210
  log III-211
  lognormal III-212
  Lorentzian III-206
  notes III-206–III-212
  poly2D III-212
  polynomial III-210
  power law III-211
  sigmoid III-211
  sin III-209
  two dimensional
   gaussian III-212
   polynomial III-212
  Voigt III-206
chi-square III-179, III-235
CM_Kn: covariance matrix waves V-125
coefficient of correlation III-226, V-125
coefficients III-179
  coefficient names III-195
  Coefficients tab in Curve Fitting dialog III-185, III-189
  coefficients wave III-184, III-195
  confidence intervals III-223
  default coefficient wave III-195
  explicit coefficients wave III-195
  for user-defined fit function III-190
  names III-190
  new wave III-195
  values III-184
complex fit functions III-248
confidence bands III-223, III-226
  nonlinear functions III-226
  statistics III-225
  waves generated by Igor III-225
confidence intervals
  calculating after fit III-224
constraints III-227, III-232, V-129
  constraint expressions III-228
  equality constraint III-229
  examples III-229
  infeasible III-230
  ODR III-238, V-132
  pitfalls III-231
  using command line III-228

  using Curve Fitting dialog III-227
  violated III-230
convolution and III-256
correlation coefficient III-233
correlation matrix III-226
covariance matrix III-226, V-125
Curve Fitting dialog III-213
CurveFit operation V-124
data folders III-232
dblexp (double exponential) V-127
derivatives III-193
destination waves III-196–III-197
  auto-trace III-196
  explicit destination III-196
  log axes III-196
  multivariate fits III-202
  no destination wave III-196
  saving III-196
differential equations and III-256
epsilon III-267
epsilon wave III-193
error estimates III-180, III-195, III-217, III-222–III-223
  ODR III-238
examples
  ellipse III-242–III-244
  experiments III-265
  function list in a string III-247
  holding a coefficient III-184
  implicit functions III-242–III-244
  ODR III-239–III-241
  residuals III-184
  simple line fit III-182
  summed exponentials III-246
excluding points III-198
exp (exponential) V-127
expected points/cycle III-209
exponentials III-187, III-207–III-209, III-265
  problems III-208
extrapolation (example) I-50
fit constants III-189
fit function expression III-191
FitFunc keyword III-252, V-255
FuncFit operation V-273
FuncFitMD operation V-276
gauss (Gaussian)
  1D V-127
  2D V-128
gaussian (examples) I-47, I-55
growing exponential III-187
handling errors III-235
Hill equation V-128
history III-184, III-212
  coefficients III-195
holding coefficients III-184
  ODR III-238
identifiability problems III-226
implicit functions III-242–III-244, V-126
  example III-242–III-244
independent variables III-191, III-255
  multiple III-242
INFs III-232
initial guesses III-180, III-189, V-131
  graphing III-193
  ODR III-237
  problems III-187
input values III-212
iteration paths, saving III-234
iteration start III-236
Levenberg-Marquardt algorithm III-179–III-180
line fit III-182, III-210, III-233, V-127
list of functions III-244
  constraints III-245
  linear dependency III-245
log axes
  and auto-trace III-196
log normal V-128
lor (Lorentzian) V-127
M_Covar wave III-226
M_FitConstraint wave III-231
manual guesses III-187
mask wave III-198
  multivariate fit III-201
maximum iterations III-234
miscellaneous options III-233
model curve III-196
multidimensional V-276
multiprocessor support
  curve fitting
    multitasking III-249–III-250
multithreading III-249
multivariate functions III-200–III-205, III-260, V-276
  and update time III-202
  examples
    built-in function III-202
    user-defined function III-203
  independent variables III-191
  model results III-202
  Poly2D III-202
  selecting data III-201
    gridded III-201
    multicolumn III-201
  selecting function III-201
NaNs III-232
nonlinear least squares III-179
ODR III-236–III-241, V-126
  constraints III-238, V-132
  error estimates III-238
  examples III-239–III-241
  holding variables III-238

    initial guesses III-237
    results III-238
    weighting III-237
  ODR is threadsafe III-237
  ODRPACK95 III-237
  only guess III-189
  output values III-184, III-212
    coefficients III-195
    ODR III-238
  overview III-179–III-180
  poly (polynomial)
    1D V-127
    2D V-128
  polynomials III-210, III-265–III-267
    two-dimensional III-212, V-403
      example III-202
  power law V-128
  prediction bands III-226
  problems
    automatic guess didn’t work III-187
    with dialog III-205
  proportional weighting III-199
  quality in line fit III-235
  quick fit III-181
    limitations III-181
  quit reason III-235
  quitting III-180
  r-squared III-221
  ranges of interest III-197
  reduced major axis III-237
  references III-267, V-134
  report III-184
    coefficient values III-195
  residuals III-184, III-217
    auto trace III-219
    auto wave III-220
    axes III-219
    calculating after fit III-220
    example I-57
    explicit residual wave III-220
    removing the auto-trace III-220
  results III-184, III-212
  RMA III-237
  robust fitting III-234
  sigmoid V-128
  simple example III-182–III-184
  simplified III-181
  sin (sine) V-127
  singular value decomposition III-179, III-265
  singularities III-265
  special variables III-232–III-236
  standard deviation for a point III-199
  statistics III-222, III-227, V-1006
  structure fit functions III-261–III-263
    examples III-262
    multivariate III-263
    syntax III-263
    WMFitInfoStruct structure III-263
  subset of data III-197–III-198
    continuous III-197
    discontinuous III-198
    multivariate fit III-201
  sums of fit functions III-244
    constraints III-245
    linear dependency III-245
  suppressing Curve Fit progress window III-234
  termination criteria III-180, III-233
  terminology III-178
  to subset of data (example) I-49
  to user-defined functions III-190–III-194
    coefficient names III-190, III-195
    Coefficients tab III-192
    compile errors III-192
    creating fit function III-190
    FitFunc keyword III-252
    forms not suitable for dialog III-254
    function expression III-191
    function format III-250–III-260
    graphing initial guesses III-192
    independent variables III-191
    initial guesses III-192
    making function always available III-194
    special comments III-253
  to XFUNCS (external functions) III-194
  total least squares III-236–III-237
  troubleshooting III-266–III-267
    constraints III-231
    Curve Fitting dialog III-205
    singularities III-265
  user-defined functions
    all-at-once functions III-256, III-260
    multivariate functions III-255, III-260
    structure fit functions III-261–III-263
  V_ variables III-232–III-236, III-265
  Voigt III-206
  W_coef wave III-196, V-131
  W_sigma wave III-195–III-196, III-222
  weighting (example) I-55
  weighting wave III-199–III-200
    ODR III-237
  XY data ranges III-197
Curve Fitting dialog III-181–III-217
  all-at-once functions III-256
  Coefficient Wave menu III-195
  Coefficients list III-185, III-189
  Coefficients tab III-185, III-189
    constraints III-227
    details III-216
    user-defined function III-192
  constraints III-227
  Data Options tab III-197
    Data Mask menu III-198

details III-216
multivariate fit III-201
detailed description III-213–III-217
external functions and III-194
fit function selection III-183
From Target checkbox III-183
Function and Data tab III-183
details III-214
multivariate functions
selecting III-201
selecting gridded data III-201
selecting multicolumn data III-201
Function menu III-183
multivariate functions III-201
global controls III-214
Graph Now button III-192
input data III-183
New Fit Function dialog III-190
compile errors III-192
Save Fit Function Now button III-191
Test Compile button III-191
Output Options tab III-186
auto-trace length III-196
create covariance matrix III-226
Destination menu III-196
destination, New Wave III-197
details III-216–III-217
error analysis III-223
residuals III-218
problems III-205
selecting data waves III-183
simple example III-182
user-defined functions III-190
problems III-254
special comments III-253
CurveFit operation V-124
data folders IV-171
custom axes
Gizmo II-450
CustomControl III-416
CustomControl operation III-424, V-134
customize at point II-306, V-625
legends III-43
cutoff frequency
filtering III-312
smoothing V-879
cutting
in page layouts II-482
in tables II-248
rows or columns in tables II-268
CWT operation V-137
cyclic redundancy check value V-998, V-1071
cylinder objects
Gizmo II-442

D

d
suffix in tables II-241, II-254
dashed lines III-496
drawing V-839
in graphs II-294
preferences III-496
SetDashPattern operation V-837
Dashed Lines dialog III-496
data
archiving V-815
creating II-65, II-92
examples I-35, I-46, I-53
in tables II-234, II-239, II-248
editing by drawing (example) I-21–I-22, I-29
loading
example I-22
from files II-126–II-177
from Igor experiment file V-500–V-505
GBLoadWave operation II-166, V-291
JCAMPLoadWave operation V-464
MLLoadWave operation II-163
waves, LoadWave operation V-508
waves, MLLoadWave operation V-589
XLLoadWave operation II-159, V-1115
reading binary files V-219
saving in files II-177–II-181, V-220
saving waves in file V-813
sparse II-367
typing into a table (example) I-13
writing formatted V-260
data acquisition IV-312
BackgroundInfo operation V-45
cameras IV-312
chart controls IV-313–IV-319
Chart operation V-61–V-63
CtrlBackground operation V-118
CtrlFIFO operation V-120
CtrlNamedBackground operation V-119
FakeData function V-215
FIFO buffers IV-313–IV-319
FIFO2Wave operation V-228
FIFOStatus operation V-229
frame grabbers IV-312
GPIB IV-312
GW Instruments IV-312
Instrutech IV-312
KillBackground operation V-468
KillFIFO operation V-469
National Instruments IV-312
NewFIFO operation V-678
NewFIFOChan operation V-678
NIDAQ Tools IV-312
oscilloscope displays II-347–II-348
serial port IV-312

SetBackground operation V-836
sound IV-312
VISA IV-312
Data Browser II-72, II-108, II-114–II-123
adopting waves II-119
browsing experiments II-21
browsing files II-117
copying objects II-115
CreateBrowser operation V-113
current data folder II-115
DataBrowserObjectsPopup IV-138
deleting data II-117
duplicating objects II-115
Execute Cmd button II-118
find dialog II-119
GetBrowserLine function V-295
GetBrowserSelection function V-296
info pane II-115
main list II-115
modal Data Browser II-119
ModifyBrowser operation V-600
moving data objects II-115
new data folder II-117
plot pane II-116
preferences II-119
saving experiment copies II-117
selecting objects II-115
shortcuts II-122
text encodings III-473
user-defined buttons II-120
data folder notes II-120
data folder references IV-78–IV-82
/SDFR flag IV-80
as function results IV-81
checking validity IV-81
CountObjectsDFR operation V-111
DataFolderRefChanges function V-143
DataFolderRefStatus function V-143
DFREF keyword IV-80
functions IV-81
GetDataFolderDFR function V-296
GetIndexedObjNameDFR function V-304
GetWavesDataFolder function V-317
in tables II-258, II-270
NewFreeDataFolder function V-680
structure fields IV-80
waves containing IV-82
data folders II-108–II-123
$ operator II-109
accessing in user functions IV-78–IV-82
assignment statements II-111, IV-3
autosave II-37
browsing in dialogs II-228
clearing IV-172
controls II-111
conventions IV-170
CountObjects operation V-110
CountObjectsDFR operation V-111
creating II-117, V-187, V-678, V-680
creating global variables in user functions IV-249
current data folder II-108–II-109, II-111, II-115, IV-171
determining V-296
setting II-108, V-837
curve fitting III-232
data folder containing wave V-317
data folder notes II-120
DataFolderDir function V-141
DataFolderExists function V-142
DataFolderList function V-142
DataFolderRefChanges function V-143
DataFolderRefStatus function V-143
dependencies II-111, III-56, IV-230
DFREF keyword IV-80
dialogs II-108
Dir operation V-160
DuplicateDataFolder operation V-187
dynamic text in annotations III-56
examples II-112–II-114
free IV-96–IV-99
full data folder paths IV-3
GetDataFolder function V-296
GetDataFolderDFR function V-296
GetWavesDataFolder function V-317
global variables II-108, II-111
KillDataFolder operation IV-172, V-469
killing II-109
listing objects in V-303–V-304
listing of II-115
loading from Igor experiment file V-500–V-505
MoveDataFolder operation V-658
MoveString operation V-663
MoveVariable operation V-665
MoveWave operation V-665
multitasking IV-330
name conflicts III-505
names II-109
CheckName function V-68
examples II-109
naming rules III-501
NewDataFolder operation V-678
NewFreeDataFolder function V-680
NVAR, SVAR and Wave keywords IV-66, IV-69
objects in data folders II-108
Packages data folder IV-170
programming IV-169–IV-172
quoting liberal names IV-169
references using $ IV-63
referring to waves II-111
related functions II-110, V-12
related operations II-110, V-6

relative data folder paths IV-3
renaming V-796
ReplaceWave operation II-114, V-801
root II-109, II-115, IV-3, V-810
saving unpacked II-18
Set Variable controls II-111
SetDataFolder operation V-837
setting and restoring current IV-171
storing runs of data IV-170
string containing name II-109
syntax II-109
quoting liberal names II-110
text encodings III-478
UniqueName function V-1048
use in commands IV-3
user-defined functions II-111
Value Display controls II-111
wave references IV-84
window macros II-111
Data Mask menu
Curve Fitting dialog III-198
data objects
Gizmo II-453
data type
in Igor Text files II-151
MatrixOp operation III-145
Redimension operation V-788
data types
of waves II-66
data units
changing II-68
SetScale operation V-853
max length II-69
data values II-63
changing units V-107
in tables II-235, II-241
in wave assignments II-74
databases
SQL II-181
DataBrowserObjectsPopup menu IV-138
DataFolderDir function V-141
DataFolderExists function V-142
DataFolderList function V-142
DataFolderRefChanges function V-143
DataFolderRefStatus function IV-81, V-143
dataset region references
HDF5 II-209
datasets
HDF5 II-202
Date Format dialog II-316
date function V-144
date/time axes II-307, II-315–II-317
date format II-316
in Modify Axis dialog II-316
manual ticks II-317
suppress date II-316
time format II-316
date/time data II-85
Excel files II-162
date2secs
example II-85
date2secs function V-145
dates
CreationDate function V-114
Date Format dialog II-130
date/time axes II-307, II-315–II-317
custom formats II-316
date2secs function V-145
DateTime function V-145
dateToJulian function V-144
format in tables II-239
formats II-246
in delimited text files II-130, II-132, II-136
in Igor Text files II-151
in notebooks III-13, III-20
in tables II-239, II-245, II-256
Julian V-144, V-468
JulianToDate function V-468
loading custom formats II-130
LoadWave operation II-131, II-144
modDate function V-592
ordering of components II-246
pasting in tables II-249
range of II-315
related functions V-7
representation of II-245, II-256, II-315
Secs2Date function V-833
SetScale operation V-854
stored in waves II-85
Table Date Format dialog II-246
units II-68–II-69
wave precision II-69
DateTime function V-145
dateToJulian function V-144
Daubechies wavelet transform V-189
dawson function V-145
dB
filtering III-305
dblexp
curve fitting function III-208
dblexp (double exponential) curve fit V-127
dblexp_peak
curve fitting function III-209
dblexp_XOffset
curve fitting function III-207
deactivate events
window hooks IV-303
debug on abort IV-214, V-146
debug on error IV-213, V-146
preventing IV-284
debugger IV-212–IV-227
background tasks IV-321

  breakpoints IV-213
  debug button IV-215
  debug on abort IV-214, V-146
  debug on error IV-213, V-146
  Debugger operation V-145
  DebuggerOptions operation V-146
  DisableThreadSafe IV-226
  enabling IV-212
  enabling independent modules IV-239
  expression evaluator IV-225
  expressions inspector IV-223–IV-224
  function variables IV-220
  Go button IV-216
  independent modules IV-239
  macro variables IV-218
  NVAR checking IV-213
  procedure pane IV-225
  setting breakpoints IV-213
  shortcuts IV-226
  single-stepping IV-216
  stack list IV-216
  Step button IV-216
  Step Into button IV-216
  Step Out button IV-216
  stepping through code IV-216
  Stop button IV-216
  string inspector IV-222–IV-223, IV-225
  structures IV-221
  SVAR checking IV-213
  threadsafe functions IV-225
  V_debugDangerously IV-225, IV-227
  variables list IV-216
    columns IV-218
  Variables pop-up menu IV-218
  WAVE checking IV-213
  wave inspector IV-222–IV-224
Debugger operation V-145
DebuggerOptions operation V-146
debugging
  DisableThreadSafe IV-226
  threadsafe functions IV-225
  wave leaks IV-206–IV-207, V-1087
debugging procedures IV-212–IV-227
  background tasks IV-321
  Debugger operation V-145
  DebuggerOptions operation V-146
  fprintf operation IV-261, V-260
  Print operation V-769
  printf operation V-770
  runtime stack information V-314
  Silent operation V-871
  Slow operation V-877
  using print statements IV-212
decibels
  filtering III-305
decimal numeric format
  in tables II-255
decimal separator
  in delimited text files II-136
  in general text files II-141
  in graphs II-288
  in tables II-255
decimal symbols
  tables II-255
decimation III-135–III-320
  FindSegmentMeans user-defined function III-174
  multidimensional II-98
  of matrices V-557
  Resample operation III-136, V-803
  techniques
    omission III-135
    smoothing III-136
decommentizing III-408
decomposed characters
  Unicode V-693
decomposing waves II-81
decompressing images V-419
deconvolution
  images V-404
decrement operator IV-6, IV-43
default
  in switch statements IV-44
default compression
  HDF5 compression II-214
default file text encoding III-465
default font
  GetDefaultFont function V-297
  GetDefaultFontSize function V-297
  GetDefaultFontStyle function V-297
  in controls V-149
  in graphs II-288
  in notebooks III-9
  in page layouts II-496
default keyword V-146
Default Output Resolution Factor II-465
default tab width
  in notebooks III-6
default tabs
  in notebooks III-6, V-696
  procedure files III-406
  procedure windows III-405
default wave properties II-67
DefaultFont operation V-147
DefaultGuiControls operation V-147
DefaultGuiFont operation V-149
DefaultTab keyword V-150
DefaultTab pragma III-406
define statement V-16
defined function V-152
DefineGuide operation V-153
Delaunay triangulation

contour plots V-605
Triangulate3d operation V-1046
delay
Sleep operation V-871
DelayUpdate
box in tables II-237, II-244
in page layouts II-483, II-487
DelayUpdate operation V-153
DELETE method
URLRequest operation V-1054
Delete Points dialog II-73
DeleteAnnotations operation V-154
DeleteFile operation V-155
DeleteFolder operation V-156
warning V-156
DeletePoints operation II-73, V-157
deleting
data in waves II-73, II-249
recreation macros II-48
delimited text files II-129–II-137
carriage return II-135
column formats II-129
column labels II-131, II-134, II-136
column position waves II-134
column types II-134, II-136
comma-separated values II-135
date columns II-132
dates II-130, II-136
decimal character II-136
delimiter characters II-131, II-136
escape codes II-135
example II-131
features of II-126–II-127
fixed field text files II-137
FORTRAN files II-137
INFs II-129
linefeed II-135
loading
matrices II-133
text waves II-134
loading process II-132
missing values II-136
NaNs II-129
nonnumeric columns II-132
numeric formats II-133
quotation marks II-135
quoted strings II-132, II-135
row labels II-134
row position waves II-134
saving multidimensional waves II-179
saving waves II-178, II-180
skipping lines II-136
spaces II-136
special characters II-134
times II-130
troubleshooting II-137
tweaks II-136
versus general text II-139
X scaling II-133
delimiter characters
in delimited text files II-126, II-131, II-136
in general text files II-138
in text data columns II-134
spaces II-136
deltax function V-157
use DimDelta instead V-160
dendrogram
example III-168
HCluster operation V-338
dendrogram wave
hierarchical clustering III-167
dense matrices III-151
dependencies IV-5, IV-230–IV-234
cascading IV-232
caveats IV-234
creating
in user functions IV-233
data folders IV-230
in user defined functions IV-234
independent modules IV-243
updating
in macros IV-123
in user functions IV-112
dependency assignments IV-230–IV-234
# operator IV-233
creating
in user functions IV-233
data folders IV-230
deferred evaluation IV-233
deleting IV-232
examples I-44, III-321, IV-230
numeric variables IV-231
SetFormula operation V-847
string variables IV-231
waves IV-232
dependency formulas IV-230–IV-234
:= operator IV-5
broken objects IV-233
creating
in user functions IV-233
data folders IV-230
deleting IV-232
dynamic text in annotations III-56
for waves II-84
GetFormula function V-303
guided tour (example) I-44
SetFormula operation IV-234, V-847
status IV-231
using $ operator IV-234
using operations IV-234
derivatives in curve fitting III-193
derived rulers III-12

desktop
    path to folder V-895
destination waves IV-85
    automatic wave references IV-72
    in curve fitting III-196–III-197
    in wave assignment II-74
    inline wave references IV-73, IV-86
    issues IV-86
    standalone wave references IV-72
    subranges of II-76
    wave references IV-86
determinant V-533
development systems IV-208
DF flag
    function results IV-81
DFREF keyword IV-80
    example IV-79
    function results IV-81
    functions IV-81
    structure fields IV-80
Diagnostics folder IV-341
dialog wave browser II-228
    filters II-229
dialogs
    background tasks IV-321
    browsing waves II-228
    from procedures V-165, V-167
    from user functions IV-144
    modeless using panels IV-158
    relation to commands I-7
    relation to menus I-7
    text areas
        magnification II-53
        zooming II-53
DIBs
    exporting graphics III-103
    saving V-826
differential equations III-322–III-336
    coupled first-order equations III-327
    curve fitting and III-256
    derivative function III-324
    discontinuities III-335
    first-order equations III-325
    higher order equations III-328
    IntegrateODE operation V-452
    interrupting calculations III-334
    stopping and restarting calculations III-334
    stopping on a condition III-335
Differentiate operation V-158
differentiation III-120
    Differentiate operation V-158
    multidimensional III-120
    of XY data III-120
    waveform data III-120
diffuse light
    Gizmo II-432
digamma function V-159
digits after decimal point
    in tables II-255
DimDelta function V-160
dimension labels II-93–II-94
    CopyDimLabels operation V-103
    example II-82
    FindDimLabel function V-240
    GetDimLabel function V-298
    in category plots II-357
    in delimited text files II-131
    in tables II-262
    length limit II-94, II-244
    long dimension labels V-209
    naming conventions II-94
    SetDimLabel operation V-838
    speed considerations II-94
    text encodings III-473
    viewing in tables II-235
    wave indexing II-82
dimension scales
    HDF5 V-342
dimension scaling
    changing II-68
        SetScale operation V-853
    checking in tables II-235
dimension units
    changing
        SetScale operation V-853
dimensions
    changing II-72
        Redimension operation V-788
    HDF5 II-204
    number of, WaveDims function V-1072
    platform-related issues III-455
DimOffset function V-160
DimSize function V-160
    example IV-200
Dir operation V-160
direct autosave II-38
direct color mode V-621
direct reference to globals IV-114
    converting to runtime lookup IV-114
directional lights
    Gizmo II-433
directories
    creating via FTP IV-275
    deleting via FTP IV-275
    downloading via FTP IV-273
    FTPCreateDirectory operation V-265
    FTPDelete operation V-267
    uploading via FTP IV-274
DisableThreadSafe IV-226
disappearing drawing objects III-65, III-68
Discrete Prolate Spheroidal Sequences V-171
disk objects

Gizmo II-443
dispersion of wave V-1084
2D V-415
Display Advanced Options Menus II-466
display list
Gizmo II-416–II-417
Display operation V-161–V-163
display volume
Gizmo II-421
DisplayHelpTopic operation V-163
DisplayProcedure operation V-163
displays
high-resolution III-508
resolution of III-507
dissimilarity
hierarchical clustering V-339
dissimilarity matrix
hierarchical clustering III-162–III-163
distance map V-259
hierarchical clustering III-162
distance sorting
Gizmo II-432
distance transform V-419
distributing
drawing objects III-65
dithering
image plots V-637
division IV-6
Do It button I-7
do-while
in user functions IV-45
do-while statements V-165
DoAlert operation V-165
document info icon
in notebooks III-5
document properties
in notebooks III-6
Document Settings dialog III-6, III-20, III-405
documents
path to folder V-895
DOG wavelet transform V-140
DoIgorMenu operation V-166
DoPrompt operation
example IV-144
help for IV-148
DoPrompt statements V-167
doScroll keyword V-319, V-321, V-866
dot product V-534
MatrixOp operation III-144
dot-underscore files V-441
double declaration V-168
local variables in user functions IV-112
variable in user functions IV-112
double keyword V-168
double precision V-788
numeric variables II-104
double-clicking
on text III-408
double-precision floating point II-66
DoUpdate operation IV-112, IV-234, V-168
in macros IV-123
DoWindow operation V-168
style macros V-171
window title V-170
downloading
directories via FTP IV-273
files via FTP IV-273
files via HTTP IV-276
web pages via HTTP IV-276
DoXOPIdle operation V-171
DPI
platform-related issues III-455
DPSS operation V-171, V-672
drag and drop II-35, IV-280
graphs II-280
dragging
a marquee in a page layout II-482
cells in tables II-243
column boundaries in tables II-252
columns in tables II-252
object handles in layouts II-481
objects in layouts II-481
DrawAction operation V-172
DrawArc operation V-174
DrawBezier operation III-75, V-174
drawing
grouped objects III-76
drawing layer tool III-65
Drawing Limitations III-442
drawing objects
Gizmo II-436
drawing order of images
ReorderImages operation V-797
drawing order of traces II-297
ReorderTraces operation V-798
drawing tools III-62–III-78
absolute coordinates III-66
Align objects III-65
anchor for text III-63
arcs III-64, V-174
Arrow Fat III-64
Arrow tool III-63
arrows III-64
axis coordinates III-68
axis relative coordinates III-67
bevels III-496
Bezier curves
creating III-75
editing III-70–III-71
bezier curves V-174
circles III-64, V-176
coordinate systems III-66–III-68, III-73, V-842

copy III-73
creating objects III-63–III-72, III-74
curves V-174
dashed lines III-64–III-65
    SetDashPattern operation V-837
deleting objects III-63, III-76, V-172
disappearing objects III-65, III-68, III-73
Distribute objects III-65
draw object commands V-172
Draw Poly tool III-69
Draw Wave tool III-73
DrawAction operation V-172
DrawArc operation V-174
DrawBezier operation V-174
drawing layer drawing tool III-65
Drawing mode III-62
DrawLine operation V-175
DrawOval operation V-176
DrawPICT operation V-176
DrawPoly operation V-176–V-177
DrawRect operation V-178
DrawRRect operation V-178
DrawText operation V-178
DrawUserShape operation V-179
duplicating objects III-63
Edit Poly tool III-69
Edit Wave tool III-73
end caps III-496, V-840
environment III-74, III-77
Environment pop-up menu III-65
fill modes III-64–III-65
Freehand Poly tool III-69
Freehand Wave tool III-73
grids III-66, V-1041
grouped objects III-65, III-74, III-77
HideTools operation V-347
in page layouts II-480
inserting objects V-172
joins III-496, V-840
kill layer III-76
layers II-479, III-68–III-69, III-76–III-77
    Overlay drawing layer III-69
    ProgAxes drawing layer III-68
    ProgAxes layer III-68
    ProgBack drawing layer III-68
    ProgBack layer III-68
    ProgFront drawing layer III-69
    ProgFront layer III-68
    selection (example) I-25
    SetDrawLayer operation V-844
    UserAxes drawing layer III-69
    UserAxes layer III-68
    UserBack drawing layer III-68
    UserBack layer III-68
    UserFront drawing layer III-69
    UserFront layer III-68
Line Properties III-65
lines III-64, V-175
Lines (and Arrows) tool III-64
miters III-496, V-840
modifying objects III-63, III-74
monotonic waves III-73
Mover pop-up menu III-65
moving, resizing objects III-63, III-69, III-73
offscreen objects III-65, III-68, III-73
Operate mode III-62
operations III-74–III-76
Oval tool III-64
ovals V-176
Overlay drawing layer III-69
paste III-73
pasted objects disappearing III-73
pasted picture resizing III-73
picture resizing III-73
plot relative coordinates III-67
Polygon tool III-64, III-69–III-73
polygons
    as input for procedures IV-164
    creating III-69, III-75
        example I-20
    DrawPoly operation V-176
    editing III-69–III-72
    exiting edit mode III-71
    last point III-69
    origin III-75
    scaling III-71
ProgAxes drawing layer III-68
ProgBack drawing layer III-68
ProgFront drawing layer III-68–III-69
programming III-73–III-77
    example III-77
    strategies III-76
Rectangle tool III-64
related operations V-5
relative coordinates III-67
replace group method III-76
Retrieve objects III-65, III-68
rotating objects III-63
Rounded Rectangle tool III-64
rounded rectangles
    DrawRRect operation V-178
selecting, deselecting objects III-63, III-69, III-74
SetDashPattern operation V-837
SetDrawEnv operation III-74, V-838–V-843
SetDrawLayer operation V-844
shortcuts III-78
Show Tools III-62
ShowTools operation V-870
text III-63
    anchor III-63
    color III-63
    properties III-63

rotation III-63
Text tool III-63
tool palette III-62
ToolsGrid operation V-1041
User Shapes tool III-65
UserAxes drawing layer III-68–III-69
UserBack drawing layer III-68
UserFront drawing layer III-68–III-69
wave editing III-73, III-75–III-76
waves as Bezier curves III-75
waves as polygons III-75
DrawLine operation V-175
DrawOval operation V-176
DrawPICT operation V-176
in independent modules IV-244
DrawPoly operation III-75, V-176–V-177
xOrg, yOrg III-75
DrawRect operation V-178
DrawRRect operation V-178
DrawText operation V-178
DrawUserShape operation V-179
formatted text III-53
DrawUserShape subtype IV-204
DSPDetrend operation V-182
DSPPeriodgram operation V-182
dummy objects in page layouts II-487
Duplicate operation II-70, V-185–V-186
examples II-70
free waves V-186
in user functions V-186
warning V-187
Duplicate Waves dialog II-70
DuplicateDataFolder operation V-187
duplicates
FindDuplicates operation V-240
duplicating
controls III-419
data folders V-187
data objects with Data Browser II-115
drawing objects III-63
waves II-70, V-185–V-186
waves in tables II-240, II-269
DWT operation V-189
dynamic IV-127
dynamic libraries
Matlab II-163
dynamic menu items IV-129
dynamic text
in annotations III-56
TagVal function V-1022
TagWaveRef function V-1023

E

e function V-190
earlyKeyboard
events in hook functions IV-302
edge detection
ImageEdgeDetection operation V-370
edge detectors
Canny V-370
Frei–Chen V-370
Kirsch V-370
Marr–Hildreth V-370
Prewitt V-370
Roberts V-370
Shen–Castan V-370
Sobel V-370
EdgeStats operation III-289, V-190–V-192
Edit operation V-192–V-193
Edit Text Cell dialog II-258
editing waves II-235
using drawing tools III-73
ei function V-194
eigenvalues
MatrixEigenV operation V-534
eigenvectors
MatrixEigenV operation V-534
elapsed time II-256, II-315–II-317, V-833
loading format II-130
elements column in tables
hiding V-649
elements keyword for ModifyTable operation II-263
ellipses
error ellipses II-305, V-200
EllipticE function V-194
EllipticK function V-194
embedding III-80–III-93
in control panels III-443
EMF III-102
saving V-826
emphasized ticks
in user tick waves II-314
Enable Transparency Blend
Gizmo II-428, II-433
Enable Updating
pictures in notebooks III-18
Enable Updating dialog III-14
end caps
for drawing tools V-840
for lines and traces III-496
for traces V-617
end effects
in convolution III-285
in smoothing III-299
End keyword
in Igor Text files II-151
in procedures V-194
EndMacro keyword V-195
EndStructure keyword V-195
endtry keyword V-195

energy loss when windowing III-278
engineering units
  printf operation V-771
enhanced metafiles
  exporting graphics III-102
  graphics technology III-507
  importing III-509
  saving V-826
enoise function V-195
Enter key
  in tables II-243
entering data in tables II-234, II-239, II-244
enums
  HDF5 II-211
environment variables
  GetEnvironmentVariable function V-298
  SetEnvironmentVariable function V-844
  UnsetEnvironmentVariable function V-1050
EPS
  CMYK III-99, III-105
  creating file III-98, III-105
  embedding fonts III-99, III-105
  exporting graphics III-97, III-103
  missing PostScript fonts III-99, III-105
  PostScript language level III-97, III-103
  saving V-826
  screen preview III-97, III-103
  Symbol font III-493
  transparency III-96, III-102
epsilon
  curve fitting III-267
epsilon wave in curve fitting III-193, III-213
equality IV-6
equality operator IV-6, IV-8
  and roundoff error IV-8
equalWave function V-196
equidistant projection V-781
erf function V-197
erfc function V-198
erfcw function V-198
error bars
  arbitrary error values II-304
  bar thickness II-304
  cap thickness II-304
  cap width II-304
  examples II-304
  in graphs II-304
  modes II-304
  NaNs in error values II-304
  parts identified V-202
  single sided II-304
  XY Error box II-304
Error Bars dialog II-304
error ellipses II-305, V-200
error estimates in curve fitting III-222
error function V-197, V-463
error handling
  text encodings text encodings
    error handling III-464
error shading II-305, V-200
error waves II-304
ErrorBars operation V-199–V-203
errors
  GetErrMessage function V-299
  GetRTErrMessage function V-313
  GetRTError function V-312
  handling IV-49, V-1047
  in background tasks IV-321
  in macros IV-122
  in user functions IV-110–IV-111
  rtFunctionErrors pragma IV-55
escape codes
  backslash characters in annotations III-58
  backslashes
    LoadWave operation II-150
  examples III-35
  for carriage return IV-14
  for linefeed IV-14
  in annotations III-35–III-38, III-42, III-53–III-58
  in axis label II-318
  in axis labels III-57
  in delimited text files II-135
  in Igor Text files II-154
  in strings IV-14
  LoadWave operation II-150
  saving text waves II-180
  text info variables III-51
  wave symbols III-42
escape sequences IV-14
  invalid text in tables II-260
  null IV-16
  Unicode IV-15
EstimatePeakSizes operation V-203
even-odd rule III-72
eventMod field III-438
example experiments II-4
examples
  all-at-once function III-256
    with convolution III-257
  confidence bands III-223
  confidence interval calculation III-224
  constraints III-229
  creating user-defined function III-190
  fit arbitrary number of Gaussian peaks III-254
  implicit function fitting III-242–III-244
  manual guesses III-187
  multivariate fit III-202
  ODR fitting III-239–III-241
  Poly2D III-202
  prediction bands III-223
  subrange destination III-197
  sums of fit functions III-246–III-248

user-defined fit code III-252
user-defined multivariate function III-203
Excel files
date/time data II-162
loading II-159, V-1115
loading 2D data II-162
excluding points in curve fitting III-198
exclusive OR operator IV-6
Execute Cmd button II-118
Execute operation IV-111, IV-201, V-204
calling external operations from functions IV-202
calling macros from functions IV-202
data folder name V-317
GetErrMessage function V-299
independent modules IV-243
use of liberal names with IV-168
Execute/P operation V-204
operation queue IV-278
ExecuteCmdOnList function IV-201
ExecuteScriptText operation V-205
EXIF metadata V-395
exists function V-207
exp
curve fitting function III-207
exp (exponential) curve fit V-127
exp function V-208
exp_XOffset
curve fitting function III-207
expanding graphs II-285
examples I-35
expanding panels III-443
expected points/cycle in curve fitting III-209
expected wave reference error IV-75
experiment file
HDF5 packed II-16
compression II-16
loading data from V-500–V-505
packed II-16
SaveExperiment operation V-819
snooping II-26
unpacked II-17
experiment file text encoding III-465
experiment files
autosave II-37
errors saving II-30
locked II-30
read-only II-30
experiment folder II-17
Experiment Information dialog II-19
experiment recreation procedures II-26
ExperimentInfo operation V-208
ExperimentModified operation V-212
experiments II-16–II-30
browsing II-21, II-117
current experiment II-16
data folders II-18
empty table II-21
errors while loading II-27
examples II-4
Experiment Information dialog II-19
ExperimentInfo operation V-208
ExperimentModified operation V-212
HDF5 packed file II-16, II-223
Igor binary wave files II-19
initialization commands II-27, IV-204
LoadData operation II-156, V-500–V-505
loading data from II-117
loading of II-26
long dimension labels V-209
long object names V-208
Macro Execute Error dialog II-27
merging II-19
miscellaneous file II-19
missing folders II-28
name of II-9
new II-21, IV-292
notebooks in III-3, III-25
opening II-19, IV-292
packed file II-16
procedure files II-19, III-396, III-398, III-400, III-403
references to files and folders II-24
reverting II-20
safe save II-29
SaveExperiment operation V-819
saving II-16–II-19, IV-286
copies II-117
HDF5 compression II-16
saving of II-29
settings II-234
sharing Igor binary wave files II-156
table in new experiments II-234
templates II-21
temporary files II-29
transferring II-24, II-117
unpacked file II-17
variables file II-19
expInt function V-212
expnoise function V-213
exponential curve fitting III-265, V-127
problems III-208
exponential functions V-7
acosh V-19
alog V-25
asinh V-42
atanh V-42
cosh V-109
coth V-110
cpowi V-111
exp V-208
ln V-500

log V-520
sinh V-871
tanh V-1023
exponential integral V-212
exponential notation
tick mark labels II-308
forcing II-311
exponentiation IV-6
exponentiation operator IV-6
Export Graphics dialog III-98, III-104
Gizmo II-465
ExportGizmo operation II-465, V-213
exporting
bitmaps III-97, III-104
BMP files V-405
BMPs III-103
choosing a format III-98, III-104
CMYK III-99, III-105
Crop to Page Contents II-498
DIB files V-826
DIBs III-103
EMF files V-826
enhanced metafiles III-102
EPS III-97, III-103
EPS files V-826
FBinWrite operation V-220
from tables II-252, II-270, V-826
SaveTableCopy operation V-829
Gizmo II-465, V-213, V-826
graphics III-96–III-100, III-102–III-106, V-405
graphics formats III-96, III-102
graphs II-324, V-826
HDF5 II-197, V-6, V-12
images V-405
ImageSave operation V-405
JPEG files V-405, V-826
metafiles III-102
notebooks III-20–III-21
numeric variables V-220
PadString function V-732
page layouts II-482, II-498, V-826
PDFs III-96, III-103
PhotoShop files V-405
PICT files V-405, V-826
HiRes V-826
PNG files V-405, V-826
PNGs III-97
PostScript language level III-97, III-103
PostScript PICT files V-826
QuickTime files V-405
SavePICT operation V-826
SGI files V-405
string variables V-220
SVGs III-97, III-104
tables as graphics II-271
Targa files V-405
TIFF files V-405, V-408
to databases II-181
waves II-177–II-181, II-270, V-220, V-829, V-1093
WMF files V-826
exporting raw
raw PNG files V-408
expressions
as operation flag values IV-12
as parameters IV-12
in debugger IV-225
precision of IV-38
strings IV-13
expressions inspector in debugger IV-223–IV-224
exterior annotations III-40
affects graph plot area III-41
exterior subwindows III-443, V-318
window hooks IV-305
external functions IV-208
creating IV-209
curve fitting to III-194
FuncRefInfo function V-278
FunctionInfo function V-279
information about V-278–V-279
external operations IV-208
calling from a user function IV-202
conditional compilation IV-108
creating IV-209, V-736
IDLE events V-171
listing V-725
long object names III-504
OperationList function V-725
ParseOperationTemplate operation V-736
external text editor II-56
Extract operation V-213

