Igor Reference

This volume contains detailed information about Igor Pro's built-in operations, functions, and keywords. They are listed alphabetically after the category sections that follow.

External operations (XOPs) and external functions (XFUNCs) are not covered here. For information about them, use the Command Help tab of the Igor Help Browser.

Built-In Operations by Category

Note: Some operations may appear in more than one category.

Graphs

AppendText	AppendToGraph	AppendToLayout	CheckDisplayed
ColorScale	ColorTab2Wave	ControlBar	Cursor
DefaultFont	DefineGuide	DelayUpdate	DeleteAnnotations
Display	DoUpdate	DoWindow	ErrorBars
GetAxis	GetMarquee	GetSelection	GetWindow
GraphNormal	GraphWaveDraw	GraphWaveEdit	HideInfo
HideTools	KillFreeAxis	KillWindow	Label
Legend	ModifyFreeAxis	ModifyGraph	ModifyWaterfall
MoveSubwindow	MoveWindow	NewFreeAxis	NewWaterfall
PauseUpdate	PrintGraphs	RemoveFromGraph	RenameWindow
ReorderImages	ReorderTraces	ReplaceText	ReplaceWave
ResumeUpdate	SaveGraphCopy	SetActiveSubwindow	SetAxis
SetMarquee	SetWindow	ShowInfo	ShowTools
StackWindows	Tag	TextBox	TileWindows

Contour and Image Plots

AppendImage	AppendMatrixContour	AppendToLayout	AppendXYZContour
CheckDisplayed	ColorScale	ColorTab2Wave	DefineGuide
DeleteAnnotations	DoUpdate	DoWindow	FindContour
HideTools	ImageLoad	ImageSave	KillWindow
ModifyContour	ModifyImage	MoveSubwindow	NewImage
PauseUpdate	RemoveContour	RemoveImage	ReplaceWave
SetActiveSubwindow	SetWindow	ShowTools	Tag
TileWindows			

Tables

AppendToLayout	AppendToTable	CheckDisplayed	DelayUpdate
DoUpdate	DoWindow	Edit	GetSelection
GetWindow	KillWindow	ModifyTable	MoveWindow
PauseUpdate	PrintTable	RemoveFromTable	RenameWindow
ResumeUpdate	SaveTableCopy	SetWindow	StackWindows
TileWindows			

Layouts

AppendLayoutObject	AppendText	AppendToLayout	ColorScale
DefaultFont	DelayUpdate	DeleteAnnotations	DoUpdate
DoWindow	GetMarquee	GetSelection	GetWindow
HideTools	KillWindow	Layout	LayoutPageAction
LayoutSlideShow	Legend	ModifyLayout	MoveWindow
NewLayout	PauseUpdate	PrintLayout	RemoveFromLayout
RemoveLayoutObjects	RenameWindow	ReplaceText	ResumeUpdate
SetMarquee	SetWindow	ShowTools	Stack
StackWindows	TextBox	Tile	TileWindows

Gizmo

AppendToGizmo	ExportGizmo	GetGizmo	GizmoInfo
GizmoScale	ModifyGizmo	NewGizmo	RemoveFromGizmo

Subwindows

DefineGuide	GetMarquee	KillWindow	MoveSubwindow
RenameWindow	SetActiveSubwindow	SetMarquee	

Other Windows

CloseHelp	CloseProc	CreateBrowser	DisplayProcedure
DoWindow	GetCamera	GetSelection	GetWindow
HideProcedures	HideTools	KillWindow	ModifyBrowser
ModifyCamera	ModifyPanel	MoveWindow	NewCamera
NewNotebook	NewPanel	Notebook	NotebookAction
OpenHelp	OpenNotebook	PrintNotebook	RenameWindow
SaveNotebook	SetWindow	ShowTools	StackWindows

All Windows

Append	AutoPositionWindow	DoWindow	GetSelection
GetUserData	GetWindow	KillWindow	Modify
MoveWindow	Remove	RenameWindow	SetWindow
StackWindows	TileWindows		

Wave Operations

AddMovieAudio	Append	AppendToGraph	AppendToTable
CheckDisplayed	ColorTab2Wave	Concatenate	CopyScales
DeletePoints	Display	Duplicate	Edit
Extract	FIFO2Wave	FindSequence	FindSequence
FindValue	GBLoadWave	GraphWaveDraw	GraphWaveEdit
InsertPoints	JCAMPLoadWave	KillWaves	LoadData
LoadWave	Make	MLLoadWave	MoveWave
Note	PlaySound	Redimension	Remove
RemoveFromGraph	RemoveFromTable	Rename	ReplaceWave
Reverse	Rotate	Save	SetDimLabel

SetScale	SetWaveLock	SetWaveTextEncoding	SoundLoadWave
SoundSaveWave	SplitWave	WAVEClear	WaveStats
wfprintf	XLLoadWave		

Analysis

APMath	BoundingBall	Convolve	Correlate
ConvexHull	Cross	CurveFit	CWT
Differentiate	DPSS	DSPDetrend	DSPPeriodogram
DWT	EdgeStats	FastGaussTransform	FastOp
FFT	FilterFIR	FilterIIR	FindContour
FindDuplicates	FindLevel	FindLevels	FindPeak
FindPointsInPoly	FindRoots	FindValue	FuncFit
FuncFitMD	Hanning	HilbertTransform	Histogram
ICA	IFFT	IndexSort	Integrate
Integrate1D	Integrate2D	IntegrateODE	Interpolate2
Interp3DPath	Interpolate3D	JointHistogram	Loess
LombPeriodogram	MakeIndex	MultiTaperPSD	NeuralNetworkRun
NeuralNetworkTrain	Optimize	PCA	PrimeFactors
Project	PulseStats	RatioFromNumber	Resample
Smooth	SmoothCustom	Sort	SortColumns
SphericalInterpolate	SphericalTriangulate	SumDimension	SumSeries
Triangulate3D	Unwrap	WaveMeanStdv	WaveStats
WaveTransform	WignerTransform	WindowFunction	

Matrix Operations

Concatenate	Extract	FFT	IFFT
ImageFilter	ImageFromXYZ	Loess	MatrixConvolve
MatrixCorr	MatrixEigenV	MatrixFilter	MatrixGaussJ
MatrixGLM	MatrixInverse	MatrixLinearSolve	MatrixLinearSolveTD
MatrixLLS	MatrixLUBkSub	MatrixLUD	MatrixLUDTD
MatrixMultiply	MatrixOp	MatrixSchur	MatrixSolve
MatrixSVBkSub	MatrixSVD	MatrixTranspose	Reverse
SplitWave	SumDimension	WaveTransform	

Analysis of Functions

FindRoots	Integrate1D	IntegrateODE	Optimize
SumSeries			

Signal Processing

Convolve	Correlate	CWT	DPSS
DSPDetrend	DSPPeriodogram	DWT	EdgeStats
FFT	FilterFIR	FilterIIR	FindLevel
FindLevels	FindPeak	Hanning	HilbertTransform
IFFT	ImageWindow	LinearFeedbackShiftRegister	LombPeriodogram
MultiTaperPSD	PulseStats	Resample	Rotate
SmoothCustom	Unwrap	WignerTransform	WindowFunction

Image Analysis

ColorScale	ColorTab2Wave	DWT	ImageAnalyzeParticles
ImageBlend	ImageBoundaryToMask	ImageEdgeDetection	ImageFileInfo
ImageFilter	ImageFocus	ImageFromXYZ	ImageGenerateROIMask
ImageGLCM	ImageHistModification	ImageHistogram	ImageInfo
ImageInterpolate	ImageLineProfile	ImageLoad	ImageMorphology
ImageNameList	ImageNameToWaveRef	ImageRegistration	ImageRemoveBackground
ImageRestore	ImageRotate	ImageSave	ImageSeedFill
ImageSnake	ImageSkeleton3D	ImageStats	ImageThreshold
ImageTransform	ImageUnwrapPhase	ImageWindow	Loess
MatrixFilter			

Statistics

EdgeStats	FPClustering	Histogram	ICA
ImageHistModification	ImageHistogram	ImageStats	JointHistogram
KMeans	PCA	PulseStats	SetRandomSeed
StatsAngularDistanceTest	StatsANOVA1Test	StatsANOVA2NRTest	StatsANOVA2RMTest
StatsANOVA2Test	StatsChiTest	StatsCircularCorrelationTest	StatsCircularMeans
StatsCircularMoments	StatsCircularTwoSampleTest	StatsCochranTest	StatsContingencyTable
StatsDIPTest	StatsDunnettTest	StatsFriedmanTest	StatsFTest
StatsHodgesAjneTest	StatsJBTest	StatsKDE	StatsKendallTauTest
StatsKSTest	StatsKWTest	StatsLinearCorrelationTest	StatsLinearRegression
StatsMultiCorrelationTest	StatsNPMCTest	StatsNPNominalSRTest	StatsQuantiles
StatsRankCorrelationTest	StatsResample	StatsSample	StatsScheffeTest
StatsShapiroWilkTest	StatsSignTest	StatsSRTest	StatsTTest
StatsTukeyTest	StatsVariancesTest	StatsWatsonUSquaredTest	StatsWatsonWilliamsTest
StatsWheelerWatsonTest	StatsWilcoxonRankTest	StatsWRCorrelationTest	WaveMeanStdv
WaveStats			

Geometry

BoundingBall	ConvexHull	FindPointsInPoly	Interp3DPath
Interpolate3D	Project	SphericalInterpolate	SphericalTriangulate
Triangulate3D			

Drawing

DrawAction	DrawArc	DrawBezier	DrawLine
DrawOval	DrawPICT	DrawPoly	DrawRect
DrawRRect	DrawText	DrawUserShape	GraphNormal
GraphWaveDraw	GraphWaveEdit	HideTools	SetDashPattern
SetDrawEnv	SetDrawLayer	ShowTools	ToolsGrid

Programming & Utilities

Abort	BackgroundInfo	Beep	BuildMenu
ChooseColor	CloseProc	CtrlBackground	CtrlNamedBackground
DefaultGUIFont	DefaultGUIControls	Debugger	DebuggerOptions
DefaultTextEncoding	DelayUpdate	DisplayHelpTopic	DisplayProcedure

DoAlert	DoIgorMenu	DoUpdate	DoXOPIdle
Execute	Execute/P	ExecuteScriptText	ExperimentModified
GetLastUserMenuInfo	GetMouse	Grep	HideIgorMenus
HideProcedures	IgorVersion	KillBackground	KillStrings
KillVariables	LoadPackagePreferences	MarkPerfTestTime	MeasureStyledText
MoveString	MoveVariable	MoveWave	MultiThreadingControl
ParseOperationTemplate	PauseForUser	PauseUpdate	Preferences
PrintSettings	PutScrapText	Quit	Rename
ResumeUpdate	SavePackagePreferences	SetBackground	SetFormula
SetIgorHook	SetIgorMenuMode	SetIgorOption	SetProcessSleep
SetRandomSeed	SetWaveLock	ShowIgorMenus	Silent
Sleep	Slow	Demo	sprintf
sscanf	String	StructGet	StructPut
ThreadGroupPutDF	ThreadStart	ToCommandLine	Variable
WAVEClear			

Files & Paths

AdoptFiles	BrowseURL	Close	CopyFile
CopyFolder	CreateAliasShortcut	DeleteFile	DeleteFolder
FBinRead	FBinWrite	fprintf	FReadLine
FGetPos	FSetPos	FStatus	FTPCreateDirectory
FTPDelete	FTPDownload	FTPUpload	GBLoadWave
GetFileFolderInfo	Grep	ImageFileInfo	ImageLoad
ImageSave	JCAMPLoadWave	KillPath	KillPICTs
KillWaves	LoadData	LoadPICT	LoadWave
MLLoadWave	MoveFile	MoveFolder	NewNotebook
NewPath	Open	OpenNotebook	OpenProc
PathInfo	ReadVariables	RemovePath	RenamePath
RenamePICT	Save	SaveData	SaveExperiment
SaveGraphCopy	SaveNotebook	SavePICT	SaveTableCopy
SetFileFolderInfo	URLRequest	wfprintf	XLLoadWave

Data Folders

cd	Dir	DuplicateDataFolder	KillDataFolder
MoveDataFolder	MoveVariable	MoveWave	NewDataFolder
pwd	RenameDataFolder	ReplaceWave	root
SetDataFolder			

Movies & Sound

AddMovieAudio	AddMovieFrame	Beep	CloseMovie
ImageFileInfo	NewMovie	PlayMovie	PlayMovieAction
PlaySnd	PlaySound	SoundInRecord	SoundInSet
SoundInStartChart	SoundInStatus	SoundInStopChart	SoundLoadWave
SoundSaveWave			

Controls & Cursors

Button	Chart	CheckBox	ControlBar
ControlInfo	ControlUpdate	Cursor	CustomControl
DefaultGUIFont	DefaultGUIControls	GetUserData	GroupBox
HideInfo	HideTools	KillControl	ListBox
ListBoxControl	ModifyControl	ModifyControlList	NewPanel
popup	PopupContextualMenu	PopupMenu	PopupMenuControl
SetVariable	ShowInfo	ShowTools	Slider
TabControl	TitleBox	ValDisplay	

FIFOs

AddFIFOData	AddFIFOVectData	Chart	ControlInfo
CtrlFIFO	FIFO2Wave	FIFOStatus	KillFIFO
NewFIFO	NewFIFOChan	SoundInStartChart	

Printing

Print	printf	PrintGraphs	PrintLayout
PrintNotebook	PrintSettings	PrintTable	sprintf
wfprintf			

Built-In Functions by Category

Note: some functions may appear in more than one category.