F

f(z)
markers II-298
factorial function V-214
Faddeeva function V-214
FakeData function V-215
false and true IV-40
FAQ II-5
fast cosine transform V-571
fast sine transform V-571
fast staggered cosine transform V-572
fast staggered sine transform V-571–V-572
FastGaussTransform operation V-215
FastOp operation V-216
faverage function III-121, V-218
subranges III-122
faverageXY function V-218
FBinRead operation IV-195, V-219
FBinWrite operation IV-195, V-220
fences

box plots II-332
FetchURL function V-221
FFT
spatial frequency filtering III-360
wave references IV-75
FFT operation V-222–V-228
FFTs III-270–III-274, V-222–V-228
compared to Fourier transform III-272
continuous wavelet transform III-282
DFT equation III-272
effect on graphs III-273
effect on wave type and number of points III-270
Hanning window function III-277
harmonic analysis V-182
IDFT equation III-272
IFFT operation V-357–V-359
image analysis III-359–III-362
number of points restrictions III-270
one-sided spectrum III-270
periodograms V-182
phase calculation V-1051
unwrapping V-1051
phase polarity III-273
power spectra III-279
scaling amplitude III-272
spectral leakage III-277
speed III-274
swap diagonal quadrants V-427
two-sided spectrum III-270
wave type restrictions III-270
windowing III-275–III-279, V-182, V-1097
for images V-435
for matrices V-435
Hanning operation V-337
X scaling and units changed III-271
FGetPos operation IV-195, V-228
fidelity
in page layouts II-486
FIFO buffers
AddFIFOData operation V-20
AddFIFOVectData operation V-20
channels IV-313
chart controls IV-313–IV-319
Chart operation V-61–V-63
CtrlFIFO operation V-120
FIFO2Wave operation V-228
FIFOStatus operation V-229
KillFIFO operation V-469
NewFIFO operation V-678
NewFIFOChan operation V-678
programming IV-315
related operations V-6
SoundInStartChart V-888
SoundInStopChart V-889
updates IV-316
valid state IV-315
FIFO files
format IV-316
FIFO2Wave operation V-228
FIFOStatus operation V-229
fifth tick
in user tick waves II-314
FILE
GET method V-1054
PUT method V-1054
file extensions
.noindex II-29
.bwav II-155
cross-platform issues
in procedures III-455
.ibw II-155
Igor binary wave file II-155
Igor binary wave files II-155
Open File dialog IV-149
platform-related issues III-448
Save File dialog IV-151
File Information dialog
for notebooks III-25
procedure file version IV-167
file paths
in procedures III-455
length limits II-21
file permissions
Macintosh II-30
saving files II-30
Windows II-30
file reference numbers IV-196, V-716, V-719
FILE scheme
URLRequest operation V-1054, V-1058
file types
cross-platform issues III-448
in procedures III-455
Open File dialog IV-149
Save File dialog IV-151
files
adopting II-24
alias creation V-111
appending V-717
carriage return II-129
carriage-return/linefeed II-129
Close operation V-71
closing IV-195
CopyFile operation V-104
copying V-104
CreateAliasShortcut operation V-111
creating V-716
formatted text IV-259–IV-261
movies V-683
creator code V-718
cross-platform compatibility III-448–III-449
current file position V-228, V-264

DeleteFile operation V-155
deleting V-155
delimited text files II-129–II-137
delimiter characters II-126
dot-underscore files V-441
downloading via FTP IV-273
downloading via HTTP IV-276
drag and drop II-35, IV-280
errors saving II-30
FIFO buffers IV-313–IV-319
file types IV-149, IV-151
finding IV-195
fixed field text files II-137
formats II-126
FORTRAN files II-137
FTPCreateDirectory operation V-265
FTPDelete operation V-267
FTPDownload operation V-268
FTPUpload operation V-271
FunctionPath function V-287
general text files II-138–II-142
GetFileFolderInfo operation V-300
headers II-126
Igor binary wave files II-154–II-157
Igor Text files II-150–II-154
ImageFileInfo operation V-372
IndexedFile function V-440
information about V-300
linefeed II-129
linefeed characters III-453
loading data files II-126–II-177, IV-280
locked II-30
MacroPath function V-525
MIME-TSV IV-281–IV-283, IV-287, IV-289
MoveFile operation V-659
moving V-659
names
    length under Windows III-450
    platform-related issues III-450
    XOP length on Windows III-451
notebooks III-25
Open File dialog IV-149
Open operation V-716–V-721
opening IV-195, IV-280, V-722, V-724
ParseFilePath function V-733
paths
    extracting V-733
    manipulating V-733
    to functions V-287
    to macros V-525
procedure
    path separators III-455
programming IV-194
read-only II-30
reading V-717
reading binary files V-219, V-261
reading text files V-261
reference numbers V-716, V-719
related functions V-12
related operations V-5
safe save II-29
Save File dialog IV-151
saving data V-815
saving data files II-177–II-181
saving waves to V-812
searching V-327
SetFileFolderInfo operation V-845
setting information about V-845
shortcut creation V-111
Spotlight II-30
status V-265
temporary files II-29
terminator characters II-126
TextFile function V-1029
transferring
    cross-platform III-448–III-449
    FTP III-448
    via FTP V-268, V-271
types III-455, IV-282, IV-287–IV-288
uploading via FTP IV-274
writing V-717
    binary files V-220
    formatted data V-260
writing to IV-196
fill page
    printing graphs II-323
fill patterns III-498
    in pop-up menu controls V-766
fill rules
    polygons III-72
fill type in graphs II-295
fills
    axes V-632
Filter Design and Application dialog III-302
FilterFIR operation V-230–V-233
FilterIIR operation V-234–V-239
filtering III-299–III-316
    band pass filters III-312
    Butterworth filters III-312
    coefficients III-300–III-301, III-306, III-308, III-311
    cutoff frequency III-312
    dB III-305
    decibels III-305
    evaluating III-314
    Filter Design and Application dialog III-302
    FilterFIR operation V-230
    FilterIIR operation V-234
    FIR filters III-300
    FMaxFlat operation V-255
    high pass band III-303
    high pass example III-308

IIR filters III-300, III-312
ImageFilter operation V-372
ImageWindow operation V-437
Kaiser's maximally flat filter V-255
low pass band III-303
low pass example III-304
MatrixFilter operation V-539
MPR filter V-789
notch example III-310
Nyquist frequency III-300, III-312
order III-313
poles and zeros III-301
Remez Exchange III-311
Remez operation V-789
Resample operation V-803
zeros and poles III-301

filters
band stop filters III-312
dialog wave browser II-229
HDF5 II-217
Open File dialog IV-149
Save File dialog IV-151

Find bar II-54

find bar
replacing text II-54

Find dialog
Data Browser II-119

Find Same II-54
in procedure windows III-404

Find Selection II-54
in procedure windows III-404

Find Text dialog III-24

Find Text in Multiple Windows dialog II-55

FindAPeak operation V-246

FindContour operation V-239

FindDimLabel function V-240

FindDuplicates operation V-240

finding
FindSequence operation V-252
FindValue operation V-253
in Data Browser II-119
in tables II-249
procedures III-404
rulers in notebooks III-12
search expressions II-3
table values II-249
text II-54
text in command window II-11
text in notebooks III-24
text in procedure windows III-404
wave value V-252–V-253

FindLevel operation III-288, V-242–V-244

FindLevels operation III-288, V-244–V-245

FindLevelXY macro III-289

FindListItem function V-245

FindPeak operation III-290–III-292, V-247–V-248

FindPointsInPoly operation V-248

FindRoots operation V-248–V-252
polynomial coefficients V-251

FindSequence operation V-252

FindValue operation V-253

FIR filters III-300
high pass example III-308
low pass example III-304
notch example III-310

fit constants III-189

fit_
curve fitting destination wave name III-196

FitFunc keyword III-252, V-255

FitFunc subtype IV-205

fixed dimension
in tables II-264

fixed field text files
blanks II-138
field widths II-138
loading II-138
number of columns II-138
padding II-137
spaces II-137

fixed-length strings
HDF5 II-219

flags
in operations IV-10, IV-12
expressions require parentheses V-15
in user functions IV-31

flat-top windows V-227

Fletcher32
HDF5 compression II-213

floating panels V-687–V-688
WinList function V-1099
WinName function V-1100

floating point
double-precision II-66
single-precision II-66

floating point data
in Igor Text files II-151
VAX II-167

floating point truncation error
and equality operator IV-8

floor function V-255

flow control
break keyword V-54
break statements IV-48
continue keyword V-85
continue statements IV-48
default keyword V-146
do-while V-165
do-while loops IV-45
extraneous text after IV-114
for loops IV-46
for-endfor V-259
for-var-in-wave V-259

if-else-endif IV-40, V-357
if-elseif-endif IV-40, V-357
if-endif IV-40, V-357
in macros IV-120
in user functions IV-40
keywords V-13
loops IV-45
range-based for loops IV-47
return keyword V-809
strswitch statements V-1002
switch statements IV-43, V-1011
try-catch-endtry statements IV-49, V-1047
while loops IV-46
FMaxFlat operation V-255
Folder button II-28
folders
alias creation V-111
browsing II-117
CopyFolder operation V-105
copying V-105
CreateAliasShortcut operation V-111
creating via FTP IV-275
data folders II-108–II-123
DeleteFolder operation V-156
deleting V-156
deleting via FTP IV-275
downloading via FTP IV-273
experiment folder II-17
FTPCreateDirectory operation V-265
FTPDelete operation V-267
GetFileFolderInfo operation V-300
Igor Extensions folder II-32
Igor Help Files folder II-31
Igor Pro Folder II-30, IV-166
Igor Pro User Files II-31
Igor Procedures folder II-32
information about V-300
missing II-28
MoveFolder operation V-661
moving V-661
names
platform-related issues III-450
SetFileFolderInfo operation V-845
setting information about V-845
shortcut creation V-111
special II-30
symbolic paths II-22–II-24
uploading via FTP IV-274
User Procedures folder II-32, IV-166
WaveMetrics Procedures folder II-32, IV-166
FontList function V-256
fonts
cross-platform issues III-453
default V-147, V-149, V-297
in controls V-149
size V-297
style V-297
functions
FontList function V-256
FontSizeHeight function V-257
FontSizeStringWidth function V-258
in an axis label II-318
in annotation text III-54
in annotations III-36
in graphs II-288, II-308
in notebooks III-8–III-9
in procedure windows III-406
in tick labels II-308
missing
substitution III-453
notebook rulers III-7, III-12
relative size
in annotations III-36
size
in an axis label II-318
in annotations III-36, III-43
style
in annotations III-37, III-54
substitution for missing III-453
Text2Bezier operation V-1023
FontSizeHeight function V-257
FontSizeStringWidth function V-258
footers
in notebooks III-6, III-20
in procedure windows III-405
programming in notebooks III-26
for loops
in user functions IV-46
for-endfor statements V-259
for-var-in-wave statements V-259
format strings IV-259
engineering units V-771
printf operation V-770–V-772
formatted notebooks
text encodings III-472
formatted text
axis labels III-53
DrawUserShape operation III-53
in control titles III-53
in ListBox controls III-53
in SetVariable controls III-53
formatted text notebooks
MBCS III-472
FORTRAN files
features of II-127
loading II-126
Fourier transforms III-270–III-274
compared to FFTs III-272
discrete wavelet transform III-283
harmonic analysis V-182
periodograms V-182
phase polarity III-273

windows V-182
X scaling and units changed III-271
FPClustering operation V-259
distance map V-259
fprintf
stderr V-261
stdout V-261
fprintf operation IV-194, IV-259–IV-261, V-260
conversion specifications IV-260
example IV-261
for debugging IV-261
frame grabbers
data acquisition IV-312
GetCamera operation V-296
ModifyCamera operation V-600
NewCamera operation V-678
frames
for notebook pictures III-14
in page layouts II-483
FReadLine operation IV-195, V-261
byte order marks V-263
cross-platform issues III-453
text encodings V-261, V-263
free axis II-278–II-279
creating V-470, V-679
killing V-470
modifying V-609
free cursors II-320
free data folder
lifetime IV-97
free data folders IV-96–IV-99
converting to global IV-99
creating V-680
deletion of waves IV-98
Multithread keyword example IV-325
NewFreeDataFolder function V-680
objects lifetime IV-98
free dimension
in tables II-264
free waves IV-91–IV-96, IV-207
converting to global IV-96
created from local waves IV-91
default name _free_ IV-95
Duplicate operation V-186
Extract operation V-214
leaks IV-94, IV-207
lifetime IV-92
Multithread keyword example IV-327
names of IV-91, IV-95, V-186, V-526, V-680
programming example IV-200
tracking IV-207
type of V-1092
wave leaks V-1087
WaveType function V-1092
Freedman-Daiconis method III-126
Frei–Chen edge operator V-370
frequency
InstantFrequency operation V-443
fresnel functions V-264
fresnelCos function V-264
fresnelCS function V-264
fresnelSin function V-264
Frobenius norm
MatrixFactor operation V-538
frustrum projection
Gizmo II-423
FSetPos operation IV-195, V-264
FStatus operation IV-195, V-265
FTP IV-272–IV-276
ASCII transfers IV-275
binary transfers IV-275
creating directories IV-275
creating help file links IV-257
deleting directories IV-275
downloading directories IV-273
downloading files IV-273
example experiment IV-272
FetchURL function V-221
FTPCreateDirectory operation V-265
FTPDelete operation V-267
FTPDownload operation V-268
FTPUpload operation V-271
GET method V-1054
image transfers IV-275
limitations IV-273
opening browser for URLs V-54
path specification IV-274–IV-275
PUT method V-1054
transfer types IV-275
transferring files III-448
troubleshooting IV-275
uploading directories IV-274
uploading files IV-274
URLRequest operation V-1054
FTPCreateDirectory operation V-265
FTPDelete operation V-267
FTPDownload operation V-268
FTPUpload operation V-271
FuncFit operation V-273
FuncFitMD operation V-276
FUNCREF keyword IV-107–IV-108, V-278
example IV-107
FuncRefInfo function V-278
Function Execution Error dialog III-415, IV-110
function keys in user menus IV-136
Function keyword V-279, V-1005
function references IV-107–IV-108
example IV-107
FUNCREF keyword V-278
FunctionInfo function V-279
FunctionList function V-284
FunctionPath function V-287

- functions IV-11, V-731
  - aborting threads IV-337
  - bits
    - setting IV-12
  - by category V-7–V-12
  - complex V-7
  - conversion V-7
  - date and time V-7
  - definition I-5
  - DoPrompt statements V-167
  - exists function V-207
  - exponential V-7
  - external IV-208
  - FindRoots operation V-248
  - FunctionList function V-284
  - hook functions IV-280–IV-292, IV-339
    - named window hooks IV-293
    - SetIgorHook operation V-848
    - SetWindow operation V-865
    - static IV-281
    - subwindows IV-294
    - unnamed window hooks IV-305
    - user-defined hook functions V-848
    - window hooks V-865
  - local variable keywords V-13
  - MultiThread keyword V-673
  - numbers V-7
  - numeric types IV-11
  - overriding V-731
  - parameters IV-10
  - plotting III-320–III-322
    - example III-320
    - using dependencies III-321
  - printing result of IV-11
  - roots of III-338–III-343, V-248
    - nonlinear, 1D III-339
    - nonlinear, 2D III-341
      - cautions III-343
    - polynomials III-338
  - rounding V-7
  - rtFunctionErrors pragma IV-55
  - runtime stack information V-314
  - special V-8
  - Static keyword V-906
  - statistics V-10
  - subtype V-66, V-763, V-858
  - syntax guide V-15
  - ThreadSafe keyword V-1033
  - trigonometric V-7
- fuzzy entropy III-355
- fuzzy means III-355

G

- GalleryGlobal keyword
  - in independent modules IV-244
- gamma function V-288
- gammaEuler function V-288
- gammaNoise function V-289
- gammInc function V-288
- gammln function V-289
- gammp function V-289
- gammq function V-290
- gaps in graphs II-303
- gauss
  - curve fitting function III-206
- gauss (Gaussian) curve fit
  - 1D V-127
  - 2D V-128
- Gauss function V-290
- gauss transform
  - discrete V-215
- Gauss-Jordan method
  - MatrixInverse operation V-541
- Gauss-Markov linear model
  - MatrixInverse operation V-541
- Gauss1D function V-291
- gauss2D
  - curve fitting function III-212
- Gauss2d function V-291
- gauss3D filter V-372
- Gaussian filtering III-294
- gaussian quadrature V-448
- GBLoadWave operation II-166, V-291
- gcd function V-294
- GDI III-507, V-866
- GDI+ III-506, V-866
- Gear method V-454
- general binary files
  - GBLoadWave operation V-291
  - loading II-166
- general numeric format
  - in tables II-255
- general perspective projection V-781
- general text files II-138–II-142
  - blocks II-138
  - column labels II-141
  - commas II-138
  - decimal character II-141
  - example II-138
  - features of II-127
  - headers II-138
  - labels II-138, II-140
  - loading matrices II-140
  - loading process II-140
  - missing values II-141
  - saving waves II-179
  - saving waves as V-813
  - skipping lines II-141
  - spaces II-138
  - tabs II-138
  - trouble-shooting II-141

    tweaks II-141
    versus delimited text II-139
    X scaling II-140
Generate Commands dialog III-29
Generate Notebook Commands dialog III-29
GeometricMean function V-295
geometry
    related operations V-4
GET method IV-277
    URLRequest operation V-1054
GetAxis operation V-295
GetBrowserLine function V-295
GetBrowserSelection function V-296
GetCamera operation V-296
GetDataFolder function V-296
GetDataFolderDFR function V-296
GetDefaultFont function V-297
GetDefaultFontSize function V-297
GetDefaultFontStyle function V-297
GetDimLabel function V-298
GetEnvironmentVariable function V-298
GetErrMessage function V-299
GetFileFolderInfo operation V-300
GetFormula function V-303
GetGizmo operation V-303
GetIndependentModuleName function V-303
    independent modules IV-243
    pop-up menus IV-242
GetIndexedObjName function V-303
GetIndexedObjNameDFR function V-304
GetKeyState function V-304
GetLastUserMenuInfo operation V-306
GetMarquee operation V-309–V-311
GetMouse operation V-312
GetRTErrMessage function V-313
GetRTError function V-312
GetRTLocation function V-313
GetRTLocInfo function V-314
GetRTStackInfo function V-314
GetScrapText operation V-315
GetSelection operation V-315–V-316
    example III-28
GetUserData function V-316
GetWavesDataFolder function V-317
GetWindow operation V-318
GetWindowBrowserSelection function V-323
GIF files
    file info V-372
Gizmo II-95, II-406–II-473
    2D waves II-406
    3D bar plots II-457
    3D scatter plot
        data format II-451
    3D scatter plot tour II-407, II-411
    3D scatter plots II-455
    3D waves II-407
    alpha II-428, II-432
    ambient light II-432
    annotations II-444–II-445, II-472
    AppendToGizmo operation V-35
    attribute list II-416–II-417
    attributes II-419
        global II-420
        internal II-419
    autoscaling II-421
    Axes Properties dialog II-450
    axis coordinate system II-421
    axis cue II-415
    axis cue objects II-448
    axis labels II-471
    axis objects II-449
    box axes II-449
    box objects II-440
    Calculate Normals II-432
    clipping II-421, II-470
    color material II-429
    color scale bars II-472, V-74
    color specification II-428
    color tables II-431
    color waves II-430
    ColorScale operation V-74
    colorscales II-445
    ColorTab2Wave II-429
    compatibility II-470
    coordinate system II-420
    coordinate transformation II-436
    crystals II-457
    custom axes II-450
    cylinder objects II-442
    data conversion V-428
    data objects II-453
    Default Output Resolution Factor II-465
    diffuse light II-432
    directional lights II-433
    disk objects II-443
    Display Advanced Options Menus II-466
    display list II-416–II-417
    display volume II-421
    display window II-414
    distance sorting II-432
    drawing objects II-436
    Enable Transparency Blend II-428, II-433
    Export Graphics dialog II-465
    ExportGizmo operation II-465, V-213
    exporting II-465, V-213, V-826
    frustrum projection II-423
    GetGizmo operation V-303
    GizmoInfo function V-323
    GL_ constants II-471
    group objects II-466
    guided tour of II-407
    home orientation II-425

  - hook functions II-472
  - image plot
    - data format II-452
  - image plots II-458
  - info window II-416
  - isosurface II-433
  - isosurface plot
    - data format II-453
  - isosurface plots II-464
  - Light Properties dialog II-433
  - lights II-433, II-469
  - line objects II-436
  - LookAt projection II-424
  - main transformation II-436
  - matrix of X values II-406
  - matrix4x4 objects II-469
  - ModifyGizmo operation V-611
  - NaNs II-453
  - NewGizmo operation V-681
  - normals II-432
  - object list II-416–II-417
  - objects II-418
  - opacity II-428
  - OpenGL II-407, II-470
  - orthographic projection II-422
  - parametric surface plot
    - data format II-452
  - path plot
    - data format II-451
  - path plots II-454
  - Path Properties dialog II-455
  - perspective projection II-423
  - pie wedge objects II-447
  - positional lights II-434
  - printing II-464
  - projections II-422
  - quad objects II-438
  - quadric objects II-467
  - quaternion II-427
  - recreation macros II-471
  - reflection II-432
  - related operations V-2
  - RemoveFromGizmo operation V-792
  - RGBA II-428
  - ribbon plot
    - data format II-451
  - ribbon plots II-462
  - Rotate to Match II-416
  - rotation II-424
  - SaveGizmoCopy operation V-821
  - SavePICT operation II-465
  - Scatter Properties dialog II-456
  - shading II-433
  - shininess II-429, II-432, II-472
  - Specify Color II-419
  - specular light II-432
  - sphere objects II-440
  - string objects II-444, II-471
  - subwindows II-469
  - subwindows in layouts II-497
  - surface plot
    - data format II-451
  - surface plot tour II-410, II-412
  - surface plots II-459
  - Sync to Gizmo II-416
  - tetrahedron objects II-445
  - textboxes II-472
  - texture objects II-467
  - tick mark labels II-471
  - tool palette II-414
    - arrow tool II-414
    - aspect ratio tool II-414
    - axis tool II-414
    - hand tool II-414
    - home tool II-415
    - rotation tool II-415
    - stop tool II-415
  - translation II-425
  - translucency II-432
  - transparency II-428, II-432, II-470
  - triangle objects II-437
  - triplet axes II-450
  - triplet waves II-406
  - troubleshooting II-469
  - Use Global Attributes II-419
  - viewing volume II-421
  - viewport projection II-424
  - Voronoi interpolation II-412
  - voxelgram plots II-463
  - VRAM II-407, II-465
  - windows II-413
- GizmoInfo function V-323
- GizmoPlot subtype IV-205
  - keyword V-323
- GL_ constants II-471
- GLM
  - MatrixInverse operation V-541
- global procedure files III-396, III-399, IV-25
  - Igor Pro User Files folder III-399
- global variables II-104, IV-61
  - created by Igor operations IV-114
  - direct reference IV-114
  - for packages IV-249
  - in user functions IV-70
  - using structures instead IV-103
- global waves IV-91, IV-96, IV-207
  - tracking IV-207, V-1087
  - WaveType function V-1092
- gnoise function V-323
- gnomonic projection V-781
- Go Back and Go Forward II-52
- Go button in debugger IV-216

- GPIB
  - data acquisition IV-312
- gradient fills III-498
- gradient plots II-294, V-614
- Graph Macros submenu II-48, II-350
- Graph menu II-277
- Graph Now button in Curve Fitting dialog III-192
- graph size problems II-289
- Graph subtype IV-204
  - keyword V-325
- graphics
  - alpha III-498
  - color blending III-498
  - features of II-127
  - fill patterns III-498
  - gradient fills III-498
  - translucency III-498
- graphics formats III-96, III-102
  - exporting III-98, III-104
- graphics technology III-506, V-866
  - alpha III-507
  - Core Graphics III-507
  - enhanced metafiles III-507
  - GDI III-507
  - GDI+ III-506
  - Quartz III-507
  - Retina III-507
  - SVG III-507
  - transparency III-507
- GraphMarquee subtype IV-204, V-310
  - keyword V-325
- graphMode
  - printing graphs II-323
- GraphNormal operation V-325
- GraphPopup menu IV-138
- graphs II-276–II-354
  - 3D II-95
  - absolute width and height II-289
  - AddDropLine macro III-77
  - adding new axes II-280
  - annotation position III-40–III-42, III-46–III-47
  - annotations III-34–III-53
  - appending waves to II-280, II-348–II-349
    - example I-23
  - AppendToGraph operation V-35
  - arrow markers II-294, V-614
  - aspect ratio II-289
  - auto ticks II-308
  - auto width and height II-289
  - autoscaling and offsets II-302
  - autoscaling modes II-285, II-287, II-348
  - axes II-278, II-306–II-319, II-324–II-353, V-626
    - AxisValFromPixel function V-45
    - colors II-308, V-634
    - GetAxis operation V-295
    - PixelFromAxisVal function V-743
    - SetAxis operation V-835
  - background color III-442, V-634
  - bars II-295
  - bars to next II-296, II-360
  - bevels III-496
  - blank II-278
  - calibrator bars III-68
  - category plot options I-31
  - CheckDisplayed operation V-68
  - close graph (examples) I-26
  - color scale bars V-74
  - ColorScale operation V-74
  - comma as decimal separator II-288
  - complex traces II-303
  - computed manual ticks II-308
  - control bar V-88
    - color V-634
    - default control appearance V-147
    - default font V-149
  - ControlNameList function V-94
  - controls III-441
    - drawing limitations III-442
  - crossing axes II-308
  - cursors II-285, II-319–II-321
    - colors II-320
    - horizontal coordinate V-340
    - moving cursor calls function IV-339–IV-340
    - vertical coordinate V-1067
    - z value from contour trace V-1120
  - dash patterns for traces V-837
  - dashed lines II-294
  - data acquisition II-347–II-348
  - date/time axes II-307, II-315–II-317
    - custom date formats II-316
  - default font II-288, V-147
  - display modes II-290
  - Display operation V-161–V-163
  - drag and drop II-280
  - drawing coordinate systems III-67–III-68
  - drawing layers III-68
  - dynamic updating II-276
  - editing waves by drawing (example) I-21–I-22, I-29
  - end caps III-496
  - error bars II-304
  - error ellipses II-305, V-200
  - error shading II-305, V-200
  - ErrorBars operation V-199–V-203
  - expand II-285
  - expansion II-352
  - expansion onscreen V-611
  - exporting II-324, V-826
  - FFT and IFFT side effects III-273
  - fill between traces II-297
  - fill to next II-296
  - fill type II-295

fixed size II-289
fling mode II-286
font II-288, II-308
free axis II-279
  creating V-470, V-679
  killing V-470
  modifying V-609
free axis (example) I-36
gaps II-284, II-303
gradient plots II-294, V-614
grids II-309–II-310
  color II-309
  colors V-634
  styles II-309
    illustrations II-310
grouping
  add to next II-296
  draw to next II-296
  keep with next II-296
  next trace defined II-296
  pop-up II-296
  stack on next II-296
  trace order affects II-297
histogram bars II-295
in notebooks III-18
info panel II-319–II-321
  HideInfo operation V-347
  ShowInfo operation V-869
joins III-496
KillFreeAxis operation V-470
legend position III-40–III-42
line size II-294–II-295
line style II-294, II-348
linking in notebooks III-18
list of V-1098, V-1100
live II-347–II-348
log axes II-315
log or linear axes II-307
magnification II-352
magnification on screen V-611
making a new category graph
  example I-30
making a new graph II-45, II-277, II-348–II-349, V-161–V-163
  example I-15
manual scaling mode II-285, II-287
manual ticks II-312–II-314
margin adjust (examples) I-37, I-39
margins II-288
marker colors II-292
marker size as f(z) II-299
markers II-288, II-291
  as f(z) II-298
  numeric codes II-300
  table of II-300
marquee for scaling II-285
masking traces V-617
  example V-626
mirror axis II-307–II-308
miters III-496
Modify Graph dialog II-288–II-290
ModifyContour operation V-600–V-606
ModifyFreeAxis operation V-609
ModifyGraph operation V-611–V-635
ModifyImage operation V-635–V-639
modifying annotations III-35
modifying styles II-290–II-306
modifying traces appearance II-290–II-306, V-613
movies IV-245
multiple axes (example) I-36, II-324–II-325
names II-277, II-350
names and titles II-45
NaNs and INFs in graphs II-284
New Graph dialog II-277, II-349
NewFreeAxis operation V-470, V-679
next trace defined II-296
of complex waves II-303
oscilloscope displays II-347–II-348
page setups II-323, II-348
panning II-286
  fling mode II-286
per unit width and height modes II-289
pixel coordinates V-45, V-743
PixelFromAxisVal function V-743
plan width and height modes II-289
plot area II-279, II-289, V-320
  affected by annotations III-41
polar graph macros III-73
pop-up menu for scaling II-285
position II-348
positioning traces II-286
preferences II-348–II-349, II-363, II-380, II-403
Print Preview II-323
printing II-323, II-348, V-773–V-774
  fill page II-323
  same aspect II-323
quick append II-348, V-36
recreating from macro II-350
recreation macros V-1101
related functions V-10
related operations V-1
relation to layouts I-3
relation to waves I-3
removing
  contour plots V-791
  image plot V-794
  traces II-283
  waves V-792
ReorderImages operation V-797
reordering traces II-297
ReorderTraces operation V-798

ReplaceWave operation V-801
replacing traces II-284
reverse plotting II-287, V-835
saving II-323
  SaveGraphCopy operation V-821
  with waves II-323, V-821
saving as macro II-350
scaling II-284–II-288
screen dumps V-828
scrolling mode V-319, V-321, V-866
setting axis range II-286–II-288
shortcuts II-353–II-354, III-446
shrink II-286
size II-348, V-319
Slider operation V-874
split axes II-347
Split Axis procedure file II-347
stacked plots II-288, II-324–II-326, V-626
  examples II-307, II-324
staggered plots (examples) II-326
sticks and markers to next II-296
sticks to next II-296
style macros II-350–II-352
subwindows III-82
subwindows in layouts II-497
surface plots II-95
swap X & Y axes II-288
tag position III-46–III-47
target window II-277
ternary II-317
textbox position III-40–III-42
tick mark control II-308
title II-277–II-278
trace color II-298
trace names II-282, V-620
TraceInfo function V-1042
TraceNameList function V-1044
TraceNameToWaveRef function V-1045
user ticks from waves II-308
vector plots II-294, V-614
volumetric II-95
wave list V-320
wave names by index V-1079
waveform data II-278
WaveName function V-1079
waves in V-1080
width II-288, II-348
window color V-634
window names V-1048
window recreation macro II-350
wintype function V-1103
wireframe plots II-95
X wave from trace V-1119
XY data II-278, V-1118–V-1119
zero line II-310
zooming II-285
  zooming, panning (examples) I-35
graphSize
  printing graphs II-323
GraphStyle subtype IV-204
  keyword V-325
GraphWaveDraw operation III-75, V-325
GraphWaveEdit operation III-75, V-326
GraphWaveNormal operation III-76
gray-level co-occurrence matrix V-375
grayscale
  TIFF files II-158
grayscale conversions
  convert2gray V-418
greater than IV-6
greatest common divisor V-294
greedy
  regular expressions IV-187
Greek letters III-491
Grep operation IV-176, V-327
  text encodings V-329
GrepList function V-333
GrepList operation IV-176
GrepString function IV-177, V-334
grids
  for controls V-1041
  for drawing III-66, V-1041
  ToolsGrid operation V-1041
grids in graphs II-309–II-310
  colors II-309
  drawing level V-627
  styles II-309
GridStyle subtype IV-204
  keyword V-334
group objects
  Gizmo II-466
GroupBox controls III-416, III-426
  drawing order of V-148
GroupBox operation V-334
guesses in curve fitting III-189
  graphing III-192
guided tour of Gizmo II-407
guided tour of Igor I-12
GuideInfo function V-336
GuideNameList function V-337
GW Instruments data acquisition package IV-312
GZIP
  HDF5 compression II-213

H

Haar wavelet transform V-140, V-189
half power point
  smoothing V-879
Hamming window function V-183, V-226, V-1097
  for images V-436
hand tool

Gizmo II-414
Hanning operation V-337
Hanning window function III-277, V-183, V-226, V-337, V-1097
for images V-436
Haralick’s texture parameters V-375
harmonic analysis V-182
Hartley Transform V-419
Hartley transform III-364
Hash function V-338
hashing
WaveHash function V-1073
HCluster operation III-162, V-338
hcsr function V-340
HDF
HDF5CloseFile operation V-341
HDF5 II-183–II-226
array data II-207
attributes II-198
HDF5 Browser II-195
large II-222
bitfields II-211
browser II-184, II-193
case sensitivity II-203
compound datatypes II-212
compression II-213
dataset region references II-209
datatypes II-204
dimension scales V-342
dimensions II-204
dumping II-197
enums II-211
filters II-217
fixed-length strings II-219
functions II-197
guided tour II-183
HDF5 packed experiment files II-16, II-223
HDF5AttributeInfo function V-341
HDF5CloseGroup operation V-341
HDF5Control operation V-341
HDF5CreateFile operation V-341
HDF5CreateGroup operation V-341
HDF5CreateLink V-342
HDF5DataInfo structure V-342
HDF5DatasetInfo function V-342
HDF5DatatypeInfo structure V-342
HDF5DimensionScale operation V-342
HDF5Dump operation V-342
HDF5DumpErrors operation V-342
HDF5FlushFile operation V-343
HDF5LibraryInfo function V-343
HDF5LibVerLowBound II-222
HDF5LinkInfo function V-343
HDF5LinkInfoStruct structure V-343
HDF5ListAttributes operation V-343
HDF5ListGroup operation V-343
HDF5LoadData operation V-344
HDF5LoadGroup operation V-344
HDF5LoadImage operation V-344
HDF5MakeHyperslabWave II-202
HDF5OpenFile operation V-344
HDF5OpenGroup operation V-344
HDF5SaveData operation V-344
HDF5SaveDataHookStruct structure V-344
HDF5SaveGroup operation V-345
HDF5SaveImage operation V-345
HDF5TypeInfo function V-345
HDF5UnlinkObject operation V-345
hyperslabs II-202
Igor capabilities II-206
Igor limitations II-206
IGORVariable attribute II-205
images II-204
large attributes II-222
library compatibility II-222
loading programmatically II-187
multitasking II-206
multithreading II-206
names
conflicts III-505
nested datatypes II-211
object references II-208
opaque data II-211
operations II-197
overview of II-184
paths II-221
procedure files II-198
references II-208
related operations II-197, V-6, V-12
row major II-204
saving programmatically II-189
string charsets II-219
string formats II-219
string padding II-219
subsets II-202
text encodings II-221
variable-length datasets II-207
variable-length strings II-219
Windows paths II-221
HDF5 Browser II-193
guided tour II-184
HDF5 Browser dump II-197
HDF5 Browser.ipf II-198
HDF5 compression II-213–II-217
chunking II-214
default compression II-214
Fletcher32 II-213
GZIP II-213
HDF5SaveDataHook IV-290
HDF5SaveDataHook hook function II-215
HDF5SaveDataHookStruct structure V-344
HDF5SaveGroup operations II-214

IgorInfo function V-361
layout II-214
Miscellaneous Settings dialog II-17
NBit II-213
SaveExperiment operation II-214
saving experiments II-16
ScaleOffset II-213
shuffle II-213
SZIP II-213
TestHDF5ExperimentCompression function II-17
HDF5 packed experiment files II-223
organization II-223
writing II-225
HDF5 packed experiments II-16
HDF5 Utilities.ipf II-198
HDF5AttributeInfo function V-341
HDF5CloseFile operation V-341
HDF5CloseGroup operation V-341
HDF5Control operation V-341
HDF5CreateFile operation V-341
HDF5CreateGroup operation V-341
HDF5CreateLink operation V-342
HDF5DataInfo structure V-342
HDF5DatasetInfo function V-342
HDF5DatatypeInfo structure V-342
HDF5DimensionScale operation V-342
HDF5Dump operation V-342
HDF5DumpErrors operation V-342
HDF5FlushFile operation V-343
HDF5LibraryInfo function V-343
HDF5LibVerLowBound II-222
HDF5LinkInfo function V-343
HDF5LinkInfoStruct structure V-343
HDF5ListAttributes operation V-343
HDF5ListGroup operation V-343
HDF5LoadData operation V-344
HDF5LoadGroup operation V-344
HDF5LoadImage operation V-344
HDF5MakeHyperslabWave function II-202
HDF5OpenFile operation V-344
HDF5OpenGroup operation V-344
HDF5SaveData operation V-344
HDF5SaveDataHook hook function II-215, IV-290
HDF5SaveDataHookStruct structure V-344
HDF5SaveGroup operation V-345
compression II-214
HDF5SaveImage operation V-345
HDF5TypeInfo function V-345
HDF5UnlinkObject operation V-345
HEAD method
URLRequest operation V-1054
headers
in delimited text files II-134
in general text files II-138
in notebooks III-6, III-20
in procedure windows III-405
in text files II-126
programming in notebooks III-26
URLRequest operation V-1054
help II-2–II-5
checking links IV-258
closing help windows II-3
compiling help files IV-257
creating help files IV-255
creating links IV-257
DisplayHelpTopic operation V-163
FAQ II-5
for user-defined functions IV-259
from command window II-12
FTP links in IV-257
help windows II-3
hiding help windows II-3
Igor Help Browser II-2
in simple input dialog IV-145
in simple input dialogs IV-148
killing help windows II-3
kinds of help II-2
ListBox operation V-493
magnification II-53
overview I-8, II-2
programmatic display of topics V-163
related topics IV-256
shortcuts II-5
subtopics IV-256
syntax of help file IV-256
technical support II-4
templates
for a function III-397
for an operation III-397
tooltips II-3
topics IV-256
URLs IV-257
Web links in IV-257
zooming II-53
help files
CloseHelp operation V-71
compiling IV-257
creating IV-255
Igor Help File Template IV-255, IV-257
Igor Help Files folder II-2, II-31
links
checking IV-258
creating IV-257
finding IV-257
magnification II-53
setting default II-53
More Help Files folder II-2
OpenHelp operation V-721
syntax IV-256
URL links IV-257
Web links IV-257

zooming II-53
setting default II-53
help windows II-3
executing commands from II-3, III-4
Go Back and Go Forward II-52
hermite function V-345
hermite functions V-345–V-346
hermiteGauss function V-346
Hermitian transposition
MatrixOp operation III-144
hex dump
in tables II-260
hexadecimal literals IV-12
hexadecimal numbers
in tables II-245, II-255, II-257
representation of II-257
HFS paths V-734
hidden procedure files III-402
changes in Igor functionality III-403
creating III-402
hide
pragma keyword V-346
hide events
window hooks IV-304
Hide Info II-319
hide keyword V-346
hide pragma IV-54
Hide Tools III-62
hideable IV-127
HideIgorMenus operation V-346
HideInfo operation V-347
HideProcedures operation V-347
HideTools operation V-347
hiding
help windows II-3
HideProcedures operation V-347
notebooks III-4, V-699
procedure windows III-399, V-642
traces II-303
vs killing windows II-46
windows II-46
hierarchical clustering III-162–III-170
dendrogram example III-168
dendrogram wave III-167
dissimilarity V-339
dissimilarity matrix III-162–III-163
distance map III-162
HCluster operation V-338
linkage III-166, V-339
high pass band
filtering III-303
high pass filters
example III-308
high resolution
displays III-507
in panels III-456
high-DPI III-508
high-resolution displays III-508
Hilbert transform III-280
HilbertTransform operation V-348
Hill equation curve fit V-128
HillEquation
curve fitting function III-210
hinges
box plots II-332
Histogram
Freedman-Daiconis method III-126
histogram III-125–III-132
accumulating III-128
adaptive histogram equalization III-354
bin range III-125–III-128
bin width III-125–III-128
destination wave III-127
example I-53, III-128
histogram equalization III-354
Igor 1.2 compatibility III-128
image analysis V-378–V-379
integrating III-132
of images III-372
histogram clusters III-355
Histogram dialog III-128
Histogram operation III-125–III-132, V-349–V-353
histograms
JointHistogram operation V-466
TextHistogram operation V-1029
history archive II-10
history area II-8–II-13
CaptureHistory function V-58
CaptureHistoryStart function V-58
colored II-10
converting to UTF-8 III-489
copying from II-10
extracting text programmatically V-58
history archive II-10
history carbon copy II-10
limiting size of II-10
magnification II-12, II-53
printing to V-769–V-770
saving II-10
searching II-11
selecting text II-10
text encodings III-470
zooming II-53
history carbon copy II-10
holding coefficients in curve fitting
example III-184
home folder II-23, II-25, II-28
for packed experiments II-16
for unpacked experiments II-17
home orientation
Gizmo II-425
home symbolic path II-23

home text files II-56
home tool
  Gizmo II-415
home waves II-87
hook functions
  earlyKeyboard events IV-302
  Gizmo II-472
  independent modules IV-240
  keyboard events IV-300
  modified events IV-299
  named window hooks IV-293
  regular modules IV-237
  subwindows IV-294
  tooltips V-868
  unnamed window hooks IV-305
HorizCrossing free axis II-278–II-279
horizontal dimension in tables II-263
horizontal index row II-261
Hough transform III-364, V-421
HSL conversions
  hsl2rgb V-421
  rgb2hsl V-424
HTML
  cascading style sheets III-22
  character encoding III-23
    Native III-23
    Shift-JIS III-23
    UTF-2 III-23
    UTF-8 III-23
  embedding in notebooks III-23
    examples III-23
  graphics in notebooks III-23
    JPEG III-23
    picture frames III-23
    PNG III-23
  HTML 4.01 specification III-22
  notebook character formatting III-23
  notebook paragraph formatting III-22
  notebooks III-21
  saving notebooks III-21
  specifications III-22
  standards III-22
  tooltips IV-311
HTTP IV-276–IV-278
  DELETE method V-1054
  downloading files IV-276
  downloading web pages IV-276
  FetchURL function V-221
  GET method IV-277, V-1054
  HEAD method V-1054
  limitations IV-276
  POST method IV-276, V-1054
  proxy servers IV-276
  PUT method V-1054
  queries IV-277
  SSL IV-276
  troubleshooting IV-278
  URLRequest operation V-1054
HTTP POST method
  URLRequest operation V-1058
HTTPS
  URLRequest operation V-1054
Hurwitz zeta function V-1120
hybridmedian filter V-372
hyperbolic functions
  arbitrary precision V-27
  cosine V-109
  cotangent V-110
  sine V-871
  tangent V-1023
hyperG0F1 function V-353
hyperG1F1 function V-353
hyperG2F1 function V-354
hypergeometric function V-353
  confluent V-353–V-354
  generalized V-354
hyperGNoise function V-354
hyperGPFQ function V-354
hyperslabs
  HDF5 II-202

I

i
  suffix in tables II-241
i function V-355
i123 conversions
  rgb2i123 V-424
.ibw extension II-155
ICA operation V-355
identifiability
  curve fitting III-226
if-else-endif V-357
  conditional operator IV-7
  extraneous text after IV-114
  in user functions IV-40
if-elseif-endif V-357
  in user functions IV-40
if-endif V-357
  in user functions IV-40
IFDL III-299
  FMaxFlat operation V-255
IFFT
  wave references IV-75
IFFT operation V-357–V-359
IFFTs
  effect on graphs III-273
  number of points restrictions III-271
  speed III-274
  wave type restrictions III-271
  X scaling and units changed III-271
.ifn file extension

notebooks III-2
Igor
symbolic path II-23
updating II-4
upgrading II-4
Igor and IgorRecent
color tables V-82
Igor binary data
exporting from tables II-252
Igor binary wave files II-154–II-157
.bwav extension II-155
copy to home II-24
features of II-127
file extensions II-155
.ibw extension II-155
in experiments II-19
KillWaves operation II-71
LoadData operation II-156
names II-155
references to II-24
saving II-179
sharing versus copying II-24
Igor Crash Reports.txt IV-341
Igor Extensions
MacOS 10.15 and later III-512
notarization III-512
Igor extensions III-511–III-512, IV-208
creating IV-209
definition I-5
definition of I-5
development systems IV-208
IGOR32 II-32
IGOR64 II-32
third party XOPs III-511
WaveMetrics XOPs III-511
Igor Extensions folder I-6, II-32
Igor Filter Design Laboratory III-299
Igor Help Browser I-8, II-2
Igor Help File Template IV-255, IV-257
Igor help files
compiling IV-257
creating IV-255
FAQ II-5
Igor Help File Template IV-255, IV-257
links
checking IV-258
creating IV-257
finding IV-257
More Help Files folder II-2
syntax IV-256
URL links IV-257
Web links IV-257
Igor Help Files folder II-2, II-31
Igor help system
Igor Help Files folder II-2
overview I-8, II-2
IGOR keyword in Igor Text files II-151
Igor mailing list II-4
Igor Pro 8.ini file III-516
Igor Pro Folder II-30
include statement IV-166
Igor Pro User Files folder II-31
example I-62
global procedure files III-399
packages IV-247
shared procedure files III-400
Igor Procedures IV-25
Igor Procedures folder II-32, III-399
Igor program name
under Windows III-448
Igor TeX III-58
Igor Text files II-150–II-154
BEGIN keyword II-151
blocks II-151
carriage return II-154
commands II-152
creating V-813
data type II-151
data type flags II-151
dates II-151
End keyword II-151
escape codes II-154
examples II-150–II-151
features of II-127
format II-151
IGOR keyword II-151
linefeed II-154
missing values II-151
multidimensional waves II-151–II-153
NaNs II-151
not thread-safe II-152
saving multidimensional waves II-179
saving waves II-179
text waves II-154
times II-151
WAVES keyword II-151
X keyword II-152
X scaling II-152
Igor Tips
for controls III-436
shortcut II-5
Igor version V-359–V-360
localization V-360
Igor XOP Toolkit IV-209
Igor-object pictures III-18
igor-object pictures
platform compatibility III-19
IGOR32 I-2, III-512
Igor extensions II-32
reading huge waves II-35
IGOR64 I-2, III-512
Igor extensions II-32

saving huge waves II-35
IgorBeforeNewHook hook function IV-290
IgorBeforeQuitHook function IV-290
IgorExchange II-4
IgorInfo function V-359
autosave V-361
HDF5 default compression V-361
serial number V-360
IgorMenuHook function IV-291
IgorQuitHook hook function IV-292
IgorStartOrNewHook hook function IV-292
example IV-292
Igortips I-8
IgorUserFiles
symbolic path II-23
IGORVariable attribute II-205
IgorVersion
example IV-110
pragma keyword V-362
IgorVersion function V-362
IgorVersion keyword V-362
IIR filters III-300, III-312, V-234–V-239
coefficients, designing V-238
cutoff frequency III-312
order III-313
ilim function V-362
imag
suffix in tables II-241
imag function V-362
example II-81
image analysis III-352–III-382
2D kernels III-365
3D morphology V-395
adaptive histogram equalization III-354, V-378–V-379
alpha blending V-366
alpha compositing V-368
average 3D filter V-372
averaging V-417
B-spline surfaces V-418
back projection V-418
background subtraction III-379, V-403
Bartlet window V-436
Bartlett window V-436
beams
definition V-420
extracting V-420
bilinear interpolation V-383–V-384
binary dilation V-395
binary erosion V-395
Blackman window V-436
blending images V-366
Catmull-Clark surfaces V-418
close V-395
closing III-369
cmap conversions
cmap2rgb V-418
CMYK conversions
CMYK2RGB V-418
co-occurrence matrix V-375
color III-379
color space conversion III-352–III-353
cmap2rgb V-418
CMYK2RGB V-418
convert2gray V-418
hsl2rgb V-421
rgb2gray V-424
rgb2hsl V-424
rgb2i123 V-424
rgb2xyz V-424
xyz2rgb V-428
color transforms III-352–III-353
columns
extracting V-420
flipping V-419
setting values from a wave V-423
summing V-427
summing all V-427
comparing images V-398
Compositeing images V-368
compressing images V-418
convolution filters III-365
gauss III-365
median III-365
sharpen III-365
convolutions III-360
correlation V-415
correlations III-362
decompressing images V-419
dilation III-368, V-395
dot product V-428
edge detection III-365–III-368
edge detectors V-370
erosion III-368, V-395
extracting color info III-353
extracting ROIs V-425
feature extraction V-426
FFT III-359–III-362
calculating derivatives III-361
calculating integrals III-361
calculating sums III-361
cautions when using III-359
convolutions III-360
correlations III-362
high pass filtering III-360
low pass filtering III-360
spatial frequency filtering III-360
windowing III-362
filling V-409
filling from 1D wave V-419
filtering V-372, V-435, V-539
find lakes V-419

finding flat areas V-419
focus V-373
fuzzy classification V-420
gaussian 3D filter V-372
grayscale conversions
  convert2gray V-418
  rgb2gray V-424
grayscale transforms III-352–III-355
Hamming window V-436
Hanning window V-436
Hartley transform III-364, V-419
histogram equalization III-354, V-378–V-379
histograms III-372, V-378–V-379
Hough transform III-364, V-421
  line detection III-364
HSL conversions
  hsl2rgb V-421
  rgb2hsl V-424
HSL segmentation III-374, V-421
hue segmentation III-353
hybridmedian filter V-372
i123 conversions
  rgb2i123 V-424
image statistics III-371
ImageAnalyzeParticles operation V-363
ImageBlend operation V-366
ImageBoundaryToMask operation V-367
ImageComposite operation V-368
ImageEdgeDetection operation V-370
ImageFocus operation V-373
ImageGenerateROIMask operation V-374
ImageGLCM operation V-375
ImageHistModification operation V-378
ImageHistogram operation V-379
ImageInterpolate operation V-382
ImageLineProfile operation III-372, V-389
ImageMorphology operation V-395
ImageRegistration operation V-398
ImageRemoveBackground operation V-403
ImageRestore operation V-404
ImageRotate operation V-405
ImageSeedFill operation V-409
ImageSkeleton3D operation V-413
ImageSnake operation V-411
ImageStats operation V-414
ImageThreshold operation V-415
ImageTransform operation III-381, V-417
ImageUnwrapPhase operation V-433
ImageWindow operation V-435
indexing V-421
inserting images V-422
interpolation III-359, V-382
k-means clustering V-472
Kaiser window V-437
Kriging V-383
line detection III-364
line profile V-389
masking V-374, V-425
mathematical transforms III-358–III-364
maximum rank 3D filter V-372
median 3D filter V-372
minimum rank 3D filter V-372
morphological operations III-368–III-371
  binary dilation V-395
  binary erosion V-395
  close V-395
  closing III-369
  dilation III-368, V-395
  erosion III-368, V-395
  ImageMorphology operation V-395
  open V-396
  opening III-369
  top hat III-370, V-396
  watershed III-371, V-396
multidimensional morphology V-395
multiplane images V-420, V-426
offsetting V-423
open V-396
opening III-369
operations V-4
padding V-423
  example V-433
particle analysis III-375, V-363
particles
  classification III-375
  morphology V-395
path profile V-389
pixels
  inverting V-422
  matching V-422
planes
  extracting V-420, V-426
    example V-433
  flipping V-420
  inserting V-422
  matching V-422
  removing V-423
  scaling V-425
  summing V-427
  summing all V-427
point finding 3D filter V-372
profiles III-372
projection slice V-423
rectification V-384
references III-382
reflectance V-426
region of interest III-378–III-379
  boundary to ROI mask III-378
  creating ROI wave III-378
  creation V-374
  marquee procedures III-379
  marquee2mask procedures III-379

    - ROI masking III-378
  - registration III-358, V-384, V-398
  - resizing V-423
  - restoring V-404
  - RGB conversions
    - cmap2rgb V-418
    - CMYK2RGB V-418
    - hsl2rgb V-421
    - rgb2gray V-424
    - rgb2hsl V-424
    - rgb2i123 V-424
    - rgb2xyz V-424
    - xyz2rgb V-428
  - ROI masking V-367
  - ROIs III-379, V-425
  - rotating image columns V-425
  - rotating image rows V-425
  - rotation III-357, V-405
  - rows
    - extracting V-420
    - flipping V-420
    - setting values from a wave V-423
    - summing V-427
    - summing all V-427
  - sampling III-359
  - scaling V-425
  - seed fill III-377
  - segmentation III-374
  - selectColor V-425
  - shading V-426
  - shifting V-423
  - shrinkRect V-426
  - spatial transforms III-357
  - stacking images V-427
  - statistics V-414
  - subimage selection III-379
  - subscene extraction V-426
  - surface extraction V-419
  - swap diagonal quadrants V-427
  - textures V-421
  - threshold III-355–III-357, V-415
    - adaptive III-355
    - bimodal III-355
    - examples III-356
    - fuzzy entropy III-355
    - fuzzy means III-355
    - histogram clusters III-355
    - iterated III-355
    - Otsu’s method III-355
  - top hat V-396
  - transforms V-417
    - adaptive histogram equalization III-354
    - color III-352–III-353
    - explicit lookup tables III-353
    - grayscale III-353–III-355
    - histogram equalization III-354
    - hue segmentation III-353
    - RGB to HSL III-353
    - to grayscale III-352
    - to RGB III-352
    - value III-353–III-355
  - unwrapping phase III-374
  - value transforms III-353–III-355
  - vol2surf V-428
  - Voronoi interpolation V-384
  - voronoi tesselation V-428
  - warping V-384
  - watershed V-396
  - wave operations III-358
    - efficiency III-358
  - wavelet transforms III-362–III-363
    - denoising III-363
    - image compression III-362
  - windowing III-362, V-435
    - example V-437
  - X projection V-428
  - XYZ conversions
    - rgb2xyz V-424
    - xyz2rgb V-428
  - Y projection V-428
  - Z projection V-428
- image loading
  - EXIF metadata V-395
- image plot
  - data format II-452
- image plots II-385–II-404
  - _calculated_ X and Y II-387, II-389
  - alpha II-400–II-401
  - appearance preferences II-403
  - Append Image Plot dialog II-404
  - AppendImage operation V-32
  - axes
    - GetAxis operation V-295
    - SetAxis operation V-835
  - axis preferences II-404
  - color
    - 24-bit II-386, II-401
    - ColorTab2Wave operation V-82
    - custom II-401
    - direct II-386, II-401
    - false II-386
    - indexed II-386, II-400–II-401
    - negative II-392
    - reversing II-392
    - RGB II-386, II-401
  - color index wave II-400–II-401
  - color scale bars II-402, III-34, III-47–III-51, V-74
  - color table waves II-399
  - color tables II-393–II-400
  - ColorScale operation V-74
  - combining with contour plots II-367
  - creating II-386–II-387, V-162

Cursor operation V-121
cursors V-121, V-741, V-785
moving cursor calls function IV-339–IV-340
dithering V-637
evenly-spaced data II-389
example II-401
filtering V-372, V-539
flipping II-391–II-392
Gizmo II-458
image instance names II-403
Image Range Adjustment II-393
ImageFilter operation V-372
ImageFromXYZ operation V-373
ImageInfo operation V-380
ImageNameList function V-397
ImageNameToWaveRef function V-398
images
color effects II-395
contrast effects II-395
contrast enhancement II-393
overlays II-393
interpolation V-637
inverting V-835
legends II-402, III-34, III-47–III-51
linear colors II-400
log colors II-387, II-400
logarithmic V-637
MatrixFilter operation V-539
missing data II-386
Modify Image Appearance dialog II-387–II-388
ModifyImage operation V-635–V-639
modifying II-387
names II-403
NaNs II-386
New Image Plot dialog II-404
NewImage operation V-681
opacity II-400–II-401
orientation II-391–II-392
pcsr function V-741
pixel size II-388, II-392
pixels II-386, II-388
preferences II-403–II-404
programming notes II-401, II-403
qcsr function V-785
rectangle edges II-388
related operations V-1
ReplaceWave operation V-801
reversing II-391–II-392
RGB waves II-400–II-401
RGBA waves II-400–II-401
Set Axis Range Dialog II-392
shortcuts II-404
smoothing V-637
translucency II-400–II-401
transparency II-400–II-401
transparent data II-386
unevenly-spaced II-389
with non-gridded data II-391
with XY centers II-389
X and Y coordinates II-388
X and Y waves II-389
image processing
contrast enhancement II-393
Image Range Adjustment II-393
ImageAnalyzeParticles operation V-363
ImageBlend operation V-366
ImageBoundaryToMask operation V-367
ImageComposite operation V-368
ImageEdgeDetection operation V-370
ImageFileInfo operation V-372
ImageFilter operation V-372
ImageFocus operation V-373
ImageFromXYZ operation V-373
ImageGenerateROIMask operation V-374
ImageGLCM operation V-375
ImageHistModification operation V-378
ImageHistogram operation V-379
ImageInfo operation V-380
ImageInterpolate operation V-382
ImageLineProfile operation V-389
ImageLoad operation V-390
ImageMorphology operation V-395
ImageNameList function V-397
ImageNameToWaveRef function IV-198, V-398
ImageRegistration operation V-398
ImageRemoveBackground operation V-403
ImageRestore operation V-404
ImageRotate operation V-405
images
average deviation V-415
average value V-415
averaging V-417
background subtraction V-403
beams
extracting V-420
color scale bars II-402, III-34, III-47–III-51
columns
extracting V-420
flipping V-419
setting values from a wave V-423
summing V-427
summing all V-427
compressing V-418
decompressing V-419
deconvolution V-404
dispersion V-415
exporting
BMP files V-405
DIB files V-826
EMF files V-826
EPS files V-826
ImageSave operation V-405

JPEG files V-405, V-826
PhotoShop files V-405
PICT files V-405, V-826
HiRes V-826
PNG files V-405, V-826
PostScript PICT files V-826
QuickTime files V-405
raw PNG files V-408
SavePICT operation V-826
SGI files V-405
Targa files V-405
TIFF files V-405, V-408
WMF files V-826
features of II-127
filling from 1D wave V-419
HDF5 II-204
ImageFileInfo operation V-372
ImageLoad operation V-390
importing V-390
indexing V-421
inserting V-422
kurtosis V-415
legends II-402, III-34, III-47–III-51
Load Image Dialog II-157
loading II-157
JPEG files V-390
PNG files V-390
Sun Raster files V-390
TIFF files V-390
TIFF stacks II-158
loading BMPs II-158
loading JPEGs II-158
loading PNGs II-157
loading Sun Raster files II-159
loading TIFFs II-158
masking V-425
maximum column location V-415
maximum row location V-415
maximum value V-415
minimum column location V-415
minimum row location V-415
minimum value V-415
multiplane images V-420, V-426
NewImage operation V-681
number of points in ROI V-415
offsetting V-423
padding V-423
example V-433
pixels
inverting V-422
planes
extracting V-420
flipping V-420
inserting V-422
removing V-423
summing V-427
summing all V-427
reflectance V-426
removing V-794
ReorderImages operation V-797
resizing V-423
restoring V-404
RMS value V-415
rotating image columns V-425
rotating image rows V-425
rotation V-405
rows
extracting V-420
flipping V-420
setting values from a wave V-423
summing V-427
summing all V-427
saving waves as II-180
selectColor V-425
shading V-426
shifting V-423
skewness V-415
smoothing V-515
stacking V-427
standard deviation V-415
textures V-421
variance V-415
voronoi tesselation V-428
ImageSave operation V-405
ImageSeedFill operation V-409
ImageSkeleton3D operation V-413
ImageSnake operation V-411
ImageStats operation V-414
ImageThreshold operation V-415
ImageTransform operation V-417
ImageUnwrapPhase operation V-433
ImageWindow operation V-435
imaginary component function V-362
implicit functions
curve fitting III-242–III-244
importing
BMPs III-509
data in tables II-239, II-248
EMFs III-509
FBinRead operation V-219
FReadLine operation V-261
from databases II-181
from Igor experiment file V-500–V-505
GBLoadWave operation II-166, V-291
graphics V-390
Load Image Dialog II-157
with drawing objects III-73
HDF5 II-197, V-6, V-12
ImageLoad operation V-390
images II-157, V-390
BMPs II-158
JPEGs II-158

Load Image Dialog II-157
PNGs II-157
Sun Raster files II-159
TIFF stacks II-158
TIFFs II-158
JCAMPLoadWave operation V-464
JPEGs III-509
limitations on graphics III-509
MLLoadWave operation II-163
notebooks III-20
numeric variables V-219
PadString function V-732
pasted picture resizing III-73
PDFs III-510
PICTs III-509–III-510
PNGs III-509
string variables V-219, V-261
TIFFs III-509
UnPadString function V-1050
waves II-126–II-177, V-219, V-508, V-589
waves in a folder (example) II-175
XLLoadWave operation II-159, V-1115
importing images
PNG versus raw PNG II-157
importing waves
setting wave names II-176
in line
in notebooks III-9
include statements III-400–III-401, IV-25, IV-166
include keyword V-17
independent modules IV-240
IndependentModule pragma IV-240
menu control IV-167
optional IV-167
version control IV-166
incomplete gamma function V-289–V-290
increment in wave assignment II-76
increment operator IV-6, IV-43
Indent Left IV-27
Indent Right IV-27
indentation
in notebooks III-7–III-8
in procedure windows III-405
of procedures IV-26
independent component analysis V-355
independent modules IV-238
containing regular modules IV-242
control action procedures IV-240
debugger IV-239
dependencies IV-243
example IV-239
Execute operation IV-243
hook functions IV-240
include statements IV-240
IndependentModuleList function V-438
limitations IV-240
pictures IV-244
pop-up menus IV-241
procedure windows IV-239
ProcGlobal keyword IV-238
ProcGlobal keywords IV-245
public functions IV-238
qualified names IV-238–IV-239
static functions IV-238–IV-239
user-defined menus IV-241
WinList function V-1099
independent variables
curve fitting III-191
IndependentModule keyword V-437
IndependentModule pragma IV-55, IV-238
GetIndependentModuleName function V-303
include statements IV-240
index sort III-134–III-135
index values
CopyScales operation V-107
in tables II-241, II-261
index columns in tables II-241
multidimensional waves V-160
pasting in tables II-249
viewing in tables II-235
index waves II-77
wave assignment statements II-77
IndexedDir function V-438
IndexedFile function V-440
dot-underscore files V-441
example II-175
sorting file lists V-441
indexing
dimension labels II-82
index waves II-77
multidimensional waves II-92, II-95–II-97
strings IV-16
text waves IV-17
using dimension labels II-93
value waves II-77
waves II-76
IndexSort
example III-135
IndexSort operation III-134–III-135, V-442
applications III-134
IndexToScale function V-442
indirect autosave II-38–II-39
inequality IV-6
INF
in wave assignments II-76
Inf function V-442
infinity function V-442
Info (File Information dialog) III-25
info pane
Data Browser II-115
info panel
HideInfo operation V-347

  - in graphs II-319–II-321
  - ShowInfo operation V-869
- info window
  - Gizmo II-416
- INFs II-83
  - detecting V-715
  - in curve fitting III-232–III-233
  - in delimited text files II-129
  - in graphs II-284
  - in tables II-247
  - in WaveStats operation III-124
  - number of V-1084
  - removing from waves V-1091
  - zapping V-570
- initial guesses in curve fitting III-180, V-131
- initialization commands II-27
- inline parameters
  - in user functions IV-33
- inline wave references IV-73
  - destination waves IV-86
- input devices
  - cross-platform issues III-452
- input/output queues
  - multitasking
    - example IV-333
- Insert File III-13, III-403
- Insert Page Break III-20
- Insert Points dialog II-73
- insert-paste II-248
- INSERTINCLUDE IV-279
- inserting
  - pictures in notebooks V-706
- inserting data II-249
  - in tables II-244
  - in waves II-73
- inserting pictures
  - in notebooks III-13
- inserting text
  - in procedure windows III-403
- insertion cell in tables II-236, II-239, II-244
- InsertPoints operation II-73, V-443
- instance names IV-20
  - contour instance in ModifyContour V-600
- InstantFrequency operation V-443
- Instrutech data acquisition package IV-312
- int declaration
  - local variables in user functions IV-33
  - variables in user functions IV-33
- int keyword V-446
- int64 declaration
  - local variables in user functions IV-33
  - variables in user functions IV-33
- int64 keyword V-446
- integer data II-85
  - in Igor Text files II-152
  - numeric format
    - in tables II-255
- integers
  - 16-bit II-66
  - 32-bit II-66
  - 64-bit II-66, IV-39
  - 64-bit integers complex IV-39
  - 8-bit II-66
  - expressions in user functions IV-38, IV-113
  - in wave assignments IV-9
  - numeric format
    - in tables II-255
  - variable in user functions IV-33, V-446, V-1048
- Integrate operation V-446
- integrate1D function V-448
- Integrate2D operation V-449
- IntegrateODE operation III-322, V-452–V-458
- integrating histogram III-132
- integration III-120
  - area function V-40
  - area under curve III-120
  - areaXY function V-41
  - faverage function V-218
  - faverageXY function V-218
  - gaussian quadrature V-448
  - Integrate operation V-446
  - integrate1D function V-448
  - Integrate2D operation V-449
  - multidimensional III-120
  - of user functions III-336
  - of XY data III-120
  - Romberg V-448
  - trapezoidal V-40–V-41, V-218, V-446, V-448
  - waveform data III-120
  - XY area procedures III-122
- interior annotations III-40
- Internet
  - FetchURL function V-221
  - FTP IV-272
  - FTP transfer V-271
  - FTP transfers V-268
  - HTTP IV-276
  - opening preferred browser V-54
- interp function V-458
  - example II-80, III-110
- Interp2D function V-458
- Interp3D function V-458
- Interp3DPath operation V-459
- Interpolate2 operation III-115, V-459
  - dialog III-117
  - example III-111
  - removing NaNs III-114
- Interpolate3D function V-461
- interpolation III-114
  - 2D
    - ContourZ function V-87
    - Interp2D function V-458

3D
Interp3D function V-458
Interpolate3D function V-461
bilinear V-383
cubic spline III-115
dialog III-117
example II-80, III-109, III-111, III-115
image analysis V-383
in image plots V-637
in wave assignments II-78, II-83
Interpolate2 operation III-115, V-459
linear III-115
methods III-114
not in multidimensional waves II-96
of multidimensional data V-459
of XY data III-109, V-458
of XYZ data V-459, V-899
Resample operation V-803
smoothing spline III-115
smoothing spline algorithm III-118
smoothing spline parameters III-119
spherical V-899
invalid text
in procedure files III-470
in tables II-259
inverse cosine V-19
inverse hyperbolic functions
cosine V-19
sine V-42
tangent V-42
inverse of a matrix V-540–V-541
inverse of a wave V-1090
inverse sine function V-41
inverse tangent V-42
inverse tangent function V-42
inverse trig functions
inverse cosine V-19, V-1090
inverse sine V-41, V-1090
inverse tangent V-42, V-1090
inverseErf function V-463
inverseErfc function V-463
invisible procedure files III-402
changes in Igor functionality III-403
creating III-402
isosurface
transparency II-433
isosurface plot
data format II-453
isosurface plots
Gizmo II-464
issues
dimensions III-455
ItemsInList function V-463
iterate-loop
loop index V-355, V-463
loop limit V-362, V-466
iterated thresholding III-355

J

j function V-463
JacobiCn function V-465
JacobiSn function V-465
JCAMP-DX files II-168
JCAMPLoadWave operation II-168, V-464
jlim function V-466
joins
for drawing tools V-840
for lines and traces III-496
for traces V-616
JointHistogram operation V-466
weighting wave V-466
JPEG files
EXIF metadata V-395
importing V-390
JPEGs
exporting V-405
file info V-372
HTML graphics III-23
importing III-509
loading II-158
saving V-826
Julian dates
dateToJulian function V-144
JulianToDate function V-468
JulianToDate function V-468
justification of text
in annotations III-37
in notebooks III-7–III-8

K

k-means clustering V-472
Kaiser Bessel window function V-183, V-226, V-437, V-1097
Kaiser window function
for images V-437
Kaiser’s maximally flat filter V-255
KDE
StatsKDE operation V-946
violin plots II-337
kernel densitiy estimate
StatsKDE operation V-946
kernel density estimate
violin plots II-337
keyboard
cross-platform issues III-452
events in hook functions IV-300
shortcuts in user menus IV-134, IV-136, V-304
keyword-value packed strings II-84, IV-173
keywords V-13
extraction
by index number V-998

FindListItem function V-245
ItemsInList function V-463
NumberByKey function V-714
RemoveByKey function V-790
RemoveFromList function V-793
RemoveListItem function V-795
ReplaceNumberByKey function V-799
ReplaceStringByKey function V-800
StringByKey function V-997
StringFromList function V-998
WhichListItem function V-1095
flow control V-13
function local variable keywords V-13
insertion
AddListItem function V-20
object references V-13
other programming keywords V-13
procedure declarations V-13
procedure subtypes V-13
syntax guide V-15
kill
defined I-22
kill events
window hooks IV-303
Kill Symbolic Paths dialog II-23
Kill Waves dialog II-71
KillBackground operation IV-323, V-468
KillControl operation V-468
KillDataFolder operation V-469
example IV-172
KillFIFO operation V-469
KillFreeAxis operation V-470
killing
background task V-468
controls III-435, V-468
data folders II-109, V-469
defined I-22
FIFO buffers V-469
graphs II-350
help windows II-3
notebooks III-4
numeric variables II-117, V-471
page layouts II-477
pictures III-511, V-470
procedure windows III-399
string variables II-117, V-471
symbolic paths II-23, V-470
tables II-241
vs hiding windows II-46
waves II-70–II-71, II-117, V-471
all of them II-71
from tables II-243
windows II-46, V-169
KillPath operation V-470
KillPICTs operation V-470
KillStrings operation II-105, V-471
KillVariables operation II-104, V-471
KillWaves operation II-70, V-471
examples II-71
Igor binary wave files II-71
KillWindow operation V-471
Kirsch edge detector V-370
KMeans operation V-472
Kolmogorov-Smirnov V-948
Korean
text encodings III-462
Kriging V-383
Kronecker product V-560
kurtosis of wave V-415, V-1084
2D V-415
kwBackgroundColor V-93
kwCmdHist
GetWindow operation V-318
kwControlBar V-93
kwControlBarBottom V-93
kwControlBarLeft V-93
kwControlBarRight V-93
kwControlBarTop V-93
kwFrame
DoWindow operation V-170
kwFrameInner
GetWindow operation V-318
kwFrameOuter
GetWindow operation V-318
kwSelectedControl V-93
kwTopWin
DoWindow operation V-170
GetWindow operation V-318
Notebook operation V-695
NotebookAction operation V-712
PrintNotebook operation V-775
SaveNotebook operation V-823
SetWindow operation V-865
SpecialCharacterInfo operation V-893
SpecialCharacterList operation V-895

L

l
suffix in tables II-241
Label operation V-474
Label Options tab II-311–II-312
labels
in delimited text files II-131
in general text files II-138, II-140
in graphs II-293–II-294
laguerre function V-474–V-475
LaguerreA function V-475
Lambert equal area projection V-781
LambertW function V-475
LAPACK matrix routines
errors III-138

MatrixEigenV operation V-534
MatrixLinearSolve operation V-542
MatrixLLS operation V-544
MatrixMultiplyAdd operation V-549
MatrixSchur operation V-579
MatrixSVD operation V-584
numeric support III-138
references III-138
V_flag III-138
LaTeX III-58
in annotations III-54
layers
drawing layers III-68–III-69
in control panels III-68
in graphs III-68
in multidimensional waves II-92, II-95
in page layouts II-476, II-479–II-480
in tables II-262
indexing V-1119
z function V-1119
maximum location V-1085
minimum location V-1085
names III-68
numbers V-786
r function V-786
Overlay drawing layer III-69
ProgAxes drawing layer III-68
ProgAxes layer III-68
ProgBack drawing layer III-68
ProgBack layer III-68
in page layouts II-479–II-480
ProgFront drawing layer III-69
ProgFront layer III-68
in page layouts II-479–II-480
UserAxes drawing layer III-69
UserAxes layer III-68
UserBack drawing layer III-68
UserBack layer III-68
in page layouts II-479–II-480
UserFront drawing layer III-69
UserFront layer III-68
in page layouts II-479–II-480
layout II-214
Layout Macros submenu II-48
Layout operation V-476
Layout subtype IV-205
keyword V-478
LayoutInfo function V-478
LayoutMarquee subtype IV-205, V-310
keyword V-480
LayoutPageAction operation V-480
LayoutSlideShow operation V-481
LayoutStyle subtype IV-205
keyword V-483
leaks
free waves IV-94
least common multiple V-294
left-shift operator IV-6, IV-43
leftx function V-483
use DimOffset instead V-160
Legend operation V-483–V-484
legendreA function V-484
legends III-42–III-43
automatically updated III-42–III-43
color as f(z) II-301
color scale bars III-34, III-47–III-51
customize at point III-43
default III-42
definition III-34
exterior III-40
affects graph plot area III-41
font size III-43
in page layouts II-483, II-495
interior III-40
making a legend (example) I-16
markers III-43
modifying III-35
offset III-41
position III-40–III-42
standard legend III-42
symbols in annotations III-57
wave symbols III-37, III-42–III-43
length limit II-8, IV-2
less than IV-6
level detection III-287–III-290
EdgeStats operation III-289, V-190–V-192
FindLevel operation III-288, V-242–V-244
FindLevels operation III-288, V-244–V-245
FindPeak operation V-247–V-248
PulseStats operation III-290, V-783–V-785
risetime measurements III-289–III-290
XY data III-289
liberal names
accessing waves and variables IV-68
data folders II-110
dimension labels II-94
in user functions IV-68
parsing by Igor IV-168
programming with IV-168–IV-169
quoting with data folders IV-169
rules III-501, IV-2
CleanupName function V-70
CreateDataObjectName function V-113
Light Properties dialog II-433
lights
Gizmo II-433
troubleshooting II-469
limit command history II-10
limit function V-485
line
curve fitting function III-210
Line Colors dialog II-369, II-371–II-372

line continuation character IV-2
line continuation characters IV-2
line fit III-182, V-127
  r-squared III-221
  through origin III-184
line objects
  Gizmo II-436
line size in graphs II-295
line styles II-294, II-348, III-496
  in pop-up menu controls V-766
  SetDashPattern operation V-837
line terminator II-128
linear colors
  in image plots II-400
linear convolution III-285
linear equations III-161
linear interpolation III-115
LinearFeedbackShiftRegister operation V-485
linefeed
  in data files II-129
  in delimited text files II-126, II-135
  in Igor Text files II-154
  in tables II-258
linefeed characters
  cross-platform issues III-453
  escape code for IV-14
  in strings IV-14
  saving text waves II-180
lines
  bevels III-496
  end caps III-496
  joins III-496
  miters III-496
LINESTYLEPOP V-766
linkage
  hierarchical clustering III-166, V-339
linking
  graphs in notebooks III-18
links
  checking in help files IV-258
  creating in help files IV-257
  finding in help files IV-257
ListBox controls III-416, III-426
ListBox operation V-488
  formatted text III-53
  help V-493
  tooltips V-493
ListBoxControl subtype IV-205
  keyword V-498
ListMatch function V-498
lists
  adding items V-20
  AddListItem function V-20
  AnnotationList function V-26
  appending items V-799–V-800
  AxisList function V-44
  ContourNameList function V-86
  counting items V-463
  CountObjects operation V-110
  CountObjectsDFR operation V-111
  CTabList function V-118
  DataFolderDir function V-141
  deleting items V-790, V-793, V-795
  FindListItem function V-245
  GetIndexedObjName function V-303
  GetIndexedObjNameDFR function V-304
  GetRTStackInfo function V-314
  ImageNameList function V-397
  IndexedFile function V-440
  ItemsInList function V-463
  ListBox controls III-416, III-426
  ListMatch function V-498
  ListToTextWave function V-499
  ListToWaveRefWave function V-499
  NumberByKey function V-714
  of controls V-94, V-608
  of windows V-1100
  PathList function V-738
  PICTList function V-742
  processing lists IV-172
  processing lists of waves IV-198–IV-201
  related functions V-11
  RemoveByKey function V-790
  RemoveFromList function V-793
  RemoveListItem function V-795
  ReplaceNumberByKey function V-799
  ReplaceString function V-799
  ReplaceStringByKey function V-800
  replacing items V-799–V-800
  selected column names V-316
  selected layout objects V-316
  sscanf operation V-902
  StringByKey function V-997
  StringFromList function V-998
  StringList function V-999
  stringmatch function V-1000
  TraceFromPixel function V-1042
  TraceNameList function V-1044
  VariableList function V-1066
  WaveList function V-1075
  WaveRefIndexed function V-1080
  WaveRefIndexedDFR function V-1080
  WaveRefWaveToList function V-1081
  waves in graph V-320
  WhichListItem function V-1095
  WinList function V-1098
ListToTextWave function V-499
ListToWaveRefWave function IV-198, V-499
literal numbers
  hexadecimal literals IV-12
literal strings
  text encodings III-480

live display of waves II-347–II-348
ln function V-500
Load Data Tweaks dialog
  auto II-141
  delimited text files II-136
  general text files II-141
Load Delimited Text dialog II-128, II-133
  loading 2D waves II-133
Load General Text dialog II-128, II-140
Load Igor Binary dialog II-128, II-156
Load Igor Text dialog II-128, II-153
Load Image Dialog II-157
Load Row Data procedure II-159
Load Wave operation
  wave names II-142
Load Waves dialog
  copy to home II-155
  creating tables II-235
  delimited text files II-132
  fixed field text files II-138
  general text files II-140
  Igor Binary II-155
  Igor Text II-152
Load Waves submenu II-128
LoadData operation II-155–II-156, V-500–V-505
  regular expressions V-503
LOADFILE IV-279
LOADFILEINNEWINSTANCE IV-279
LOADHELPEXAMPLE IV-279
loading experiments II-26
  error handling II-27
loading waves II-126–II-177
  automation II-170
  BMPs II-158
  choosing names II-133, II-140
  creating tables II-235
  delimited text files II-129–II-137
  drag and drop II-35, IV-280
  example II-171, II-173, II-175
  FBinRead operation V-219
  fixed field text files II-137
  FORTRAN files II-137
  from row-oriented files II-159
  general text files II-138–II-142
  Igor binary wave files II-154–II-157
  Igor Text files II-150–II-154
  images II-157
  JCAMP-DX files II-168
  JPEG files V-390
  JPEGs II-158
  number of loaded waves V-189, V-294, V-464, V-515, V-592, V-658, V-1117
  PNG files V-390
  PNGs II-157
  programming II-170
  S_fileName string II-170
  S_path II-170
  S_waveNames string II-170
  setting wave names II-176
  spectroscopic data II-168
  Sun Raster files II-159, V-390
  TIFF files V-390
  TIFF stacks II-158
  TIFFs II-158
  V_flag variable II-170
  variables II-170
LoadPackagePreferences operation V-505
LoadPICT operation V-506–V-508
LoadWave operation II-154, V-508–V-515
  backslashes II-150
  column formats II-146
  column-specifc settings II-145
  copy to home II-24
  dates II-144
  escape codes II-150
  example II-171
  field widths II-146
  loading very large files II-150
  text encodings II-129, II-149, III-475, V-509
  using file name in wave names II-142
  wave names II-146
  wave number type II-146
local functions
  Static keyword V-906
local variables II-105, IV-61
  automatic creation IV-61
  complex keyword V-82
  double keyword V-168
  in macros IV-119
  in user functions IV-34
  initialization
    in macros IV-119
    in user functions IV-34
  int keyword V-446
  int64 keyword V-446
  keywords V-13
  name conflicts
    in macros IV-119
    in user functions IV-34
  uint64 keyword V-1048
local waves IV-91, IV-96, IV-207
  tracking IV-207, V-1087
  type of V-1092
  WaveType function V-1092
lock icon
  in procedure files III-397
locking
  notebooks III-5
  procedure files III-397, V-642
Loess operation V-515
loess smoothing III-297
log

curve fitting function III-211
log axes II-307, II-315
loglin checkbox II-315
LogLin mode II-315
minor tick labels II-315
tick mark labels II-319
unexpected behavior II-315
log colors
for traces II-298
for traces in graphs V-616
in contour plots II-373, V-604
in image plots II-387, II-400, V-637
log function V-520
log normal curve fit V-128
$\log_{10}$ V-520
$\log_e$ V-500
logical operators IV-6, IV-41
AND IV-7, IV-41
NOT IV-41
OR IV-7, IV-41
side effects IV-7
LogLin axes II-315
loglin checkbox II-315
lognormal
curve fitting function III-212
logNormalNoise function V-521, V-523
LogRatio function III-172
LombPeriodogram operation V-521
long date
in notebooks III-13
long dimension labels II-94, V-209
long object names III-502–III-504, V-208
compatibility III-503
external operations III-504
Look for Folder button II-28
LookAt projection
Gizmo II-424
loops IV-45
break statements IV-48
continue statements IV-48
for loops IV-46
in macros IV-120
in user functions IV-45
nested IV-45
range-based for loops IV-47
lor
curve fitting function III-206
lor (Lorentzian) curve fit V-127
Lorentzian
curve fitting function III-206
low pass band
filtering III-303
low pass filters
example III-304
lower-case string conversion V-523
LowerStr function V-523
LOWESS smoothing III-297, V-515
LU decomposition V-546–V-547
Lucy-Richardson deconvolution V-404