Numbers

e	Inf	NaN	numtype
Pi	VariableList		

Trig

acos	asin	atan	atan2
cos	cot	csc	sawtooth
sec	sin	sinc	sqrt
tan			

Exponential

acosh	alog	asinh	atanh
cosh	coth	cpowi	csch
exp	ln	log	sech
sinh	tanh		

Complex

cabs	cequal	cmplx	conj
cpowi	imag	magsqr	p2rect
r2polar	real		

Rounding

abs	cabs	ceil	floor
limit	max	min	mod
round	sign	trunc	

Conversion

char2num	cmplx	ConvertGlobalStringTextEncoding	ConvertTextEncoding
date2secs	imag	LowerStr	magsqr
NormalizeUnicode	num2char	num2istr	num2str
p2rect	pnt2x	r2polar	real
Secs2Date	Secs2Time	str2num	UpperStr
x2pnt			

Time and Date

CreationDate	date	dateToJulian	date2secs
DateTime	JulianToDate	ModDate	Secs2Date
Secs2Time	StartMSTimer	StopMSTimer	ticks
time			

Matrix Analysis

MatrixCondition	MatrixDet	MatrixDot	MatrixRank
MatrixTrace			

Wave Analysis

area	areaXY	BinarySearch	BinarySearchInterp
ContourZ	FakeData	faverage	faverageXY
interp	Interp2D	Interp3D	mean
median	p	poly	poly2D
PolygonArea	q	r	s
sum	t	Variance	x
y	z		

About Waves

BinarySearch	BinarySearchInterp	ContourInfo	ContourNameToWaveRef
ContourZ	CreationDate	CsrInfo	CsrWave
CsrWaveRef	CsrXWave	CsrXWaveRef	deltax
DimDelta	DimOffset	DimSize	EqualWaves
exists	FindDimLabel	GetDimLabel	GetWavesDataFolder
GetWavesDataFolderDFR	hcsr	ImageInfo	ImageNameToWaveRef
IndexToScale	leftx	ModDate	NameOfWave
NewFreeWave	note	numpnts	p
pcsr	pnt2x	q	qcsr
r	rightx	s	ScaleToIndex
t	TagVal	TagWaveRef	TraceInfo
TraceNameToWaveRef	WaveCRC	WaveDims	WaveExists
WaveInfo	WaveList	WaveName	WaveRefIndexed
WaveRefsEqual	WaveTextEncoding	WaveType	WaveUnits
x	x2pnt	xcsr	XWaveName
XWaveRefFromTrace	y	v	zcsr

Special

airyA	airyAD	airyB	airyBD
Besseli	Besselj	Besselk	Bessely
bessI	bessJ	bessK	bessY
beta	betai	binomial	binomialln
binomialNoise	chebyshev	chebyshevU	CosIntegral
dawson	digamma	Dilogarithm	ei
enoise	erf	erfc	erfcw
expInt	ExpIntegralE1	expnoise	factorial
fresnelCos	fresnelCS	fresnelSin	gamma
gammaInc	gammaNoise	gammln	gammp
gammq	Gauss	Gauss1D	Gauss2D
gcd	gnoise	hermite	hermiteGauss
hyperG0F1	hyperG1F1	hyperG2F1	hyperGNoise
hyperGPFQ	inverseErf	inverseErfc	JacobiCn
JacobiSn	laguerre	laguerreA	laguerreGauss
LambertW	legendreA	logNormalNoise	lorentzianNoise
MandelbrotPoint	MarcumQ	poissonNoise	poly

poly2D	SinIntegral	sphericalBessJ	sphericalBessJD
sphericalBessY	sphericalBessYD	sphericalHarmonics	sqrt
VoigtFunc	zeta	ZernikeR	

Statistics

binomialln	binomialNoise	enoise	erf
erfc	expnoise	faverage	faverageXY
gamma	gammaInc	gammaNoise	gammln
gammp	gammq	gnoise	inverseErf
inverseErfc	lorentzianNoise	logNormalNoise	mean
max	min	norm	poissonNoise
StatsCorrelation	StatsBetaCDF	StatsBetaPDF	StatsBinomialCDF
StatsBinomialPDF	StatsCauchyCDF	StatsCauchyPDF	StatsChiCDF
StatsChiPDF	StatsCMSSDCDF	StatsCorrelation	StatsDExpCDF
StatsDExpPDF	StatsErlangCDF	StatsErlangPDF	StatsErrorPDF
StatsEValueCDF	StatsEValuePDF	StatsExpCDF	StatsExpPDF
StatsFCDF	StatsFPDF	StatsFriedmanCDF	StatsGammaCDF
StatsGammaPDF	StatsGeometricCDF	StatsGeometricPDF	StatsGEVCDF
StatsGEVPDF	StatsHyperGCDF	StatsHyperGPDF	StatsInvBetaCDF
StatsInvBinomialCDF	StatsInvCauchyCDF	StatsInvChiCDF	StatsInvCMSSDCDF
StatsInvDExpCDF	StatsInvEValueCDF	StatsInvExpCDF	StatsInvFCDF
StatsInvFriedmanCDF	StatsInvGammaCDF	StatsInvGeometricCDF	StatsInvKuiperCDF
StatsInvLogisticCDF	StatsInvLogNormalCDF	StatsInvMaxwellCDF	StatsInvMooreCDF
StatsInvNBinomialCDF	StatsInvNCChiCDF	StatsInvNCFCDF	StatsInvNormalCDF
StatsInvParetoCDF	StatsInvPoissonCDF	StatsInvPowerCDF	StatsInvQCDF
StatsInvQpCDF	StatsInvRayleighCDF	StatsInvRectangularCDF	StatsInvSpearmanCDF
StatsInvStudentCDF	StatsInvTopDownCDF	StatsInvTriangularCDF	StatsInvUSquaredCDF
StatsInvVonMisesCDF	StatsInvWeibullCDF	StatsKuiperCDF	StatsLogisticCDF
StatsLogisticPDF	StatsLogNormalCDF	StatsLogNormalPDF	StatsMaxwellCDF
StatsMaxwellPDF	StatsMedian	StatsMooreCDF	StatsNBinomialCDF
StatsNBinomialPDF	StatsNCChiCDF	StatsNCChiPDF	StatsNCFCDF
StatsNCFPDF	StatsNCTCDF	StatsNCTPDF	StatsNormalCDF
StatsNormalPDF	StatsParetoCDF	StatsParetoPDF	StatsPermute
StatsPoissonCDF	StatsPoissonPDF	StatsPowerCDF	StatsPowerNoise
StatsPowerPDF	StatsQCDF	StatsQpCDF	StatsRayleighCDF
StatsRayleighPDF	StatsRectangularCDF	StatsRectangularPDF	StatsRunsCDF
StatsSpearmanRhoCDF	StatsStudentCDF	StatsStudentPDF	StatsTopDownCDF
StatsTriangularCDF	StatsTriangularPDF	StatsTrimmedMean	StatsUSquaredCDF
StatsVonMisesCDF	StatsVonMisesPDF	StatsWaldCDF	StatsWaldPDF
StatsWeibullCDF	StatsWeibullPDF	StudentA	StudentT
sum	Variance	WaveMax	WaveMin
wnoise			

Windows

AnnotationInfo	AnnotationList	AxisInfo	AxisList
AxisValFromPixel	ChildWindowList	ContourInfo	CsrInfo
CsrWave	CsrXWave	GetBrowserLine	GetBrowserSelection
GuideInfo	GuideNameList	GizmoScale	hcsr
ImageInfo	LayoutInfo	PanelResolution	pcsr

PixelFromAxisVal	ProcedureText	qcsr	SpecialCharacterInfo
SpecialCharacterList	TagVal	TraceInfo	vcsr
WinList	WinName	WinRecreation	WinType
xcsr	XWaveName	zcsr	

Strings

AddListItem	char2num	cmpstr	FontSizeHeight
FontSizeStringWidth	GrepList	GrepString	IndexedDir
IndexedFile	ListToTextWave	LowerStr	num2char
num2istr	num2str	PadString	PossiblyQuoteName
RemoveEnding	RemoveFromList	RemoveListItem	ReplaceStringByKey
SelectString	str2num	StringByKey	StringCRC
StringFromList	StringList	StringMatch	strlen
strsearch	TextFile	TrimString	UnPadString
UpperStr	URLDecode	URLEncode	WhichListItem

Names

CheckName	CleanupName	ContourNameList	ContourNameToWaveRef
ControlNameList	CTabList	FontList	FunctionList
GetDefaultFont	GetIndependentModuleName	GetIndexedObjName	GetWavesDataFolder
GetWavesDataFolderDFR	ImageNameList	ImageNameToWaveRef	IndependentModuleList
IndexedDir	IndexedFile	MacroList	NameOfWave
StringList	TextEncodingCode	TextEncodingName	TraceFromPixel
TraceNameList	TraceNameToWaveRef	UniqueName	VariableList
WaveList	WaveName	WinList	WinName
XWaveName			

Lists

AnnotationList	AxisList	ChildWindowList	ContourNameList
ControlNameList	CountObjects	CountObjectsDFR	DataFolderDir
FindListItem	FontList	FunctionInfo	FunctionList
GetIndexedObjName	GetWindow	GuideNameList	ImageNameList
IndependentModuleList	ItemsInList	ListMatch	ListToTextWave
ListToWaveRefWave	MacroList	NumberByKey	OperationList
PathList	PICTList	RemoveByKey	RemoveFromList
RemoveListItem	ReplaceNumberByKey	ReplaceStringByKey	SortList
StringByKey	StringFromList	StringList	TableInfo
TraceNameList	VariableList	WaveList	WaveRefIndexed
WaveRefWaveToList	WhichListItem	WinList	

Programming

CaptureHistory	CaptureHistoryStart	ControlNameList	DDEExecute
DDEInitiate	DDEPokeString	DDEPokeWave	DDERequestString
DDERequestWave	DDEStatus	DDETerminate	exists
FakeData	FuncRefInfo	FunctionInfo	GetDefaultFont
GetDefaultFontSize	GetDefaultFontStyle	GetEnvironmentVariable	GetErrMessage