M

M_3DParticleInfo V-366
format V-366
M_3DVertexList V-1046
M_4DTranspose V-427
M_A V-543–V-544, V-579
M_Abs V-1090
M_Acos V-1090
M_Affine V-383
M_alphaBlend V-366
M_ANOVA1 V-907
M_ANOVA2NRResults V-908–V-909
M_ANOVA2Results V-910
M_ANOVA2RMResults V-909
M_Asin V-1090
M_Atan V-1090
M_AveImage V-417
M_AverageRGBImage V-417
M_B V-543–V-544
M_BackProjection V-418
M_Balanced V-529
M_Boxy V-428
M_C V-740–V-741
M_CCBScalar V-418
M_CCBSplines V-418
M_CConjugate V-1090
M_Chunk V-420
M_Circles V-428
M_clustersCM V-259
M_CMYK2RGB V-418
M_CochranTestResults V-923
M_ColorIndex V-424
M_colors V-82
M_Combinations V-974
M_ControlCorrTestResults V-958
M_Convolution V-532
M_Corr V-533
M_Cos V-1090
M_Covar III-213, III-226, V-125, V-131, V-533
M_CrystalToRect V-1090
M_CWT V-137
M_D V-740–V-741
M_DelaunayTriangles V-428
M_Detrend V-182
M_DistanceMap V-259, V-419
M_DPSS V-171
M_DunnettMCElevations V-952
M_DunnettTestResults V-927
M_DWT V-190
M_eigenVectors V-535
M_ExtractedSurface V-419, V-432

M_FFT V-222
M_FGT V-215
M_FitConstraint III-231, V-124, V-131
M_flipped V-1090
M_FriedmanRanks V-931
M_FriedmanTestResults V-931
M_FuzzySegments V-432
M_GLCM V-375
M_GradImage V-412
M_Hartley V-419
M_Hilbert V-348
M_Hough V-421
M_HSL2RGB V-421
M_Hull V-101
M_I123 V-424
M_ICAComponents V-355
M_ICAUnMix V-355
M_Image2Gray V-418
M_ImageEdges V-370
M_ImageHistEq V-378
M_ImageLineProfile V-389
M_ImageMorph V-395
M_ImagePlane V-420
    multitasking example IV-325, IV-327
M_ImageThresh V-415–V-416
M_IndexImage V-424
M_InsertedWave V-422
M_InstantFrequency V-445
M_InterClusterDistance V-260
M_InterpolatedImage V-382, V-384
M_Inverse V-540–V-541, V-1090
M_Inverted V-422
M_JointHistogram V-466
M_KMClasses V-472–V-473
M_L_eigenVectors V-536
M_LakeFill V-419
M_leftEigenVectors V-537
M_LineProfileStdv V-389
M_Lower V-546–V-547
M_LUFactors V-546
M_Magnitude V-1090
M_MagSqr V-1090
M_matchPlanes V-422
M_matrixW V-355
M_max V-1091
M_min V-1091
M_Moments V-363
M_MovieChunk V-745
M_MovieFrame V-745
M_NNResults V-676
M_NPCCResults V-962
M_NPMCDHWResults V-963
M_NPMConResults V-963
M_NPMCSNKResults V-963
M_NPMCTukeyResults V-964
M_OffsetImage V-423
M_PaddedImage V-423
M_paddedImage V-431
M_Particle V-364, V-366
M_ParticleArea V-364, V-366
M_ParticleMarker V-363, V-365
M_ParticlePerimeter V-364, V-366
M_PCA V-356
M_Phase V-1091
M_PhaseLUT V-434
M_PixelatedImage V-383
M_product V-548
M_ProjectionSlice V-423
M_Pt V-546–V-547
M_R V-740–V-741
M_R_eigenVectors V-536
M_RawCanny V-371
M_rawMoments V-366
M_Reconstructed V-404
M_ReducedWave V-423
M_RegMaskOut V-402
M_RegOut V-402
M_RemovedBackground V-403
M_Resampled V-972
M_RGB2Gray V-424
M_RGB2HSL V-424
M_RGB2XYZ V-424
M_RGBOut V-418
M_rightEigenVectors V-537
M_ROIMask V-367, V-374
M_RotatedImage V-405
M_ScaledPlanes V-425
M_ScatterMesh V-385
M_ScheffeTestResults V-975
M_SeedFill V-409
M_SelectColor V-425
M_ShadedImage V-426
M_Shrunk V-426
M_shrunkBox V-426
M_Sin V-1091
M_Skeleton V-413
M_SphericalTriangulation V-899
M_sqrt V-1091
M_Stack V-427
M_StackDot V-428
M_StatsQuantilesSamples V-973
M_StdvImage V-417
M_STFT V-994
M_SumDimension V-1007
M_SumPlanes V-427
M_SV V-544
M_TDLinearSolution V-544
M_TetraPath V-1046
M_TukeyCorrTestResults V-958
M_TukeyMCElevations V-952
M_TukeyMCSlopes V-952
M_TukeyTestResults V-983

M_U V-583–V-584
M_UnwrappedPhase V-433
M_Upper V-546–V-547
M_V V-579, V-583
M_VolumeTranspose V-427
M_VoronoiEdges V-387, V-428
M_VT V-584
M_WaveStats V-973, V-1082–V-1083
  format V-1085
M_WaveStatsSamples V-973
M_Weights1 V-677
M_Weights2 V-677
M_Wigner V-1095
M_WindowedImage V-435–V-436
M_x V-540, V-546, V-580, V-583
M_xProjection V-428
M_XYZ2RGB V-428
M_yProjection V-428
M_zProjection V-428, V-430
Macintosh
  creator code for Igor III-449
  date system II-245, II-256
  dot-underscore files V-441
  file permissions II-30
  memory management III-512
  system requirements III-513
  system version info V-360
  XOP compatibility III-512
Macintosh System Requirements III-513
MacLab data acquisition package IV-312
Macro Execute Error dialog
  during experiment load II-27
Macro keyword IV-118, V-523
macro submenus in Windows menu II-48
MacroInfo function V-523
MacroList function V-524
MacRoman text encoding III-460–III-461, III-481
MacroPath function V-525
macros IV-116–IV-124
  $ operator IV-121
  aborting IV-124
  based on history (example) I-59
  body code IV-120
  calling from a user function IV-202
  compared to functions IV-116
  conditional statements IV-120
  default values V-715, V-1005
  DelayUpdate operation V-153
  deleting II-48
  DoUpdate operation IV-123, V-168
  DoXOPIdle operation V-171
  errors IV-122
  exists function V-207
  flow control IV-120
  initialization of local variables IV-119
  local variable declaration IV-119
  locating a window macro II-48
  loops IV-120
  MacroInfo function V-523
  MacroList function V-524
  menus showing IV-121
  missing parameter dialog IV-121
  names IV-118
  parameter lists IV-119
  parameters III-170, IV-10
  path to V-525
  PauseUpdate operation IV-123, V-739
  procedure type IV-118
  ProcedureText function V-779
  Prompt statements IV-121, V-782–V-783
  recreation macros V-168
  ResumeUpdate operation IV-123, V-809
  retrieve code
    ProcedureText function V-779
  return statements IV-120
  runtime stack information V-314
  S_ variables IV-123
  scope of variables II-103–II-105, IV-119, IV-123
  Silent operation IV-122, V-871
  Sleep operation V-871
  Slow operation IV-122, V-877
  style macros II-48, II-350–II-352, V-171
  subtype II-47–II-48, IV-118, V-66, V-763, V-858
  syntax IV-118–IV-120
  updates IV-123
  V_ variables IV-123
  variables
    scope II-103–II-105, IV-119, IV-123
  wave parameters IV-121
  window recreation II-47–II-48, II-350
magnification
  command window II-12
  history area II-12
  in command line II-53
  in dialogs
    text areas II-53
  in help II-53
  in help files II-53
  in history area II-53
  in notebooks II-53
  in page layouts II-477
  in procedure windows II-53
  setting default II-53
magnitude III-274–III-275, V-1090
magnitude squared V-1090
  of complex number V-525
magsqr function V-525
  example II-81
mailing list II-4
main transformation
  Gizmo II-436
major ticks

in user tick waves II-314
Make operation V-526
automatic creation of Wave IV-72
examples II-68
in user functions IV-72
overwriting caveats II-68
Make Waves dialog II-67
MakeIndex
example III-135
MakeIndex operation III-134–III-135, V-527
applications III-134
making waves II-65, II-67, II-92, V-526
MandelbrotPoint function V-528
map projections
Project operation V-781
MarcumQ function V-529
margins
in graphs II-288
in notebooks III-6
sizes in page layouts II-478
marker size as f(z) II-299
MARKERPOP V-766
markers II-291
as f(z) II-298
box plots V-596
centering in annotations III-43
colors II-292
fill colors II-292
in annotations III-55
in graphs II-288, II-291, II-299
in pop-up menu controls V-766
numeric codes II-300
size in annotations III-43
stroke color II-292
table of II-300
violin plots V-651–V-652
width in annotations III-43
MarkPerfTestTime operation V-529
marquee
in page layouts II-489
marquee menus IV-137
as input for procedures IV-163
marquee2mask procedures III-379
marquees
in graphs II-285, V-309–V-311, V-852
in page layouts I-41, II-479, II-482, V-309–V-311, V-852
Marr–Hildreth edge detector V-370
mask wave in curve fitting III-198, III-201
matching characters III-408
Matlab dynamic libraries II-163
Matlab files
loading II-163, V-589
loading as HDF5 II-165
MLLoadWave64Support II-164
version 7.3 II-165
matrices
addition III-138
analysis
condition V-531
continuous wavelet transform V-137
convolution V-532
CWT operation V-137
determinant V-533
discrete wavelet transform V-189
DWT operation V-189
eigenvalues V-534
eigenvectors V-534
FFT V-222–V-228
Gauss-Jordan V-540
IFFT V-357–V-359
least squares V-542–V-544
linear equations V-542–V-544
LU decomposition V-546–V-547
back substitution V-546
multiply V-548–V-549
rank V-578
related operations V-3–V-4, V-7
Schur factorization V-579
solving V-579
SV decomposition V-584
back substitution V-583
trace V-585
transpose V-585
balancing V-529, V-578
Bartlet window V-436
Bartlett window V-436
Blackman window V-436
built from waveforms II-97
columns
extracting V-420
flipping V-419
summing V-427
summing all V-427
combining V-82
Concatenate operation V-82
condition V-531
converting to waveform II-98
correlation V-532
covariance V-532
dot product V-534
efficient execution III-140, V-550
exporting
BMP files V-405
JPEG files V-405
PhotoShop files V-405
PICT files V-405
PNG files V-405
QuickTime files V-405
raw PNG files V-408
SGI files V-405
Targa files V-405

TIFF files V-405, V-408
Extract operation V-213
extracting a column II-97
extracting a row II-97
FFT of II-98
filling from 1D wave V-419
from XYZ data V-373
Hamming window V-436
Hanning window V-436
Hartley transform V-419
Hough transform V-421
image filtering V-372, V-539
in Igor Text files II-151–II-153
indexing V-421
interpolation V-87, V-458
inverse of V-540–V-541
Kaiser window V-437
LAPACK routines III-138
loading
from delimited text files II-133
from general text files II-140
loading from Excel file II-162
making from 1D waves in tables II-267
MatrixGLM operation V-541
MatrixOp operation III-140, V-550
maximum
chunk location V-1085
column location V-1084
layer location V-1085
row location V-1084
minimum
chunk location V-1085
column location V-1084
layer location V-1085
row location V-1084
multiplication III-138
by scalar III-138
MatrixMultiply operation V-548
MatrixMultiplyAdd operation V-549
offsetting V-423
padding V-423
example V-433
plane extraction V-426
planes
summing all V-427
printing V-769
redimensioning II-98
reshaping 1D to 2D V-789
resizing V-423
Reverse operation V-809
rotating
columns V-425
rows V-425
rows
extracting V-420
flipping V-420
summing V-427
summing all V-427
saving
as delimited text II-178
shifting V-423
smoothing V-515
speeding up III-140, V-550
splitting V-901
SplitWave operation V-901
summing
sum V-427
swap diagonal quadrants V-427
terminology II-92
windowing V-435
matrix data
converting from XYZ data II-367
matrix multiplication V-548–V-549
MatrixOp operation III-144
matrix of Z values
Gizmo II-406
matrix4x4 objects II-469
MatrixBalance operation V-529
MatrixCondition function V-531
MatrixConvolve operation V-532
matrixCorr function V-532
matrixDet function V-533
matrixDot function V-534
MatrixEigenV operation V-534
MatrixFilter operation V-539
MatrixGaussJ operation V-540
MatrixGLM operation V-541
MatrixInverse operation V-541
MatrixLinearSolve operation V-542
MatrixLinearSolveTD operation V-543
MatrixLLS operation V-544
MatrixLUBkSub operation V-546
MatrixLUD operation V-546
MatrixLUDTD operation V-547
MatrixMultiply operation V-548
MatrixMultiplyAdd operation V-549
MatrixOp
Cholesky decomposition V-555
cross-covariance V-556
decimation V-557
fast cosine transform V-571
fast sine transform V-571
fast staggered cosine transform V-572
fast staggered sine transform V-571–V-572
Kronecker product V-560
MatrixOp operation V-550
data types III-145
dot product III-144
examples III-148
functions by category III-149
Hermitian transposition III-144
matrix multiplication III-144

matrix transposition III-144
memory allocation V-574
multithreading III-148
operators III-143
optimization III-148
quaternions III-146, V-573
using III-140–III-151
wave dimensions III-143
MatrixReverseBalance operation V-578
MatrixSchur operation V-579
MatrixSolve operation V-579
MatrixSparse operation V-580
ADD operation III-157
index data type III-156
inputs III-155
list of operations III-154
MM operation III-157
MV operation III-158
operation data type III-155
operations III-156
SMSM operation III-158
TOCOO operation III-159
TOCSC operation III-159
TOCSR operation III-160
TODENSE operation III-160
transformations III-156
TRSV operation III-161
MatrixSparse operations
example III-153
MatrixSVBkSub operation V-583
MatrixSVD operation V-584
matrixTrace function V-578, V-585
MatrixTranspose operation V-585
max function V-586
MAX_OBJ_NAME V-14
MAX_UNITS V-14
MAX_WIN_PATH V-14
max3D filter V-372
MaxBytesReadingWave II-35
MAXCMDLEN V-14
maximized windows V-320
MDI frame IV-286
maximum among several waves
WavesMax user-defined function III-173
maximum location in wave V-415, V-1084
maximum of two numbers V-586
maximum of wave V-1091
MBCS III-462
formatted text notebooks III-472
MD4 hash algorithm V-338, V-1073
MD5 hash algorithm V-338, V-1073
MDI frame window IV-285
mean function III-121, V-586
subranges III-122
mean of wave segments
FindSegmentMeans user-defined function III-174
mean of waveform data III-120
measurements
EdgeStats operation V-190–V-192
PulseStats operation V-783–V-785
median
box plots II-331
Median function III-133
median function V-586
median smoothing III-296
median3D filter V-372
memory
deallocating IV-205, V-1070
fragmentation III-512
free V-359
management III-512
virtual III-512
menu bar names II-44
menu definitions IV-126–IV-141
(character IV-134
! character IV-134
& character IV-135
- character IV-134
/ character IV-134
; character IV-134
accelerators IV-135
BuildMenu operation IV-130
contextualmenu IV-127
creating a new menu IV-129
creating submenus IV-127
DataBrowserObjectsPopup menu IV-138
disabling items IV-134, V-851
DoIgorMenu operation V-166
dynamic IV-127
dynamic menu items IV-129
extending built-in menus IV-128
function keys IV-136
GetKeyState function V-304
GraphPopup menu IV-138
hideable IV-127
independent modules IV-241
keyboard shortcuts IV-134, IV-136, V-304
marking items IV-134
marquee menus IV-137
menus and multiple procedure files IV-131
multiple menu items IV-131
operation queue IV-278
optional menu items IV-130
rebuilding from procedure V-54
regular modules IV-238
separating items IV-134
SetIgorMenuMode operation V-851
special characters IV-133–IV-136, IV-145
enabling and disabling IV-135
Windows IV-134–IV-135

  - specialized menu items IV-132
    - examples IV-132
  - syntax IV-126
  - TablePopup menu IV-138
  - trace menus IV-137
  - WindowBrowserWindowsPopup menu IV-140
  - Windows IV-134–IV-135
- Menu keyword IV-126, V-589
- menus
  - activating V-166
  - consolidating items into a submenu IV-131
  - disabling items V-851
  - GetKeyState function V-304
  - hiding V-346
  - include statements IV-167
  - information about V-306
  - invoking V-166
  - packages III-401
  - pop-up menu controls III-416, III-426–III-428
  - programming V-166, V-306, V-346, V-869
  - relation to commands I-7
  - relation to dialogs I-7
  - showing V-869
  - showing macros IV-121
  - why the menu bar changes II-44
- Mercator projection V-781
  - Transverse V-781
- MERGEEXPERIMENT IV-279
- merging experiments II-19
- Mersenne Twister V-196, V-323–V-324, V-853
- metafiles III-102
- meteorological wind barbs V-614
- Metropolis algorithm V-729
- MexHat wavelet transform V-140
- microphone
  - recording from IV-312
- Microsoft Visual C++ IV-208
- MIME-TSV files IV-281–IV-283, IV-287, IV-289
- min function V-589
- min3D filter V-372
- minidump files IV-341
- minimization III-343–III-348
  - 1D functions III-343
  - 1D nonlinear functions III-344, V-728
  - bracketing extrema III-345
  - Brent’s method V-728
  - caveats III-347
  - dogleg method V-726, V-728
  - finite differences V-728
  - finite differences method V-726
  - function format V-727
  - line search method V-726, V-728
  - More-Hebdon method V-726, V-728
  - multidimensional functions III-343
  - multidimensional nonlinear functions III-346, V-728
  - multivariate functions III-346, V-728
  - Optimize operation V-725
  - output variables V-730
  - output waves III-347, V-730
  - quasi-Newton method V-728
  - references III-349, V-731
  - secant (BFGS) method V-726, V-728
  - stopping tolerances III-347
  - tolerances III-347, V-729
    - setting V-727
  - univariate
    - stopping criterion V-728
  - univariate functions III-344, V-728
  - user-defined functions
    - 1D III-344
    - multidimensional III-346
- minimized windows
  - MDI frame IV-286
- minimum location in wave V-415, V-1084
- minimum of two numbers V-589
- minimum of wave V-1091
- Minkowski difference V-754
- Minkowski sum V-754
- Minkowski vector dissimilarity V-339
- minor ticks
  - in user tick waves II-314
- mirror axis in graphs II-307–II-308
- Misc pop-up menu in page layouts II-483
- miscellaneous file
  - in experiments II-19
- Miscellaneous Settings dialog III-500
  - autosave II-36
  - Default Data Precision II-239
  - HDF5 default compression II-17
  - Loaded Igor Binary Data II-24
  - Repeat Wave Style Prefs in Graphs II-348
  - Save and Restore Recent Colors III-498
  - waves II-68
- Missing Folder dialog II-28
- missing folders II-28
- missing parameter dialog IV-121
  - equivalent for user functions IV-144
- missing parameters
  - in functions V-167
  - in macros V-782–V-783
- missing values II-83
  - in analysis III-112
  - in delimited text files II-129, II-132, II-136
  - in general text files II-139, II-141
  - in graphs II-284, II-303
  - in Igor Text files II-151
  - NaNs II-247
  - removing by interpolation III-113
  - working around III-113
- Missing Wave File dialog II-29
- miters

for lines and traces III-496
mitres
for drawing tools V-840
for traces V-616
MLLoadWave
Matlab dynamic libraries II-163
MLLoadWave64Support II-164
MLLoadWave operation II-163, V-589
MLLoadWave64Support II-164
mod function V-592
modal and modeless user interface IV-144
modal Data Browser II-119
modDate function V-592
modeless dialogs IV-158
modified
events in hook functions IV-299
Modify Annotation dialog III-35
Dynamic pop-up menu III-38
for page layouts II-494
Insert: group III-35
Special pop-up menu III-36
Modify Axis dialog II-306–II-312
Auto/Man Ticks tab II-308
Axis Label tab II-318
Axis Range tab II-286–II-288
autoscaling modes II-287
Quick Set buttons II-287
Axis tab II-307–II-308
date/time items II-316
draw between II-307
global controls II-306
Label Options tab II-311–II-312
live update II-318
log axes II-315
tick marks II-315
manual ticks II-312
computed manual ticks II-312
date/time axes II-317
user ticks from waves II-314
mirror axes II-307
multiple axis selection II-306
Tick Options tab II-310
Ticks and Grids tab II-308–II-310
Modify Columns dialog II-253
Modify Contour Appearance dialog II-368–II-370
Label Tweaks II-378
Modify Draw Environment dialog III-65
Modify Graph dialog II-288–II-290, III-36
Modify Image Appearance dialog II-387–II-388
Modify Objects dialog II-487
Modify operation V-593
Modify Polygon dialog III-71
Modify Trace Appearance dialog II-112, II-290–II-306
customize at point II-306
Set as f(z) II-298
ModifyBoxPlot operation V-593
ModifyBrowser operation V-600
modal Data Browser II-120
ModifyCamera operation V-600
ModifyContour operation V-600–V-606
ModifyControl operation V-606
ModifyControlList operation V-608
ModifyFreeAxis operation V-609
ModifyGizmo operation V-611
ModifyGraph operation V-611–V-635
axes appearance V-626
colors V-634
general graph window settings V-611
live keyword II-348
traces
customize at point V-625
traces appearance V-613
ModifyImage operation V-635–V-639
ModifyLayout operation V-639–V-640
ModifyPanel operation V-640
ModifyProcedure operation V-642
ModifyTable operation V-645–V-649
multidimensional waves II-263
ModifyViolinPlot operation V-649
ModifyWaterfall operation II-326, V-657
ModuleName keyword IV-54, V-658
ModuleName pragma IV-54, IV-236
modules IV-236
modulus functions
mod V-592
undone III-316, V-1050
monitors
dimensions, platform-related III-455
number, depth, and size V-359
resolution V-832
More Extensions folder I-6
More Help Files folder II-2
More Levels dialog II-369
Morlet wavelet transform V-140
complex V-140
morphological operations
3D V-395
multidimensional V-395
mouse
cross-platform issues III-452
GetMouse operation V-312
mouse wheel
panning in graphs axes
panning using mouse wheel II-286
rotation in Gizmo II-424
scaling graph axes II-286
move
data folders V-658
numeric variables V-665
string variables V-663
waves V-665

windows V-666
with Data Browser II-115
Move Backward II-481
Move Forward II-481
MoveDataFolder operation V-658
converting free data folder IV-99
MoveFile operation V-659
MoveFolder operation V-661
warning V-661
MoveString operation V-663
MoveSubwindow operation V-663
MoveVariable operation V-665
MoveWave
free waves IV-96
MoveWave operation V-665
MoveWindow operation V-666
movies IV-245
AddMovieAudio operation V-21
AddMovieFrame operation V-21
CloseMovie V-71
codecs V-683
compression factor V-683
creatiing IV-245
extracting frames IV-246
from pictures IV-246
MP4 V-683
NewMovie operation V-683
of graphs IV-245
playing IV-245
PlayMovie operation V-744
PlayMovieAction operation V-745
related operations V-6
moving
a window on-screen II-50
all windows on-screen II-50
moving average III-295, V-878
moving cursor calls function IV-339–IV-340
MP4 movies V-683
MPFXEMGPeak function V-667
MPFXExpConvExpPeak function V-668
MPFXGaussPeak function V-669
MPFXLorentzianPeak function V-670
MPFXVoigtPeak function V-671
MPR filter V-789
multidimensional data
copy/paste in tables II-265–II-270
multidimensional waves II-92–II-99
3D copy and paste II-269
analysis II-95
average 3D filter V-372
averaging V-417
beams
extracting V-420
built from matrices II-97
chunk indexing in assignments V-1011
chunk numbers in assignments V-811
chunks II-92, II-95
maximum location V-1085
minimum location V-1085
columns
extracting V-420
flipping V-419
indexing in assignments V-1119
maximum location V-1084
minimum location V-1084
names II-261
numbers in assignments V-785
setting values from a wave V-423
styles in tables II-237
summing V-427
summing all V-427
concatenating V-82
create-paste in tables II-269
creating II-92–II-93, V-526
in tables II-234
cutting columns in tables II-268
decimation II-98
deleting points II-92
DimDelta function V-160
dimensions
how many? V-1072
labels II-93–II-94, V-240, V-298, V-838
names II-92
numbers II-92
DimOffset function V-160
DimSize function V-160
duplicating II-93
editing in tables II-261–II-270
extracting
matrix II-97
matrix from a 3D wave II-269
row slice II-97
FFT of II-98
filling from 1D wave V-419
format in text file V-814
FuncFitMD operation V-276
gaussian 3D filter V-372
GetDimLabel function V-298
Hartley transform V-419
Hough transform V-421
hybridmedian filter V-372
in Igor Text files II-151–II-153
index columns in tables II-242
index values V-160
indexing II-95–II-97, V-421
initial values II-92
insert-paste in tables II-266–II-267
inserting images V-422
inserting points II-92
interpolation V-87, V-458–V-459, V-461
interpolation, not II-96
labels II-93–II-94

    - example II-82
    - FindDimLabel function V-240
    - GetDimLabel function V-298
    - in delimited text files II-131
    - in tables II-262
    - length limit II-94, II-244
    - naming conventions II-94
    - SetDimLabel function V-838
    - viewing in tables II-235
    - wave indexing II-82
- layers II-92, II-95
    - indexing in assignments V-1119
    - maximum location V-1085
    - minimum location V-1085
    - numbers in assignments V-786
- matrix to matrix conversion II-98
- maximum rank 3D filter V-372
- median 3D filter V-372
- minimum rank 3D filter V-372
- number of points V-160
- padding
    - example V-433
- pixels
    - inverting V-422
- planes
    - extracting V-420, V-426
        - example V-433
    - flipping V-420
    - inserting V-422
    - matching V-422
    - removing V-423
    - scaling V-425
    - summing V-427
    - summing all V-427
- point finding 3D filter V-372
- polynomial function V-749
- projection slice V-423
- redimensioning II-92–II-93, II-98
- replace-paste in tables II-266
- rows
    - extracting V-420
    - flipping V-420
    - maximum location V-1084
    - minimum location V-1084
    - setting values from a wave V-423
    - summing V-427
    - summing all V-427
- saving
    - as delimited text II-179
    - as Igor Text II-179
    - in text files II-181
- shrinkBox V-426
- spherical interpolation V-899
- splitting V-901
- stacking V-427
- storing into a subrange II-97
- storing matrix as 3D layer II-96
- styles in tables II-254
- subset of II-93
- surface extraction V-419
- swap diagonal quadrants V-427
- terminology II-92
- transpose 4D wave V-427
- transpose volume V-427
- treating as 1D II-99
- vol2surf V-428
- voronoi tesselation V-428
- wave assignments II-96–II-97
- WaveDims function V-1072
- waveform to matrix conversion II-98
- X projection V-428
- Y projection V-428
- Z projection V-428

multiple return statements IV-36

multiplication IV-6
- quaternion II-427

multitasking IV-329–IV-339
- aborting threads IV-337
- curve fitting III-249–III-250
- data folders IV-330
- examples IV-331–IV-339
- free data folders example IV-325
- free waves example IV-327
- HDF5 II-206
- MatrixOp operation III-148
- MultiThread keyword V-673
- Multithread keyword IV-323–IV-328
- MultiThreadControl operation V-674
- network operations IV-271
- preemptive IV-106, IV-329–IV-339
- processor count V-1033
- queues IV-330
- return value V-1033
- starting threads V-1033
- structures example IV-328
- TBB IV-323
- thread groups
    - creating V-1031
    - data folders V-1031
    - posting data to V-1031
    - releasing V-1032
    - stopping V-1032
- ThreadGroupCreate function V-1031
- ThreadGroupGetDF function V-1031
- ThreadGroupGetDFR function V-1031
- ThreadGroupPutDF operation V-1031
- ThreadGroupRelease function V-1032
- ThreadGroupWait function V-1032
- ThreadReturnValue function V-1033
- ThreadSafe keyword V-1033
- ThreadStart function V-1033
- ThreadStart operation V-1033

user functions IV-106, IV-329–IV-339
MultiThread
Mandelbrot demo V-528
MultiThread keyword V-673
Multithread keyword IV-323–IV-328
free data folders example IV-325
free waves example IV-327
structures example IV-328
MultiThreadControl operation V-674
multithreading
curve fitting III-249
HDF5 II-206
multivariate functions
complex curve fitting III-248
curve fitting III-200, III-260, V-276
Gauss2D V-291
mushroom cloud icon (it's a tree, really) III-65

N

n V-465
name spaces III-502
NameOfWave function V-675
example IV-200
names III-501–III-504
$ operator IV-18, IV-62
allowed characters III-501, IV-2
annotations III-39, III-53, V-26
case insensitive III-501
CheckName function V-68
CleanupName function V-70
column names II-241
conflicts III-505
HDF5 III-505
ContourNameList function V-86
ControlNameList function V-94
CreateDataObjectName function V-113
creating in tables II-239, II-248
CTabList function V-118
data folders II-109
dimension labels II-94
FontList function V-256
FunctionList function V-284
graphs II-277, II-350
ImageNameList function V-397
in graphs IV-20
ContourNameToWaveRef function V-87
ImageNameToWaveRef function V-398
in Igor binary wave files II-155
in page layouts II-477, IV-20
instance names IV-20, V-613
layers III-68
length III-501
liberal rules III-501, IV-2
loading
delimited text II-133
general text II-140
local variables
in macros IV-119
in user functions IV-34
long object names V-208
macro names IV-118
MacroList function V-524
name spaces III-502
notebooks III-25
objects II-486
OperationList function V-725
parsing by Igor IV-168
paths
IndexedDir function V-438
pictures III-509–III-510
programming with liberal names IV-168–IV-169
quoting III-501
related functions V-11
Rename Objects dialog III-504
Rename operation II-72, V-796
RenameDataFolder operation V-796
RenamePath operation V-797
RenamePICT operation V-797
RenameWindow operation V-797
renaming with Data Browser II-115
rulers III-11
standard rules III-501
suffixes in tables II-241, II-254
table names II-240
trace names V-613
UniqueName function V-1048
uniqueness III-502
user function names IV-31
using $string IV-62
using $stringName IV-18
variables II-102
waves II-63, II-65
in tags III-38
window names II-45
namespaces
independent modules IV-238, IV-242
regular modules IV-236
NaN (Not a Number) function V-676
NaNs II-83
as missing values II-247
comparison with IV-6
detecting V-715
entering in tables II-247
in analysis III-112
in curve fitting III-232–III-233
in delimited text files II-129
in error waves II-304
in general text files II-139
in Gizmo II-453
in graphs II-284, II-303

in Igor Text files II-151
in tables II-247
in WaveStats operation III-124
not an integer II-83
number of V-1084
removing
by interpolation III-114
by median smoothing III-114
from waves V-1092
removing from waves III-113
replacing in waves III-113
working around III-113
zapping V-570
National Instruments data acquisition package IV-312
natural logarithm function V-500
navigation
Go Back and Go Forward II-52
navigation bar
in procedure windows III-397
NBit
HDF5 compression II-213
negation IV-6
negation operator IV-6
nested datatypes
HDF5 II-211
nested loops IV-45
network
UNC paths III-451
Unix paths III-452
networking
aborting operations IV-270
multitasking IV-271
neural networks
NeuralNetworkRun operation V-676
NeuralNetworkTrain operation V-676
NeuralNetworkRun operation V-676
NeuralNetworkTrain operation V-676
neurons
ImageSkeleton3D operation V-413
New Fit Function dialog III-190
New Free Axis dialog II-278
New Graph dialog II-277, II-349
New Graph dialog graphs
New Graph dialog II-278
New Notebook dialog III-3
New Page Layout dialog II-476
New Symbolic Path
example II-22
New Symbolic Path dialog II-22
New Table dialog II-234–II-235
New window II-45
NewDataFolder operation V-678
NEWEXPERIMENT IV-279
NewFIFO operation V-678
NewFIFOChan operation V-678
NewFreeAxis operation V-470, V-679
NewFreeDataFolder function V-680
NewGizmo operation V-681
NewImage operation V-681
NewLayout operation V-682
newline
in regular expressions IV-184
NewMovie operation IV-245, V-683
NewNotebook operation V-684
example III-26
text encodings V-685
NewPanel operation V-687
control panel units III-444
points versus pixels III-444
NewPath
platform-related issues III-449
NewPath operation V-690
example II-22, II-175
in experiment files II-28
NewtCamera operation V-678
NewWaterfall operation II-326, V-691
Nicolet WFT files II-128
NIDAQ Tools IV-312
NIGPIB2 XOP IV-312
noise functions
enoise V-195
gnoise V-323
LinearFeedbackShiftRegister operation V-485
Mersenne Twister V-196, V-324
SetRandomSeed operation V-853
Xoshiro256** V-196, V-324
nonlinear least squares curve fitting III-179
norm function V-692
normal ruler in notebooks III-9, III-11
Normal+ ruler III-12
normalization
of waves II-79
Unicode V-693
normalized Gaussian function V-290
NormalizeUnicode function V-693
normals
Gizmo II-432
NOT
logical operator IV-41
Not a Number function V-676
notarization
Igor extensions III-512
notch filters
example III-310
notched box plots II-335
note function V-694
Note operation V-694
notebook files
autosave II-37
Notebook operation V-695–V-711
accessing contents V-710

byte order marks V-697
document property parameters V-696
example III-26–III-27
miscellaneous parameters V-698
paragraph property parameters V-700
pictures III-26
selection parameters V-702
setting contents V-711
text property parameters V-705
writeProtect V-697
writing
graphics parameters V-705
special character parameters V-709–V-710
text parameters V-711
notebook operation
autoSave V-698
NotebookAction operation V-711
Notebooks
recreating subwindows V-711
notebooks III-2–III-32
accessing contents V-710
actions III-14–III-18, V-711, V-893, V-895
commands III-16–III-17
editing III-16
help text III-15
helper procedures III-17
NotebookAction operation V-711
pictures III-15
SpecialCharacterInfo function V-893
SpecialCharacterList function V-895
styles III-15
adopting II-25, V-696
adopting programmatically V-23
aligning pictures III-9
as subwindows III-91
as worksheets III-4
auto-save V-698
automation III-25–III-30
background color III-6
byte order marks V-697
changeableByCommandOnly III-11
character properties III-8
converting to UTF-8 III-489
creating derived rulers III-12
creating new rulers III-11
dates III-13, III-20
Default button III-9
default tab width III-6
default tabs III-6, V-696
document info icon III-5
document properties III-6, V-696
embedding HTML code III-23
examples III-23
HTMLCode ruler III-23
executing commands from III-4
experiments III-3, III-25
exporting III-20–III-21
external text editor II-56
File Information dialog III-25
file names III-25, V-686
finding
pictures V-702
rulers III-12
text III-24, V-702
fonts III-8–III-9
footers III-6, III-20
programming III-26, V-698
formatted III-2
generating commands III-29
Go Back and Go Forward II-52
graphs III-18
headers III-6, III-20
programming III-26, V-698
hiding III-4, V-699
HTML III-21
cascading style sheets III-22
character formatting III-23
graphics III-23
JPEG III-23
picture frames III-23
PNG III-23
HTML 4.01 specification III-22
paragraph formatting III-22
.ifn file extension III-2
Igor object pictures III-18, III-28
importing III-20
in line III-9
indentation III-7–III-8
Insert Page Break III-20
inserting pictures III-13, V-706
justification of text III-7–III-8
killing III-4
linking graphs III-18
list of V-1098, V-1100
locking III-5
magnification II-53
making a new notebook II-45
margins III-6
names II-45, III-25, V-686
NewNotebook operation V-684
normal ruler III-9, III-11
Notebook operation V-695–V-711
NotebookAction operation V-711
opening III-3
as RTF III-21
OpenNotebook operation V-722
page breaks III-19
page layouts III-18
page numbers III-13, III-20
page setups III-30
paragraph properties III-7, V-700
picture compatibility III-19

pictures III-13, III-18–III-19, V-705
    type of III-458
    updating III-28, V-699, V-709
plain III-2
    styles III-8
    tabs III-8
platform-related issues III-458
preferences III-30
printing III-20, V-775
programming III-25–III-30
    examples III-26
read-only property III-10
recreation macros III-91
redefining rulers III-12
references to II-24, III-3
removing rulers III-12
replacing text III-24
retrieving text III-28
RTF III-20
Ruler pop-up menu III-11
rulers III-5, III-7, III-11–III-12, V-700
    fonts III-7, III-12
    names III-11
    text color III-7, III-12
    text sizes III-7, III-12
    text styles III-7, III-12
SaveNotebook operation V-823
saving
    as HTML III-21
    as RTF III-21
saving pictures III-14, V-708
scaling pictures III-14
selections V-315, V-702
Set Text Format to Ruler Default III-9
shortcuts III-32
showing III-4
spacing III-7–III-8
special character names III-14
special characters III-13, V-699, V-709–V-710
SpecialCharacterInfo function V-893
SpecialCharacterList function V-895
subscripts III-9
superscripts III-9
tab characters III-7–III-8
tables III-18
template files III-30
text
    color III-8
    formats III-9
    inserting V-711
    properties V-705
    retrieving III-28
    sizes III-8–III-9
    styles III-8
text color III-9
TEXT files III-2
text formats III-9
text size III-9
text style III-9
times III-13, III-20
titles II-45, V-686
transferring rulers III-12
.txt file extension III-2
unlocking III-5
updating
    pictures III-18
    special characters III-14, V-699, V-709
vertical offsets III-8–III-9
window names V-1048
window titles III-13, III-20, III-25
wintype function V-1103
WMT0 files III-2
write-protect V-697
write-protect icon III-5
write-protect property III-10
zooming II-53
notes
    data folder notes II-120
    in windows V-320
null
    in strings IV-16
num2char function V-713
num2istr function V-713
num2str function II-73, V-714
number functions V-7
number-to-string conversion IV-260, V-713–V-714, V-902
NumberByKey function IV-173, V-714
numeric expressions
    continuation characters IV-35, IV-113
    hexadecimal literals IV-12
numeric formats
    commas in tables II-255
    dates in tables II-245, II-256
    hexadecimal in tables II-257
    in delimited text files II-133
    in tables II-255
    octal in tables II-257
    pasting in tables II-249
    precision in tables II-245
    times in tables II-256
numeric precision II-104
    APMath operation V-27
    arbitrary V-27
    changing II-72
    copying from table II-252
    default for waves II-67
    expression evaluation IV-38
    in functions IV-31
    in Igor Text files II-151
    in tables II-239, II-245, II-255

integer expressions in user functions IV-38, IV-113
of calculations IV-8
of waves II-66, II-68, V-788, V-1073, V-1092
Redimension operation V-788
saving waves II-179
numeric types IV-8
changing II-72
coercion in user functions IV-111
of functions IV-11
of waves V-1073, V-1092
numeric variables II-104–II-105
complex II-104, V-1065
copying II-115
default values V-715
deleting II-104
dependency assignments IV-231
global II-104, III-417–III-418, III-428–III-429, V-1065
in operation flags V-15
initializing II-104, V-1065
killing II-104, II-117
KillVariables operation V-471
MoveVariable operation V-665
moving to data folder II-115
names
CheckName function V-68
numeric type II-104
precision II-104
renaming II-115, III-504, V-796
Set Variable controls III-417, III-428
Slider controls III-418, III-429
UniqueName function V-1048
ValDisplay operation V-1060–V-1064
Variable declaration II-104, V-1065
VariableList function V-1066
numpnts function V-714
use DimSize for multidimensional waves V-160
numtype function V-715
NumVarOrDefault function V-715
example IV-147
saving parameters IV-147
NVAR keyword IV-65, V-715
/Z flag IV-67
automatic creation IV-70
automatic creation with rtGlobals IV-70
checking in debugger IV-213
example IV-69
explicit creation IV-70
failures IV-67
NVAR_Exists function V-716
use with $ IV-62
use with data folders IV-69, IV-249
NVAR_Exists IV-67, V-716
Nyquist frequency
filtering III-300, III-312
smoothing V-879