GetFormula	GetKeyState	GetRTError	GetRTErrMessage
GetRTLocation	GetRTLocInfo	GetRTStackInfo	GetScrapText
GuideInfo	GuideNameList	Hash	i
IgorInfo	ilim	j	jlim
NameOfWave	numtype	NumVarOrDefault	NVAR_Exists
PanelResolution	ParamIsDefault	PICTInfo	PixelFromAxisVal
ProcedureText	ScreenResolution	SelectNumber	SelectString
SetEnvironmentVariable	SpecialDirPath	StartMSTimer	StopMSTimer
StringCRC	StrVarOrDefault	SVAR_Exists	TableInfo
TagVal	ThreadGroupCreate	ThreadGroupGetDF	ThreadGroupGetDFR
ThreadGroupRelease	ThreadGroupWait	ThreadProcessorCount	ThreadReturnValue
UnsetEnvironmentVariable	WaveCRC	WinType	

Data Folders

CountObjects	DataFolderDir	DataFolderExists	DataFolderRefsEqual
DataFolderRefStatus	GetDataFolder	GetDataFolderDFR	GetIndexedObjName
GetWavesDataFolder	GetWavesDataFolderDFR	NewFreeDataFolder	

I/O (files, paths, and PICTs)

FetchURL	FunctionPath	IndexedDir	IndexedFile
ParseFilePath	PathList	PICTInfo	PICTList
SpecialDirPath	TextFile	URLDecode	URLEncode

Built-In Keywords

Procedure Declarations

End	EndMacro	EndStructure	Function
Macro	Picture	Proc	Structure
Window			

Procedure Subtypes

ButtonControl	CameraWindow	CDFFunc	CheckBoxControl
CursorStyle	FitFunc	GizmoPlot	Graph
GraphMarquee	GraphStyle	GridStyle	Layout
LayoutMarquee	LayoutStyle	ListBoxControl	Panel
PopupMenuControl	SetVariableControl	SliderControl	TabControl
Table	TableStyle		

Object References

DFREF	FUNCREF	NVAR	STRUCT
SVAR	WAVE		

Function Local Variable Keywords

Complex	Double	Int	Int64
String	STRUCT	UInt64	Variable

Flow Control

AbortOnRTE	AbortOnValue	break	catch
continue	default	do-while	endtry
for-endfor	if-elseif-endif	if-endif	return
strswitch-case-endswitch	switch-case-endswitch	try	try-catch-endtry

Other Programming Keywords

#define	#if-#elif-#endif	#if-#endif	#ifdef-#endif
#ifndef-#endif	#include	#pragma	#undef
Constant	DoPrompt	GalleryGlobal	hide
IgorVersion	IndependentModule	Menu	ModuleName
MultiThread	Override	popup	ProcGlobal
Prompt	root	rtGlobals	Static
Strconstant	String	Submenu	TextEncoding
ThreadSafe	Variable	version	

Built-in Structures

Point	PointF	Rect	RectF
RGBColor	RGBAColor	WMAxisHookStruct	WMBackgroundStruct
WMButtonAction	WMCheckboxAction	WMCustomControlAction	WMDrawUserShapeStruct
WMFitInfoStruct	WMGizmoHookStruct	WMListboxAction	WMMarkerHookStruct
WMPopupAction	WMSetVariableAction	WMSliderAction	WMTabControlAction
WMWinHookStruct			

Hook Functions

See Chapter IV-10, **Advanced Topics**, **User-Defined Hook Functions** on page IV-264.

AfterCompiledHook	AfterFileOpenHook	AfterMDIFrameSizedHook	AfterWindowCreatedHook
BeforeDebuggerOpensHook	BeforeExperimentSaveHook	BeforeFileOpenHook	IgorBeforeNewHook
IgorBeforeQuitHook	IgorMenuHook	IgorQuitHook	IgorStartOrNewHook

Alphabetic Listing of Functions, Operations and Keywords

This section alphabetically lists all built-in functions, operations and keywords. Much of this information is also accessible online in the Command Help tab of the Igor Help Browser.

External operations (XOPs) and external functions (XFUNCs) are not covered here. For information about them, use the Command Help tab of the Igor Help Browser and the XOP help file in the same folder as the XOP file.

Reference Syntax Guide

In the descriptions of functions and operations that follow, italics indicate parameters for which you can supply numeric or string expressions. Non-italic keywords must be entered literally as they appear. Commas, slashes, braces and parentheses in these descriptions are always literals. Brackets surround optional flags or parameters. Ellipses (…) indicate that the preceding element may be repeated a number of times.

Italicized parameters represent values you supply. Italic words ending with "*Name*" are names (wave names, for example), and those ending with "*Str*" are strings. Italic words ending with "*Spec*" (meaning "specification") are usually further defined in the description. If none of these endings are employed, the italic word is a numeric expression, such as a literal number, the name of a variable or function, or some valid combination.

Strings and names are different, but you can use a string where a name is expected using "string substitution": precede a string expression with the $ operator. See **String Substitution Using $** on page IV-17.

A syntax description may span several lines, but the actual command you create must occupy a single line.

Many operations have optional "flags". Flags that accept a value (such as the Make operation's /N=*n* flag) sometimes require additional parentheses. For example:

```
Make/N=1 aNewWave
```

is acceptable because here *n* is the literal "1". To use a numeric expression (anything other than a literal number) for *n*, parentheses are needed:

```
Make/N=(numberOfPoints) aNewWave              // error if no parentheses!
```

For more about using functions, operations and keywords, see Chapter IV-1, **Working with Commands**, Chapter IV-2, **Programming Overview**, and Chapter IV-10, **Advanced Topics**.

#define

```
#define symbol
```

The #define statement is a conditional compilation directive that defines a *symbol* for use only with #ifdef or #ifndef expressions. #undef removes the definition.

Details

The defined *symbol* exists only in the file where it is defined; the only exception is in the main procedure window where the scope covers all other procedures except independent modules. See **Conditional Compilation** on page IV-100 for information on defining a global *symbol*.

#define cannot be combined inline with other conditional compilation directives.

See Also

The #**undef**, #**ifdef**-#**endif**, and #**ifndef**-#**endif** statements.

Conditional Compilation on page IV-100.

#if-#elif-#endif

```
#if expression1
   <TRUE part 1>
#elif expression2
   <TRUE part 2>
[...]
[#else
   <FALSE part>]
#endif
```

In a #if-#elif-#endif conditional compilation statement, when an expression evaluates as TRUE (absolute value > 0.5), then only code corresponding to the TRUE part of that expression is compiled, and then the conditional statement is exited. If all expressions evaluate as FALSE (zero) then *FALSE part* is compiled when present.

Details

Conditional compiler directives must be either entirely outside or inside function definitions; they cannot straddle a function fragment. Conditionals cannot be used within Macros.

See Also

Conditional Compilation on page IV-100 for more usage details.

#if-#endif

```
#if expression
   <TRUE part>
[#else
   <FALSE part>]
#endif
```

A #if-#endif conditional compilation statement evaluates *expression*. If *expression* is TRUE (absolute value > 0.5) then the code in *TRUE part* is compiled, or if FALSE (zero) then the optional *FALSE part* is compiled.

Details

Conditional compiler directives must be either entirely outside or inside function definitions; they cannot straddle a function fragment. Conditionals cannot be used within Macros.

See Also

Conditional Compilation on page IV-100 for more usage details.

#ifdef-#endif

```
#ifdef symbol
   <TRUE part>
[#else
   <FALSE part>]
#endif
```

A #ifdef-#endif conditional compilation statement evaluates *symbol*. When *symbol* is defined the code in *TRUE part* is compiled, or if undefined then the optional *FALSE part* is compiled.

Details

Conditional compiler directives must be either entirely outside or inside function definitions; they cannot straddle a function fragment. Conditionals cannot be used within Macros.

symbol must be defined before the conditional with #define.

See Also

The #**define** statement and **Conditional Compilation** on page IV-100 for more usage details.

#ifndef-#endif

```
#ifndef symbol
   <TRUE part>
[#else
   <FALSE part>]
#endif
```

An #ifndef-#endif conditional compilation statement evaluates *symbol*. When *symbol* is undefined the code in *TRUE part* is compiled, or if defined then the optional *FALSE part* is compiled.

Details

Conditional compiler directives must be either entirely outside or inside function definitions; they cannot straddle a function fragment. Conditionals cannot be used within Macros.

symbol must be defined before the conditional with #define.

See Also

The #**define** statement and **Conditional Compilation** on page IV-100 for more usage details.

#include

```
#include "file spec" or <file spec>
```

A #include statement in a procedure file automatically opens another procedure file. You should use #include in any procedure file that you write if it requires that another procedure file be open. A #include statement must always appear flush against the left margin in a procedure window.

Parameters

file spec is the procedure file name, which can incorporate a full or partial path. The form used depends of the procedure file location: <*file spec*> is in "Igor Pro 7 Folder/WaveMetrics Procedures" and "*file spec*" is in "Igor Pro 7 Folder/User Procedures" or "Igor Pro User Files/User Procedures".

See Also

The Include Statement on page IV-155 for usage details.

Igor Pro User Files on page II-31.

#pragma

```
#pragma pragmaName = value
```

#pragma introduces a compiler directive, which is a message to the Igor procedure compiler. A #pragma statement must always appear flush against the left margin in a procedure window.

Igor ignores unknown pragmas such as pragmas introduced in later versions of the program.

Currently Igor supports the following pragmas:

```
#pragma rtGlobals = value
#pragma version = versionNumber
#pragma IgorVersion = versionNumber
#pragma hide = value
#pragma ModuleName = name
#pragma IndependentModule = name
#pragma rtFunctionErrors = value
#pragma TextEncoding = "textEncodingName"
```

See Also

Pragmas on page IV-48

The rtGlobals Pragma on page IV-48

The version Pragma on page IV-50

The IgorVersion Pragma on page IV-50

The hide Pragma on page IV-50

The ModuleName Pragma on page IV-50

The IndependentModule Pragma on page IV-51

The rtFunctionErrors Pragma on page IV-51

The TextEncoding Pragma on page IV-51

#undef

`#undef` ***`symbol`***

A #undef statement removes a nonglobal *symbol* created previously by #define. See **Conditional Compilation** on page IV-100 for information on undefining a global *symbol.*

See Also

The #**define** statement and **Conditional Compilation** on page IV-100 for more usage details.

Abort

`Abort` `[`***`errorMessageStr`***`]`

The Abort operation aborts procedure execution.

Parameters

The optional *errorMessageStr* is a string expression, which, if present, specifies the message to be displayed in the error alert.

Details

Abort provides a way for a procedure to abort execution when it runs into an error condition.

See Also

Aborting Functions on page IV-103 , **Aborting Macros** on page IV-115, and **Flow Control for Aborts** on page IV-45. The **DoAlert** operation.

AbortOnRTE

`AbortOnRTE`

The AbortOnRTE flow control keyword raises an abort with a runtime error.

AbortOnRTE should be used after a command that might give rise to a runtime error.

You can place AbortOnRTE immediately after a command that might give rise to a runtime error that you want to handle instead of allowing Igor to handle it by halting procedure execution. Use a **try-catch-endtry** block to catch the abort, if it occurs.

AbortOnRTE has very low overhead and should not significantly slow program execution.

Details

In terms of programming style, you should consider using AbortOnRTE (preceded by a semicolon) on the same line as the command that may give rise to an abort condition.

When using AbortOnRTE after a related sequence of commands, then it should be placed on its own line.

When used with try-catch-endtry, you should place a call to **GetRTError**(1) in your catch section to clear the runtime error.

Example

Abort if the wave does not exist:

```
WAVE someWave; AbortOnRTE
```

See Also

Flow Control for Aborts on page IV-45 and **AbortOnRTE Keyword** on page IV-45 for further details.

The **try-catch-endtry** flow control statement.

AbortOnValue

`AbortOnValue` ***`abortCondition, abortCode`***

The AbortOnValue flow control keyword will abort function execution when the *abortCondition* is nonzero and it will then return the numeric *abortCode*. No dialog will be displayed when such an abort occurs.

Parameters

abortCondition can be any valid numeric expression using comparison or logical operators.

abortCode is a nonzero numeric value returned to any abort or error handling code by AbortOnValue whenever it causes an abort.

Details

When used with try-catch-endtry, you should place a call to **GetRTError**(1) in your catch section to clear the runtime error.

See Also

Flow Control for Aborts on page IV-45 and **AbortOnValue Keyword** on page IV-45 for further details.

The **AbortOnRTE** keyword and the **try-catch-endtry** flow control statement.

abs

`abs(`*`num`*`)`

The abs function returns the absolute value of the real number *num*. To calculate the absolute value of a complex number, use the cabs function.

See Also

The **cabs** function.

acos

`acos(`*`num`*`)`

The acos function returns the inverse cosine of *num* in radians in the range [0,π].

In complex expressions, *num* is complex and acos returns a complex value.

See Also

cos

acosh

`acosh(`*`num`*`)`

The acosh function returns the inverse hyperbolic cosine of *num*. In complex expressions, *num* is complex and acosh returns a complex value.

AddFIFOData

`AddFIFOData` *`FIFOName, FIFO_channelExpr`* `[,` *`FIFO_channelExpr`*`]`...

The AddFIFOData operation evaluates *FIFO_channelExpr* expressions as double precision floating point and places the resulting values into the named FIFO.

Details

There must be one *FIFO_channelExpr* for each channel in the FIFO.

See Also

FIFOs are used for data acquisition. See **FIFOs and Charts** on page IV-291.

Other operations used with FIFOs: **NewFIFO**, **NewFIFOChan**, **CtrlFIFO**, and **FIFOStatus**.

AddFIFOVectData

`AddFIFOVectData` *`FIFOName, FIFO_channelKeyExpr`* `[,` *`FIFO_channelKeyExpr`*`]`...

The AddFIFOVectData operation is similar to AddFIFOData except the expressions use a keyword to allow either a single numeric value for a normal channel or a wave containing the data for a special image vector channel.

Details

There must be one *FIFO_channelKeyExpr* for each channel in the FIFO.

A *FIFO_channelKeyExpr* may be one of:

```
num = numericExpression
vect = wave
```

For best results, the wave should have the same number of points as used to define the FIFO channel and the same number type. See the **NewFIFOChan** operation.

See Also

FIFOs and Charts on page IV-291.

AddListItem

`AddListItem(`*`itemStr, listStr`* `[,` *`listSepStr`* `[,` *`itemNum`*`]])`

The AddListItem function returns *listStr* after adding *itemStr* to it. *listStr* should contain items separated by the *listSepStr* character, such as “abc;def;”.

Use AddListItem to add an item to a string containing a list of items separated by a single character, such as those returned by functions like **TraceNameList** or **AnnotationList**, or to a line from a delimited text file.

listSepStr and *itemNum* are optional; their defaults are “;” and 0, respectively.

Details

By default *itemStr* is added to the start of the list. Use the optional list index *itemNum* to add *itemStr* at a different location. The returned list will have *itemStr* at the index *itemNum* or at ItemsInList(*returnedListStr*)-1 when *itemNum* equals or exceeds ItemsInList(*listStr*).

itemNum can be any value from -infinity (`-Inf`) to infinity (`Inf`). Values from -infinity to 0 prepend *itemStr* to the list, and values from ItemsInList(*listStr*) to infinity append *itemStr* to the list.

itemStr may be `""`, in which case an empty item (consisting of only a separator) is added.

If *listSepStr* is `""`, then *listStr* is returned unchanged (unless *listStr* contains only list separators, in which case an empty string (`""`) is returned).

listStr is treated as if it ends with a *listSepStr* even if it doesn’t.

In Igor6, only the first byte of *listSepStr* was used. In Igor7 and later, all bytes are used.

Examples

```
Print AddListItem("hello","kitty;cat;")        // prints "hello;kitty;cat;"
Print AddListItem("z", "b,c,", ",", 1)         // prints "b,z,c,"
Print AddListItem("z", "b,c,", ",", 999)       // prints "b,c,z,"
Print AddListItem("z", "b,c,", ",", Inf)       // prints "b,c,z,"
Print AddListItem("", "b-c-", "-")             // prints "-b-c-"
```

See Also

The **FindListItem**, **FunctionList**, **ItemsInList**, **RemoveByKey**, **RemoveFromList**, **RemoveListItem**, **StringFromList**, **StringList**, **TraceNameList**, **VariableList**, and **WaveList** functions.

AddMovieAudio

`AddMovieAudio` ***`soundWave`***

The AddMovieAudio operation adds audio samples to the audio track of the currently open movie.

In Igor Pro 7.00 and later, this operation is not supported on Macintosh.

Parameters

soundWave contains audio samples with an amplitude from -128 to +127 and with the same time scale as the prototype *soundWave* used to open the movie.

Details

You can create movies with 16-bit and stereo sound by providing a sound wave in the appropriate format. To specify 16-bit sound, the wave type must be signed 16-bit integer (/W flag in **Make** or **Redimension**). To specify stereo, use a wave with two columns (or any other number of channels as desired).

See Also

Movies on page IV-230.

The **NewMovie** operation.

AddMovieFrame

`AddMovieFrame [/PICT=`*`pictName`*`]`

The AddMovieFrame operation adds the top graph, page layout, Gizmo window, or the specified picture to the currently open movie.

Support for page layout and Gizmo windows was added in Igor Pro 7.00.

When you write a procedure to generate a movie, you need to call the **DoUpdate** operation after all modifications to the target window and before calling AddMovieFrame. This allows Igor to process any changes you have made to the window.

In Igor7 or later, the target window at the time you call NewMovie is remembered and is used by AddMovieFrame even if it is not the target window when you call AddMovieFrame.

If the /PICT flag is provided, then the specified picture from the picture gallery (see **Pictures** on page III-448) is used in place of the target window.

See Also

Movies on page IV-230.

The **NewMovie** operation.

AdoptFiles

`AdoptFiles` `[`***`flags`***`]`

The AdoptFiles operation adopts external files and waves into the current experiment.

When the experiment is next saved, the files and waves are saved in the experiment file for a packed experiment or in the experiment folder for an unpacked experiment. References to the external files are eliminated.

AdoptFiles cannot be called from a function except via Execute/P.

Flags

/A	Adopts all external notebooks and user procedure files and all waves in the experiment. WaveMetrics Procedure files are not adopted. /A is equivalent to /NB/UP/DF.
/DF	Adopts all waves saved external to the experiment.

/DF=*dataFolderPathStr*	Adopts all waves saved external to the experiment that are in the specified data folder.
/I	Shows the Adopt All dialog and adopts what the user selects there.
/NB	Adopts all external notebook files.
/UP	Adopts all external user procedure files.

/W=*winTitleOrName*	Adopts the specified notebook or procedure file. /W was added in Igor Pro 7.02.
	winTitleOrName is a name, not a string, so you construct /W like this:
	`/W=$"New Polar Graph.ipf"`
	or:
	`/W=Notebook0`
	When working with independent modules, *winTitleOrName* is a procedure window title followed by a space and, in brackets, an independent module name. See **Independent Modules** on page IV-224 for details.

/WP	Adopts all WaveMetrics Procedure procedure files.
/WV=*wave*	Adopts only the specified wave.

Details

Only files and waves saved external to the current experiment are adopted. See **References to Files and Folders** on page II-22 for a discussion of such standalone files.

The number of objects actually adopted is returned in V_Flag.

To adopt just one wave, use:

```
AdoptFiles/WV=wave
```

To adopt just one notebook or procedure window use AdoptFiles/W=*winTitleOrName*.

Command Line and Macro Examples

```
// Using AdoptFiles from the command line or from a macro
AdoptFiles/I                    // Show the Adopt All dialog.
AdoptFiles/A/WP                 // Adopt everything that can be adopted.
AdoptFiles/DF/NB/UP/WP          // Adopt everything that can be adopted.
AdoptFiles/DF=root:subfolder    // Adopt any externally saved waves in root:subfolder.
AdoptFiles/W=$"Proc0.ipf"       // Adopt Proc0.ipf if it is saved externally.
AdoptFiles/WV=GetWavesDataFolder(wave0,2)   // Adopt wave0 if it is saved externally.
```

Function Examples

```
// Using AdoptFiles from a user-defined function - you must use Execute/P
Execute/P "AdoptFiles/A"          // Schedule adoption of all user files and waves
Execute/P "AdoptFiles/WV="+GetWavesDataFolder(w,2)    // Schedule adoption of wave w
```

See Also

Adopt All on page II-24, **Adopting Notebook and Procedure Files** on page II-23, **Avoiding Shared Igor Binary Files** on page II-23, **Operation Queue** on page IV-263.

airyA

`airyA(x [, accuracy])`

The airyA function returns the value of the Airy *Ai*(*x*) function:

$$Ai(x) = \frac{1}{\pi}\sqrt{\frac{x}{3}}K_{1/3}\left(\frac{2}{3}x^{3/2}\right),$$

where *K* is the modified Bessel function.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **airyAD** and **airyB** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

airyAD

`airyAD(x [, accuracy])`

The airyAD function returns the value of the derivative of the Airy function.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **airyA** function.

airyB

`airyB(x [, accuracy])`

The airyB function returns the value of the Airy $Bi(x)$ function:

$$Bi(x)=\sqrt{\frac{x}{3}}\left[I_{-1/3}\left(\frac{2}{3}x^{3/2}\right)+I_{1/3}\left(\frac{2}{3}x^{3/2}\right)\right],$$

where I is the modified Bessel function.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **airyBD** and **airyA** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

airyBD

`airyBD(x [, accuracy])`

The airyBD function returns the value of the derivative $Bi'(x)$ of the Airy function.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **airyB** function.

alog

`alog(num)`

The alog function returns 10^{num}.

AnnotationInfo

`AnnotationInfo(winNameStr, annotationNameStr [, options])`

The AnnotationInfo function returns a string containing a semicolon-separated list of information about the named annotation in the named graph or page layout window or subwindow.

The main purpose of AnnotationInfo is to use a tag or textbox as an input mechanism to a procedure. This is illustrated in the “Tags as Markers Demo” sample experiment, which includes handy utility functions (supplied by AnnotationInfo Procs.ipf).

Parameters

winNameStr can be `""` to refer to the top graph or layout window or subwindow.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

options is an optional parameter that controls the text formatting in the annotation output. The default value is 0.

Omit *options* or use 0 for *options* to escape the returned annotation text, which is appropriate for printing the output to the history or for using the text in an Execute operation.

Use 1 for *options* to not escape the returned annotation text because you intend to extract the text for use in a subsequent command such as Textbox or Tag.

Details

The string contains thirteen pieces of information. The first twelve pieces are prefaced by a keyword and colon and terminated with a semicolon. The last piece is the annotation text, which is prefaced with a keyword and a colon but is not terminated with a semicolon.