O

object list
Gizmo II-416–II-417
object names
long names III-502–III-504
object references
FUNCREF keyword V-278
HDF5 II-208
in tables II-258, II-270
NVAR keyword V-715
STRUCT keyword V-1002
SVAR keyword V-1010
Wave keyword V-1069
Object Status dialog IV-230
objects
broken objects IV-233
exists function V-207
Gizmo II-418
how objects relate I-3
in experiments II-16
in page layouts II-485, V-477
properties II-498
indexing in style macros IV-20
long object names V-208
name spaces III-502
names III-501–III-504, V-477
conflicts III-505
instance names IV-20
overview I-2
pictures in notebooks III-18
updating III-28
Rename Objects dialog III-504
unique names III-502
octal numbers
in tables II-245, II-255, II-257
representation of II-257
ODBC II-181
ODRPACK95 III-237
offscreen drawing objects III-65, III-68
offsetting traces in graphs II-302–II-303
log axes II-302
multipliers II-302
preventing II-302
tags III-38
undoing II-302
only guess in curve fitting III-189
opacity
color table waves II-400
image plots II-400–II-401
in Gizmo II-428
in graph traces II-298
opaque data
HDF5 II-211

- Open Experiment dialog II-19
- Open File dialog IV-149, V-717
  - multi-selection V-718
- Open File submenu II-45
- Open operation IV-194, IV-196, V-716–V-721
  - appending V-717
  - file reference numbers V-716
  - open for appending IV-196
  - open for reading IV-196
  - paths V-717
  - reading V-717
  - symbolic paths V-717
  - writing V-717
- OpenGL II-407
  - changes to II-470
  - GL_ constants II-471
- OpenHelp operation V-721
- opening
  - data files II-126
  - experiments II-19
  - files II-45, IV-196
  - notebooks III-3
  - procedure files III-398
  - RTF as notebooks III-21
- OpenNotebook operation V-722
  - text encodings V-723
- OpenProc operation V-724
  - text encodings V-724
- operands IV-8
- operation queue IV-278
  - COMPILEPROCEDURES IV-279
  - Execute/P operation V-204
  - INSERTINCLUDE IV-279
  - LOADFILE IV-279
  - LOADFILEINNEWINSTANCE IV-279
  - LOADHELPEXAMPLE IV-279
  - MERGEEXPERIMENT IV-279
  - NEWEXPERIMENT IV-279
  - RELOAD CHANGED PROCS IV-280
- OperationList function V-725
- operations IV-10
  - bits
    - setting IV-12
  - by category V-1–V-6
  - definition I-5
  - destination waves IV-85
  - exists function V-207
  - external IV-208
  - flags
    - expressions require parentheses V-15
  - in user functions IV-111
  - listing V-725
  - on functions III-320
    - differential equations III-322
    - minimization III-343
    - root finding III-338
  - OperationList function V-725
  - syntax IV-10
  - syntax guide V-15
  - trace names IV-88
- operators IV-6
  - ! IV-6, IV-41
  - != IV-6
  - $ IV-6
  - %^ IV-6
  - & IV-6, IV-41
  - && IV-6, IV-41
  - * IV-6
  - + IV-6
  - - IV-6
  - / IV-6
  - ^ IV-6
  - < IV-6
  - << IV-41
  - <= IV-6
  - == IV-6
    - and roundoff error IV-8
  - > IV-6
  - >= IV-6
  - >> IV-41
  - ?: IV-6
  - ~ IV-6, IV-41
  - | IV-6, IV-41
  - | | IV-6, IV-41
  - addition IV-6
  - associativity IV-6
  - bitwise
    - << IV-41
    - >> IV-41
    - AND IV-6–IV-7, IV-41
    - complement IV-41
    - OR IV-6–IV-7, IV-41
    - XOR IV-6–IV-7
  - comparison IV-7
  - complement IV-6
  - conditional II-82, IV-6–IV-7
  - decrement IV-6, IV-43
  - division IV-6
  - equality IV-6
  - exponentiation IV-6
  - greater than IV-6
  - greater than or equal IV-6
  - increment IV-6, IV-43
  - inequality IV-6
  - left shift IV-6, IV-43
  - less than IV-6
  - less than or equal IV-6
  - logical
    - AND IV-6–IV-7, IV-41
    - NOT IV-6, IV-41
    - OR IV-6–IV-7
  - MatrixOp operation III-143

multiplication IV-6
negation IV-6
obsolete IV-8
post-decrement IV-6, IV-43
post-increment IV-6, IV-43
pre-decrement IV-6, IV-43
pre-increment IV-6, IV-43
precedence IV-6, IV-8, IV-41
right shift IV-6, IV-43
strings
substitution IV-6
subtraction IV-6
Optimize operation V-725–V-731
optional include statements IV-167
optional parameters IV-33
determining usage V-733
example IV-60
options
in operations IV-10, IV-12
OR
bitwise operator IV-6–IV-7, IV-41
logical operator IV-6–IV-7, IV-41
orthographic projection V-781
Gizmo II-422
Otsu's method III-355
outliers
box plots II-331, II-334, V-594
RemoveOutliers user-defined function III-171
Overlay drawing layer III-69
Override Experiment III-466
plain text files III-467–III-468
wave text encodings III-473
Override keyword V-731
user functions IV-106
overriding functions
Override keyword V-731

P

p function V-731
in multidimensional waves II-92–II-93, II-96
in wave assignments II-74
p2rect function V-732
packages III-401, IV-246–IV-249
lightweight IV-249
LoadPackagePreferences operation V-505
managing globals IV-249
naming IV-251
opening V-691
path to folder V-895
preferences IV-251–IV-255
loading IV-251, V-505
saving IV-251, V-825
in binary files IV-252
in experiments IV-254
private data IV-249
procedure modules IV-236
SavePackagePreferences operation V-825
WaveMetrics IV-247
Packages data folder IV-170, IV-249
packed experiments II-16
adopting files II-25
adopting files programmatically V-23
autosave II-37
HDF5 II-223
LoadData operation II-156
packed strings
adding items V-20
AddListItem function V-20
appending items V-799–V-800
counting items V-463
deleting items V-790, V-793, V-795
FindListItem function V-245
ItemsInList function V-463
NumberByKey function V-714
RemoveByKey function V-790
RemoveFromList function V-793
RemoveListItem function V-795
ReplaceNumberByKey function V-799
ReplaceStringByKey function V-800
replacing items V-799–V-800
StringByKey function V-997
StringFromList function V-998
WhichListItem function V-1095
PadString function V-732
page breaks
in notebooks III-19
page layouts II-476–II-499
$ and object names V-477
active page II-478
adding pages II-478
adjusting objects II-481
aligning
graph axes II-490
objects II-481
anchors for annotations II-494
annotation tool III-35
annotations II-483, II-494–II-496, III-34–III-53
position III-41
appending
objects II-486
pictures II-496
appending into marquee (example) I-41
AppendLayoutObject operation V-33
AppendToLayout operation V-36
arranging objects II-487
arrow tool II-481
auto-sizing objects II-481
background color II-477
bounding box II-489
changing drawing layers II-480
closing II-477

color scale bars V-74
ColorScale operation V-74
compatibility with Igor6 II-478
constrain during drag II-481
copying objects II-482, II-493
creating II-476
creating from functions V-682
Crop to Page Contents II-498
cutting objects II-482
default font II-496, V-147
DelayUpdate II-483, II-487
deleting pages II-478
drawing icon II-479
drawing layers III-68
drawing tools II-480
dummy objects II-487
dynamic updating II-476
exporting II-498, V-826
exporting section II-482
fidelity II-486
frames II-483
in notebooks III-18
info V-478
killing II-477
layers II-476, II-479–II-480
    ProgBack layer II-479–II-480
    ProgFront layer II-479–II-480
    UserBack layer II-479–II-480
    UserFront layer II-479–II-480
layout icon II-479
Layout operation V-476
LayoutInfo function V-478
LayoutPageAction operation V-480
LayoutSlideShow operation V-481
legend position III-41
legends II-483, II-495, V-484
list of V-1098, V-1100
magnification II-477
making a new layout II-45
    example I-19
margins II-478
margins of V-481
marquee II-489
marquees II-482
Misc pop-up menu II-483
modifying annotations III-35
modifying objects II-487
ModifyLayout operation V-639–V-640
names II-486, V-477
    instance names IV-20
names and titles II-45
NewLayout operation V-682
objects II-485
    properties II-498
page setups II-478
page size II-479
page sizes II-478
page sorter II-478
pages
    LayoutPageAction operation V-480
pasting color scale annotations II-494
pasting objects II-482, II-493
pasting pictures II-494
positioning annotations II-494
preferences II-498
preserve arrangement II-490
printing II-478–II-479, V-774
ProgBack drawing layer II-479–II-480
ProgFront drawing layer II-479–II-480
recreating II-477
recreation macros V-1101
related operations V-2
relation to graphs I-3
relation to tables I-3
RemoveLayoutObjects operation V-795
removing objects from II-486, V-793, V-795
resizing objects II-481
selected pages II-478
selecting multiple objects II-481
selecting objects II-481
selections V-315
size of V-480
slideshows V-481
Stack operation V-905
stacking graphs II-490
stacking objects II-482, II-487, V-905
style macros II-498
subwindow syntax III-92
subwindows III-90
textbox position III-41
textboxes II-483
thumbnails II-478
Tile operation V-1035–V-1036
tiling II-487
tiling guided tour II-488
tiling objects II-482, II-487, V-1035–V-1036
tool palette II-481–II-483
tools II-479
UserBack drawing layer II-479–II-480
UserFront drawing layer II-479–II-480
window names II-477, V-1048
window titles II-477
wintype function V-1103
zooming II-477
page numbers
    in notebooks III-13, III-20
Page Setup dialog
    graphs II-323, II-348
page setups
    in graphs II-323, II-348
    in notebooks III-30
    in page layouts II-478

    in tables II-271
    logical paper size V-320
    logical print table size V-320
    platform-related issues III-450
    preference in tables II-272
    settings V-775
page sorter
    in page layouts II-478
pages
    in page layouts II-478
    sizes in page layouts II-478
palette color
    TIFF files II-158
Panel subtype IV-205
    keyword V-732
PanelResolution function III-457, V-732
PanelResolution keyword III-456
panels III-442–III-446
    aborts IV-160
    as modeless dialog IV-158
    background color III-443, V-640
    control panel units III-444
    ControlNameList function V-94
    default control appearance V-147
    default font V-149
    drawing layers III-68
    expansion factor III-443
    floating III-443, V-687–V-688
    from user function IV-158
    list boxes V-488
    list of V-1098, V-1100
    making a new panel II-45
    ModifyPanel operation V-640
    names and titles II-45
    NewPanel operation V-687
    notebook subwindows III-91
    PanelResolution function III-457
    PauseForUser operation IV-156
    points versus pixels III-444
    preferences III-445
    progress windows IV-156
    recreation macros V-1101
    resolution of III-443, III-456, V-732
    screen dumps V-828
    shortcuts III-446
    Slider operation V-874
    subwindows III-82
    window names V-1048
    wintype function V-1103
panning graphs II-286
    examples I-35
    fling mode II-286
paragraph properties
    in notebooks III-7
parameter lists
    in macros IV-119
    in user functions IV-32
    inline in user functions IV-33
    optional IV-33, IV-60, V-733
parameters
    bits
        setting IV-12
    complex IV-32
    complex in user functions IV-112
    curve fitting III-179
    declaration
        in macros IV-119
        in user functions IV-32
    double in user functions IV-112
    expressions IV-12
    for functions IV-10
    for macros IV-10
    in commands IV-2, IV-11–IV-21
    inline in user functions IV-33
    int in user functions IV-33
    int64 in user functions IV-33
    optional IV-33, IV-60, V-733
    ParamIsDefault function V-733
    pass-by-reference IV-58–IV-59
    pass-by-value IV-58
    saving for reuse IV-147
    string IV-32
    string substitution IV-18–IV-19
    uint64 in user functions IV-33
    waves in macros IV-121
parametric surface plot
    data format II-452
ParamIsDefault function V-733
ParseFilePath function V-733
ParseOperationTemplate operation V-736
parsing
    liberal names IV-168
particle analysis
    images V-363
Parzen window function V-183, V-226, V-1097
pass-by-reference IV-58–IV-59
    waves IV-60
pass-by-value IV-58
passwords
    in URLs IV-267–IV-268
    security IV-270
    URLRequest operation V-1054
pasting
    data in tables II-239
    in page layouts II-482, II-493
    in tables II-248
path plot
    data format II-451
path plots
    Gizmo II-454
Path Properties dialog
    Gizmo II-455

Path Status dialog II-23
PathInfo operation V-737
PathList function V-738
paths
  cross-platform path separators III-451
  desktop V-895
  documents V-895
  extracting V-733
  HDF5 II-221
  HFS V-734
  length limit under Windows III-451
  manipulating V-733
  Open operation V-717
  packages V-895
  ParseFilePath function V-733
  path separators III-455
  platform-related issues III-449
  Posix V-734
  preferences V-895
  SpecialDirPath function V-895
  temporary V-895
  to functions V-287
  to macros V-525
  UNC V-734
  UNC paths III-451
  UniqueName function V-1048
  UNIX V-734
  Unix paths III-452
  user files V-895
PATTERNPOP V-766
patterns III-498
Paul wavelet transform V-140
pause Igor
  Sleep operation V-871
PauseForUser
  control panel example IV-156
  cursor example IV-152, IV-154
PauseForUser operation IV-152, V-738
  alternative to IV-154
  examples IV-152
PauseUpdate operation IV-123, V-739
PCA operation V-739
pcsr function V-741
PDFs
  blurry images III-96, III-103
  embedding fonts III-99
  exporting graphics III-96, III-103
  importing graphics III-510
  missing PostScript fonts III-99
  Symbol font III-493
peaks
  detection
    FindAPeak operation V-246
    FindPeak operation III-290–III-292, V-247–V-248
    Multi-peak Fit experiment III-290
    Technical Notes III-290
  EstimatePeakSizes operation V-203
  fitting
    MPFXEMGPeak function V-667
    MPFXExpConvExpPeak function V-668
    MPFXGaussPeak function V-669
    MPFXLorentzianPeak function V-670
    MPFXVoigtPeak function V-671
Pearson correlation coeffient III-222, III-233
pencil icon
  in notebooks III-5
  in procedure windows III-397
percent encoding IV-268
  URLRequest operation V-1058
percent-encoding
  URL function V-1053
  URLDecode function V-1053
percentiles
  smoothing III-296, V-878
performance testing IV-340, V-529
  GetRTLocation V-313
  GetRTLocInfo V-314
periodic functions
  cos V-109
  cot V-110
  csc V-115
  csch V-115
  fresnelCos V-264
  fresnelCS V-264
  fresnelSin V-264
  sawtooth V-831
  sec V-832
  sech V-832
  sin V-871
  sinc V-871
  tan V-1023
periodograms III-279
  calculation V-182
perspective projection
  Gizmo II-423
phase III-274–III-275, V-1091
  continuous III-316
  unwrapping III-274, III-374, V-433
    ImageUnwrapPhase operation V-433
    Unwrap operation V-1050
PhotoShop files
  exporting V-405
  file info V-372
Pi function V-742
PICTInfo function V-742
PICTList function V-742
PICTs
  DrawPICT operation V-176
  exporting V-405
  exporting file III-98, III-105, V-826
  file info V-372

importing III-509–III-510
in page layouts II-496
LoadPICT operation V-506–V-508
saving V-826
HiRes V-826
PostScript V-826
picture formats III-449, III-509
picture gallery III-510
SavePICT operation V-827
Picture keyword V-743
pictures III-509–III-511
aligning in notebooks III-9
cross-platform
in notebooks III-458
deleting (killing) III-511
DrawPICT operation V-176
exporting tables II-271
finding in notebooks V-702
gray boxes III-450, III-510
Igor object pictures III-18
importing III-509–III-510
in annotations III-54
in independent modules IV-244
in notebooks III-13, III-18–III-19, V-699, V-702, V-705, V-709–V-710
in page layouts II-494, II-496
inserting in notebooks III-13, V-706
KillPICTs operation V-470
LoadPICT operation V-506–V-508
movies from IV-246
names III-509–III-510
CheckName function V-68
naming rules III-501
Notebook operation III-26
PICTInfo function V-742
PICTList function V-742
picture formats III-449, III-509
platform compatibility III-19
related functions V-12
renaming III-504, V-797
SavePICT operation V-826–V-827
saving from notebooks III-14, V-708
scaling in notebooks III-14
UniqueName function V-1048
unnamed III-511
updating Igor object pictures III-28
updating in notebooks III-28
Pictures dialog III-510
pie wedge objects
Gizmo II-447
pitfalls
constrained curvefitting III-231
PixelFromAxisVal function V-743
pixels
AxisValFromPixel function V-45
image plots II-386, II-388
in panels III-456
inverting V-422
logical screen dimensions III-455
matching V-422
PixelFromAxisVal function V-743
versus points in panels III-444
Place Picture III-510
plain text
external text editor II-56
plain text files
byte order marks III-471
Override Experiment III-467–III-468
text encoding conversion errors III-469
text encodings III-466
UTF-8 III-470
planes
extracting V-420, V-426
matching V-422
scaling V-425
summing V-427
summing all V-427
platform-related issues III-448–III-458
backslash in file paths III-451
procedures III-455
experiment files III-448
file compatibility III-448–III-449
file extensions III-448
file names III-450
file paths in procedures III-455
file system III-450
folder names III-450
Macintosh file type III-448
notebook pictures III-19
notebooks III-458
page setup compatibility III-450
path separators III-451
Paths III-449
picture compatibility III-449
pictures in notebooks III-458
PNG graphic format III-450
symbolic path III-449
text encodings III-453
transferring files III-448–III-449
UNC paths III-451
Unix paths III-452
window position III-455
Windows-specific III-448
PlayMovie operation IV-245, V-744
PlayMovie without QuickTime installed V-744
PlayMovieAction operation IV-246, V-745
PlaySnd operation IV-245, V-746
PlaySound operation IV-245, V-747
plot area in graphs II-279, III-40
affected by annotations III-41
plot pane
Data Browser II-116

plot relative drawing III-67
PNGs III-97
  Convert to PNG III-458
  converting to III-19, III-450
  creating III-458
  cross-platform compatibility III-450
  exporting III-450, V-405
  file info V-372
  HTML graphics III-23
  importing III-509, V-390
  in notebooks III-19
  loading II-157
  on Clipboard III-97–III-98
  PNG versus raw PNG II-157
  raw PNG files V-408
  resolution III-458
  saving V-826
pnt2x function V-748
point column in tables II-252
  hiding V-649
  multidimensional waves II-262
point number-to-x value conversion V-748
point numbers II-63
  BinarySearch function V-51
  BinarySearchInterp function V-51
  fractional point numbers II-78
  increment II-76
  indexing II-74, II-76
  p function V-731
  pnt2x function V-748
  relation to X values II-63
  x2pnt function V-1114
point scaling II-64
  setting V-854
Point structure IV-103, V-749
point3D filter V-372
PointF structure V-749
points
  versus pixels III-455
  versus pixels in panels III-444, III-456
points versus pixels
  NewPanel operation III-444
Poisson window function V-183, V-226, V-1097
poissonNoise function V-749
polar graph macros III-73
polar-to-rectangular conversion V-732
poles and zeros III-301
poly
  curve fitting function III-210
poly curve fit function
  1D V-127
  2D V-128
poly function V-749
poly_XOffset V-127
  curve fitting function III-210
poly2D
  curve fitting function III-212
Poly2D curve fit function
  example III-202
poly2D function V-749
PolygonArea function V-750
PolygonOp operation V-751
polygons
  area V-754
  BezierToPolygon operation V-50
  difference V-754
  even-odd rule III-72
  exclusive OR V-754
  fill rules III-72
  FindPointsInPoly operation V-248
  intersection V-754
  Minkowski difference V-754
  Minkowski sum V-754
  PointsInPoly V-754
  PolygonArea function V-750
  PolygonOp operation V-751
  union V-754
  Voronoi polygons V-384, V-387, V-899
  winding rule III-72
polynomials
  2D series function V-749
  chebyshev function V-63
  chebyshevU function V-64
  curve fitting III-210, III-265–III-267
    two-dimensional III-212, V-403
  FindRoots V-251
  FindRoots operation V-248
  hermite function V-345
  hermiteGauss function V-346
  laguerre function V-474–V-475
  LaguerreA function V-475
  legendreA function V-484
  roots of III-338
  series function V-749
  x offset III-210, V-127
  ZernikeR function V-1120
polyOpWaveX V-751–V-752
pop-up menu controls
  colors V-766
  fill patterns V-766
  line styles V-766
  marker styles V-766
  menu items III-427
  mode III-427
  programming III-426–III-428
  update problems V-94
  updating III-428
  using III-416
  value keyword III-427
pop-up menus
  CTabList function V-118
  DataBrowserObjectsPopup menu IV-138

  - example of creating IV-162
  - GraphPopup menu IV-138
  - in macro dialogs V-782–V-783
  - in simple input dialogs IV-145
  - independent modules IV-241
  - special characters IV-133–IV-136, IV-145
    - Windows IV-134–IV-135
  - TablePopup menu IV-138
  - TraceNameList function IV-145
  - WaveList function IV-145
  - WindowBrowserWindowsPopup menu IV-140
- popup keyword V-755
  - in Prompt statements IV-145
- PopupContextualMenu
  - example IV-162
- popupcontextualmenu V-756
- PopupContextualMenu operation V-756
- PopupMenu operation III-426–III-428, V-761
  - value keyword III-427, V-763
- PopupMenuControl subtype IV-205, V-763
  - keyword V-768
- Porter-Duff
  - ImageComposite operation V-368
- position
  - managing windows II-50
  - moving windows to preferred II-50
  - retrieving all windows II-50
  - retrieving windows II-50
  - window III-455
- positional lights
  - Gizmo II-434
- Posix paths V-734
- PossiblyQuoteList function IV-169
- PossiblyQuoteName function V-768
  - example IV-168
- POST method IV-276
  - URLRequest operation V-1054, V-1058
- poster-sized graphs II-323
- PostScript
  - exporting
    - Macintosh OS X III-100
    - Windows OS III-106
  - exporting file III-98, III-105
  - fonts
    - embedding III-99, III-105
    - exporting
      - Macintosh OS X III-99–III-100
      - Windows OS III-105–III-106
  - language level III-97, III-103
- power
  - curve fitting function III-211
- power law curve fit V-128
- power spectra III-279
- power spectral density III-279
  - MultiTaperPSD operation V-672
- pragmas IV-52
  - DefaultTab III-406
  - DefaultTab keyword V-150
  - hide III-402, IV-54
  - hide keyword V-346
  - IgorVersion keyword V-362
  - IndependentModule IV-55, IV-238
  - IndependentModule keyword V-437
  - ModuleName IV-54, IV-236
  - ModuleName keyword IV-54, V-658
  - pragma keyword V-17
  - rtFunctionErrors IV-55
  - rtGlobals IV-52, IV-65
  - rtGlobals keyword V-811
  - TextEncoding pragma IV-55
  - version IV-54
  - version keyword V-1068
- pre-decrement operator IV-6, IV-43
- pre-increment operator IV-6, IV-43
- precedence
  - in conditional statements IV-41
  - of operators IV-6, IV-8
- precomposed characters
  - Unicode V-693
- predefined symbols IV-110
- prediction bands
  - for curve fitting III-226
- preferences III-516–III-517
  - auto-trace destination waves III-196
  - capturing III-516–III-517
  - category plots II-363–II-364
  - column styles II-272
  - command window II-12
  - contour plots II-380–II-381
  - Curve Fitting dialog III-201
  - dashed lines III-496
  - Data Browser II-119
  - directory III-516
  - graphs II-348–II-349, II-363, II-380, II-403
  - Igor Pro 8.ini file III-516
  - image plots II-403–II-404
  - in page layouts II-498
  - in procedure windows III-407
  - in procedures IV-203, V-768
  - load waves (example) II-172
  - notebooks III-30
  - packages IV-251–IV-255
    - loading IV-251, V-505
    - saving IV-251, V-825
      - in binary files IV-252
      - in experiments IV-254
  - path to folder V-895
  - Preferences operation IV-203, V-768
  - tables II-272
  - using III-516
  - vs style macros II-350–II-351
  - wave styles II-348

XY Plots II-348
Preferences operation IV-203, V-768
example IV-204
preserve arrangement
page layouts II-490
Prewitt compass gradient filters V-370
PrimeFactors operation V-769
principle component analysis V-739
Print operation V-769
structure elements V-770
Print Preview
for graphs II-323
printers
listing available V-775
printf operation IV-259–IV-261, V-770–V-773
conversion specifications IV-260, V-770–V-772
engineering units V-771
WaveMetrics extension V-771
PrintGraphs operation V-773–V-774
printing
Gizmo II-464
graphs II-323, II-348
notebooks III-20
page layouts II-478–II-479
poster-sized graphs II-323
PrintSettings operation V-775
procedure windows III-405
related operations V-6
selection from table II-271
settings V-775
structure elements V-770
tables II-271, V-779
to history area V-769–V-770
PrintLayout operation V-774
PrintNotebook operation V-775
example III-27
PrintSettings operation V-775
PrintTable operation V-779
private
static functions IV-105
problems II-4
Proc keyword IV-118, V-779
Proc Pictures IV-56–IV-58
creating IV-57
global IV-57, V-781
ProcGlobal keyword IV-57, V-781
using in commands IV-57
procedure declarations
End keyword V-194
EndMacro keyword V-195
Function keyword V-279, V-1005
Macro keyword V-523
Picture keyword V-743
Proc keyword V-779
Window keyword V-1097
procedure files III-396
adopting II-25, III-403
adopting all II-25
adopting programmatically V-23
autosave II-37
auxiliary III-396, III-398
converting to UTF-8 III-489
creating III-397–III-398
cross-platform issues III-454
DefaultTab keyword V-150
DefaultTab pragma III-406
easy access III-401
external text editor II-56
global III-396, III-399
hidden III-402
changes in Igor functionality III-403
creating III-402
hiding IV-54
Igor Pro User Files folder III-399–III-400
in experiments II-19, III-396, III-398, III-400, III-403
include statements III-400–III-401
inserting text III-403
invisible III-402
changes in Igor functionality III-403
creating III-402
lock icon III-397
locking III-397, V-642
opening III-398
packages III-401
ProcedureVersion function V-780
read-only III-397, V-642
references to II-24
rtGlobals automatic in new III-398
shared III-396, III-398, III-400
TextEncoding pragma IV-55
unlocking III-397
UTF-8 IV-56
version control IV-166
versions IV-54
write-protect V-643
procedure modules IV-236
procedure subtypes
ButtonControl keyword V-57
CameraWindow keyword V-58
CheckBoxControl keyword V-67
CursorStyle keyword V-124
FitFunc keyword V-255
GizmoPlot keyword V-323
Graph keyword V-325
GraphMarquee keyword V-325
GraphStyle keyword V-325
GridStyle keyword V-334
Layout keyword V-478
LayoutMarquee keyword V-480
LayoutStyle keyword V-483
ListBoxControl keyword V-498

Panel keyword V-732
PopupMenuControl keyword V-768
SetVariableControl keyword V-59, V-858
SliderControl keyword V-877
TabControl keyword V-1014
Table keyword V-1014
TableStyle keyword V-1015
procedure windows III-396–III-412
built-in III-396
CloseProc operation V-72
closing III-399
closing programmatically V-72
code marker comments III-408
color III-406
command line completion IV-3
default tabs III-405
executing commands from III-4
experiment recreation macros II-26
finding text III-404
fonts III-406
footers III-405
Go Back and Go Forward II-52
headers III-405
hiding III-399, V-642
HideProcedures operation V-347
include statements IV-166
indentation III-405
independent modules IV-239
initialization commands II-27
killing III-399
magnification II-53
making a new procedure window II-45
ModifyProcedure operation V-642
names and titles II-45
new window (example) I-59
OpenProc operation V-724
preferences III-407
printing III-405
procedures navigation bar III-397
ProcedureVersion function V-780
replacing text III-405
rtGlobals statements IV-65
showing III-398
DisplayProcedure operation V-163
syntax coloring III-407
templates III-397
text sizes III-406
text styles III-406
write-protect icon III-397
zooming II-53
procedures
aborting IV-49, IV-51, IV-112, V-18–V-19, V-1047
auto-compiling III-404, IV-26
bitwise and logical operators IV-41
case statements IV-43
Constant keyword V-85
Strconstant keyword V-996
comparisons IV-41
compiling III-397, IV-26
creating III-397
cross-platform issues
file extensions III-455
file types III-455
debugger IV-212–IV-227
debugging
using print statements IV-212
enabling preferences IV-203
experiment initialization commands IV-204
experiment recreation II-26
file path separators III-455
finding III-397, III-404
global IV-25
Igor Procedures IV-25
include statements IV-166
indentation IV-26
IndependentModule keyword V-437
local variable declaration IV-34
logical operators IV-7
loops IV-45
MacroList function V-524
menu definitions IV-126–IV-141, V-54
ModuleName keyword IV-54, V-658
modules IV-236
names IV-31
operation queue IV-278
organizing IV-24–IV-25
overview I-5
parameter declarations IV-32
parameter lists IV-32
parameters IV-10
pause for user IV-152
PauseForUser operation V-738
ProcedureText function V-779
ProcedureVersion function V-780
processing lists of waves IV-198–IV-201
programming with liberal names IV-168–IV-169
retrieve code
ProcedureText function V-779
saving parameters for reuse IV-147
scanning IV-26
subtypes IV-32, IV-204–IV-205
switch statements IV-43
Constant keyword V-85
Strconstant keyword V-996
User Procedures IV-25
using cursors IV-163
using marquee menus IV-163
using polygons for input IV-164
utility procedures IV-25
WaveMetrics Procedures IV-25
writing utility procedures IV-167

- procedures navigation bar III-397
- ProcedureText function V-779
- ProcedureVersion function V-780
- ProcGlobal keyword IV-57, IV-244–IV-245, V-781
  - independent modules IV-238
  - pop-up menus IV-242
  - regular modules IV-236
- profiling IV-340
  - GetRTLocation V-313
  - GetRTLocInfo V-314
- ProgAxes drawing layer III-68
- ProgBack drawing layer III-68
  - in page layouts II-479–II-480
- ProgFront drawing layer III-68–III-69
  - in page layouts II-479–II-480
- program name
  - under Windows III-448
- programming
  - analysis procedures III-170–III-175
  - annotations III-52
  - AppleScript execution V-205
  - closing files IV-195
  - commands IV-2–IV-21
  - conditional compilation IV-108
  - controls III-419–III-442
  - debugging IV-212–IV-227
  - displaying help topics V-163
  - drawing tools III-73–III-77
  - examples III-171–III-175
  - Execute operation V-204
  - Execute/P operation V-204
  - ExecuteScriptText operation V-205
  - files IV-194
  - finding files IV-195
  - functions
    - user-defined IV-30–IV-114
  - generating notebook commands III-29
  - GetRTStackInfo function V-314
  - interaction IV-144–IV-164
  - keywords
    - #define V-16
    - #if-#elif-#endif V-16
    - #if-#endif V-16
    - #ifdef-#endif V-17
    - #ifndef-#endif V-17
    - #include V-17
    - #pragma V-17
    - #undefine V-18
    - Constant V-85
    - DefaultTab V-150
    - DoPrompt V-167
    - hide V-346
    - IgorVersion V-362
    - IndependentModule V-437
    - Menu V-589
    - ModuleName IV-54, V-658
    - Override V-731
    - popup V-755
    - ProcGlobal V-781
    - Prompt V-782
    - root V-810
    - rtGlobals V-811
    - Static V-906
    - Strconstant V-996
    - Submenu V-1006
    - TextEncoding V-1027
    - version V-1068
  - loading files in folder (example) II-175
  - loading waves II-170
    - examples II-171, II-173
  - macros IV-116–IV-124
  - menus IV-126–IV-141
  - notebooks III-25–III-30
    - examples III-26
  - opening files IV-195
  - overview I-8, IV-24–IV-27
  - predefined symbols IV-110
  - related functions V-11
  - related operations V-5
  - retrieving text from notebooks III-28
  - runtime stack information V-314
  - SetIgorOption operation V-851
  - syntax coloring III-407
  - techniques IV-166–IV-209
  - text encodings III-480
  - Unicode III-480
  - updating in notebooks III-28
  - user interaction IV-144–IV-164
  - user-defined functions IV-30–IV-114
  - UTF-8 III-480
  - waves as parameters III-170
  - writing to text files IV-196
- progress windows IV-156
- Project operation V-781
- projections
  - Gizmo II-422
  - Project operation V-781
- Prompt statements V-782–V-783
  - help for IV-148
  - in macros IV-121
  - in user functions IV-144
  - location of IV-147
  - pop-up menus IV-145
- proportional weighting
  - curve fitting III-199
- prototype functions IV-107
- Provide Wave Names II-133, II-140
- proxy servers IV-276
  - URLRequest operation V-1056
- Pseudo-Coifman wavelet transform V-189
- pseudorandom numbers
  - LinearFeedbackShiftRegister operation V-485

- pseudorandom sequences
  - LinearFeedbackShiftRegister operation V-485
- public functions IV-236, IV-244
  - independent modules IV-238
  - regular modules IV-236
- pulse measurements
  - EdgeStats operation III-289
  - EstimatePeakSizes operation V-203
  - FindLevel operation V-242–V-244
  - FindLevels operation V-244–V-245
  - FindPeak operation V-247–V-248
  - PulseStats operation III-290
- pulse statistics
  - EdgeStats operation V-190
  - PulseStats operation V-783
- PulseStats operation V-783–V-785
  - NaN results V-784
- PUT method
  - URLRequest operation V-1054
- PutScrapText operation V-785

Q

- q function V-785
  - in multidimensional waves II-92–II-93, II-96
- qcsr function V-785
- Qt
  - graphics technology III-506
  - SVG III-507
- quad objects
  - Gizmo II-438
- quadric objects
  - Gizmo II-467
- qualified names IV-237
  - independent modules IV-238–IV-239
  - triple names IV-242–IV-243
- quartiles
  - box plots II-331
- Quartz
  - font embedding III-99
  - graphics technology III-507
- quaternion
  - Gizmo II-427
  - multiplication II-427
- quaternions
  - convert to rotation matrix V-573
  - in MatrixOp III-146, V-573
  - spherical linear interpolation V-573
- queues
  - multitasking IV-330
- quick append II-348
- QuickTime IV-245
  - file info V-372
- QuickTime files
  - exporting V-405
- QuickTime movies
  - NewMovie operation V-683
  - related operations V-6
- quit Igor hook function IV-292
- Quit operation V-786
- Quit Scan button IV-122
- quotation marks
  - for liberal names III-501
  - in delimited text files II-135
  - saving text waves II-180
- quoted strings
  - in delimited text files II-132, II-135

R

- r function V-786
  - in multidimensional waves II-92–II-93, II-96
- r-squared III-221
- r2polar function V-786
  - example II-81
- radio button controls
  - programming III-424
  - using III-416
- random functions
  - gnoise V-323
  - LinearFeedbackShiftRegister operation V-485
  - Mersenne Twister V-323, V-853
  - SetRandomSeed operation V-853
- random numbers
  - enoise V-195
- Range-Based For Loop V-259
- range-based for loops V-259
  - in user functions IV-47
- ranges of interest
  - curve fitting to III-197
    - example III-197
  - in Duplicate Waves dialog II-70
  - using cursors in graphs II-320
- ratio of integers V-786
- RatioFromNumber operation V-786
- raw PNG II-157
- read-only
  - ModifyProcedure operation V-642
  - notebooks III-10
- readme
  - data folder notes II-120
- ReadVariables operation V-788
- real
  - columns in tables II-241
  - suffix in tables II-241
- real component function V-788
- real function V-788
  - example II-81
- real number type II-66
- Recent Colors III-497
- Recent Windows submenu II-46
- recreating

  - graphs II-350
  - page layouts II-477
  - tables II-241
- recreation macros II-47–II-48, II-350, V-1101
  - deleting II-48
  - DoWindow operation V-168
  - Gizmo II-471
  - notebook subwindows III-91
- Rect structure IV-103, V-787
- rectangular-to-polar conversion V-786
- RectF structure V-787
- rectification
  - images V-384
- Redefine Ruler from Selection III-12
- redefining
  - rulers III-12
- Redimension operation II-72, II-98, V-788
- Redimension Waves dialog II-72
  - from tables II-267, II-269
- redirects
  - URLRequest operation V-1055
- reduced major axis curve fitting III-237
- references
  - convolution V-103
  - curve fitting III-267, V-134
  - HDF5 II-208
  - to files and folders II-24
  - to global procedure files III-400
  - to Igor binary wave files II-24, II-155–II-156
    - why it’s bad II-156
  - to notebooks III-3
  - to procedure files III-398, III-400
- reflection
  - Gizmo II-432
- region of interest III-378
  - creating V-374
  - creating ROI wave III-378
  - example V-375
  - masking V-367
  - objects V-374
  - statistics V-414
- regular expressions IV-176–IV-194
  - alternation IV-183
  - assertions IV-190–IV-192
    - lookahead IV-190
    - lookbehind IV-191
    - multiple IV-191
  - atomic grouping IV-188
  - back references IV-189
  - backslashes IV-179–IV-181
    - generic character types IV-181
    - nonprinting characters IV-179
    - simple assertions IV-181
  - brackets IV-182
  - character classes IV-178, IV-182
  - characters
    - circumflex IV-181
    - dollar IV-181
    - dot IV-182
    - newline IV-184
    - period IV-182
    - vertical bar IV-183
  - circumflex character IV-181
  - comments IV-193
  - dollar character IV-181
  - dot character IV-182
  - full stop IV-182
  - greedy IV-187
  - Grep operation IV-176, V-327
  - GrepList function V-333
  - GrepList operation IV-176
  - GrepString function IV-177, V-334
  - LoadData operation V-503
  - match option settings IV-184
  - metacharacters IV-178
  - newline IV-184
  - period character IV-182
  - posessive qualifiers IV-188
  - POSIX character classes IV-183
  - quantifiers IV-186
  - recursive patterns IV-193
  - references IV-194
  - repetition IV-186
  - SplitString function V-900
  - SplitString operation IV-177
  - subpatterns IV-186
    - conditional IV-192
    - named IV-186
    - subroutines IV-194
  - vertical bar character IV-183
- regular modules IV-236
  - control action procedures IV-237
  - hook functions IV-237
  - ProcGlobal keyword IV-236
  - ProcGlobal keywords IV-244
  - public functions IV-236
  - qualified names IV-237
  - user-defined menus IV-238
  - within independent modules IV-242
- RELOAD CHANGED PROCS IV-280
- remainder function V-592
- Remez Exchange III-311
- Remez operation V-789
- Remove Columns dialog II-242
- Remove from Graph dialog II-283
- Remove Objects dialog II-486
- Remove operation V-790
- Remove Ruler dialog III-12
- RemoveByKey function V-790
- RemoveContour operation V-791
- RemoveEnding function V-792
- RemoveFromGizmo operation V-792