Keyword	Information Following Keyword
ABSX	X location, in points, of the anchor point of the annotation. For graphs, this is relative to the top-left corner of the graph window. For layouts, it is relative to the top-left corner of the page.
ABSY	Y location, in points, of the anchor point of the annotation. For graphs, this is relative to the top-left corner of the graph window. For layouts, it is relative to the top-left corner of the page.
ATTACHX	For tags, it is the X value of the wave at the point where the tag is attached, as specified with the Tag operation. For textboxes, color scales, and legends, this will be zero and has no meaning.
AXISX	X location of the anchor point of the annotation. For tags or color scales in graphs, it is in terms of the X axis against which the tagged wave is plotted. For textboxes and legends in graphs, it is in terms of the first X axis. For layouts, this has no meaning and is always zero.
AXISY	Y location of the anchor point of the annotation. For layouts, this has no meaning and is always zero. For tags or color scales in graphs, it is in terms of the Y axis against which the tagged wave is plotted. For textboxes and legends in graphs, it is in terms of the first Y axis.
AXISZ	Z value of the image or contour level trace to which the tag is attached or NaN if the trace is not a contour level trace or the annotation is not a tag.
COLORSCALE	Parameters used in a ColorScale operation to create the annotation.
FLAGS	Flags used in a Tag, Textbox, ColorScale, or Legend operation to create the annotation.
RECT	The outermost corners of the annotation (values are in points): RECT:*left*, *top*, *right*, *bottom*
TEXT	Text that defines the contents of the annotation or the main axis label of a color scale.
TYPE	Annotation type: “Tag”, “TextBox”, “ColorScale”, or “Legend”.
XWAVE	For tags, it is the name of the X wave in the XY pair to which the tag is attached. If the tag is attached to a single wave rather than an XY pair, this will be empty. For textboxes, color scales, and legends, this will be empty and has no meaning.
XWAVEDF	For tags, the full path to the data folder containing the X wave associated with the trace to which the tag is attached. For textboxes, color scales, and legends, this will be empty and has no meaning.

Keyword	Information Following Keyword
YWAVE	For tags, it is the name of the trace or image to which the tag is attached. See **ModifyGraph (traces)** and **Instance Notation** on page IV-19 for discussions of trace names and instance notation. For color scales, it is the name of the wave displayed in associated the contour plot, image plot, f(z) trace, or the name of the color scale's cindex wave. For textboxes and legends, this will be empty and has no meaning.
YWAVEDF	Full path to the data folder containing the Y wave or blank if the annotation is not a tag or color scale.

AnnotationList

`AnnotationList(`*`winNameStr`*`)`

The AnnotationList function returns a semicolon-separated list of annotation names from the named graph or page layout window or subwindow.

Parameters

winNameStr can be `""` to refer to the top graph or layout window.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

APMath

`APMath [`*`flags`*`]` ***`destStr = Expression`***

The APMath operation provides arbitrary precision calculation of basic mathematical expressions. It converts the final result into the assigned string *destStr*, which can then be printed or used to represent a value (at the given precision) in another APMath operation.

Parameters

destStr	Specifies a destination string for the assignment expression. If *destStr* is not an existing variable, it is created by the operation. When executing in a function, *destStr* will be a local variable if it does not already exist.
Expression	Algebraic expression containing constants, local, global, and reference variables or strings, as well as wave elements together with the operators shown below.

APMath Operators

`+`	Scalar addition	Lowest precedence
`-`	Scalar subtraction	Lowest precedence
`*`	Scalar multiplication	Medium precedence
`/`	Scalar division	Medium precedence
`^`	Exponentiation	Highest precedence

APMath Functions

`sqrt(`*`x`*`)`	Square root of *x*.
`cbrt(`*`x`*`)`	Cube root of *x*.
`pi`	Value of π (without parentheses).
`sin(`*`x`*`)`	Sine of *x*.
`cos(`*`x`*`)`	Cosine of *x*.
`tan(`*`x`*`)`	Tangent of *x*.
`asin(`*`x`*`)`	Inverse sine of *x*.

`acos(x)`	Inverse cosine of *x*.
atan(x)	Inverse tangent of *x*.
`atan2(y,x)`	Inverse tangent of *y*/*x*.
`log(x)`	Logarithm of *x*.
`log10(x)`	Logarithm based 10 of *x*.
`exp(x)`	Exponential function e^*x*.
`pow(x,n)`	*x* to the power *n* (*n* not necessarily integer).
`sinh(x)`	Hyperbolic sine of *x*.
`cosh(x)`	Hyperbolic cosine of *x*.
`tanh(x)`	Hyperbolic tangent of *x*.
`asinh(x)`	Inverse hyperbolic sine of *x*.
`acosh(x)`	Inverse hyperbolic cosine of *x*.
`atanh(x)`	Inverse hyperbolic tangent of *x*.
`ceil(x)`	Smallest integer larger than *x*.
`comp(x,y)`	Returns 0 for *x* == *y*, 1 if *x* > *y* and -1 if *y* > *x*.
`factorial(n)`	Factorial of integer *n*.
`floor(x)`	Greatest integer smaller than *x*.
`gcd(x,y)`	Greatest common divisor of *x* and *y*.
`lcd(x,y)`	Lowest common denominator of *x* and *y* (given by *x***y*/gcd(*x*,*y*).
`sgn(x)`	Sign of *x* or zero if *x* == 0.

Flags

/EX=*exDigits*	Specifies the number of extra digits added to the precision digits (/N) for intermediate steps in the calculation.
/N=*numDigits*	Specifies the precision of the final result. To add digits to the intermediate computation steps, use /EX.
/V	Verbose mode; prints the result in the history in addition to performing the assignment.
/Z	No error reporting.

Details

By default, all arbitrary precision math calculations are performed with *numDigits*=50 and *exDigits*=6, which yields a final result using at least 56 decimal places. Because none of the built-in variable types can express numbers with such high accuracy, the arbitrary precision numbers must be stored as strings. The operation automatically converts between strings and constants. It evaluates all of the numerical functions listed above using the specified accuracy. If you need functions that are not supported by this operation, you may have to precompute them and store the results in a local variable.

The operation stores the result in *destStr*, which may or may not exist prior to execution. When you execute the operation from the command line, *destStr* becomes a global string in the current data folder if it does not already exist. If it exists, then the result of the operation overwrites its value (as with any normal string assignment). In a user function, *destStr* can be a local string, an SVAR, or a string passed by reference. If *destStr* is not one of these then the operation creates a local string by that name.

Arbitrary precision math calculations are much slower (by a factor of about 300) than equivalent floating point calculations. Execution time is a function of the number of digits, so you should use the /N flag to limit the evaluation to the minimum number of required digits.

Examples

Evaluate pi to 50 digits:

```
APMath/V aa=pi
```

Evaluate ratios of large factorials:

```
APMath/v aa=factorial(500)/factorial(499)
```

Evaluate ratios of large exponentials:

```
APMath/v aa=exp(-1000)/exp(-1001)
```

Division of mixed size values:

```
APMath/v aa=1-sgn(1-(1-0.00000000000000000001234)/(1-0.000000000000000000012345)))
```

you'll get a different result trying to evaluate this using double precision.

Difference between built-in pi and the arbitrary precision pi:

```
Variable/G biPi=pi
APMath/v aa=biPi-pi
```

Precision control:

```
Function test()
   APMath aa=pi                  // Assign 50 digit pi to the string aa.
   APMath/v bb=aa                // Create local string bb equal to aa.
   APMath/v bb=aa-pi             // Subtract arb. prec. pi from aa.
                                 // note the default exDigits=6.
   APMath/v/n=50/ex=0 bb=aa-pi   // setting exDigits=0.
End
```

Numerical recreation:

```
APMath/v/n=16 aa=111111111^2
```

Append

`Append`

The Append operation is interpreted as **AppendToGraph**, **AppendToTable**, or **AppendToLayout**, depending on the target window. This does not work when executing a user-defined function. Therefore we now recommend that you use **AppendToGraph**, **AppendToTable**, or **AppendLayoutObject** rather than Append.

AppendImage

`AppendImage [/G=`*`g`*`/W=`*`winName`*`][`*`axisFlags`*`]` ***`matrix`*** **`[vs {`*`xWaveName, yWaveName`*`}]`**

The AppendImage operation appends the matrix as an image to the target or named graph. By default the image is plotted versus the left and bottom axes.

Parameters

matrix is either an NxM 2D wave for false color or indexed color images, or it can be a 3D NxMx3 wave containing a layer of data for red, a layer for green and a layer for blue. It can also be a 3D NxMx4 wave with the fourth plane containing alpha values.

If *matrix* contains multiple planes other than three or four or if it contains three or four and multiple chunks, the **ModifyImage** plane keyword can be used to specify the desired subset to display.

If you provide *xWaveName* and *yWaveName*, *xWaveName* provides X coordinate values, and *yWaveName* provides Y coordinate values. This makes an image with uneven pixel sizes. In both cases, you can use * to specify calculated values based on the dimension scaling of *matrix*. See **Details** if you use *xWaveName* or *yWaveName*.

Flags

axisFlags	Flags /L, /R, /B, and /T are the same as used by **AppendToGraph**.
/G=*g*	Controls the interpretation of three-plane images as direct RGB. *g*=1 Suppresses the auto-detection of three or four plane images as direct (RGB) color. *g*=0 Same as no /G flag (default).
/W=*winName*	Appends to the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

When appending an image to a graph each image data point is displayed as a rectangle. You can supply optional X and Y waves to define the coordinates of the rectangle edges. *These waves need to contain one more data point than the X (row) or Y (column) dimension of the matrix*. The waves must also be either strictly increasing or strictly decreasing. See **Image X and Y Coordinates** on page II-301 for details.

For false color, the values in the matrix are linearly mapped into a color table. See the ModifyImage ctab keyword. For indexed color, the values in the matrix are interpreted as Z values to be looked up in a user-supplied 3 column matrix of colors. See the ModifyImage cindex keyword. Direct color NxMx3 waves contain the actual red, green, and blue values for each pixel. NxMx4 waves add an alpha channel. If the number type is unsigned bytes, then the range of intensity ranges from 0 to 255. For all other number types, the intensity ranges from 0 to 65535.

By default, nondirect color matrices are initially displayed as false color using the Grays color table and autoscale mode.

If the matrix is complex, the image is displayed in terms of the magnitude of the Z value, that is, sqrt(real^2 + imag^2).

See Also

Image X and Y Coordinates on page II-301, **Color Blending** on page III-440.

The **NewImage**, **ModifyImage**, and **RemoveImage** operations. For general information on image plots see Chapter II-15, **Image Plots**.

AppendLayoutObject

`AppendLayoutObject [`***`flags`***`]` ***`objectType objectName`***

The AppendLayoutObject operation appends a single object to the top layout or to the layout specified via the /W flag. It targets the active page or the page specified by the /PAGE flag.

Unlike the AppendToLayout operation, AppendLayoutObject can be used in user-defined functions. Therefore, AppendLayoutObject should be used in new programming instead of AppendToLayout.

Parameters

objectType identifies the type of object to be appended. It is one of the following keywords: `graph`, `table`, `picture`, `gizmo`.

objectName is the name of the graph, table, picture or Gizmo window to be appended.

Use a space between *objectType* and *objectName*. A comma is not allowed.

Flags

/D=*fidelity*	Draws layout objects in low fidelity (*fidelity*=0) or high fidelity (*fidelity*=1; default). This affects drawing on the screen only, not exporting or printing. Low fidelity is somewhat faster but less accurate and should be used only for graphs that take a very long time to draw.

/F=*frame*	Specifies the type of frame enclosing the object. *frame* =1 Single frame (default). *frame* =2 Double frame. *frame* =3 Triple frame. *frame* =4 Shadow frame.
/T=*trans*	Sets the transparency of the object background to opaque (*trans* =0; default) or transparent (*trans* =1). For transparency to be effective, the object itself must also be transparent. Annotations have their own transparent/opaque settings. Graphs are transparent only if their backgrounds are white. PICTs may have been created transparent or opaque. Opaque PICTs cannot be made transparent.
/R=(*l*, *t*, *r*, *b*)	Sets the size and position of the object. If omitted, the object is placed with a default size and position. *l*, *t*, *r*, and *b* are the left, top, right, and bottom coordinates of the object, respectively. Coordinates are expressed in units of points, relative to the top/left corner of the paper.
/PAGE=*page*	Appends the object to the specified page. Page numbers start from 1. To target the active page, omit /PAGE or use *page*=0. The /PAGE flag was added in Igor Pro 7.00.
/W=*winName*	Appends the object to the named page layout window. If /W is omitted or if *winName* is `$""`, the top page layout is used.

See Also

NewLayout, **ModifyLayout**, **LayoutPageAction**, **RemoveLayoutObjects**, **TextBox**, **Legend**

AppendMatrixContour

AppendMatrixContour [***axisFlags***] [**/F=*formatStr* /W=*winName***] ***zWave***
    [**vs** {***xWave*, *yWave***}]

The AppendMatrixContour operation appends to the target or named graph a contour plot of a matrix of z values with autoscaled contour levels, using the Rainbow color table.

Note: There is no DisplayContour operation. Use `Display; AppendMatrixContour`.

Parameters

zWave must be a matrix (2D wave).

To contour a set of XYZ triplets, use **AppendXYZContour**.

If you provide the *xWave* and *yWave* specification, *xWave* provides X values for the rows, and *yWave* provides Y values for the columns. This results in an “uneven grid” of Z values.

If you omit the *xWave* and *yWave* specification, Igor uses the *zWave*’s X and Y scaled indices as the X and Y values. Igor also uses the *zWave*’s scaled indices if you use * (asterisk symbol) in place of *xWave* or *yWave*.

In a macro, to modify the appearance of contour levels before the contour is calculated and displayed with the default values, append "`;DelayUpdate`" and immediately follow the AppendMatrixContour command with the appropriate **ModifyContour** commands. All but the last ModifyContour command should also have `;DelayUpdate` appended. DelayUpdate is not needed in a function, but DoUpdate is useful in a function to force the contour traces to be built immediately rather than the default behavior of waiting until all functions have completed.

On the command line, the Display command and subsequent AppendMatrixContour commands and any ModifyContour commands can be typed all on one line with semicolons between:

```
Display; AppendMatrixContour MyMatrix; ModifyContour ...
```

Flags

axisFlags	Flags /L, /R, /B, /T are the same as used by **AppendToGraph**.
/F=*formatStr*	Determines the names assigned to the contour level traces. See **Details**.
/W=*winName*	Appends to the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

AppendMatrixContour creates and displays contour level traces. You can modify these all together using the Modify Contour Appearance dialog or individually using the Modify Trace Appearance dialog. In most cases, you will not need to modify the individual traces.

By default, Contour level traces are automatically named with names that show the *zWave* and the contour level, for example, "zWave=1.5". You will see these trace names in the Modify Trace Appearance dialog and in Legends. In most cases, the default trace names will be just fine.

If you want to control the names of the contour level traces (which you might want to do for names in a Legend), use the /F=*formatStr* flag. This flag uses a format string as described for the **printf** operation. The default format string is `"%.17s=%g"`, resulting in trace names such as "zWave=1.5". *formatStr* must contain at least `%f` or `%g` (used to insert the contour level) or `%d` (used to insert the zero-based index of the contour level). Include `%s`, to insert the *zWave* name.

Here are some examples of format strings.

formatStr	Examples of Resulting Name	Format
`"%g"`	"100", "1e6", "-2.05e-2"	(<level>)
`"z=%g"`	"z=100", "z=1e6", "z=-2.05e-2"	(z=<level>)
`"%s %f"`	"zWave 100.000000"	(<wave>, space, <level>)
`"[%d]=%g"`	"[0]=100", "[1]=1e6"	([<index>]=<level>)

Examples

```
Make/O/N=(25,25) w2D                    // Make a matrix
SetScale x -1, 1, w2D                   // Set row scaling
SetScale y -1, 1, w2D                   // Set column scaling
w2D = sin(x) * cos(y)                   // Store values in the matrix
Display; AppendMatrixContour w2D
ModifyContour w2D autoLevels={*,*,9}    // Roughly 9 automatic levels
```

See also

Display, **AppendToGraph**, **AppendXYZContour**, **ModifyContour**, **RemoveContour**, **FindContour**.

For general information on contour plots, see Chapter II-14, **Contour Plots**.

AppendText

AppendText [**/W=*winName*/N/NOCR** [=***n***]] ***textStr***

The AppendText operation appends a carriage return and *textStr* to the most recently created or changed annotation, or to the named annotation in the target or graph or layout window. Annotations include tags, textboxes, color scales, and legends.

Parameters

textStr can contain escape codes to control font, font size and other stylistic variations. See **Annotation Escape Codes** on page III-53 for details.

Flags

/N=*name*	Appends *textStr* to the named tag or textbox.
/NOCR[=*n*]	Omits the initial appending of a carriage return (allows a long line to be created with multiple AppendText commands). /NOCR=0 is the same as no /NOCR, and /NOCR=1 is the same as just /NOCR.
/W=*winName*	Appends to an annotation in the named graph, layout window, or subwindow. Without /W, AppendText appends to an annotation in the topmost graph or layout window or subwindow. This must be the first flag specified when AppendText is used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

A textbox, tag, or legend can contain at most 100 lines. A color scale can have at most one line, and this line is the color scale's main axis label.

See Also

The **Tag**, **TextBox**, **ColorScale**, **ReplaceText**, and **Legend** operations.

Annotation Escape Codes on page III-53.

AppendToGizmo

AppendToGizmo [***flags***] ***keyword*** [=***value***]

The AppendToGizmo operation appends a Gizmo object or attribute operation to the top Gizmo window or to the Gizmo window specified by the /N flag.

Documentation for the AppendToGizmo operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "AppendToGizmo"
```

AppendToGraph

AppendToGraph [***flags***] ***waveName*** [, ***waveName***]…[**vs** ***xwaveName***]

The AppendToGraph operation appends the named waves to the target or named graph. By default the waves are plotted versus the left and bottom axes.

Parameters

The *waveName*s parameters are the names of existing waves.

vs *xwaveName* plots the data values of *waveName*s against the data values of *xwaveName*.

Subsets of data, including individual rows or columns from a matrix, may be specified using **Subrange Display Syntax** on page II-250.

You can provide a custom name for a trace by appending /TN=traceName to the waveName specification. This is useful when displaying waves with the same name but from different data folders. See **User-defined Trace Names** on page IV-82 for more information.

Flags

/B [=*axisName*]	Plots X coordinates versus the standard or named bottom axis.
/C=(*r*,*g*,*b*)	*r*, *g*, and *b* specify the amount of red, green, and blue in the color of the appended waves as an integer from 0 to 65535.
/L [=*axisName*]	Plots Y coordinates versus the standard or named left axis.
/NCAT	Causes trace to be plotted normally on what otherwise is a category plot. X values are just category numbers but can be fractional. Category numbers start from zero. This can be used to overlay the original data points for a box plot. See **Combining Numeric and Category Traces** on page II-273 for details.
/Q	Uses a special, quick update mode when appending to a pair of existing axes. A side effect of this mode is that waves that are appended are marked as not modified. This will prevent other graphs containing these waves, if any, from being updated properly.
/R [=*axisName*]	Plots Y coordinates versus the standard or named right axis.
/T [=*axisName*]	Plots X coordinates versus the standard or named top axis.
/TN=*traceName*	Allows you to provide a custom trace name for a trace. This is useful when displaying waves with the same name but from different data folders. See **User-defined Trace Names** on page IV-82 for details.
/VERT	Plots data vertically. Similar to SwapXY (**ModifyGraph (axes)**) but on a trace-by-trace basis.
/W=*winName*	Appends to the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

See Also

The **Display** operation.

AppendToLayout

`AppendToLayout` `[`***`flags`***`]` ***`objectSpec`*** `[`**`,`** ***`objectSpec`***`]`…

The AppendToLayout operation appends the specified objects to the top layout.

The AppendToLayout operation can not be used in user-defined functions. Use the AppendLayoutObject operation instead.

Parameters

The optional *objectSpec* parameters identify a graph, table, textbox or PICT to be added to the layout. An object specification can also specify the location and size of the object, whether the object should have a frame or not, whether it should be transparent or opaque, and whether it should be displayed in high fidelity or not. See the **Layout** operation for details.

Flags

/G=*g*	Specifies grout, the spacing between tiled objects. Units are points unless /I, /M, or /R are specified.
/I	*objectSpec* coordinates are in inches.
/M	*objectSpec* coordinates are in centimeters.
/R	*objectSpec* coordinates are in percent of printing part of the page.
/S	Stacks objects.

/T	Tiles objects.

See Also

The **Layout** and **AppendLayoutObject** operations for use with user-defined functions.

AppendToTable

`AppendToTable [/W=`*`winName`*`] `*`columnSpec`*` [, `*`columnSpec`*`]…`

The AppendToTable operation appends the specified columns to the top table. *columnSpec*s are the same as for the **Edit** operation; usually they are just the names of waves.

Flags

/W=*winName*	Appends columns to the named table window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

See Also

Edit for details about *columnSpec*s, and **RemoveFromTable**.

AppendXYZContour

`AppendXYZContour [/W=`*`winName`*` /F=`*`formatStr`*`][`*`axisFlags`*`] `*`zWave`*` [vs {`*`xWave, yWave`*`}]`

The AppendXYZContour operation appends to the target or named graph a contour of a 2D wave consisting of XYZ triples with autoscaled contour levels and using the Rainbow color table.

To contour a matrix of Z values, use **AppendMatrixContour**.

Note: There is no DisplayContour operation. Use `Display; AppendXYZContour`.

Parameters

If you provide the *xWave* and *yWave* specification, *xWave* provides X values for the rows, and *yWave* provides Y values for the columns, *zWave* provides Z values and all three waves must be 1D. All must have at least four rows and must have the same number of rows.

If you omit the *xWave* and *yWave* specification, *zWave* must be a 2D wave with 4 or more rows and 3 or more columns. The first column is X, the second is Y, and the third is Z. Any additional columns are ignored.

If any of X, Y, or Z in a row is blank, (NaN), that row is ignored.

In a macro, to modify the appearance of contour levels before the contour is calculated and displayed with the default values, append "`;DelayUpdate`" and immediately follow the AppendXYZContour command with the appropriate **ModifyContour** commands. All but the last ModifyContour command should also have `;DelayUpdate` appended. DelayUpdate is not needed in a function, but DoUpdate is useful in a function to force the contour traces to be built immediately rather than the default behavior of waiting until all functions have completed.

On the command line, the **Display** command and subsequent AppendXYZContour commands and any **ModifyContour** commands can be typed all on one line with semicolons between:

```
Display; AppendXYZContour zWave; ModifyContour ...
```

Flags

axisFlags	Flags /L, /R, /B, and /T are the same as used by **AppendToGraph**.
/F=*formatStr*	Determines names assigned to the contour level traces. This is the same as for **AppendMatrixContour**.
/W=*winName*	Appends to the named graph window or subwindow. When omitted, action affects the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

AppendXYZContour creates and displays contour level traces. You can modify these as a group using the Modify Contour Appearance dialog or individually using the Modify Trace Appearance dialog. In most cases, you will have no need to modify the traces individually.

See **AppendMatrixContour** for a discussion of how the contour level traces are named.

Examples

```
Make/O/N=(100) xW, yW, zW              // Make X, Y, and Z waves
xW = sawtooth(2*PI*p/10)               // Generate X values
yW = trunc(p/10)/10                    // Generate Y values
zW = sin(2*PI*xW)*cos(2*PI*yW)         // Generate Z values
Display; AppendXYZContour zW vs {xW, yW}; DelayUpdate
ModifyContour zW autoLevels={*,*,9}    // roughly 9 automatic levels
```

See Also

The **Display** operation. **AppendToGraph** for details about other axis flags. The **AppendMatrixContour**, **ModifyContour**, and **RemoveContour** operations. For general information on contour plots, see Chapter II-14, **Contour Plots**.

area

area(*waveName* [, *x1*, *x2*])

The area function returns the signed area between the named wave and the line y=0 from x=*x1* to x=*x2* using trapezoidal integration, accounting for the wave's X scaling. If your data are in the form of an XY pair of waves, see **areaXY**.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If *x1* or *x2* are not within the X range of *waveName*, area limits them to the nearest X range limit of *waveName*.

If any values in the X range are NaN, area returns NaN.

The function returns NaN if the input wave has zero points.

Reversing the order of *x1* and *x2* changes the sign of the returned area.

The area function is intended to work on 1D real or complex waves only.

The area function returns a complex result for a complex input wave. The real part of the result is the area of the real components in the input wave, and the imaginary part of the result is the area of the imaginary components.

Examples