RemoveFromGraph operation V-792
RemoveFromLayout operation V-793
RemoveFromList function V-793
RemoveFromTable operation V-794
RemoveImage operation V-794
RemoveLayoutObjects operation V-795
RemoveListItem function V-795
RemoveNaNsXY function III-113
RemoveOutliers function III-171
RemovePath operation V-796
Rename Objects dialog II-72, III-504
Rename operation V-796
  renaming waves II-72
RenameDataFolder operation V-796
RenamePath operation V-797
RenamePICT operation V-797
RenameWindow operation V-797
renaming
  waves in tables II-234, II-239
  with Data Browser II-115
ReorderImages operation V-797
reordering traces II-297
ReorderTraces operation V-798
repeat end effect method III-299
repeating sliders III-418, V-875, V-877
Replace Text dialog III-24
replace-paste II-248
ReplaceNumberByKey function V-799
ReplaceString function V-799
ReplaceStringByKey function V-800
ReplaceText operation V-801
ReplaceWave operation V-801
  examples II-114, II-284
replacing
  in notebooks III-24
  in tables II-251
  replace text shortcut III-24
  table values II-251
  text in multiple windows II-55
  text in procedure windows III-405
  text using find bar II-54
  traces in graph II-114
ReplicateString function V-802
Resample operation V-803–V-808
reserved characters
  in URLs IV-268
residuals in curve fitting III-184, III-217
resolution
  of panels III-456
  screen III-507
restoring
  images V-404
ResumeUpdate operation IV-123, V-809
Retina
  graphics technology III-507
Retina displays III-507
Retry button IV-122
Return key
  in tables II-243
return keyword V-809
return statements
  in macros IV-120
  in user functions IV-35
  multiple return statements IV-36
Reverse operation V-809
reverting
  experiments II-20
RGB conversions
  cmap2rgb V-418
  CMYK2RGB V-418
  hsl2rgb V-421
  rgb2gray V-424
  rgb2hsl V-424
  rgb2i123 V-424
  rgb2xyz V-424
  xyz2rgb V-428
RGB images
  exporting V-408
  TIFF files II-158
RGB waves II-400–II-401
  color tables II-400
RGBA
  in Gizmo II-428
RGBA images
  exporting V-408
  TIFF files II-158
RGBA values IV-13
RGBA waves II-400–II-401
  color tables II-400
RGBAColor structure V-810
RGBColor structure IV-103, V-810
ribbon plot
  data format II-451
ribbon plots
  Gizmo II-462
Richardson-Lucy deconvolution V-404
Riemann window function V-183, V-227, V-1097
Riemann zeta function V-1120
right-shift operator IV-6, IV-43
rightx function V-810
risetime measurements III-289–III-290
RMA curve fitting III-237
Robert’s row and column edge detector V-370
Romberg integration V-448
root keyword V-810
roots
  finding III-338–III-343
  FindRoots operation V-248
  of nonlinear 1D functions III-339
  of nonlinear 2D functions III-341, III-343
  of polynomials III-338
  references III-349

Rotate dialog III-316
Rotate operation III-316, V-810
  X scaling change III-316
Rotate to Match II-416
rotated text III-63
rotating
  annotations III-40
  drawn objects III-63
rotation
  Gizmo II-416, II-424
    arrow keys II-424
    mouse wheel II-424
rotation matrices
  convert from quaternion V-573
rotation tool
  Gizmo II-415
round function V-811
rounding
  integer wave calculations IV-9
rounding functions V-7
  abs V-19
  cabs V-58
  ceil V-59
  floor V-255
  mod V-592
  round V-811
  sign V-870
  trunc V-1047
roundoff error IV-8
  and equality operator IV-8
row column in tables
  hiding V-649
row indices
  in tables II-261
row labels
  in delimited text files II-134
row major
  HDF5 II-204
row numbers
  p function V-731
row position waves
  in delimited text files II-134
row-major order II-99, V-294
rows
  extracting V-420
  flipping V-420
  in multidimensional waves II-92, II-95
  in waveform data II-62
  maximum location V-1084
  minimum location V-1084
  relation to X values II-63
  summing all V-427
RRect radius drawing setting III-64
RTF
  margins III-6
  notebooks III-20
  opening notebooks III-21
  saving notebooks III-21
rtFunctionErrors pragma IV-55
rtGlobals IV-52, IV-65
  automatic in new procedure files III-398, IV-65
  converting to rtGlobals=1 IV-114
  pragma keyword V-811
  rtGlobals=0 IV-114
  rtGlobals=1 IV-65–IV-67
  strict wave reference mode IV-75
rug plot II-336, II-340
rug plots II-340
rulers
  creating derived rulers III-12
  creating new rulers III-11
  finding where used III-12
  fonts in notebooks III-7, III-12
  HTMLCode III-23
  in notebooks III-5, III-7, III-11–III-12
  names III-11
  naming rules III-501
  redefining III-12
  removing III-12
  Ruler pop-up menu III-11
  text color in notebooks III-7, III-12
  text sizes in notebooks III-7, III-12
  text styles in notebooks III-7, III-12
  transferring III-12
Run Autosave Now II-39
Runge–Kutta–Fehlberg method V-454
runtime lookup of globals IV-65–IV-67
  converting from direct reference IV-114
  example IV-69
  failures IV-67
runtime stack
  information about V-314
runtime vs compile time IV-64
runtime wave type checking IV-53

S

s function V-811
  in multidimensional waves II-92–II-93, II-96
S_ variables
  in macros IV-123
  in user functions IV-61
S_aliasPath V-301
S_BrowserList II-120
S_columnName V-307
S_columnWidths V-91
S_creator V-302
S_CursorAInfo IV-340
S_CursorBInfo IV-340
S_dataFolder V-91, V-316
S_Filename V-268

S_fileName IV-149, V-105–V-106, V-112, V-265, V-294, V-394, V-464, V-515, V-592, V-660, V-662, V-706, V-709, V-718, V-891, V-1057, V-1117
loading waves (example) II-171
LoadWave operation II-170
Open operation V-718, V-720
S_fileType V-302
S_fileVersion V-302
S_firstColumnPath V-307
S_font V-588
S_graphName V-307
S_headers V-1057
S_Info III-234
S_info V-119, V-229, V-265, V-394, V-507, V-517–V-518, V-747, V-850, V-891
S_marqueeWin V-310
S_Name V-844
S_name V-150, V-154, V-162, V-193, V-312, V-681, V-683, V-688, V-692, V-702, V-706, V-709
S_outputFullPath V-1052
S_path V-105–V-106, V-112, V-155–V-156, V-265, V-294, V-301, V-394, V-464, V-515, V-592, V-660, V-662, V-737, V-818, V-847, V-891, V-1117
LoadWave operation II-170
S_pictureInfo V-722
S_recreation V-90, V-172
S_selection V-316, V-757–V-759
S_serverResponse V-1057
S_SoundInName V-889
S_tableName V-307
S_title V-90
S_traceName V-307
S_TraceOffsetInfo V-625
S_UserData V-90
S_Value V-208–V-209, V-827
S_value V-45, V-89, V-91, V-93, V-166, V-307, V-319–V-321, V-328–V-329, V-766, V-776, V-900, V-1117
S_waveNames V-294, V-394, V-464, V-515, V-592, V-891, V-902, V-1117
loading waves (example) II-171
LoadWave operation II-170
S_worksheetName V-1117
S_xWave V-326
S_yWave V-326
same aspect
printing graphs II-323
Save and then kill button III-4
save as template II-21
Save Delimited Text dialog II-178, II-180
Save EPS File dialog III-98, III-105
for page layouts II-498
Save Experiment dialog II-18
Save File dialog IV-151, V-718
Save Graphics
pictures in notebooks III-14
Save operation V-812
Save PICT File dialog III-98, III-105
for page layouts II-498
save save II-29
Save Waves submenu II-177
SaveData operation V-815
SaveExperiment operation V-819
compression II-214
SaveGizmoCopy operation V-821
SaveGraphCopy operation V-821
SaveNotebook operation V-823
text encodings V-823
SavePackagePreferences operation V-825
SavePICT operation V-826
PictGallery V-827
Gizmo II-465
SaveTableCopy operation V-829
saving
pictures from notebooks V-708
saving data
SaveData operation V-815
saving experiments II-16–II-19, II-29
errors II-30
SaveExperiment operation V-819
Spotlight II-30
saving graphs
save graph copy II-323
SaveGraphCopy operation V-821
with waves II-323, V-821
saving notebooks
as HTML III-21
as RTF III-21
SaveNotebook operation V-823
saving pictures
from notebooks III-14
saving tables
save table copy II-270
SaveTableCopy operation V-829
saving waves II-177–II-181
64-bit capable wave record II-35
appending II-179
archiving V-815
as image files II-180
BigWaveLoadMode II-35
carriage return II-180
column labels II-178
delimited text files II-178
escape characters II-180
FBinWrite operation V-220
from Data Browser II-117
general text files II-179
graphics V-405
Igor binary wave files II-179
Igor Text files II-179

images V-405
ImageSave operation V-405
linefeed II-180
MaxBytesReadingWave II-35
multidimensional waves II-181, V-814
numeric precision II-179
quotation marks II-180
row labels II-178
Save operation V-812
SaveData operation V-815
saving huge waves II-35
table formatting V-813
tabs II-180
text encodings II-181
text waves II-180
use table formatting II-179
UseNewDiskHeaderBytes II-35
UseNewDiskHeaderCompress II-35
wfprintf operation V-1093
with Gizmo plot V-821
with graphs II-323, V-821
Savitzky-Golay smoothing III-294, V-878
sawtooth function V-831
ScaleOffset
HDF5 compression II-213
ScaleToIndex function V-832
scaling pictures
in notebooks III-14
in page layouts II-481
with drawing tools III-63
scanning
procedures IV-26
Scatter Properties dialog
Gizmo II-456
schemes
in URLs IV-267
URLRequest operation V-1054
scientific notation
axis labels II-319
prevention in graph labels II-319
tick mark labels II-308
forcing II-311
scientific numeric format
in tables II-255
screen
dimensions, platform-related III-455
resolution III-507
resolution of panels III-456
screen dumps V-828
screen preview
for EPS III-97, III-103
ScreenResolution function V-832
scripts
calling Igor from IV-265
scrolling graphs (examples) I-35
scrolling mode
in control panels V-319, V-866
in graphs V-319, V-321, V-866
Search Backwards III-24
Search Selected Text Only III-24, III-405
searching
Igor files II-3
sec function V-832
sech function V-832
Secs2Date function V-833
Secs2Time function V-833
Secure Socket Layer IV-276
security
for passwords IV-270
Select All
in Modify Columns dialog II-253
in tables II-243
Select Control III-419
Select Current Folder button II-28
selected pages
in page layouts II-478
selecting
cells in tables II-243, V-315
controls III-419–III-420
drawing objects III-63
multiple controls III-420
objects in page layouts V-315
Select All in tables II-243
text in notebooks V-315
SelectNumber function V-834
SelectString function V-834
Send to Back II-481
serial number
IgorInfo function V-360
serial port
data acquisition IV-312
series III-109
SumSeries operation V-1008
Set as f(z) dialog II-298
Set Text Format dialog III-8
Set Variable controls
programming III-428–III-429
SetActiveSubwindow operation V-835
SetAxis operation V-835
SetBackground operation IV-233, IV-323, V-836
SetDashPattern operation V-837
SetDataFolder operation V-837
SetDimLabel operation V-838
SetDrawEnv operation III-74, V-838–V-843
grouping with gstart and gstop III-74
save III-74
SetDrawLayer operation III-74, V-844
SetEnvironmentVariable function V-844
SetFileFolderInfo operation V-845
SetFormula operation IV-233–IV-234, V-847
SetIdlePeriod operation V-848
SetIgorHook operation V-848

- SetIgorMenuMode operation V-851
- SetIgorOption operation V-851
  - enable debugger for independent modules IV-239
  - PanelResolution keyword III-456
  - Procedure Window submenu IV-239
  - syntax coloring III-407
- SetMarquee operation V-852
- SetProcessSleep operation V-852
- SetRandomSeed operation V-853
- SetScale operation V-853–V-854
  - dates V-854
  - setting X scaling II-63
- SetVariable operation III-428–III-429, V-854–V-1064
  - example IV-159
  - formatted text III-53
- SetVariableControl subtype IV-205, V-858
  - keyword V-59, V-858
- SetWaveLock operation V-858
- SetWaveTextEncoding operation III-477, V-859
- SetWindow operation V-865
- SGI files
  - exporting V-405
  - file info V-372
- SHA-1 hash algorithm V-1073
- SHA-224 hash algorithm V-338, V-1073
- SHA-256 hash algorithm V-338, V-1073
- SHA-384 hash algorithm V-338, V-1073
- SHA-512 hash algorithm V-338, V-1073
- shading
  - Gizmo II-433
- Shapiro-Wilk test V-976
- shared procedure files III-396, III-400
  - Igor Pro User Files folder III-400
- shared text files II-56
- shared waves II-87
- Shen–Castan edge detector V-370
- Shift JIS text encoding III-460–III-461, III-481
  - backslash characters III-484
  - history area III-470
- Shift-Escape
  - aborting procedures IV-51
- shininess
  - Gizmo II-429, II-432, II-472
- shoelace algorithm V-750
- short date
  - in notebooks III-13
- short-time Fourier Transform V-994
- shortcuts
  - axes II-353–II-354
  - command window II-13
  - contour plots II-382–II-383
  - controls III-446
  - CreateAliasShortcut operation V-111
  - creating V-111
  - data browser II-122
  - debugger IV-226
  - drawing tools III-78
  - graphs II-353–II-354, II-383, II-404, III-446
  - help II-5
  - Igor Tips II-5
  - image plots II-404
  - page layouts II-499
  - panels III-446
  - procedure windows III-411
  - templates II-5, III-397
  - traces II-353–II-354
  - windows II-60
- Show Axis Cue II-415
- Show Default Text Encoding III-465
- show events
  - window hooks IV-304
- Show Experiment Info III-465
- Show Info II-319
- Show Multivariate Functions
  - in Curve Fitting dialog III-201
- Show Tools III-62
- ShowIgorMenus operation V-869
- ShowInfo operation V-869
- showing
  - notebooks III-4
  - procedure windows III-398
    - DisplayProcedure operation V-163
- ShowTools operation V-870
- shrinking graphs II-286
- shuffle
  - HDF5 compression II-213
- sigmoid
  - curve fitting function III-211
- sigmoid curve fit V-128
- sign function V-870
- sign of a wave V-1091
- signal processing III-270–III-292
  - continuous wavelet transform III-282, V-137
  - convolution III-284
  - Convolve operation V-101
  - correlation III-286
  - CWT operation V-137
  - decimating III-136, V-803
  - discrete wavelet transform III-283, V-189
  - down-sampling V-803
  - DPSS operation V-171, V-672
  - DSP support procedures III-274
  - DSPDetrend operation V-182
  - DSPPeriodogram operation V-182
  - DWT operation V-189
  - FFT III-270
  - FFT operation V-222
  - FilterFIR operation V-230–V-233
  - FilterIIR operation V-234–V-239
  - Hanning operation V-337
  - Hanning window III-277

Hilbert transform III-280
HilbertTransform operation V-348
IFFT operation V-357
IIR filters V-234–V-239
coefficients, designing V-238
ImageWindow operation V-435
InstantFrequency operation V-443
interpolating V-803
LombPeriodogram operation V-521
lowpass filtering V-803
magnitude III-274
operations V-4
periodograms III-279, V-182, V-521
phase III-274
power spectra III-279
Resample operation V-803–V-808
Rotate operation V-810
SmoothCustom operation V-883
STFT operation V-994
time frequency analysis III-280
trend removal V-182
Unwrap operation V-1050
up-sampling V-803
Wigner transform III-281
WignerTransform operation V-1095
windowing III-275, V-182
significant digits
in tables II-255
Silent operation IV-122, V-871
compatibility mode V-871
obsolete for comments IV-113
simple input dialog IV-144
Help button IV-145
help for IV-148
pop-up menus IV-145
saving parameters for reuse IV-147
simulated annealing V-725, V-729
simultaneous equations III-161
sin
curve fitting function III-209
sin (sine) curve fit V-127
sin function V-871
sin of a wave V-1091
sinc function V-871
single precision V-788
single-precision floating point II-66
single-stepping in debugger IV-216
singular value decomposition III-179, III-265, V-584
MatrixInverse operation V-541
singularities in curve fitting III-265
sinh function V-871
size
expanding windows to full II-50
managing windows II-50
moving windows to preferred II-50
skeleton
ImageSkeleton3D operation V-413
skewness of wave V-1084
2D V-415
Skip this block button II-140
Skip this Path button II-29
Skip this Wave button II-27
skipping lines
in delimited text files II-136
in general text files II-141
Sleep operation V-871
slices
in tables II-262
Slider controls III-418, III-429
action procedures III-430
event handling III-430
Slider operation V-874
Slider operation V-874
SliderControl subtype IV-205
keyword V-877
sliders
repeating III-418, V-875, V-877
slideshows
page layouts V-481
sliding average V-878
Slow operation IV-122, V-877
Smooth operation V-878–V-882
SmoothCustom operation III-298, V-883
smoothing III-292–III-299
binomial III-294, V-878
box III-295, V-53, V-878
BoxSmooth operation V-53
convolution III-298
custom coefficients III-298, V-230, V-234
cutoff frequency V-879
end effects III-299
FilterFIR operation V-230–V-233
FilterIIR operation V-234–V-239
gaussian III-294, V-878
half power point V-879
in image plots V-637
Least Squares Polynomial III-294, V-878
locally-weighted regression V-515
loess III-297
Loess operation V-515
LOWESS III-297, V-515
median III-114, III-296
nonparametric regression V-515
Nyquist frequency V-879
percentile III-296
percentiles V-878
polynomial V-878
Savitzky-Golay III-294, V-878
Smooth operation V-878–V-882
SmoothCustom operation V-883
WavesAverage user-defined function III-173
smoothing spline interpolation III-115

algorithm III-118
parameters III-119
snapshots V-828
Sobel edge detector V-370
Sort operation III-132, V-883
SortColumns operation V-884
sorting III-132–III-135
description of III-133
file lists V-441
guided tour (example) I-48
index sort III-134–III-135
index wave applications III-134
IndexSort example III-135
IndexSort operation III-134–III-135, V-442
MakeIndex example III-135
MakeIndex operation III-134–III-135, V-527
many waves III-134
multiple keys III-133
Reverse operation V-809
simplest case III-133
Sort operation III-132, V-883
SortColumns operation V-884
SortList function V-886
text III-133
case-sensitive III-133
example III-133
unsorting III-134
XY data III-133
SortList function V-886
sorting file lists V-441
sound
WAVE files V-892
SoundInRecord operation V-887
SoundInSet operation V-887
SoundInStartChart operation V-888
SoundInStatus operation V-888
SoundInStopChart operation V-889
SoundLoadWave operation V-890
sounds
AddMovieAudio operation V-21
AIFF files V-892
Beep operation V-46
input IV-245
NewMovie operation V-683
output IV-245
PlaySnd operation V-746
PlaySound operation V-747
recording from microphone IV-312
related operations V-6
SoundInRecord operation V-887
SoundInSet operation V-887
SoundInStartChart operation V-888
SoundInStatus operation V-888
SoundInStopChart operation V-889
SoundLoadWave operation V-890
SoundSaveWave operation II-180, V-892
SoundSaveWave operation II-180, V-892
source waves
in wave assignment II-74
subranges of II-76
spaces
as delimiters II-136
in delimited text files II-136
in fixed field text files II-136–II-137
in general text files II-138
in text data columns II-134
spacing
in notebooks III-7–III-8
sparse data
converting to matrix II-367
sparse matrices III-151–III-162
ADD operation III-157
COO format III-152
CSC format III-153
CSR format III-152
example III-153
format conversion III-159–III-160
formats III-151
ImageFromXYZ operation V-373
index data type III-156
inputs III-155
list of operations III-154
MatrixSparse operation V-580
MM operation III-157
multiplication III-157–III-158
MV operation III-158
operation data type III-155
operations III-156
simultaneous equations III-161
SMSM operation III-158
TOCOO operation III-159
TOCSC operation III-159
TOCSR operation III-160
TODENSE operation III-160
transformations III-156
TRSV operation III-161
spatial frequency filtering III-360
special characters
in notebooks III-13, V-709–V-710
in tables II-261
special folders II-30
special functions V-8
airyA V-24
airyAD V-24
airyB V-25
airyBD V-25
besseli V-46
besselj V-46
besselk V-47
bessely V-47
bessI V-47
beta V-49

  - betai V-49
  - binomial V-52
  - binomialln V-52
  - binomialNoise V-52
  - chebyshev V-63
  - chebyshevU V-64
  - Ci V-109
  - dawson V-145
  - digamma V-159
  - ei V-194
  - EllipticE V-194
  - EllipticK V-194
  - erf V-197
  - erfc V-198
  - erfcw V-198
  - expInt V-212
  - expnoise V-213
  - factorial V-214
  - fresnelCos V-264
  - fresnelCS V-264
  - fresnelSin V-264
  - gamma V-288
  - gammaEuler V-288
  - gammaNoise V-289
  - gammInc V-288
  - gammln V-289
  - gammp V-289
  - gammq V-290
  - hermite V-345
  - hermiteGauss V-346
  - hyperG0F1 V-353
  - hyperG1F1 V-353
  - hyperG2F1 V-354
  - hyperGNoise V-354
  - hyperGPFQ V-354
  - inverseErf V-463
  - inverseErfc V-463
  - JacobiCn V-465
  - JacobiSn V-465
  - laguerre V-474–V-475
  - LaguerreA V-475
  - LambertW V-475
  - legendreA V-484
  - logNormalNoise V-521, V-523
  - poissonNoise V-749
  - sphericalBessJ V-897
  - sphericalBessJD V-897
  - sphericalBessY V-897
  - sphericalBessYD V-898
  - sphericalHarmonics V-898
  - sqrt function V-902
  - StatsVonMisesNoise V-987
  - wnoise V-1113
  - ZernikeR V-1120
- special numbers
  - e V-190
  - Inf V-442
  - NaN V-676
  - Pi V-742
- Special submenu III-14
- SpecialCharacterInfo function V-893
- SpecialCharacterList function V-895
- SpecialDirPath function V-895
- Specify Color
  - Gizmo II-419
- spectral leakage
  - in FFTs III-277
- specular light
  - Gizmo II-432
- sphere objects
  - Gizmo II-440
- spherical interpolation V-899
- spherical linear interpolation V-573
- sphericalBessJ function V-897
- sphericalBessJD function V-897
- sphericalBessY function V-897
- sphericalBessYD function V-898
- sphericalHarmonics function V-898
- SphericalInterpolate operation V-899
- SphericalTriangulate operation V-899
- splines wavelet transform V-189
- SplitAxis Dialog II-347
- SplitString operation IV-177, V-900
- splitting
  - waves V-901
- splitting up waves II-70
- SplitWave operation V-901
- Spotlight II-30
- spreadsheets
  - compared to tables II-234
- sprintf operation IV-259–IV-261, V-902
  - conversion specifications IV-260
- SQL II-181
- sqrt function V-902
- sqrt of a wave V-1091
- square root V-902, V-1091
- sscanf operation V-902
- SSL IV-276
- stack in debugger IV-216
- Stack operation V-905
- stacking
  - graphs in page layouts II-490
  - in page layouts II-482, II-487, V-905
  - windows II-50, V-905
- StackWindows operation V-905
- standard deviation
  - for a point in curve fitting III-179, III-199
  - of a wave V-415, V-1077, V-1084
  - of an image V-415
  - of parameters in curve fitting III-222
- standard error of mean V-1084
- StartMSTimer function V-906

static
  constants IV-51
  functions V-906
  user functions
    ModuleName pragma IV-54
static functions IV-105, IV-236, IV-244
  independent modules IV-238–IV-239, IV-243
  regular modules IV-236
Static keyword V-906
  user functions IV-51, IV-105, IV-244
stationery
  experiment files II-21
  notebook files III-30
statistics III-384–III-394
  classification V-472
  clustering V-472
  complementary error function V-198
  complex III-124
  confidence intervals V-1006
  curve fitting III-222–III-227, V-1006
  error function V-197, V-463
  functions V-10
    binomialln V-52
    binomialNoise V-52
    enoise V-195
    erf V-197
    erfc V-198
    erfcw V-198
    expnoise V-213
    faverage V-218
    faverageXY V-218
    gamma V-288
    gammaNoise V-289
    gammInc V-288
    gammln V-289
    gammp V-289
    gammq V-290
    GeometricMean V-295
    gnoise V-323
    hyperGNoise V-354
    inverseErf V-463
    inverseErfc V-463
    logNormalNoise V-521, V-523
    max V-586
    mean V-586
    median V-586
    min V-589
    norm V-692
    poissonNoise V-749
    StatsBetaCDF V-911
    StatsBetaPDF V-911
    StatsBinomialCDF V-912
    StatsBinomialPDF V-912
    StatsCauchyCDF V-912
    StatsCauchyPDF V-913
    StatsChiCDF V-913
    StatsChiPDF V-913
    StatsCMSSDCDF V-922
    StatsCorrelation V-925
    StatsDExpCDF V-926
    StatsDExpPdf V-926
    StatsErlangCDF V-928
    StatsErlangPDF V-928
    StatsErrorPDF V-928
    StatsEValueCDF V-929
    StatsEValuePDF V-929
    StatsExpCDF V-929
    StatsExpPDF V-929
    StatsFCDF V-930
    StatsFPDF V-930
    StatsFriedmanCDF V-930
    StatsGammaCDF V-932
    StatsGammaPDF V-933
    StatsGeometricCDF V-933
    StatsGeometricPDF V-933
    StatsGEVCDF V-934
    StatsGEVPDF V-934
    StatsHyperGCDF V-935
    StatsHyperGPDF V-935
    StatsInvBetaCDF V-936
    StatsInvBinomialCDF V-936
    StatsInvCauchyCDF V-936
    StatsInvChiCDF V-936
    StatsInvCMSSDCDF V-937
    StatsInvDExpCdf V-937
    StatsInvEValueCDF V-937
    StatsInvExpCDF V-938
    StatsInvFCDF V-938
    StatsInvFriedmanCDF V-938
    StatsInvGammaCDF V-939
    StatsInvGeometricCDF V-939
    StatsInvKuiperCDF V-939
    StatsInvLogisticCDF V-939
    StatsInvLogNormalCDF V-940
    StatsInvMaxwellCDF V-940
    StatsInvMooreCDF V-940
    StatsInvNBinomialCDF V-940
    StatsInvNCChiCDF V-940
    StatsInvNCFCDF V-941
    StatsInvNormalCDF V-941
    StatsInvParetoCDF V-941
    StatsInvPoissonCDF V-941
    StatsInvPowerCDF V-941
    StatsInvQCDF V-942
    StatsInvQpCDF V-942
    StatsInvRayleighCdf V-942
    StatsInvRectangularCDF V-943
    StatsInvSpearmanCDF V-943
    StatsInvStudentCDF V-943
    StatsInvTopDownCDF V-943
    StatsInvTriangularCDF V-943
    StatsInvUSquaredCDF V-944

StatsInvVonMisesCDF V-944
StatsInvWeibullCDF V-945
StatsKuiperCDF V-949
StatsLogisticCDF V-955
StatsLogisticPDF V-955–V-956
StatsLogNormalCDF V-956
StatsMaxwellCDF V-956
StatsMaxwellPDF V-957
StatsMedian V-957
StatsMooreCDF V-957
StatsNBinomialCDF V-959
StatsNBinomialPDF V-959
StatsNCChiCDF V-960
StatsNCChiPDF V-960
StatsNCFCDF V-960
StatsNCFPDF V-960
StatsNCTCDF V-961
StatsNCTPDF V-961
StatsNormalCDF V-961
StatsNormalPDF V-962
StatsParetoCDF V-965
StatsParetoPDF V-965
StatsPermute V-965
StatsPoissonCDF V-966
StatsPoissonPDF V-966
StatsPowerCDF V-966
StatsPowerNoise V-967
StatsPowerPDF V-967
StatsQCDF V-967–V-968
StatsRayleighCDF V-971
StatsRayleighPDF V-971
StatsRectangularCDF V-971
StatsRectangularPDF V-971
StatsRunsCDF V-974
StatsSpearmanRhoCDF V-977
StatsStudentCDF V-979
StatsStudentPDF V-979
StatsTopDownCDF V-980
StatsTriangularCDF V-980
StatsTriangularPDF V-980
StatsTrimmedMean V-981
StatsUSquaredCDF V-984
StatsVonMisesCDF V-987
StatsVonMisesNoise V-987
StatsVonMisesPDF V-987
StatsWaldCDF V-988
StatsWaldPDF V-988
StatsWeibullCDF V-990
StatsWeibullPDF V-990
StudentA V-1006
StudentT V-1006
sum V-1007
variance V-1066
WaveMax V-1077
WaveMin V-1078
WaveMinAndMax V-1078
wnoise V-1113
incomplete gamma function V-289–V-290
k-means clustering V-472
maximum V-1091
minimum V-1091
of complex waves V-1082
of matrices
maximum chunk location V-1085
maximum column location V-1084
maximum layer location V-1085
maximum row location V-1084
minimum chunk location V-1085
minimum column location V-1084
minimum layer location V-1085
minimum row location V-1084
of waves III-122–III-124, V-1082
average deviation V-1084
average value V-1084
dispersion V-1084
geometric mean V-295
kurtosis V-1084
maximum V-1077, V-1084, V-1091
maximum location V-1084
mean V-586
median V-586
min and max V-1078
minimum V-1078, V-1084, V-1091
minimum location V-1084
number of points V-1084
RMS V-1084
root mean squared V-1084
skewness V-1084
standard deviation V-1084
standard error of mean V-1084
sum V-1007
sum of values V-1084
variance V-1066, V-1084
operations V-4
EdgeStats V-190
Histogram V-349
ImageGLCM V-375
ImageHistModification V-378
ImageHistogram V-379
ImageStats V-414
JointHistogram V-466
PulseStats V-783
SetRandomSeed V-853
StatsAngularDistanceTest V-906
StatsANOVA1Test V-907
StatsANOVA2NRTest V-908
StatsANOVA2RMTest V-909
StatsANOVA2Test V-910
StatsChiTest V-913
StatsCircularCorrelationTest V-914
StatsCircularMeans V-916
StatsCircularMoments V-918

StatsCircularTwoSampleTest V-921
StatsCochranTest V-923
StatsContingencyTable V-924
StatsDExpPdf V-926
StatsDunnettTest V-927
StatsFriedmanTest V-931
StatsFTest V-931
StatsHodgesAjneTest V-933
StatsJBTest V-945, V-948
StatsKDE V-946
StatsKendallTauTest V-947
StatsKWTest V-949
StatsLinearCorrelationTest V-950
StatsLinearRegression V-952
StatsMultiCorrelationTest V-957
StatsNPMCTest V-962
StatsNPNominalSRTest V-964
statsQuantiles V-968
StatsRankCorrelationTest V-970
StatsResample V-972
StatsSample V-974
StatsScheffeTest V-975
StatsShapiroWilkTest V-976
StatsSignTest V-977
StatsSRTest V-977
StatsTTest V-981
StatsTukeyTest V-983
StatsVariancesTest V-985
StatsWatsonUSquaredTest V-988
StatsWatsonWilliamsTest V-989
StatsWheelerWatsonTest V-990
StatsWilcoxonRankTest V-991
StatsWRCorrelationTest V-993
WaveMeanStdv V-1077
WaveStats V-1082
random V-323
random numbers V-195
references III-393
special functions V-288–V-290
Student’s T distribution V-1006
StatsAngularDistanceTest operation V-906
StatsANOVA1Test operation V-907
StatsANOVA2NRTest operation V-908
StatsANOVA2RMTest operation V-909
StatsANOVA2Test operation V-910
StatsBetaCDF function V-911
StatsBetaPDF function V-911
StatsBinomialCDF function V-912
StatsBinomialPDF function V-912
StatsCauchyCDF function V-912
StatsCauchyPDF function V-913
StatsChiCDF function V-913
StatsChiPDF function V-913
StatsChiTest operation V-913
StatsCircularCorrelationTest operation V-914
StatsCircularMeans operation V-916
StatsCircularMoments operation V-918
StatsCircularTwoSampleTest operation V-921
StatsCMSSDCDF function V-922
StatsCochranTest operation V-923
StatsContingencyTable operation V-924
StatsCorrelation function V-925
StatsDExpCDF function V-926
StatsDExpPdf function V-926
StatsDIPTest operation V-926
StatsDunnettTest operation V-927
StatsErlangCDF function V-928
StatsErlangPDF function V-928
StatsErrorPDF function V-928
StatsEValueCDF function V-929
StatsEValuePDF function V-929
StatsExpCDF function V-929
StatsExpPDF function V-929
StatsFCDF function V-930
StatsFPDF function V-930
StatsFriedmanCDF function V-930
StatsFriedmanTest operation V-931
StatsFTest operation V-931
StatsGammaCDF function V-932
StatsGammaPDF function V-933
StatsGeometricCDF function V-933
StatsGeometricPDF function V-933
StatsGEVCDF function V-934
StatsGEVPDF function V-934
StatsHodgesAjneTest operation V-933
StatsHyperGCDF function V-935
StatsHyperGPDF function V-935
StatsInvBetaCDF function V-936
StatsInvBinomialCDF function V-936
StatsInvCauchyCDF function V-936
StatsInvChiCDF function V-936
StatsInvCMSSDCDF function V-937
StatsInvDExpCdf function V-937
StatsInvEValueCDF function V-937
StatsInvExpCDF function V-938
StatsInvFCDF function V-938
StatsInvFriedmanCDF function V-938
StatsInvGammaCDF function V-939
StatsInvGeometricCDF function V-939
StatsInvKuiperCDF function V-939
StatsInvLogisticCDF function V-939
StatsInvLogNormalCDF function V-940
StatsInvMaxwellCDF function V-940
StatsInvMooreCDF function V-940
StatsInvNBinomialCDF function V-940
StatsInvNCChiCDF function V-940
StatsInvNCFCDF function V-941
StatsInvNormalCDF function V-941
StatsInvParetoCDF function V-941
StatsInvPoissonCDF function V-941
StatsInvPowerCDF function V-941
StatsInvQCDF function V-942

StatsInvQpCDF function V-942
StatsInvRayleighCdf function V-942
StatsInvRectangularCDF function V-943
StatsInvSpearmanCDF function V-943
StatsInvStudentCDF function V-943
StatsInvTopDownCDF function V-943
StatsInvTriangularCDF function V-943
StatsInvUSquaredCDF function V-944
StatsInvVonMisesCDF function V-944
StatsInvWeibullCDF function V-945
StatsJBTest operation V-945, V-948
StatsKDE operation V-946
  KDE V-946
  kernel density estimate V-946
StatsKendallTauTest operation V-947
StatsKuiperCDF function V-949
StatsKWTest operation V-949
StatsLinearCorrelationTest operation V-950
StatsLinearRegression operation V-952
StatsLogisticCDF function V-955
StatsLogisticPDF function V-955–V-956
StatsLogNormalCDF function V-956
StatsMaxwellCDF function V-956
StatsMaxwellPDF function V-957
StatsMedian function V-957
StatsMooreCDF function V-957
StatsMultiCorrelationTest operation V-957
StatsNBinomialCDF function V-959
StatsNBinomialPDF function V-959
StatsNCChiCDF function V-960
StatsNCChiPDF function V-960
StatsNCFCDF function V-960
StatsNCFPDF function V-960
StatsNCTCDF function V-961
StatsNCTPDF function V-961
StatsNormalCDF function V-961
StatsNormalPDF function V-962
StatsNPMCTest operation V-962
StatsNPNominalSRTest operation V-964
StatsParetoCDF function V-965
StatsParetoPDF function V-965
StatsPermute function V-965
StatsPoissonCDF function V-966
StatsPoissonPDF function V-966
StatsPowerCDF function V-966
StatsPowerNoise function V-967
StatsPowerPDF function V-967
StatsQCDF function V-967–V-968
statsQuantiles operation V-968
StatsRankCorrelationTest operation V-970
StatsRayleighCDF function V-971
StatsRayleighPDF function V-971
StatsRectangularCDF function V-971
StatsRectangularPDF function V-971
StatsResample operation V-972
StatsRunsCDF function V-974
StatsSample operation V-974
StatsScheffeTest operation V-975
StatsShapiroWilkTest operation V-976
StatsSignTest operation V-977
StatsSpearmanRhoCDF function V-977
StatsSRTest operation V-977
StatsStudentCDF function V-979
StatsStudentPDF function V-979
StatsTopDownCDF function V-980
StatsTriangularCDF function V-980
StatsTriangularPDF function V-980
StatsTrimmedMean function V-981
StatsTTest operation V-981
StatsTukeyTest operation V-983
StatsUSquaredCDF function V-984
StatsVariancesTest operation V-985
StatsVonMisesCDF function V-987
StatsVonMisesNoise function V-987
StatsVonMisesPDF function V-987
StatsWaldCDF function V-988
StatsWaldPDF function V-988
StatsWatsonUSquaredTest operation V-988
StatsWatsonWilliamsTest operation V-989
StatsWeibullCDF function V-990
StatsWeibullPDF function V-990
StatsWheelerWatsonTest operation V-990
StatsWilcoxonRankTest operation V-991
StatsWRCorrelationTest operation V-993
status line help
  for controls III-436
stderr
  fprintf V-261
stdout
  fprintf V-261
stellar images
  deconvolution V-404
Step button in debugger IV-216
Step Into button in debugger IV-216
Step Out button in debugger IV-216
stereographic projection V-781
STFT operation V-994
Stop button in debugger IV-216
stop tool
  Gizmo II-415
StopMSTimer function V-995
stopping
  from control panel IV-160
  macros IV-124
  user functions IV-112
str2num function II-73, V-996
Strconstant keyword V-996
  in switch statements IV-44
  string declaration IV-51
strict wave reference mode IV-52, IV-75
string charsets
  HDF5 II-219