```
Make/O/N=100 data; SetScale/I x 0,Pi,data
data=sin(x)
Print area(data,0,Pi)          // the entire X range, and no more
Print area(data)               // same as -infinity to +infinity
Print area(data,Inf,-Inf)      // +infinity to -infinity
```

The following is printed to the history area:

```
Print area(data,0,Pi)          // the entire X range, and no more
  1.99983
Print Print area(data)         // same as -infinity to +infinity
  1.99983
Print area(data,Inf,-Inf)      // +infinity to -infinity
  -1.99983
```

The `-Inf` value was limited to 0 and `Inf` was limited to `Pi` to keep them within the X range of data.

See Also

The figure "Comparison of area, faverage and mean functions over interval (12.75,13.32)", in the **Details** section of the **faverage** function.

Integrate, **areaXY**, **faverage**, **faverageXY**, **PolygonArea**

areaXY

areaXY(*XWaveName, YWaveName* [, *x1, x2*])

The areaXY function returns the signed area between the named *YWaveName* and the line y=0 from x=*x1* to x=*x2* using trapezoidal integration with X values supplied by *XWaveName*.

This function is identical to the **area** function except that it works on an XY wave pair and does not work with complex waves.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If *x1* or *x2* are outside the X range of *XWaveName*, areaXY limits them to the nearest X range limit of *XWaveName*.

If any values in the Y range are NaN, areaXY returns NaN.

If any values in the entire X wave are NaN, areaXY returns NaN.

The function returns NaN if the input wave has zero points.

Reversing the order of *x1* and *x2* changes the sign of the returned area.

If *x1* or *x2* are not found in *XWaveName*, a Y value is found by linear interpolation based on the two bracketing X values and the corresponding values from *YWaveName*.

The values in *XWaveName* may be increasing or decreasing. AreaXY assumes that the values in *XWaveName* are monotonic. If they are not monotonic, Igor does not complain, but the result is not meaningful. If any X values are NaN, the result is NaN.

See the figure “Comparison of area, faverage and mean functions over interval (12.75,13.32)”, in the **Details** section of the **faverage** function.

The areaXY operation is intended to work on 1D waves only.

Examples

```
Make/O/N=101 Xdata, Ydata
Xdata = x*pi/100
Ydata = sin(Xdata[p])
Print areaXY(Xdata, Ydata,0,Pi)         // the entire X range, and no more
Print areaXY(Xdata, Ydata)              // same as -infinity to +infinity
Print areaXY(Xdata, Ydata,Inf,-Inf)     // +infinity to -infinity
```

The following is printed to the history area:

```
Print areaXY(Xdata, Ydata,0,Pi)         // the entire X range, and no more
  1.99984
Print areaXY(Xdata, Ydata)              // same as -infinity to +infinity
  1.99984
Print areaXY(Xdata, Ydata,Inf,-Inf)     // +infinity to -infinity
  -1.99984
```

The `-Inf` value was limited to `0`, and `Inf` was limited to `Pi` to stay within the X range of data.

See Also

Integrate, **area**, **faverage**, **faverageXY**, **PolygonArea**

asin

asin(*num*)

The asin function returns the inverse sine of num in radians in the range [-π/2,π/2].

In complex expressions, *num* is complex, and asin returns a complex value.

See Also

sin

asinh

asinh(*num*)

The asinh function returns the inverse hyperbolic sine of *num*. In complex expressions, *num* is complex, and asinh returns a complex value.

atan

`atan(`*`num`*`)`

The atan function returns the inverse tangent of *num* in radians. In complex expressions, *num* is complex, and atan returns a complex value. Results are in the range -π/2 to π/2.

See Also

tan, **atan2**

atan2

`atan2(`*`y1, x1`*`)`

The atan2 function returns the angle in radians whose tangent is *y1*/*x1*. Results are in the range -π to π.

See Also

tan, **atan**

atanh

`atanh(`*`num`*`)`

The atanh function returns the inverse hyperbolic tangent of *num*. In complex expressions, *num* is complex, and atanh returns a complex value.

AutoPositionWindow

`AutoPositionWindow [/E/M=`*`m`*`/R=`*`relWindow`*`] [`*`windowName`*`]`

The AutoPositionWindow operation positions the window specified by *windowName* relative to the next lower window of the same kind or relative to the window given by the /R flag. If *windowName* is not specified, AutoPositionWindow acts on the target window.

Flags

/E	Uses entire area of the monitor. Otherwise, it takes into account the command window.
/M=*m*	Specifies the window positioning method. *m*=0: Positions *windowName* to the right of the other window, if possible. If there is no room, then it positions *windowName* just below the other window but at the left edge of the display area. If that is not possible, then the position is not affected. *m*=1: Positions *windowName* just under the other window lined up on the left edge, if possible. If there is no room, then it positions *windowName* just to the right of the other window lined up on the bottom edges. If neither are possible then it positions *windowName* as far to the bottom and right as it will go.
/R=*relWindow*	Positions *windowName* relative to *relWindow*.

AxisInfo

`AxisInfo(`*`graphNameStr, axisNameStr`*`)`

The AxisInfo function returns a string containing a semicolon-separated list of information about the named axis in the named graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

axisNameStr is the name of the graph axis.

Details

The string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with a semicolon. The keywords are:

Keyword	Information Following Keyword
AXFLAG	Flag used to select the axis in any of the operations that display waves (Display, AppendMatrixContour, AppendImage, etc.).
AXTYPE	Axis type, such as “left”, “right”, “top”, or “bottom”.
CATWAVE	Wave supplying the categories for the axis if this is a category plot.
CATWAVEDF	Full path to data folder containing category wave.
CWAVE	Name of wave controlling named axis.
CWAVEDF	Full path to data folder containing controlling wave.
HOOK	Name set by ModifyFreeAxis with hook keyword.
ISCAT	Truth that this is a category axis (used in a category plot).
ISTFREE	Truth that this is truly free axis (created via NewFreeAxis).
MASTERAXIS	Name set by ModifyFreeAxis with master keyword.
RECREATION	List of keyword commands as used by ModifyGraph command. The format of these keyword commands is: *keyword*(x)=*modifyParameters*;
SETAXISCMD	Full SetAxis command.
SETAXISFLAGS	Flags that would be used with the SetAxis function to set the particular auto-scaling behavior that the axis uses. If the axis uses a manual axis range, SETAXISFLAGS is blank.
UNITS	Axis units, if any.

The format of the RECREATION information is designed so that you can extract a keyword command from the keyword up to the “`;`”, prepend “`ModifyGraph`”, replace the “`x`” with the name of an actual axis and then **Execute** the resultant string as a command.

Examples

```
Make/O data=x;Display data
Print StringByKey("CWAVE", AxisInfo("","left"))      // prints data
```

See Also

The **StringByKey** and **NumberByKey** functions.

The **GetAxis** and **SetAxis** operations.

The #include <Readback ModifyStr> procedures are useful for parsing strings returned by AxisInfo.

AxisList

`AxisList(`*`graphNameStr`*`)`

The AxisList function returns a semicolon-separated list of axis names from the named graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Examples