String declaration II-105, V-996
automatic creation of SVAR IV-70
in user functions IV-70
local variables in macros IV-119
local variables in user functions IV-34
not in a macro loop IV-119
string expressions
continuation characters IV-35, IV-113
string formats
HDF5 II-219
string inspector in debugger IV-222–IV-223, IV-225
string objects
Gizmo II-444, II-471
string padding
HDF5 II-219
string variables II-105–II-106, IV-13–IV-19
$ operator IV-18, IV-62
assignment IV-6, IV-17
char2num function V-60
CmpStr function V-73
constants IV-51
ConvertGlobalStringTextEncoding operation V-95
converting to names IV-18–IV-19, IV-62
converting to UTF-8 III-487
copying II-115
creating V-996
declaring II-105, V-996
default values V-1005
deleting II-105
dependency assignments IV-231
exists function V-207
expressions IV-13
global II-105, III-417, III-428, V-996
GrepList function V-333
GrepString function V-334
indexing IV-16
initializing II-105, V-996
in macros IV-119
in user functions IV-34
killing II-105, II-117, V-471
KillString operation V-471
length of IV-14, V-1001
list of
GetIndexedObjName function V-303
GetIndexedObjNameDFR function V-304
ListBox controls III-416, III-426
listing in data folder V-141
ListMatch function V-498
ListToTextWave function V-499
loading from Igor experiment file V-500–V-505
LowerStr function V-523
MoveString operation V-663
moving to data folder II-115
names
CheckName function V-68
naming rules III-501
num2char function V-713
num2istr function V-713
num2str function V-714
parsing V-902
pass-by-reference IV-59
ProcedureText function V-779
related functions V-11
renaming II-115, III-504, V-796
saving V-815
Set Variable controls III-417, III-428
SortList function V-886
SplitString operation V-900
sscanf operation V-902
static IV-51
Static keyword V-906
str2num function V-996
Strconstant V-996
StringByKey function V-997
StringFromList function V-998
StringList function V-999
stringmatch function V-1000
strlen function V-1001
strsearch function V-1001
StrVarOrDefault function V-1005
substitution IV-18–IV-19
precedence IV-19
text encodings III-478
text waves IV-14
UniqueName function V-1048
string-to-number conversion V-996
StringByKey function IV-173, V-997
stringCRC function V-998
StringFromList function IV-172, V-998
example IV-198
load waves (example) II-171–II-172
StringList function V-999
stringmatch function V-1000
strings IV-13–IV-19
$ operator IV-18, IV-62
and sprintf IV-260
as lists IV-172
assignment IV-6, IV-17
binary data in strings IV-175, V-1000, V-1071
char2num function V-60
character-by-character operations IV-173
characters versus bytes III-483
CmpStr function V-73
command in string: Execute operation V-204
concatenation IV-6, IV-8
constants IV-51
conversion to upper case V-1053
converting to names IV-18–IV-19, IV-62
declaring II-105
escape characters IV-14
escape sequences IV-14

exists function V-207
expressions IV-13
FontSizeHeight function V-257
FontSizeStringWidth function V-258
Grep operation V-327
GrepList function V-333
GrepString function V-334
in user functions IV-31
indexing IV-16
initializing
  in macros IV-119
  in user functions IV-34
keyword-value packed strings II-84, IV-173
length of IV-14, V-1001
ListBox controls III-416, III-426
ListToTextWave function V-499
LowerStr function V-523
naming rules III-501
null IV-14, IV-16
num2char function V-713
num2istr function V-713
num2str function V-714
PadString function V-732
parsing V-902
pass-by-reference IV-59
ProcedureText function V-779
processing lists of waves IV-198–IV-201
related functions V-11
RemoveEnding function V-792
ReplaceString function V-799
replacing items V-799
ReplicateString function V-802
searching V-327, V-333–V-334, V-1001
SelectString function V-834
SortList function V-886
SplitString operation V-900
splitting V-900
sprintf operation V-902
Static keyword V-906
str2num function V-996
StringByKey function V-997
StringFromList function V-998
StringList function V-999
stringmatch function V-1000
StringToUnsignedByteWave function V-1000
strlen function V-1001
strsearch function V-1001
substitution IV-18–IV-19, IV-62
  in user functions IV-83
  precedence IV-19
substrings IV-16
text encodings III-480
text waves IV-14
TrimString function V-1047
Unicode III-480
Unicode escape sequences IV-15
UnPadString function V-1050
UTF-16 literals IV-12, IV-14
UTF-8 III-480
utility functions IV-19
WaveDataToString function V-1071
StringToUnsignedByteWave function V-1000
strlen function V-1001
  characters versus bytes III-483
StrSearch function V-1001
strswitch statements IV-43, V-1002
  default keyword V-146
STRUCT keyword V-1002
StructFill operation V-1002
  example IV-104
StructGet operation V-1003
  example IV-104
  use in multitasking IV-328
StructPut operation V-1004
  example IV-104
  use in multitasking IV-328
structure fit functions III-261–III-263
  examples III-262
  multivariate III-263
  syntax III-263
  WMFitInfoStruct structure III-263
structures IV-99–IV-105
  accessing members IV-101
  action procedures IV-103
  alignment IV-100
  applications IV-103
  arrays of
    disallowed IV-101
  built-in IV-103
    HDF5DataInfo V-342
    HDF5DatatypeInfo V-342
    HDF5LinkInfoStruct V-343
    HDF5SaveDataHookStruct V-344
    Point V-749
    PointF V-749
    Rect V-787
    RectF V-787
    RGBAColor V-810
    RGBColor V-810
    WMAxisHookStruct V-609, V-1103
    WMBackgroundStruct V-1103
    WMButtonAction V-57, V-1104
    WMCheckboxAction V-66, V-1104
    WMCustomAction V-136
    WMCustomControAction V-137
    WMCustomControlAction V-1105
    WMDrawUserShapeStruct V-1106
    WMFitInfoStruct V-1107
    WMGizmoHookStruct II-473, V-1107
    WMListboxAction V-494, V-1108
    WMMarkerHookStruct IV-309, V-1109
    WMPopupAction V-1109

    - WMSetVariableAction V-858, V-1110
    - WMSliderAction V-876, V-1111
    - WMTabControlAction V-1013, V-1112
    - WMWinHookStruct IV-293, IV-297, V-1113
  - controls III-437, IV-103
  - debugging IV-221
  - declaring IV-100
  - defining IV-100
  - DFREF keyword IV-80
  - EndStructure keyword V-195
  - examples IV-102, IV-104
  - exporting V-220
    - into a string V-1004
    - into a wave V-1004
  - for axes
    - WMAxisHookStruct V-609
  - for controls
    - WMButtonAction V-57
    - WMCheckboxAction V-66
    - WMCustomAction V-136
    - WMCustomControAction V-137
    - WMListboxAction V-494
    - WMSetVariableAction V-858
    - WMSliderAction V-876
    - WMTabControlAction V-1013
  - global IV-104
  - importing V-219
    - from a string V-1003
    - from a wave V-1003
  - initialization IV-100
  - initializing V-1002
  - limitations IV-104
  - local references V-1002
  - member arrays IV-100
  - member types IV-100
  - Multithread keyword example IV-328
  - named window hooks
    - WMWinHookStruct IV-293
  - passing IV-101
  - passing to XOPs IV-104
  - Point structure V-749
  - PointF structure V-749
  - predefined IV-103
  - printing V-769–V-770
  - printing structure elements V-770
  - reading V-220
  - Rect structure V-787
  - RectF structure V-787
  - RGBAColor structure V-810
  - RGBColor structure V-810
  - static IV-100
  - STRUCT keyword V-1002
  - StructFill operation V-1002
  - StructGet operation V-1003
  - StructPut operation V-1004
  - structure fit functions III-261–III-263
    - examples III-262
    - multivariate III-263
    - syntax III-263
    - WMFitInfoStruct structure III-263
  - user data IV-104
  - using IV-101
  - windows IV-103
  - WMAxisHookStruct structure V-1103
  - WMBackgroundStruct structure V-1103
  - WMButtonAction structure V-1104
  - WMCheckboxAction structure V-1104
  - WMCustomControlAction structure V-1105
  - WMDrawUserShapeStruct structure V-1106
  - WMFitInfoStruct structure V-1107
  - WMGizmoHookStruct structure V-1107
  - WMListboxAction structure V-1108
  - WMMarkerHookStruct structure V-1109
  - WMPopupAction structure V-1109
  - WMSetVariableAction structure V-1110
  - WMSliderAction structure V-1111
  - WMTabControlAction structure V-1112
  - WMWinHookStruct structure V-1113
  - writing V-221
- StrVarOrDefault function V-1005
  - example IV-147
  - saving parameters IV-147
- StudentA function V-1006
- StudentT function V-1006
- Student’s T distribution V-1006
- style macros II-48, II-350–II-352, V-1101
  - adding annotations II-352
  - DoWindow operation V-171
  - for page layouts II-498
  - for tables II-272
  - limitations II-352
  - load waves II-173
  - object indexing IV-20
  - vs preferences II-350–II-351
- styles
  - plain notebooks III-8
- Submenu keyword IV-126, V-1006
- submenus
  - consolidating items IV-131
  - creating IV-127
- subminor ticks
  - in user tick waves II-314
  - log axes II-308–II-309
- subpatterns
  - in regular expressions IV-186
  - named IV-186
- subrange
  - display syntax II-321
  - displaying II-321
  - limitations II-322
- subranges
  - of text waves IV-17

of waves II-76
wave assignment statements II-76
subscripts
in an axis label II-318
in annotations III-37, III-54
in notebooks III-9
subset of data
using cursors in graphs II-320
subtraction IV-6
subtypes IV-118, IV-204–IV-205
ButtonControl IV-205
CameraWindow IV-205
CDFFunc IV-205
CheckBoxControl IV-205, V-66
CursorStyle IV-204
DrawUserShape IV-204
FitFunc IV-205
GizmoPlot IV-205
Graph IV-204
GraphMarquee IV-204, V-310
GraphStyle IV-204
GridStyle IV-204
Layout IV-205
LayoutMarquee IV-205, V-310
LayoutStyle IV-205
ListBoxControl IV-205
Panel IV-205
PopupMenuControl IV-205, V-763
SetVariableControl IV-205, V-858
SliderControl IV-205
TabControl IV-205
Table IV-205
TableStyle IV-205
user functions IV-32
window macros II-47
subwindows III-80–III-93
ChildWindowList function V-69
closing V-471
command syntax III-92
contextual menus III-82
converting to III-90
creating III-82
DefineGuide operation V-153
draw mode III-80, III-84
examples III-86
exterior III-443, V-318, V-687, V-689
examples V-689
frame style III-84
frames III-80
Gizmo II-469
graphs III-82
GuideInfo function V-336
GuideNameList function V-337
guides III-80, III-83, III-85
built-in III-83, V-153
creating V-153
deleting V-153
information about V-336
list of V-337
moving V-153
user-defined III-84–III-85
in control panels III-443
killing V-471
layout mode III-80, III-85
limitations III-82
listing V-69
MoveSubwindow operation V-663
notebook subwindows III-91
operate mode III-80, III-84
page layouts III-82, III-90, III-92
panels III-82
path to active V-318
positioning III-83
programming III-92
recreating notebooks V-711
renaming V-797
repositioning V-663
selection III-80
SetActiveSubwindow operation V-835
setting active V-835
sizing in commands III-92
tables III-82
terminology III-80
tutorial III-86
window hook functions IV-294
sum function V-1007
SumDimension operation V-1007
SumSeries operation V-1008
Sun Raster files
file info V-372
importing V-390
loading II-159
superscripts
in an axis label II-318
in annotations III-37, III-54
in notebooks III-9
surface plot
data format II-451
Gizmo tour II-410, II-412
Voronoi interpolation II-412
surface plots II-95
Gizmo II-459
SVAR keyword IV-65, V-1010
/Z flag IV-67
automatic creation IV-70
automatic creation with rtGlobals IV-70
checking in debugger IV-213
example IV-69
explicit creation IV-70
failures IV-67
SVAR_Exists function V-1011
use with $ IV-62

use with data folders IV-69
with data folders IV-249
SVAR_Exists IV-67, V-1011
SVG
graphics technology III-507
SVGs
exporting graphics III-97, III-104
switch statements IV-43, V-1011
break IV-44
Constant keyword V-85
default IV-44
default keyword V-146
Strconstant keyword V-996
Symbol font III-491–III-493
EPS III-493
Greek letters III-491
Igor6 compatibility III-492
PDFs III-493
Symbolic Path Status dialog II-23
symbolic paths II-22–II-24
automatically created II-23
creating manually II-22, V-690
example II-22, II-175
home II-23
Igor II-23
IgorUserFiles II-23
killing II-23
KillPath operation V-470
naming rules III-501
CheckName function V-68
NewPath operation V-690
Open operation V-717
PathInfo operation V-737
PathList function V-738
platform-related issues III-449
related functions V-12
related operations V-5
renaming III-504, V-797
Sync to Gizmo II-416
syntax coloring III-407
changing III-407
syntaxColorSelection keyword V-705
system locale III-461
system requirements III-513
system text encoding III-461
SZIP
HDF5 compression II-213

T

t function V-1011
in multidimensional waves II-92–II-93, II-95
T scaling
changing V-107
SetScale operation V-853
CopyScales operation V-107
SetScale operation V-853
T units
changing V-107, V-853
T values
t function V-1011
T_ReplacedDuplicates V-241–V-242
T_Tags V-391–V-392
tab characters
escape code for IV-14
in annotations III-39
in delimited text files II-131
in general text files II-138
in notebooks III-7–III-8
in strings IV-14
in tables II-258
in text data columns II-134
indentation conventions IV-26
plain notebooks III-8
saving text waves II-180
Tab key
in tables II-243
tab-delimited data
exporting from tables II-252
pasting II-239
saving waves as V-813
TabControl III-418, III-431
creating III-431
drawing order of V-148
TabControl operation V-1011
TabControl subtype IV-205
keyword V-1014
Table Date Format dialog II-239
Table Macros submenu II-48
Table menu II-234, II-236
Table pop-up menu II-236
table selection
Compose Expression dialog III-138
Table subtype IV-205
keyword V-1014
TableInfo function V-1015
TablePopup menu IV-138
tables II-234–II-274
3D copy and paste II-269
appending columns II-242
AppendToTable operation V-37
arrow keys II-243
can’t enter X values II-244
carriage return characters II-258
cell area II-236
cell ID area II-243
CheckDisplayed operation V-68
Choose Dimensions dialog II-263
chunks of data II-263
clearing values II-247
closing II-241
columns

autosizing
by double-clicking II-252
by menus II-253
limitations II-253
formats II-255
pasting II-249
headings by index V-1079
indices II-261
names II-241, II-261
positions II-252
styles II-272
titles II-254
widths II-252
comma as decimal separator II-255
compared to spreadsheets II-234
complex columns II-241
Compose Expression dialog III-138
control characters II-260
copy/paste multidimensional data II-265–II-270
copying II-247, II-252
create-paste of multidimensional data II-269
creating II-234, V-192–V-193
blank values II-247
multidimensional waves II-234
text waves II-234
waves by copy/paste II-240, II-269
while loading waves II-235
cutting II-248
rows or columns II-268
d suffix II-241
data values II-235
date format II-239
dates II-239, II-245, II-256
formats II-246
decimal numeric format II-255
decimal separators II-238
decimal symbols II-255
DelayUpdate box II-237, II-244
deleting data II-249
digits after decimal point II-255
dimension labels II-235, II-262
dynamic updating II-234, II-237
Edit operation V-192–V-193
Edit Text Cell dialog II-258
editing existing waves II-235
elapsed time II-256
elements column hiding V-649
empty table in new experiments II-21
Enter key II-243
entering dates II-246
entering new waves II-234, II-239
entering values II-244
escape sequences for invalid text II-260
exporting as graphics II-271, V-826
exporting data II-252
save table copy II-270
SaveTableCopy operation V-829
extracting a matrix from a 3D wave II-269
finding values II-249
fixed dimension II-264
fractional seconds II-256
free dimension II-264
general numeric format II-255
hex dump II-260
hexadecimal numbers II-257
hiding parts of II-237
horizontal dimension II-263
horizontal index row II-261
i suffix II-241
imag suffix II-241
in new experiments II-234
in notebooks III-18
index columns II-241, II-249
for multidimensional waves II-242
INFs II-247
insert-paste of multidimensional data II-266–II-267
inserting data II-244, II-249
insertion cell II-236, II-239, II-244
integer numeric format II-255
invalid text II-259
killing II-241
killing waves II-243
l suffix II-241
layers of data II-262
linefeed characters II-258
list of V-1098, V-1100
making a matrix from 1D waves II-267
making a new table II-45
example I-13
ModifyTable operation II-254, V-645–V-649
multidimensional waves II-261–II-270
names and titles II-45
NaNs II-247
new table II-45
example I-13
numeric formats II-255
numeric precision II-239, II-255
object references II-258, II-270
octal numbers II-257
page setup preference II-272
page setups II-271
parts of a table II-235
pasting
data II-239, II-248
index columns II-249
with column names II-240
point column II-252, II-262
hiding V-649
precision II-245
preferences II-272
printing II-271, V-779

printing selection II-271
properties V-1015
real columns II-241
real suffix II-241
recreating II-241
recreation macros V-1101
Redimension Waves dialog II-267, II-269
related operations V-1
relation to layouts I-3
relation to waves I-3, II-234
removing columns II-242, V-794
replace-paste of multidimensional data II-266
replacing values II-251
Return key II-243
row column hiding V-649
row indices II-261
scientific numeric format II-255
Select All II-243
Select All in Modify Columns dialog II-253
selecting cells II-243
selection
Compose Expression dialog III-138
selections V-315
setting to NaN II-247
shortcuts II-273
showing parts of II-237
showing X values II-235
significant digits II-255
slices of data II-262
special characters II-261
specifying column by index number V-645
style macros II-272
styles II-253–II-254
for multidimensional waves II-254
subwindows III-82
subwindows in layouts II-497
tab characters II-258
Tab key II-243
tab-delimited data II-239
Table pop-up menu II-236
TableInfo function V-1015
target cell ID II-236
target cells II-243
terminator characters II-258
text color V-648
text waves II-258
thousands separators II-238
time–of–day II-256
times II-239, II-256
troubleshooting II-240, II-244
Use Special Symbols for Control Characters II-261
vertical dimension II-263
views of multidimensional waves II-263
wave references II-270
WaveName function V-1079
waves in V-1080
whitespace characters II-258
window names II-240, V-1048
window titles II-235, II-240
wintype function V-1103
X columns II-241
x suffix II-241, II-254
X values II-235
y suffix II-241, II-254
TableStyle subtype IV-205
keyword V-1015
Tag operation V-1017–V-1022
/Q and contour plots II-379
tags III-43–III-47
anchors III-46
arrows III-45–III-46
as contour labels II-377–II-378
attached to wave III-44
attachment point III-38, III-44
changing III-45
axis escape codes III-57
definition III-34
deleting III-45, V-154
dynamic text III-38, III-55
lines III-45–III-46
lines too close to markers III-46
making a tag (example) I-17
modifying III-35
offscreen III-46
position III-46–III-47
offscreen III-47
standoff III-46
Tag operation V-1017–V-1022
TagVal function III-38, V-1022
TagWaveRef function III-38, V-1023
trace offsets III-38
wave name III-38
TagVal function III-38, V-1022
in contour labels II-378
TagWaveRef function III-38, IV-198, V-1023
tan function V-1023
tan of a wave V-1091
tanh function V-1023
tank icon (it’s a bulldozer, really) III-65
Targa files
exporting V-405
file info V-372
target cells in tables II-243
target window II-44
graphs II-277
TBB IV-323
MultiThreadControl operation V-674
TDMS files
loading II-128
saving II-178
technical support II-4

FAQ II-5
mailing list II-4
World Wide Web II-5
template files
formatted notebooks III-30
templates
experiments II-21
file extensions II-21
in procedure windows III-397
temporary files
saving experiments II-29
termination criteria in curve fitting III-180
terminator II-128
terminator character
in tables II-258
terminator characters
in delimited text files II-126
ternary graphs II-317
TestHDF5ExperimentCompression function II-17
tetrahedron objects
Gizmo II-445
TeX III-58
in annotations III-54
text
cross-platform issues III-453
DrawText operation V-178
in annotations III-35
retrieving from notebooks III-28
text color
in notebooks III-9
text encoding names III-490
text encodings III-460–III-493
Asian III-462
binary data III-475
byte order marks III-471
character encoding schemes III-462
code points III-462
ConvertGlobalStringTextEncoding operation V-95
converting to UTF8 III-485
ConvertTextEncoding operation V-98
data folders III-478
default file text encoding III-465
dimension labels III-473
experiment file text encoding III-465
formatted notebooks III-472
FReadLine operation V-261, V-263
Grep operation V-329
HDF5 II-221
history area III-470
inspecting in Data Browser III-473
invalid text III-464, III-470
invalid text in tables II-259
literal strings III-480
LoadWave operation II-129, II-149, III-475, V-509
MBCS III-462
misidentification III-463
mixed III-464
NewNotebook operation V-685
OpenNotebook operation V-723
OpenProc operation V-724
override experiment III-466
plain text files III-466
problems III-463
programming III-480
SaveNotebook operation V-823
saving text waves II-181
SetWaveTextEncoding operation V-859
string variables III-478
text encoding names III-490
text waves II-87
TextEncoding pragma IV-55
TextEncodingCode function V-1028
TextEncodingName function V-1028
Unicode conversion errors III-460
UTF-16 III-463
UTF-32 III-463
UTF-8 III-463
wave names III-473
wave units III-473
waves III-472–III-478
WaveTextEncoding function V-1086
western III-461, III-466
TEXT files
notebooks III-2
text files
converting to UTF-8 III-489
delimited text files II-129–II-137
delimiter characters II-126
description of II-126
fixed field text files II-137
FORTRAN files II-137
FReadLine operation V-261
general text files II-138–II-142
headers II-126
home text files II-56
Igor Text files II-150–II-154
shared text files II-56
terminator characters II-126
TextFile function V-1029
text formats
in notebooks III-9
text info variables III-51, III-55–III-56
examples
simple III-52
Text Markers dialog II-294
text markers in graphs II-293–II-294
text operations
Close operation V-71
DrawText operation V-178
FGetPos operation V-228

  fprintf operation IV-259–IV-261, V-260
  FReadLine operation V-261
  FSetPos operation V-264
  FStatus operation V-265
  IndexedFile function V-440
  Open operation IV-196, V-716–V-721
  Print operation V-769
  printf operation IV-259–IV-261, V-770–V-773
  sprintf operation IV-259–IV-261, V-902
  sscanf operation V-902
  TextFile function V-1029
  wfprintf operation IV-259–IV-261, V-1093
text size
  in notebooks III-9
text sizes
  in notebooks III-8–III-9
  in procedure windows III-406
text style
  in notebooks III-9
text styles
  in notebooks III-8
  in procedure windows III-406
text transfers IV-275
text waves
  accessing IV-68, IV-83, V-1069
  allowable content of II-86
  assignments IV-18, IV-68, IV-83, V-1069
  containing binary data III-475–III-477
  converting to/from numeric wave II-73
  creating IV-18
  creating in tables II-234
  in category plots II-356
  in Igor Text files II-154
  in tables II-258
  in user functions IV-68, IV-83, V-1069
  indexing IV-17
  labels in graphs II-293–II-294
  loading from delimited text files II-134
  making II-86–II-87
    examples II-68, II-86
  markers in graphs II-293–II-294
  preallocation V-526
  redimensioning II-72
  saving in text files II-180
  SetScale pitfall V-854
  speeding up V-526
  strings IV-14
  subranges IV-17–IV-18
  text encoding when saving II-181
  text encodings II-87
Text2Bezier operation V-1023
TextBox operation V-1024–V-1027
textboxes
  definition III-34
  deleting V-154
  exterior III-40
    affects graph plot area III-41
  Gizmo II-472
  in page layouts II-483
  interior III-40
  modifying III-35
  offset III-41
  position III-40–III-42
  TextBox operation V-1024–V-1027
TextEncoding keyword V-1027
TextEncoding pragma IV-55
TextEncodingCode function V-1028
TextEncodingName function V-1028
TextFile function V-1029
TextHistogram operation V-1029
texture objects II-467
textures
  creating V-421
  ImageGLCM V-375
thinning
  ImageSkeleton3D operation V-413
thousands separators
  in tick mark labels V-633
ThreadGroupCreate function V-1031
ThreadGroupGetDF function V-1031
ThreadGroupGetDFR IV-331
ThreadGroupGetDFR function V-1031
ThreadGroupPutDF IV-331
ThreadGroupPutDF operation V-1031
ThreadGroupRelease function V-1032
ThreadGroupWait function V-1032
ThreadProcessorCount function V-1033
ThreadReturnValue function V-1033
threads IV-329–IV-339
  aborting IV-337
  Igor Text files II-152
  MatrixOp operation III-148
  network operations IV-271
  worker functions IV-330
ThreadSafe functions IV-106, IV-329–IV-339
  debugging IV-225
  DisableThreadSafe IV-226
ThreadSafe keyword V-1033
ThreadStart operation V-1033
thumbnails
  in page layouts II-478
tick mark labels II-307
  axis scaling II-319
  color II-308, V-634
  engineering mode II-309
  exponential mode II-309
  exponential notation II-308
    forcing II-311
  font II-308
  Gizmo II-471
  hiding II-312
  log axes II-319

  - low trip, high trip II-308
  - minor tick labels on log axes II-315
  - no leading zero II-311
  - no trailing zeroes II-311
  - scaling II-308
  - scientific notation II-309
  - TickWavesFromAxis V-1034
  - tweaks II-311
  - units
    - suppress II-311
  - units in every label II-311
  - use thousands separator II-311, V-633
  - x10^n prevention II-319
  - zero as 0 II-311
- tick marks
  - auto ticks II-308
  - color II-308
  - computed manual ticks II-308
  - control of II-308
  - control of minor II-308
  - crossing II-309
  - hiding II-308
  - inside II-309
  - length II-309
  - location II-309
  - manual II-312–II-314
    - canonic tick II-312
    - date/time axes II-317
    - from waves II-313–II-314
    - tick increment II-312
  - minor ticks on log axes II-315
  - on mirror axes II-307
  - outside II-309
  - problems II-316
  - separation II-308
  - subminor on log axes II-308
  - thickness II-308–II-309
  - TickWavesFromAxis V-1034
  - types illustrated II-309
  - user ticks from waves II-308, II-313–II-314
    - specifying tick type II-314
- Tick Options tab II-310
- Ticks and Grids tab II-308–II-310
  - date/time items II-316
- ticks function IV-246, V-1034
- TickWavesFromAxis operation V-1034
- TIFF
  - CMYK III-99, III-105
- TIFF files
  - bilevel II-158
  - CMYK II-158
  - EXIF metadata V-395
  - exporting V-405, V-408
    - RGB waves V-408
    - RGBA waves V-408
    - stacks V-408
    - tags V-405
  - file info V-372
  - grayscale II-158
  - importing V-390
  - palette color II-158
  - RGB II-158
  - RGBA II-158
  - stacks II-158
- TIFFs
  - importing III-509
  - loading II-158
- Tile operation V-1035–V-1036
- Tile or Stack Windows dialog II-50
- TileWindows operation V-1036–V-1037
- tiling
  - in page layouts II-482, II-487, V-1035–V-1036
  - page layouts II-487–II-488
  - windows II-50, V-1036–V-1037
- tiltles
  - controls III-53
- time format in date/time axes II-316
- time frequency analysis III-280
- time function V-1037
- time–of–day II-256, II-315–II-317, V-833
  - loading format II-130
- timeouts
  - in network operations IV-270
  - URLRequest operation V-1056
- timers IV-246
  - MarkPerfTestTime operation V-529
  - StartMSTimer function V-906
  - StopMSTimer function V-995
  - ticks function V-1034
- times
  - date/time axes II-307, II-315–II-317
  - DateTime function V-145
  - elapsed time II-130, II-256, II-315–II-317, V-833
  - in delimited text files II-130
  - in Igor Text files II-151
  - in notebooks III-13, III-20
  - in tables II-239, II-256
    - formats II-256
  - pasting in tables II-249
  - related functions V-7
  - representation of II-315
  - Secs2Time function V-833
  - ticks function V-1034
  - time function V-1037
  - time–of–day II-130, II-256, II-315–II-317, V-833
  - units II-68–II-69
  - UTC V-145
  - wave precision II-69
- tips I-8
- TitleBox controls III-418, III-433
- TitleBox operation V-1038
- titles

  - column titles in tables II-254
  - TitleBox operation V-1038
- To Clip button I-7
- To Cmd Line button I-7
- ToCommandLine operation V-1040
- tool palette
  - Gizmo II-414
  - HideTools operation V-347
  - in page layouts II-479, II-481–II-484
  - ShowTools operation V-870
- toolbar I-7
- ToolsGrid operation V-1041
- tooltips I-8, II-3
  - custom V-868
  - for columns II-3
  - for commands IV-3
  - for traces II-3
  - hook functions V-868
  - HTML IV-311
  - ListBox operation V-493
- total least squares curve fitting III-237
- total pages
  - in notebooks III-13
- trace instance names II-278
- trace menus IV-137
- trace name parameters
  - user-defined IV-89
- trace names II-282
  - as parameters IV-88
  - box plots II-330
  - custom V-620
  - programming IV-87
  - programming example IV-89
  - violin plots II-330
- Trace Offset dialog II-302
- trace symbols
  - in axis labels II-318
- TraceFromPixel function V-1042
- TraceInfo function IV-173, V-1042
- TraceNameList function V-1044
  - example IV-197, IV-199
  - in dialogs IV-145
- TraceNameToWaveRef function IV-85, IV-198, V-1045
  - example IV-146, IV-197
- traces
  - alpha II-298
  - appearance in graphs II-290–II-306, II-348, II-364, II-380, V-613
  - AppendToGraph operation V-35
  - apppending II-280
  - color II-298, V-36, V-621
    - of each point V-621
  - cursors
    - z value from contour trace V-1120
  - customize at point II-306, V-625
- dashed lines II-294
- data folder II-112
- defined II-277
- display modes II-290, V-36
- drag and drop II-280
- drawing layer III-68
- editing by drawing (example) I-21–I-22, I-29
- end caps III-496, V-617
- f(z) II-298
- fill between II-297
- gaps II-284, II-303
- hiding II-295, II-303
- hiding portions trick II-347
- in graphs V-1044
- index in style macros II-351
- instance names III-38, IV-20, V-613
- joins III-496, V-616
- labels II-293–II-294
- line size II-294–II-295
- line style II-294, II-348, II-364, II-380
- list of V-1044, V-1075
- live II-347–II-348
- log colors II-298, V-616
- markers II-288, II-291, II-299
  - as f(z) II-298
  - coloring uniquely V-621
- miters III-496, V-616
- ModifyGraph operation V-613
- names as parameters IV-88
- next defined II-296
- offsetting in graphs II-302–II-303
  - example I-24
  - multipliers II-302
  - preventing II-302
  - undoing II-302
- opacity II-298
- preferences II-348, II-364, II-380
- programming IV-87
- programming example IV-89
- property from auxiliary wave II-298
- related functions V-8
- related operations V-1
- removing II-283, V-792
- reordering II-297
- ReorderTraces operation V-798
- ReplaceWave operation V-801
- replacing II-114, II-284
- selecting for modification II-290
- sending to back II-297
- Set as f(z) II-298
- SetDashPattern operation V-837
- setting all the same II-290
- shortcuts II-353–II-354
- symbol in an axis label II-318–II-319, III-37
- text markers II-293–II-294
- tooltips II-3

- trace instance names II-278
- trace names II-282, V-620
- TraceFromPixel function V-1042
- TraceInfo function V-1042
- TraceNameList function V-1044
- TraceNameToWaveRef function V-1045
- transparency II-298
- user data IV-89
- user-defined trace names IV-89
- wave reference from V-1045
- X wave from trace V-1119

transferring
- experiments II-24
- files III-448–III-449
  - FTP III-448, V-268, V-271
- files via FTP IV-272
- files via HTTP IV-276
- rulers III-12

translation
- Gizmo II-425

translucency III-498
- color table waves II-400
- Gizmo II-432
- image plots II-400–II-401

transparency
- annotations III-39
- color table waves II-400
- EPS III-96, III-102
- Gizmo II-432
- graphics technology III-507
- image plots II-400–II-401
- in Gizmo II-428
- in graph traces II-298
- troubleshooting in Gizmo II-470

transpose
- 4D wave V-427
- volumes V-427

transposition
- MatrixOp operation III-144

Transverse Mercator projection V-781

trapezoidal integration
- area function V-40
- areaXY function V-41
- faverage function V-218
- faverageXY function V-218
- Integrate operation V-446
- integrate1D operation V-448

tree icon (what it is) III-65

trend removal
- DSPDetrend operation V-182

trial exponent for axis label II-319

triangle objects
- Gizmo II-437

triangle window function III-278

Triangulate3d operation V-1046

triangulation
- convexHull operation V-101
- Delaunay V-605
- Triangulate3d operation V-1046

triangulaton
- ModifyContour operation V-602
- perturbation in contour plots V-605

trigonometric functions V-7
- acos V-19
- arbitrary precision V-27
- asin V-41
- atan V-42
- atan2 V-42
- cos V-109, V-1090
- cot V-110
- csc V-115
- csch V-115
- sawtooth V-831
- sec V-832
- sech V-832
- sin V-871
- tan V-1023, V-1091

trigonometric transforms
- MatrixOp V-576

TrimString function V-1047

triple-clicking
- on text III-408

triplet axes
- Gizmo II-450

triplet waves
- Gizmo II-406

troubleshooting II-4, IV-340–IV-341
- crash log file IV-341
- curve fitting III-265–III-267
- delimited text files II-137
- general text files II-141
- Gizmo II-469
- tables II-240, II-244

true and false IV-40

TrueType
- embedding in EPS III-99, III-105
- embedding in PDFs III-99
- exporting tables
  - Macintosh OS X III-100
  - Windows OS III-106

truncate to integer (trunc) function V-1047

try keyword V-1047

try-catch-endtry statements IV-49, V-1047
- catch keyword V-58
- endtry keyword V-195

Tukey
- box plots II-331

tutorial I-12

.txt file extension
- notebooks III-2

U

- uint64 declaration
  - local variables in user functions IV-33
  - variables in user functions IV-33
- uint64 keyword V-1048
- UltraHD displays III-507
  - resolution of panels III-456
- UNC paths III-451, V-734
- undefine statement V-18
- Unicode III-460, III-462–III-463
  - byte order marks III-471
  - canonical equivalence V-693
  - compatible characters V-693
  - decomposed characters V-693
  - normalization V-693
  - NormalizeUnicode function V-693
  - precomposed characters V-693
  - programming III-480
  - replacement character III-465
  - UTF-16 III-463
  - UTF-16 literals IV-12, IV-14
  - UTF-32 III-463
  - UTF-8 III-463
- Unicode conversion errors III-460
- UniqueName function V-1048
- units
  - dat II-69
    - precision II-69
  - of waves V-1093
- Universal Name Convention III-451
- Unix
  - paths III-452
  - shell scripts IV-264
    - example IV-264
- UNIX paths V-734
- unlocking
  - notebooks III-5
  - procedure files III-397
- unpacked experiments
  - adopting files II-25
  - adopting files programmatically V-23
  - autosave II-37, II-41
  - experiment file II-17
  - LoadData operation II-156
- UnPadString function V-1050
- UnsetEnvironmentVariable function V-1050
- unsigned integers V-1048
- Unwrap operation III-316, V-1050
- unwrapping phase III-274, V-433
- UnzipFile operation V-1051
- Update All Now III-14, III-18
- Update Selection Now III-14, III-18
- updates
  - during macros IV-123
  - during user functions IV-112
  - for Igor II-4
- updating
  - annotations III-38–III-39
  - controls
    - ControlUpdate operation V-94
  - ControlUpdate operation III-436
  - DelayUpdate operation V-153
  - DoUpdate operation V-168
  - during procedures V-153, V-168, V-739, V-809
  - Igor II-4
  - PauseUpdate operation V-739
  - pictures in notebooks III-18
  - pop-up menu controls III-428
  - ReplaceWave operation V-801
  - ResumeUpdate operation V-809
  - special characters in notebooks III-14, V-709
  - traces in graph II-114
- upgrading
  - Igor II-4
- uploading
  - directories via FTP IV-274
  - files via FTP IV-274
- upper-case string conversion V-1053
- UpperStr function V-1053
- URLDecode function V-1053
- URLEncode function IV-268, V-1053
- URLRequest operation V-1053
  - authentication V-1054
  - FILE scheme V-1058
  - headers V-1054
  - HTTP POST method V-1058
  - passwords V-1054
  - percent encoding V-1058
  - POST method V-1058
  - proxy servers V-1056
  - redirects V-1055
  - schemes V-1054
  - timeouts V-1056
  - usernames V-1054
- URLs IV-267
  - in help files IV-257
  - percent encoding IV-268
  - reserved characters IV-268
  - URLDecode function V-1053
  - URLEncode function V-1053
  - URLRequest operation V-1053
- Use Global Attributes
  - Gizmo II-419
- Use Selection for Find II-54
  - in procedure windows III-404
- Use Special Symbols for Control Characters II-261
- use table formatting checkbox II-179
- UseNewDiskHeaderBytes II-35
- UseNewDiskHeaderCompress II-35
- user data
  - controls III-440

Buttons V-57
CheckBoxes V-66, V-136
examples III-440
GroupBox V-336
ListBox V-492
pop-up menus V-763
SetVariable V-857
Sliders V-876
TabControl V-1013
GetUserData function V-316
storing global variables IV-104
traces IV-89
windows IV-293, V-868
examples IV-293
getting V-316
user files
path to folder V-895
user functions IV-30–IV-114
$ operator IV-62
& operator IV-59
aborting IV-49, IV-51, IV-112, V-18–V-19, V-1047
accessing global variables IV-70
accessing waves III-170–III-174, IV-82–IV-85
automatic local variables IV-61
bitwise and logical operators IV-41
body code IV-35
break in switch IV-44
break statements IV-48
calling a macro IV-202
calling an external operation IV-202
case statements IV-43
coercion of data types IV-111
compared to macros IV-116
comparisons IV-41
compile time vs runtime IV-64
complex IV-31
complex return value IV-31
complex wave references IV-64
complex waves IV-68, IV-83
conditional compilation IV-108
conditional statements IV-40
constants IV-51
continuation characters IV-35, IV-113
continue statements IV-48
control panels IV-158
creating global variables in data folders IV-249
curve fitting to III-190, V-273
data folder reference result IV-31
data folder references IV-78–IV-82
data folders II-111
debugging
debugger IV-212–IV-227
GetRTStackInfo function V-314
runtime stack information V-314
default values V-715, V-1005
dependency assignment IV-233, V-847
dialogs IV-144
direct reference to globals IV-114
do-while loops IV-45
DoUpdate IV-112
Duplicate operation V-186
error handling IV-111
errors IV-110
examples III-171–III-175
FindSegmentMeans III-174
LogRatio III-172
modifying a wave III-171
RemoveOutliers III-171
WavesAverage III-173
WavesMax III-173
WaveSum III-171
Execute operation IV-202
exists function V-207
FitFunc keyword V-255
flags IV-31
flow control IV-40
for loops IV-46
FuncRefInfo function V-278
Function Execution Error dialog IV-110
function references IV-107–IV-108
FunctionInfo function V-279
FunctionList function V-284
GetRTStackInfo function V-314
global variables IV-65–IV-71
help for IV-259
hook functions IV-280–IV-292
if-else-endif IV-40
if-elseif-endif IV-40
in curve fitting
details III-250–III-260
IndependentModule keyword V-437
information about V-278–V-279
initialization of local variables IV-34
inline parameter lists IV-33
invoking operations IV-111
liberal names IV-68
local variable declaration IV-34
local variables IV-61
loops IV-45
multiple return statements IV-36
MultiThread keyword V-673
multivariate curve fitting to V-276
names IV-31
operation queue IV-278
optional parameters IV-33, IV-60, V-733
overriding IV-106, V-731
parameter lists IV-32
pass-by-reference IV-59
path to V-287
pause for user IV-152
PauseForUser operation V-738
performance V-529

pop-up menus IV-145
precision of calculations IV-31
predefined symbols IV-110
preemptive multitasking IV-106, IV-329–IV-339
preferences IV-203
ProcedureText function V-779
processing waves III-171–III-175
programming with liberal names IV-168–IV-169
Prompt reuse IV-147
Prompt statements IV-144, IV-147
prototype functions IV-107
range-based for loops IV-47
retrieve code
  ProcedureText function V-779
return statements IV-35
returning waves III-170
reusing prompts IV-147
rtFunctionErrors pragma IV-55
rtGlobals=0 IV-114
runtime stack information V-314
S_ variables IV-61
scope of variables II-103–II-105, IV-34, IV-61
simple input dialog IV-144
static IV-105, IV-244
  ModuleName keyword IV-54, V-658
Static keyword V-906
string IV-31
string return value IV-31
subtypes IV-32
switch statements IV-43
syntax for curve fitting III-251
syntax of IV-31
text waves IV-68, IV-83
ThreadSafe IV-106, IV-329–IV-339
ThreadSafe keyword V-1033
timing V-529
updates IV-112
V_ variables IV-61
variables
  declaration IV-34
  global IV-65–IV-71
  local IV-61
  scope II-103–II-105, IV-34, IV-61
wave access IV-65–IV-71
wave as parameter IV-82
wave parameters IV-60
wave reference functions IV-85
wave reference result IV-31
wave references IV-71–IV-78
waves as parameters III-170, V-675
while loops IV-46
user interface I-6
User Procedures IV-25
User Procedures folder II-32, III-400–III-401, IV-25, IV-166
user shapes drawing tool III-65
user-defined buttons
  Data Browser II-120
user-defined drawing shapes V-179
user-defined menus
  contextual menu example IV-162
user-defined trace names IV-89
UserAxes drawing layer III-68–III-69
UserBack drawing layer III-68
  in page layouts II-479–II-480
UserFront drawing layer III-68–III-69
  in page layouts II-479–II-480
usernames
  in URLs IV-267–IV-268
  URLRequest operation V-1054
UTC V-145
  GetFileFolderInfo V-300
  SetFileFolderInfo V-846
UTF-16 III-463
  Unicode escape sequences IV-15
UTF-16 literals IV-12, IV-14
UTF-32 III-463
  Unicode escape sequences IV-15
UTF-8 III-460–III-461, III-463, III-481
  character-by-character operations IV-173
  Convert to UTF-8 Text Encoding dialog III-485
  converting notebooks III-489
  converting procedure files III-489
  converting string variables III-487
  converting text files III-489
  converting to III-485
  converting waves III-486
  history area III-489
  invalid text III-470
  plain text files III-470
  programming III-480
  TextEncoding pragma IV-56
  Unicode escape sequences IV-15
utility procedures IV-25
  writing IV-167

V

V_ variables II-103
  curve fitting III-232–III-236
  in macros IV-123
  in user functions IV-61
V_AbortCode IV-49, V-1048
V_adev III-123, V-415, V-973, V-1084
V_Alpha V-70, V-91, V-93, V-307, V-318–V-319, V-759, V-766
V_alreadyOpen V-722
V_ascent V-588
V_autoMultiThread V-674
V_avg III-122, V-415, V-973, V-1084

V_Blue V-70, V-91, V-93, V-307, V-318–V-319, V-759, V-766
V_bottom V-310, V-319, V-321
V_bytesRead V-506
V_CenterX V-53
V_CenterY V-53
V_CenterZ V-53
V_chisq III-213, III-233, III-235
V_correlation V-416
V_creationDate V-301
V_debugDangerously IV-225, IV-227
V_debugOnAbort V-146
V_debugOnError V-146
V_defaultTextEncoding V-151
V_denominator V-786
V_descent V-588
V_disable V-90
V_EdgeAmp4_0 V-191
V_EdgeDLoc3_1 V-191
V_EdgeLoc1 V-191
V_EdgeLoc2 V-191
V_EdgeLoc3 V-191
V_EdgeLvl0 V-191
V_EdgeLvl1 V-191
V_EdgeLvl2 V-191
V_EdgeLvl3 V-191
V_EdgeLvl4 V-191
V_EdgeSlope3_1 V-191
V_enable V-146
V_endChunk III-124, V-1085
V_endCol III-124, V-316, V-1085
V_endLayer III-124, V-1085
V_endParagraph V-316
V_endPos V-172, V-316
V_endRow III-123, V-316, V-1085
V_Error V-891, V-1003
V_FIFOChunks V-229
V_FIFOnchans V-229
V_FIFORunning V-229
V_filePos V-228, V-265
V_FitError III-233, III-235
V_FitIterStart III-233, III-236
V_FitMaxIters III-234
V_FitNumIters III-234
V_FitOptions III-213, III-233–III-234
V_FitQuitReason III-233, III-235, V-182
V_FitTol III-233
V_Flag II-120, V-29, V-208–V-209, V-246, V-255, V-790, V-1003, V-1088
V_flag V-21, V-45, V-54, V-68, V-70–V-71, V-89–V-90, V-105–V-106, V-112, V-140, V-150, V-154–V-156, V-163, V-165–V-167, V-169, V-172, V-182, V-184, V-189, V-191, V-204, V-228–V-229, V-235, V-243, V-245, V-247, V-251, V-265, V-268, V-271, V-294–V-295, V-301, V-306, V-310, V-312, V-316, V-321, V-394, V-414, V-464, V-487, V-506–V-507, V-515, V-518, V-540–V-541, V-543–V-545, V-579–V-580, V-592, V-658, V-660, V-662, V-684, V-696, V-702–V-703, V-706, V-709, V-720, V-722, V-730, V-737, V-744–V-745, V-747, V-757–V-759, V-768, V-784, V-818, V-821–V-822, V-826, V-828, V-831, V-847, V-887–V-889, V-891, V-900, V-907–V-910, V-914, V-917, V-919, V-923–V-924, V-927, V-931–V-932, V-934, V-945, V-947–V-949, V-958, V-964, V-976, V-982, V-984–V-985, V-989, V-991–V-992, V-995, V-1052, V-1057, V-1117
  loading waves (example) II-171
  LoadWave operation II-170
V_fontSize V-588
V_fontStyle V-588
V_Green V-70, V-91, V-93, V-307, V-318–V-319, V-759, V-766
V_Height V-90, V-93
V_horizScroll V-91
V_IQR V-973
V_isAliasShortcut V-301
V_isFile V-301
V_isFolder V-301
V_isInvisible V-166, V-301
V_isReadOnly V-301
V_isStationery V-301
V_iterations V-879
V_kind V-758
V_kurt III-123, V-415, V-973, V-1084
V_LeadingEdgeLoc V-247
V_left V-90, V-310, V-312, V-319, V-321
V_LevelsFound V-245
V_LevelX V-243
V_logEOF V-265, V-302
V_LUPolarity V-547
V_marquee V-311
V_max III-123, V-260, V-295, V-415, V-730–V-731, V-973, V-1084
V_maxChunkLoc III-123, V-1085
V_maxColLoc III-123, V-415, V-1084
V_maxLayerLoc III-123, V-1085
V_maxloc III-123, V-730, V-1084
V_maxRowLoc III-123, V-415, V-1084
V_Median V-973
V_min III-123, V-295, V-415, V-547, V-730–V-731, V-973, V-1084
V_minChunkLoc III-123, V-1085
V_minColLoc III-123, V-415, V-1084
V_minLayerLoc III-123, V-1085
V_minloc III-123, V-730, V-1084
V_minRowLoc III-123, V-415, V-1084
V_modificationDate V-301
V_mouseX V-307
V_mouseY V-307

V_nextRun V-45
V_nextValue V-487
V_nheld III-233
V_no_MIME_TSV_Load IV-283
V_npnts III-122, III-213, V-415, V-740, V-1084
  curve fitting III-233
V_nterms III-233
V_numConversionsFailed V-98, V-864
V_numConversionsSkipped V-98, V-864
V_numConversionsSucceeded V-98, V-864
V_numerator V-786
V_numImages V-394
V_numINFs III-122, III-233, V-973, V-1084
V_numMarkedAsBinary V-860, V-864
V_numNaNs III-122, III-233, V-973, V-1084
V_NumParticles V-365
V_numRegions V-435
V_numResidues V-433, V-435
V_numRoots V-252
V_numWaves V-1088
V_numWavesAffected V-864
V_NVAR_SVAR_WAVE_Checking V-146
V_ODEFunctionCalls V-457
V_ODEMinStep V-457
V_ODEStepCompleted III-334, V-457
V_ODEStepSize V-457
V_ODETotalSteps V-457
V_OptNumFunctionCalls V-731
V_OptNumIters V-731
V_OptTermCode V-730
V_overrideExperiment V-151
V_PeakLoc V-247
V_peakP V-246
V_PeakVal V-247
V_PeakWidth V-247
V_peakX V-246
V_period V-45
V_Pr III-222, III-233
V_prob V-976
V_PulseAmp4_0 V-784
V_PulseLoc1 V-784
V_PulseLoc2 V-784
V_PulseLoc3 V-784
V_PulseLvl0 V-784
V_PulseLvl123 V-784
V_PulseLvl4 V-784
V_PulsePolarity V-784
V_PulseWidth2_1 V-784
V_PulseWidth3_1 V-784
V_PulseWidth3_2 V-784
V_q III-233, III-235
V_Q25 V-973
V_Q75 V-973
V_r2 III-221, III-233
V_Rab III-233
V_rab III-226, V-125
V_Radius V-53
V_Red V-70, V-91, V-93, V-307, V-318–V-319, V-759, V-766
V_responseCode V-1057
V_right V-310, V-319, V-321
V_rising V-243
V_rms III-123, V-415, V-973, V-1084
V_Root V-252
V_Root2 V-252
V_rowHeight V-91
V_SANumIncreases V-730
V_SANumReductions V-730
V_sdev III-123, V-415, V-973, V-1084
V_selCol V-91
V_sem III-123, V-1084
V_siga III-233
V_sigb III-233
V_skew III-123, V-415, V-973, V-1084
V_SoundInAGC V-887, V-889
V_SoundInChansAv V-889
V_SoundInGain V-888–V-889
V_SoundInSampSize V-889
V_startChunk III-124, V-1085
V_startCol III-123, V-316, V-1085
V_startLayer III-124, V-1085
V_startParagraph V-316
V_startPos V-172, V-254, V-316
V_startRow III-123, V-91, V-316, V-1085
V_statistic V-976
V_structSize V-506, V-826
V_subscriptExtraHeight V-588
V_Sum V-547, V-973
V_sum III-123, V-1084
V_superscriptExtraHeight V-588
V_SVConditionNumber V-585
V_tapValue V-487
V_TBBufZone III-39, IV-292, V-1025
V_threshold V-416
V_tol III-233, III-265
V_top V-90, V-310, V-312, V-319, V-321
V_TrailingEdgeLoc V-247
V_Value V-29
V_value V-90, V-150, V-252, V-254, V-306, V-318, V-427, V-436, V-491, V-745, V-1098
V_version V-302
V_vertScroll V-91
V_Width V-90
V_YatRoot V-252
V_YatRoot2 V-252
ValDisplay controls
  using III-418
ValDisplay operation III-433–III-435, IV-233, V-1060–V-1064
Value Display controls
  bar display III-419, III-434–III-435
  examples III-434–III-435

  - height III-434
  - LED display III-434
  - limits III-419, III-435
  - numeric readout III-434
  - programming III-433–III-435
  - title III-435
  - updating
    - problems III-442, V-94
  - widths explained III-433
- value waves II-77
- Variable declaration II-104, V-1065
  - automatic creation of NVAR IV-70
  - in user functions IV-70
  - local variables in macros IV-119
  - local variables in user functions IV-34
  - not in a macro loop IV-119
- variable-length datasets
  - HDF5 II-207
- variable-length strings
  - HDF5 II-219
- VariableList function V-1066
- variables II-102–II-106
  - complex II-104
  - default values V-715
  - exists function V-207
  - global II-102–II-105, II-111, III-417–III-418, III-428–III-429
    - creating II-102
    - in data folders II-108
    - in user functions IV-65–IV-71
    - numeric variables V-1065
    - runtime lookup IV-65–IV-67
    - string variables V-996
    - uses of II-102
    - using structures instead IV-103
    - versus local IV-61
  - in debugger IV-218, IV-220
  - list of
    - GetIndexedObjName function V-303
    - GetIndexedObjNameDFR function V-304
  - listing in data folder V-141
  - LoadData operation II-156
  - loading from Igor experiment file V-500–V-505
  - loading waves II-170
  - local II-105
    - in macros IV-119
    - in user functions IV-34
    - versus global IV-61
  - MoveVariable operation V-665
  - name conflicts III-505
  - names II-102
    - CheckName function V-68
  - naming rules III-501
  - numeric II-104–II-105
  - NVAR_Exists function V-716
  - pass-by-reference IV-59
  - references using $ IV-62
  - runtime lookup of globals IV-65–IV-67
  - saving V-815
  - saving procedure parameters IV-147
  - scope II-103–II-105, IV-34, IV-61, IV-119, IV-123
  - Static keyword V-906
  - string II-105–II-106
  - SVAR_Exists function V-1011
  - system II-102–II-103
  - user II-103
  - uses for global variables II-102
  - V_ variables II-103
  - veclen II-103
  - writing formatted data V-260
- variables file
  - in experiments II-19
- variance
  - of images V-415
  - of waves V-415, V-1084
- variance function V-1066
- VAX floating point II-167
- vcsr function V-1067
  - in wave assignments II-84
- VDT2 XOP IV-312
- veclen II-103
- vector plots II-294, V-614
  - wind barbs V-614
- vectors
  - cross product V-115
- version
  - IgorVersion function V-362
  - in include statement IV-166
  - of Igor V-359–V-360
  - of procedure file IV-166
  - pragma keyword V-1068
  - ProcedureVersion function V-780
- version pragma IV-54
- VertCrossing free axis II-278–II-279
- vertical dimension in tables II-263
- vertical offsets
  - in notebooks III-8–III-9
- viewing volume
  - Gizmo II-421
- viewport projection
  - Gizmo II-424
- violin plots II-330, II-337–II-342, II-345, V-649
  - appearance II-339
  - AppendViolinPlot operation V-37
  - asymmetric II-341
  - creating V-37
  - KDE II-337
  - markers V-651–V-652
  - per point settings II-343
  - trace names II-330
- virtual memory III-513
- VISA

data acquisition IV-312
VISA XOP IV-312
Visual C++ IV-208
Voigt
curve fitting function III-206
VoigtFunc function V-1068
VoigtPeak function V-1069
volumes
shrinkBox V-426
transpose volume V-427
volumetric plots II-95
Voronoi interpolation V-384, V-602
surface plot II-412
voronoi tesselation V-428
voxelgram plots
Gizmo II-463
voxels
shrinkBox V-426
VRAM II-407
Gizmo II-465

W

W_Abs V-1090
W_Acos V-1090
W_alphaValues V-537
W_AngleWave V-919
W_ANOVA1BnF V-907
W_ANOVA1Welch V-907
W_Asin V-1090
W_Atan V-1090
W_BackgroundCoeff V-403
W_Beam V-420
W_betaValues V-537
W_BoundaryIndex V-365
W_BoundaryX V-365
W_BoundaryY V-365
W_CConjugate V-1090
W_circularity V-365
W_CircularMeans V-916
W_CircularStats V-918
W_coef III-213
in curve fitting III-195–III-196, V-124, V-131
W_Compressed V-418
W_ContingencyTableResults V-924
W_COOColumns III-152
W_COORows III-152
W_COOValues III-152
W_Cos V-1090
W_Covar III-226
W_Cross V-115
W_CSCPointerB III-153
W_CSCRows III-153
W_CSCValues III-153
W_CSRColumns III-152
W_CSRPointerB III-152
W_CSRValues III-152
W_CumulativeVAR V-739
W_CWTScaling V-138–V-139
W_DebugTimerIDs V-529
W_DebugTimerVals V-529
W_DeCompressed V-419
W_Detrend V-182
W_Diagonal V-548
W_DWT V-190
W_Eigen V-740–V-741
W_eigenValues V-535–V-536, V-579
W_ExtractedCol V-420
W_ExtractedRow V-420
W_extremum III-347, V-731
W_FFT V-222
W_FindLevels V-244
W_fitConstants III-190, III-213
W_FitConstraint III-231, V-124, V-131
W_flipped V-1090
W_FPCenterIndex V-259
W_FPClusterIndex V-259
W_FuzzyClasses V-420
W_Hilbert V-348
W_Histogram V-349, V-351
W_HodgesAjne V-933
W_IE V-740–V-741
W_IEigenValues V-579
W_ImageHist V-379
W_ImageHistB V-379
W_ImageHistG V-379
W_ImageHistR V-379
W_ImageLineProfile V-389
W_ImageObjArea V-365
W_ImageObjPerimeter V-365
W_IND V-740–V-741
W_Index V-329
W_index V-328
W_IndexedValues V-421
W_inPoly V-248
W_InstantAmplitude V-445
W_InstantFrequency V-445
W_IntAvg V-363
W_Interpolated V-459
W_IntMax V-363
W_IntMin V-363
W_Inverse V-1090
W_InvW V-584
W_IPIV V-543
W_JackKnifeStats V-972
W_JBResults V-945
W_KMDispersion V-472
W_KMMembers V-472–V-473
W_KSResults V-948
W_KWTestResults V-907, V-949
W_LDiagonal V-548
W_LFSR V-487

W_LinearOrderStats V-919
W_LinearRegressionMC V-955
W_LineProfileDisplacement V-389
W_LineProfileStdv V-389
W_LineProfileX V-389
W_LineProfileY V-389
W_LombPeriodogram V-521
W_LombProb V-521
W_LUPermutation V-546
W_Magnitude V-1090
W_MagSqr V-1090
W_MatrixOpInfo V-535
W_MatrixRCONDE V-535
W_MatrixRCONDV V-535
W_max V-1091
W_MeanStdv V-1077
W_min V-1091
W_MultiTaperDF V-672
W_MultiTaperF V-672
W_MultiTaperPSD V-672
W_MultiThreadingArraySizes V-674–V-675
W_NNResults V-676
W_normalizedArea V-1091
W_ObjPerimeter V-364
W_OptGradient III-347, V-731
W_ParamConfidenceInterval V-132
W_PCAEV V-356
W_Periodogram V-182
W_Phase V-1091
W_PIV V-548
W_polyRoot V-252
W_PolyX V-172
W_PolyY V-172
W_PrimeFactors V-769
W_PSL V-740–V-741
W_QuantilesIndex V-968
W_QuantileValues V-968
W_reciprocalConditionE V-537
W_reciprocalConditionV V-537
W_rectangularity V-365
W_RegParams V-402
W_REigenValues V-579
W_ReplacedDuplicates V-241
W_Resampled V-972–V-973
W_RMS V-740–V-741
W_roi_to_1d V-425
W_Root V-252
W_RSD V-740–V-741
W_Scale V-530
W_Sgn V-1091
W_sigma III-213
  in curve fitting V-131
W_sigma wave in curve fitting III-195–III-196, III-222
W_Sin V-1091
W_SoundInRates V-889
W_SphericalInterpolation V-899
W_SpotX V-365
W_SpotY V-365
W_sqrt V-1091
W_SqrtN V-351
W_StatsChiTest V-913
W_StatsCircularTwoSamples V-914, V-921
W_StatsFTest V-931
W_StatsKendallTauTest V-947
W_StatsLinearCorrelationTest V-950
W_StatsLinearRegression V-952
W_StatsMultiCorrelationTest V-958–V-959
W_StatsNPSRTest V-964
W_StatsQuantiles V-969, V-973
W_StatsRankCorrelationTest V-970
W_StatsSRTest V-977
W_StatsTTest V-981
W_StatsVariancesTest V-985
W_StatsWRCorrelationTest V-993
W_sumCols V-427
W_SumDimension V-1007
W_sumRows V-427
W_tan V-1091
W_Texture V-421
W_TextureParams V-375
W_TriangulationData V-384, V-387
W_U2Diagonal V-548
W_UDiagonal V-548
W_W V-542, V-583–V-584
W_WatsonUtest V-922, V-988
W_WatsonWilliams V-989
W_WaveList wave V-320
W_WaveTransform V-1090
W_WheelerWatson V-990
W_WilcoxonTest V-907, V-991
W_XContour V-240
W_XHull V-101
W_xmax V-366
W_xmin V-366
W_XPolyn V-325
W_XPolynn III-73
W_XProjection V-781
W_YatRoot V-252
W_YContour V-240
W_YHull V-101
W_ymax V-366
W_ymin V-366
W_YPolyn V-325
W_YPolynn III-73
W_YProjection V-781
W_ZScores V-1083
wait
  Sleep operation V-871
wait until user clicks
  Sleep/B V-872
warping

images V-384
watch cursor
nonspinning V-872
waterfall
subranges II-322
waterfall plots II-326
creating II-326, V-691
cursors V-741, V-785
displaying II-326
modifying V-657
ModifyWaterfall operation V-657
NewWaterfall operation V-691
pcsr function V-741
qcsr function V-785
“poor man’s” II-302
wave assignment statements
index waves II-77
indexing II-76
subranges II-76
wave assignments II-74
* II-76
64-bits IV-39
bounds checking IV-53
chunk indexing V-1011
chunk numbers V-811
column indexing V-1119
column numbers V-785
complex 64-bits IV-39
Compose Expression dialog III-137–III-138
concatenating waves II-80
curly braces II-78
data values II-74
decomposing waves II-81
dependency formulas II-84, IV-232
destination waves II-74
evaluation of II-74
example II-75
FastOp operation V-216
impulses II-82
increment II-76
INF II-76
interpolation II-78, II-83
layer indexing V-1119
layer numbers V-786
lists of values II-78
mismatched waves II-83, III-175
multidimensional waves II-96–II-97
MultiThread keyword V-673
omitting indices II-76
p function II-74, II-96
point numbers V-731
q function II-96
r function II-96
row indexing in assignments V-1114
row numbers V-731
s function II-96
source waves II-74
speeding up V-216
subranges II-76
synthesis II-82
unexpected results II-84
x function II-74
X indexing in assignments V-1114
XY data II-80
wave declaration
automatic creation of NVAR IV-70
WAVE files V-892
wave inspector in debugger IV-222–IV-224
wave instance names IV-20
in annotations III-38
WAVE keyword IV-65, V-1069
/Z flag IV-67
automatic creation with rtGlobals IV-70, IV-72
checking in debugger IV-213
example IV-69
failures IV-67
function results IV-76
in procedures III-170
inline IV-73
reference type flags IV-74
reference types IV-73
standalone IV-72
use with $ IV-62
use with data folders IV-69
wave reference waves IV-77
with data folders IV-249
wave leaks
debugging IV-206–IV-207, V-1087
example IV-207
free wave leaks IV-94
IgorInfo(16) V-362
tracking IV-207
WaveTracking operation IV-207, V-1087
wave names
LoadWave operation II-142
text encodings III-473
wave notes II-84
copy/paste in tables II-240, II-269
keyword-value packed strings II-84
note function V-694
Note operation V-694
Wave parameters III-170
in macros IV-121
wave reference functions IV-85, IV-197
ListToWaveRefWave function V-499
WaveExists function V-1073
WaveRefIndexed function V-1080
WaveRefIndexedDFR function V-1080
WaveRefsEqual function V-1081
WaveRefWaveToList function V-1081
XWaveRefFromTrace function V-1119
wave references IV-71–IV-78

as destination waves IV-85–IV-86
as function results IV-76
automatic creation IV-72
checking in debugger IV-213
creation with $ IV-62, IV-83
explicit IV-75
in tables II-258, II-270
inline IV-73
passed as function parameter IV-82
problems IV-75
real versus complex IV-64
runtime wave type checking IV-53
standalone IV-72
strict mode IV-52, IV-75
WAVE keyword IV-65
waves containing IV-77
wave scaling
IndexToScale function V-442
ScaleToIndex function V-832
Wave Stats dialog III-124
wave symbols III-42
centering in annotations III-43
customize at point III-43
in an axis label II-318–II-319, III-37
in annotations III-37, III-42
markers III-43
size in annotations III-43
width in annotations III-43
wave text encodings III-472–III-478
converting to UTF-8 III-486
problems III-474
SetWaveTextEncoding operation III-477, V-859
WaveTextEncoding function V-1086
wave tracking
counter mode IV-207
tracker mode IV-207
wave units
text encodings III-473
WAVEClear keyword V-1070
waveCRC function V-1071
WaveDataToString function V-1071
WaveDims function V-1072
WaveExists IV-67
WaveExists function V-1073
example IV-200
waveform data
computing areas III-120
computing means III-120
converting to matrix II-98
definition I-2
description of II-62
generating from XY data III-109
loading waves (example) II-171
versus XY data III-108
WaveHash function V-1073
WaveInfo function V-1073
wavelet transforms
continuous III-282, V-137
CWT operation V-137
discrete III-283, V-189
DWT operation V-189
image analysis III-362–III-363
inverse V-189
WaveList function V-1075
in dialogs IV-145
liberal names with IV-169
WaveMax function V-1077
WaveMeanStdv operation V-1077
WaveMetrics
Igor mailing list II-4
packages IV-247
support II-4
WaveMetrics Procedures IV-25
WaveMetrics Procedures folder II-32, III-401, IV-25, IV-166
packages III-401
read only III-397
WaveMin function V-1078
WaveMinAndMax function V-1078
WaveModCount function V-1079
WaveName function V-1079
WaveRefIndexed function IV-197, V-1080
example IV-200
WaveRefIndexedDFR function IV-197, V-1080
WaveRefsEqual function IV-198, V-1081
WaveRefWaveToList function IV-198, V-1081
waves II-62–II-88
64-bit capable wave record II-35
64-bits IV-39
accessing in user functions IV-65–IV-78
adopting II-25
adopting in Data Browser II-119
adopting programmatically V-23
appearance in graphs II-290–II-306, II-348, II-364, II-380
AppendToGraph operation V-35
AppendToTable operation V-37
as Bezier curves III-75
as polygons III-75, V-248
as XY pair I-3, V-1118–V-1119
autosave II-37
auxiliary
setting markers II-298
average deviation V-415, V-1084
average value III-173, V-415, V-1084
axis controlled by wave II-279, II-318, V-43, V-85, V-381
browsing II-72
in dialogs II-228
combining V-82
comparing V-196
complex II-81

- complex 64-bits IV-39
- Concatenate operation V-82
- concatenating II-80
- contour names V-86
- converting to/from text II-73
- copying II-115
- CopyScales operation V-107
- creating II-65, II-92, V-185, V-526
  - by drawing
    - GraphWaveDraw operation V-325
  - examples I-35, I-46, I-53
  - in tables II-239, II-248
- CreationDate function V-114
- CsrWave function V-116
- CsrWaveRef function V-117
- CsrXWaveRef function V-117
- dat units II-69
  - precision II-69
- data
  - scaling V-853
  - units V-853
  - values II-63
    - finding point number V-51
- data folder containing wave V-317
- data full scale II-69
- data types II-66
- date format II-245, II-315
- date/time II-85
- deallocating memory IV-205, V-1070
- decimation II-98, III-135
- decomposing II-81
- default properties II-67
- definition I-2, II-62
- deleting data II-73, II-249, V-157
- dependency formulas II-84, IV-232
- DimDelta function V-160
- dimensions
  - how many V-1072
  - labels V-240, V-298
  - scaling V-853
- DimOffset function V-160
- DimSize function V-160
- dispersion V-415, V-1084
- display of complex III-273
- drag and drop II-280
- Duplicate operation V-185–V-186
- duplicating II-70
  - in tables II-240, II-269
- dynamic updating in tables II-234, II-237
- editing by drawing III-73, III-75–III-76
  - examples I-21–I-22, I-29
  - GraphWaveDraw operation V-326
- error waves II-304
- exists function V-207
- Extract operation V-213
- f(z) II-298
- finding values V-252–V-253
- FindSequence operation V-252
- FindValue operation V-253
- fixed length V-526, V-788
- flip V-1090
- free IV-91–IV-96, IV-207
  - Extract operation V-214
- free waves V-1092
- freeing memory IV-205, V-1070
- freezing
  - brr V-858
- global IV-207, V-1087
- global waves IV-91, IV-96, V-1092
- graphing II-276–II-354
- hiding in graphs II-295
- home waves II-87
- image names V-397
- in experiments V-1075
- in graphs V-1075
  - CheckDisplayed operation V-68
- in tables V-1075
  - exporting II-270, V-829
- in user functions III-170–III-174, IV-82–IV-85
- in windows V-1075
- index V-1090
- index in style macros II-351
- index waves II-77
- indexing II-76
  - dimension labels II-93
  - using dimension labels II-82
- information about II-72, V-1073
- initialization II-79
- inserting data II-73, II-249, V-443
- instance names III-38, IV-20
- integer data types II-85
- integer wave calculations IV-9
- inverse V-1090
- killing II-70–II-71, II-117
- KillWaves operation V-471
- kurtosis V-415, V-1084
- labels II-93–II-94
  - example II-82
  - FindDimLabel function V-240
  - GetDimLabel function V-298
  - in delimited text files II-131
  - in tables II-262
  - length limit II-94, II-244
  - naming conventions II-94
  - SetDimLabel operation V-838
  - speed considerations II-94
  - viewing in tables II-235
  - wave indexing II-82
- list of V-1075
  - CountObjects operation V-110
  - CountObjectsDFR operation V-111
  - GetIndexedObjName function V-303

GetIndexedObjNameDFR function V-304
properties II-88
waves in graph V-320
listing in data folder V-141
ListToWaveRefWave function V-499
live II-347–II-348
loading
from Igor experiment file V-500–V-505
local IV-207, V-1087
local waves IV-91, IV-96, V-1092
locking V-858
magnitude squared V-1090
Make operation V-526
making II-65, II-67, II-92, V-185, V-526
MatrixOp operation III-143
maximum
column location V-415
location V-1084
row location V-415
value V-415, V-1084
minimum
column location V-415
location V-1084
row location V-415
value V-415, V-1084
mismatched waves II-83, III-175
modification date/time V-592
MoveWave operation V-665
moving to data folder II-115
name conflicts III-505
name from wave reference V-675
names II-63, II-65, III-38
as parameters IV-85
CheckName function V-68
in Igor binary wave files II-155
instance names IV-20
related functions V-11
names of free waves IV-95
naming rules III-501
normalization II-79
number of INFs V-1084
number of NaNs V-1084
number of points II-67, III-109, V-160, V-526, V-714, V-788, V-1084
default II-67
numeric precision II-66, II-68, V-788
default II-67
numeric types V-1092
numpnts function V-714
offsets in graphs II-302–II-303
multipliers II-302
preventing II-302
undoing II-302
overview of waves I-2–I-3
overwriting caveats II-68
parameters
in user functions IV-60
to functions V-675
pass-by-reference IV-60
passed as function parameter IV-82
point numbers II-63, V-1114
preventing modification V-858
printing V-769
processing lists of waves IV-198–IV-201
properties default II-67
Redimension operation V-788
redimensioning II-72, V-788
related functions V-8
related operations V-2–V-3
relation to graphs I-3
relation to tables I-3, II-234
removing INFs V-1091
removing NaNs V-1092
removing zeros V-1092
renaming II-72, II-115, III-504, V-796
in tables II-234, II-239
reshaping 1D to 2D V-789
Reverse operation V-809
RMS value V-415, V-1084
rotate V-1090
Rotate operation V-810
rows II-62
runtime lookup of globals IV-65–IV-67
saving
wfprintf operation V-1093
saving huge waves II-35
scaling II-68, V-160, V-853
searching V-252–V-253
SetDimLabel operation V-838
SetScale operation V-853–V-854
SetWaveLock operation V-858
shared waves II-87
shift V-1090
sign V-1091
skewness V-415, V-1084
splitting V-901
splitting up II-70
SplitWave operation V-901
standard deviation V-1077, V-1084
standard error of mean V-1084
statistics III-122–III-124, V-1077, V-1082
styles II-290–II-306, II-350–II-352
preferences II-348, II-364, II-380
sum function V-1007
sum of values V-1084
symbols in legends III-37, III-42
synthesis II-82
temporary II-108
terminology I-2, II-62
text encodings III-472–III-473
time format II-245, II-315
trace instance names III-38

transforms V-1090
uniform spacing I-2
UniqueName function V-1048
units string of V-1093
value waves II-77
variance V-1084
wave instance names IV-20
wave reference functions IV-197
ContourNameToWaveRef V-87
ImageNameToWaveRef V-398
WAVEClear keyword V-1070
WaveDims function V-1072
WaveExists function V-1073
WaveInfo function V-1073
WaveList function V-1075
WaveModCount function V-1079
WaveName function V-1079
WaveRefIndexed function V-1080
WaveRefIndexedDFR function V-1080
WaveRefsEqual function V-1081
WaveRefWaveToList function V-1081
WaveStats operation III-122
WaveTransform operation V-1090
X scaling I-2, II-63, V-157, V-483, V-810
X values II-62–II-63
x0, dx II-63
zero padding V-789
WAVES keyword in Igor Text files II-151
waves text encodings
Override Experiment III-473
WavesAverage function III-173
WavesMax function III-173
WaveStats
multidimensional waves II-99
WaveStats operation V-1082
for finding the mean III-121
NaNs and INFs III-124
special variables III-122
WaveSum function III-171
WaveTextEncoding function V-1086
WaveTracker
example IV-207
WaveTracking operation IV-207, V-1087
WaveTransform
removing NaNs III-113
WaveTransform operation V-1090
WaveType function V-1092
WaveUnits function V-1093
web pages
downloading via HTTP IV-276
HTTP queries IV-277
weighting wave
curve fitting III-199
JointHistogram operation V-466
western text encodings III-461, III-466
wfprintf operation IV-195, IV-259–IV-261, V-1093
conversion specifications IV-260
example IV-261
WhichListItem function V-1095
while loops
in user functions IV-45–IV-46
whiskers
box plots II-331–II-332, V-595
whitespace characters
in tables II-258
Whole Word III-24
width of graphs II-288
Wigner Distribution Function III-281
Wigner transform III-281
WignerTransform operation V-1095
wildcard character V-999, V-1066, V-1075
WinList function V-1098
wind barbs II-329, V-614
winding rule III-72
Window Browser
WindowBrowserWindowsPopup menu IV-140
window browser II-49
Window Control dialog II-49, II-240, II-277
window hooks
deactivate events IV-303
exterior subwindows IV-305
hide events IV-304
kill events IV-303
show events IV-304
Window keyword IV-118, V-1097
data folders II-111
window names II-45
changing V-169
CheckName function V-68
graphs II-277, V-162
references using $ IV-63
UniqueName function V-1048
window position
platform-related issues III-455
window recreation
DoWindow operation V-168
page layouts II-477
tables II-241
Window Tiling Area dialog II-50
window titles II-45
command window II-9
DoWindow operation V-170
for notebooks III-13, III-20, III-25
for page layouts II-477
for tables II-235, II-240
graphs II-277–II-278
WindowBrowserWindowsPopup menu IV-140
WindowFunction operation V-1097
windowing
amplitude compensation III-278
Bartlet function V-225
Bartlett function V-183, V-225, V-1097

Blackman function V-183, V-225, V-1097
Cos function V-225
DSPPeriodgram operation V-182
energy loss III-278
flat-top windows V-227
Hamming function V-183, V-226, V-1097
Hanning function V-183, V-226, V-1097
Hanning window function III-277
images V-435
Bartlet V-436
Bartlett V-436
Blackman V-436
Hamming V-436
Hanning V-436
Kaiser V-437
Kaiser Bessel function V-183, V-226, V-437, V-1097
matrices V-435
other window functions III-279
Parzen function V-183, V-226, V-1097
Poisson function V-183, V-226, V-1097
Riemann function V-183, V-227, V-1097
triangle window function III-278
used with FFT III-275–III-279
WindowFunction operation V-1097
Windows
file permissions II-30
high-DPI III-508
windows II-44–II-60
activating II-46
active state V-318
AutoPositionWindow operation V-42
Bring to Front II-50
Close Window dialogs II-47, II-350
closing II-46, V-471
control dialog II-49
DefineGuide operation V-153
DoWindow operation V-168
expand to full size II-50
GetUserData function V-316
GetWindow operation V-318
GetWindowBrowserSelection function V-323
GuideInfo function V-336
GuideNameList function V-337
guides
built-in V-153
creating V-153, V-336
deleting V-153
information about V-336
list of V-337
moving V-153
help windows II-3
hiding II-46
HideProcedures operation V-347
hook function V-320
named V-320, V-866
unnamed V-866
hook functions
named IV-293
subwindows IV-294
unnamed IV-305
killing II-46, V-471
list of V-1098, V-1100
macro submenus II-48
MacroList function V-524
making a new window II-45
managing position and size II-50
maximized state V-320
menu II-45–II-51
move to preferred position II-50
MoveWindow operation V-666
names
of notebooks III-25, V-686
of tables II-240
naming rules III-501
note V-320, V-867
overview II-44
Recent Windows submenu II-46
recreation macros II-47–II-48, II-350, V-168, V-1101
related functions V-10
related operations V-2
renaming V-797
retrieving II-50
retrieving all II-50
saving II-47
Send to Back II-50
SetWindow operation V-865
shortcuts II-60
showing II-46
DisplayProcedure operation V-163
size V-321
stacking II-50, V-905
style macros V-171
target window II-44
Tile or Stack Windows dialog II-50
tiling II-50, V-1036–V-1037
user data IV-293, V-868
examples IV-293
getting V-316
window browser II-49
Windows menu II-45–II-51
Windows metafiles
saving V-826
Windows OS
ActiveX Automation IV-265
date system II-245
file extensions
cross-platform issues
in procedures III-455
in procedure files III-455
template files II-21

file paths
in procedures III-455
issues III-448
Igor program name III-448
system requirements III-513
system version info V-360
Windows paths
HDF5 II-221
Windows-1252 text encoding III-460–III-461, III-481
WinList function V-1098
independent modules V-1099
WinName function V-1100
WinRecreation function V-1101
wintype function V-1103
wireframe plots II-95
WMAxisHookStruct V-609
WMAxisHookStruct structure V-1103
WMBackgroundStruct structure IV-319–IV-320, V-1103
WMButtonAction structure V-1104
WMCheckboxAction structure V-1104
WMCustomControlAction structure V-1105
WMDrawUserShapeStruct structure V-1106
WMF
saving V-826
WMFitInfoStruct structure V-1107
WMGizmoHookStruct II-473
WMGizmoHookStruct structure V-1107
WMListboxAction structure V-1108
WMMarkerHookStruct IV-309
WMMarkerHookStruct structure V-1109
WMMenus.ipf IV-247
WMPopupAction structure V-1109
WMSetVariableAction structure V-1110
WMSliderAction structure V-1111
WMT0 files
notebooks III-2
WMTabControlAction structure V-1112
WMTooltipHookStruct IV-310, V-1113
WMWinHookStruct IV-297
keyboard example IV-300
WMWinHookStruct structure V-1113
wnoise function V-1113
worker functions for threads IV-330
worksheets
using a notebook as III-4
World Wide Web II-5
creating help file links IV-257
Igor as browser IV-281–IV-283, IV-287, IV-289
NIDAQ Tools information IV-312
opening preferred browser V-54
WaveMetrics
home page II-5
Wrap Around Search III-24
wrap end effect method III-299
write-protect
in notebooks III-10, V-697
ModifyProcedure operation V-643
write-protect icon
in notebooks III-5
in procedure windows III-397

X

x
suffix in tables II-241, II-254
x function V-1114
in multidimensional waves II-92–II-93, II-95
in wave assignments II-74
X keyword in Igor Text files II-152
x offset
in polynomial fitting III-210, V-127
X scaling I-2, II-63
changed by FFTs III-271
changed by IFFTs III-271
changing II-68, V-107
SetScale operation V-853
checking
in tables II-235
CopyScales operation V-107
default II-67
deltax function V-157
for XY data II-63
in Igor Text files II-152
leftx function V-483
loading text files II-133, II-140
pnt2x function V-748
rightx function V-810
Rotate operation III-316
SetScale operation V-853
x2pnt function V-1114
X units
changed by FFTs III-271
changed by IFFTs III-271
changing II-68, V-107
SetScale operation V-853
default II-67
X values II-62–II-63
can’t enter in tables II-244
conversion to point numbers V-1114
definition II-63
in tables II-235, II-241
in wave assignments II-74
indexing II-74, II-76
interpolation II-78
pnt2x function V-748
relation to point numbers II-63
tag attached at III-44
viewing in tables II-235
x function V-1114
x2pnt function V-1114
X waves

  - CsrXWave function V-117
  - CsrXWaveRef function V-117
- x2pnt function V-1114
- Xcode IV-208
- xcsr function V-1114
- XLLoadWave operation II-159, V-1115
- XOP Toolkit IV-209
- XOPs IV-209
- XOR
  - bitwise operator IV-6–IV-7
- Xoshiro256** V-196, V-324
- XWaveName function V-1118
- XWaveRefFromTrace function IV-197, V-1119
- XY data
  - areaXY function V-41
  - converting to waveform data II-80, III-109
  - definition I-3
  - description of II-63
  - faverageXY function V-218
  - in curve fitting III-197
  - in graphs II-278, V-1118–V-1119
  - interpolation of III-109, V-458
  - loading waves (example) II-173
  - overview I-3
  - versus waveform data III-108
  - X scaling of II-63
  - XY area procedures III-122
- XY Pair to Waveform panel III-109
- XYToWave1 user-defined function III-111
- XYToWave2 user-defined function III-112
- XYZ (color) conversions
  - rgb2xyz V-424
  - xyz2rgb V-428
- XYZ data
  - bounding sphere V-53
  - converting to matrix II-367
  - interpolation V-459, V-899

Y

- y
  - suffix in tables II-241
- Y full scale
  - default II-67
- y function V-1119
  - in multidimensional waves II-92–II-93, II-95
- Y scaling
  - changing V-107
    - SetScale operation V-853
  - CopyScales operation V-107
  - SetScale operation V-853
  - use in Axis Range tab II-287
- Y units
  - changing V-107
    - SetScale operation V-853
  - default II-67
- Y values
  - lists of II-78
  - y function V-1119

Z

- z function V-1119
  - in multidimensional waves II-92–II-93, II-95
- Z scaling
  - changing V-107
    - SetScale operation V-853
  - CopyScales operation V-107
  - SetScale operation V-853
- Z units
  - changing V-107
    - SetScale operation V-853
- Z values
  - z function V-1119
- Zapf Dingbats font III-493
- zcsr function V-1120
- ZernikeR function V-1120
- zero end effect method III-299
- zero line in graphs II-310
- zero padding waves V-789
- zeros
  - removing
    - from waves V-1092
- zeros and poles III-301
- zeros of functions
  - finding III-338–III-343
  - FindRoots operation V-248
- zeta function V-1120
- zip files
  - UnzipFile operation V-1051
- zooming
  - command line II-53
  - dialogs
    - text areas II-53
  - graphs II-285
    - example I-35
  - help II-53
  - help files II-53
  - history area II-53
  - notebooks II-53
  - page layouts II-477
  - procedure windows II-53
  - setting default II-53