```
Make/O data=x;Display/L/T data
Print AxisList("")              // prints left;top;
```

AxisValFromPixel

`AxisValFromPixel(`*`graphNameStr`*`, `*`axNameStr`*`, `*`pixel`*`)`

The AxisValFromPixel function returns an axis value corresponding to the local graph pixel coordinate in the graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

If the specified axis is not found and if the name is "left" or "bottom" then the first vertical or horizontal axis will be used. Sources for *pixel* value may be the GetWindow operation or a user window hook with the mousemoved and mousedown event messages (see the **SetWindow** operation).

If *graphNameStr* references a subwindow, *pixel* is relative to top left corner of base window, not the subwindow.

See Also

The **PixelFromAxisVal** and **TraceFromPixel** functions; the **GetWindow** and **SetWindow** operations.

BackgroundInfo

`BackgroundInfo`

The BackgroundInfo operation returns information about the current unnamed background task.

BackgroundInfo works only with the unnamed background task. New code should used named background tasks instead. See **Background Tasks** on page IV-298 for details.

Details

Information is returned via the following variables:

`V_flag`	0: No background task is defined.
	1: Background task is defined, but not running (is idle).
	2: Background task is defined and is running.
`V_period`	DeltaTicks value set by CtrlBackground. This is how often the background task runs.
`V_nextRun`	Ticks value when the task will run again. 0 if the task is not scheduled to run again.
`S_value`	Text of the numeric expression that the background task executes, as set by SetBackground.

See Also

The **SetBackground**, **CtrlBackground**, **CtrlNamedBackground**, **KillBackground**, and **SetProcessSleep** operations, and the **ticks** function. See **Background Tasks** on page IV-298 for usage details.

Beep

`Beep`

The Beep operation plays the current alert sound (*Macintosh*) or the system beep sound (*Windows*).

Besseli

`Besseli(`*`n`*`,`*`z`*`)`

The Besseli function returns the modified Bessel function of the first kind, I*n*(*z*), of order *n* and argument *z*. Replaces the bessI function, which is supported for backwards compatibility only.

If *z* is real, Besseli returns a real value, which means that if *z* is also negative, it returns NaN unless *n* is an integer.

For complex *z* a complex value is returned, and there are no restrictions on *z* except for possible overflow.

Details

The calculation is performed using the SLATEC library. The function supports fractional and negative orders *n*, as well as real or complex arguments *z*.

See Also

The **Besselj**, **Besselk**, and **Bessely** functions.

Besselj

`Besselj(n,z)`

The Besselj function returns the Bessel function of the first kind, J*n* (*z*), of order *n* and argument *z*. Replaces the bessJ function, which is supported for backwards compatibility only.

If *z* is real, Besselj returns a real value, which means that if *z* is also negative, it returns NaN unless *n* is an integer.

For complex *z* a complex value is returned, and there are no restrictions on *z* except for possible overflow.

Details

The calculation is performed using the SLATEC library. The function supports fractional and negative orders *n*, as well as real or complex arguments *z*.

See Also

The **Besseli**, **Besselk**, and **Bessely** functions.

Besselk

`Besselk(n,z)`

The Besselk function returns the modified Bessel function of the second kind, K*n*(*z*), of order *n* and argument *z*. Replaces the bessK function, which is supported for backwards compatibility only.

If *z* is real, Besselk returns a real value, which means that if *z* is also negative, it returns NaN unless *n* is an integer.

For complex *z* a complex value is returned, and there are no restrictions on *z* except for possible overflow.

Details

The calculation is performed using the SLATEC library. The function supports fractional orders *n*, as well as real or complex arguments *z*.

See Also

The **Besseli**, **Besselj**, and **Bessely** functions.

Bessely

`Bessely(n,z)`

The Bessely function returns the Bessel function of the second kind, Y*n*(*z*), of order *n* and argument *z*. Replaces the bessY function, which is supported for backwards compatibility only.

If *z* is real, Bessely returns a real value, which means that if *z* is also negative, it returns NaN unless *n* is an integer.

For complex *z* a complex value is returned, and there are no restrictions on *z* except for possible overflow.

Details

The calculation is performed using the SLATEC library. The function supports fractional and negative orders *n*, as well as real or complex arguments *z*.

See Also

The **Besseli**, **Besselj**, and **Besselk** functions.

bessI

`bessI(n, x [, algorithm [, accuracy]])`

Obsolete — use **Besseli**.

The bessI function returns the modified Bessel function of the first kind, I*n*(*x*) of order *n* and argument *x*.

For real *x*, the optional parameter *algorithm* selects between a faster, less accurate calculation method and slower, more accurate methods. In addition, when *algorithm* is zero or absent, the order *n* is truncated to an integer.

When *algorithm* is included and is 1, *accuracy* can be used to specify the desired fractional accuracy. See Details about algorithms.

If *x* is complex, a complex result is returned. In this case, *algorithm* and *accuracy* are ignored. The order *n* can be fractional, and must be real.

Details

The *algorithm* parameter has three options, each selecting a different calculation method:

Algorithm	What You Get
0 (default)	Uses a calculation method that has fractional accuracy better than 10^{-6} everywhere and is generally better than 10^{-8}. This method does not handle fractional order *n*; the order is truncated to an integer before the calculation is performed. Algorithm 0 is fastest by a large margin.
1	Allows fractional order. The calculation is performed using methods described in *Numerical Recipes in C*, 2nd edition, pp. 240-245. Using algorithm 1, *accuracy* specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 1e-7 (that is, 10^{-7}), that means that you wish that the absolute value of $(f_{actual} - f_{returned})/f_{actual}$ be better than 10^{-7}. Asking for less accuracy gives some increase in speed. You pay a heavy price for higher accuracy or fractional order. When *algorithm* is nonzero, calculation time is increased by an order of magnitude for small *x*; at larger *x* the penalty is even greater. If accuracy is greater than 10^{-8} and *n* is an integer, algorithm 0 is used. The algorithm calculates bessI and bessK simultaneously. Both values are stored, and if a call to bessI is followed by a call to bessK (or bessK is followed by bessI) with the same *n*, *x*, and *accuracy* the previously-stored value is returned, making the second call very fast.
2	Fractional order is allowed. The calculation is performed using code from the SLATEC library. The accuracy achievable is often better than algorithm 1, but not always. Algorithm 2 is 1.5 to 3 times faster than algorithm 1, but still slower than algorithm 0. The accuracy parameter is ignored.

The achievable accuracy of algorithms 1 and 2 is a complicated function *n* and *x*. To see a summary of achievable accuracies choose File→Example Experiments→Testing and Misc→Bessel Accuracy menu item.

bessJ

bessJ(*n*, *x* [, *algorithm* [, *accuracy*]])

Obsolete — use **Besselj**.

The bessJ function returns the Bessel function of the first kind, J*n*(*x*) of order *n* and argument *x*.

For real *x*, the optional parameter *algorithm* selects between a faster, less accurate calculation method and slower, more accurate methods. In addition, when *algorithm* is zero or absent, the order *n* is truncated to an integer.

When *algorithm* is included and is 1, *accuracy* can be used to specify the desired fractional accuracy. See Details about algorithms.

If *x* is complex, a complex result is returned. In this case, *algorithm* and *accuracy* are ignored. The order *n* can be fractional, and must be real.

Details

See the **bessI** function for details on algorithms, accuracy and speed of execution.

When *algorithm* is 1, pairs of values for bessJ and bessY are calculated simultaneously. The values are stored, and a subsequent call to bessY after a call to bessJ (or vice versa) with the same *n*, *x*, and *accuracy* will be very fast.

bessK

`bessK(`*`n, x`*` [, `*`algorithm`*` [, `*`accuracy`*`]])`

Obsolete — use **Besselk**.

The bessK function returns the modified Bessel function of the second kind, K*n*(*x*) of order *n* and argument *x*.

For real *x*, the optional parameter *algorithm* selects between a faster, less accurate calculation method and slower, more accurate methods. In addition, when *algorithm* is zero or absent, the order *n* is truncated to an integer.

When *algorithm* is included and is 1, *accuracy* can be used to specify the desired fractional accuracy. See Details about algorithms.

If *x* is complex, a complex result is returned. In this case, *algorithm* and *accuracy* are ignored. The order *n* can be fractional, and must be real.

Details

See the **bessI** function for details on algorithms, accuracy and speed of execution.

When *algorithm* is 1, pairs of values for bessJ and bessY are calculated simultaneously. The values are stored, and a subsequent call to bessY after a call to bessJ (or vice versa) with the same *n*, *x*, and *accuracy* will be very fast.

bessY

`bessY(`*`n, x`*` [, `*`algorithm`*` [, `*`accuracy`*`]])`

Obsolete — use **Bessely**.

The bessY function returns the Bessel function of the second kind, Y*n*(*x*) of order *n* and argument *x*.

For real *x*, the optional parameter *algorithm* selects between a faster, less accurate calculation method and slower, more accurate methods. In addition, when *algorithm* is zero or absent, the order *n* is truncated to an integer.

When *algorithm* is included and is 1, *accuracy* can be used to specify the desired fractional accuracy. See Details about algorithms.

If *x* is complex, a complex result is returned. In this case, *algorithm* and *accuracy* are ignored. The order *n* can be fractional, and must be real.

Details

See the **bessI** function for details on algorithms, accuracy and speed of execution.

When *algorithm* is 1, pairs of values for bessJ and bessY are calculated simultaneously. The values are stored, and a subsequent call to bessY after a call to bessJ (or vice versa) with the same *n*, *x*, and *accuracy* will be very fast.

beta

`beta(`*`a, b`*`)`

The beta function returns for real or complex arguments as

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)},$$

with *Re*(*a*), *Re*(*b*)>0. Γ is the gamma function.

See Also

The **gamma** function.

betai

`betai(`*`a, b, x`*` [, `*`accuracy`*`])`

The betai function returns the regularized incomplete beta function I*x*(*a*,*b*),

$$I_x(a,b) = \frac{B(x;a,b)}{B(a,b)}.$$

Here

$$B(x;a,b) = \int_0^x t^{a-1}(1-t)^{b-1}\,dt.$$

where $a,b > 0$, and $0 \le x \le 1$.

Optionally, *accuracy* can be used to specify the desired fractional accuracy.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of $(f_{actual} - f_{returned})/f_{actual}$ be less than 10^{-7}.

Larger values of *accuracy* (poorer accuracy) result in evaluation of fewer terms of a series, which means the function executes somewhat faster.

A single-precision level of accuracy is about 3×10^{-7}, double-precision is about 2×10^{-16}. The betai function will return full double-precision accuracy for small values of *a* and *b*. Achievable accuracy declines as *a* and *b* increase:

a	*b*	*x*	**betai**	**Accuracy Achievable**
1	1.5	0.5	0.646447	2×10^{-16} (full double precision)
8	10	0.5	0.685470	6×10^{-16}
20	21	0.5	0.562685	2×10^{-15}
20	21	0.1	1.87186×10^{-10}	5×10^{-15}

BinarySearch

`BinarySearch(`*`waveName, val`*`)`

The BinarySearch function performs a binary search of the one-dimensional *waveName* for the value *val*. BinarySearch returns an integer point number p such that *waveName*[p] and *waveName*[p+1] bracket *val*. If *val* is in *waveName*, then *waveName*[p]==*val*.

Details

BinarySearch is useful for finding the point in an XY pair that corresponds to a particular X coordinate.

WaveName must contain monotonically increasing or decreasing values.

BinarySearch returns -1 if *val* is not within the range of values in the wave, but would numerically be placed before the first value in the wave.

BinarySearch returns -2 if *val* is not within the range of values in the wave, but would fall after the last value in the wave.

BinarySearch returns -3 if the wave has zero points.

Examples

```
Make/O data = {1, 2, 3.3, 4.9}       // Monotonic increasing
Print BinarySearch(data,3)           // Prints 1
// BinarySearch returns 1 because data[1] <= 3 < data[2].
```

```
Make/O data = {9, 4, 3, -6}          // Monotonic decreasing
Print BinarySearch(data,2.5)         // Prints 2
// BinarySearch returns 2 because data[2] >= 2.5 > data[3].
Print BinarySearch(data,10)          // Prints -1, precedes first value
Print BinarySearch(data,-99)         // Prints -2, beyond last value
```

See Also

The **BinarySearchInterp** and **FindLevel** operations. See **Indexing and Subranges** on page II-71.

BinarySearchInterp

`BinarySearchInterp(`*`waveName, val`*`)`

The BinarySearchInterp function performs a binary interpolated search of the named wave for the value *val*. The returned value, pt, is a floating-point point index into the named wave such that *waveName*[pt] == *val*.

Details

BinarySearchInterp is useful for finding the point in an XY pair that corresponds to a particular X coordinate.

WaveName must contain monotonically increasing or decreasing values.

When the named wave does not actually contain the value *val*, BinarySearchInterp locates a value below *val* and a value above *val* and uses reverse linear interpolation to figure out where *val* would fall if a straight line were drawn between them. It includes that fractional amount in the resulting point index.

BinarySearchInterp returns NaN if *val* is not within the range of values in the wave.

Examples

```
Make/O data = {1, 2, 3.3, 4.9}          // Monotonic increasing
Print BinarySearchInterp(data,3)        // Prints 1.76923
Print data[1.76923]                     // Prints 3

Make/O data = {9, 4, 3, 1}              // Monotonic decreasing
Print BinarySearchInterp(data,2.5)      // Prints 2.25
Print data[2.25]                        // Prints 2.5
```

See Also

The **BinarySearch** and **FindLevel** operations. See **Indexing and Subranges** on page II-71.

binomial

`binomial(`*`n`*`, `*`k`*`)`

The binomial function returns the ratio:

$$\frac{n!}{k!(n-k)!}$$

It is assumed that *n* and *k* are integers and $0 \le k \le n$ and ! denotes the factorial function.

Note that although the binomial function is an integer-valued function, a double-precision number has 53 bits for the mantissa. This means that numbers over 2^{52} (about 4.5x10^{15}) will be accurate to about one part in 2x10^{16}.

binomialln

`binomialln(`*`a`*`, `*`b`*`)`

The binomialln function returns the natural log of the binomial coefficient for *a* and *b*.

$$\text{binomialln}(a,b) = \ln(a!) - \ln(b!) - \ln((a-b)!)$$

See Also

Chapter III-12, **Statistics** for an overview of the various functions and operations; **binomial**, **StatsBinomialPDF**, **StatsBinomialCDF**, and **StatsInvBinomialCDF**.

binomialNoise

`binomialNoise(`*`n`*`, `*`p`*`)`

The binomialNoise function returns a pseudo-random value from the binomial distribution

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}, \qquad \begin{matrix} 0 \le p \le 1 \\ x = 1,2,\ldots n \end{matrix}$$

whose mean is *np* and variance is *np*(1-*p*).

When *n* is large such that p^n is zero to machine accuracy the function returns NaN. When *n* is large such that *np*(1-*p*)>5 and 0.1<*p*<0.9 you can replace the binomial variate with a normal variate with mean *np* and standard deviation sqrt(*n***p**(1-*p*)).

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat the same sequence. For repeatable “random” numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-344.

Chapter III-12, **Statistics** for an overview of the various functions and operations.

BoundingBall

`BoundingBall [/F/Z]` ***`scatterWave`***

The BoundingBall operation calculates a bounding circle or the bounding sphere for a set of scatter points. The operation accepts 2D waves that have two, three or more columns; data in the additional columns are ignored.

When *scatterWave* consists of two columns the operation computes the bounding circle. Otherwise it computes the bounding 3D sphere.

Parameters

scatterWave is a two-dimensional wave with X coordinates in column 0, Y in column 1, and optional Z coordinates in column 2.

Flags

/F	This flag applies to 3D scatter only. It uses an algorithm from “An Efficient Bounding Sphere” by Jack Ritter originally from *Graphics Gems*. Unfortunately it does not give an accurate bounding ball but something that is sufficiently large. This algorithm is less accurate but it produces a ball which is sufficiently large to contain all the points.
/Z	No error reporting.

Details

The center and radius of the bounding sphere are stored in the variables: V_CenterX, V_CenterY, V_CenterZ, and V_Radius.

If you are not using the /F flag, the operation also accepts a 2 column wave consisting of X, Y pairs for calculating the center and radius of a bounding circle in the plane.

Example

```
Make/N=(33,2) ddd=enoise(4)              // Create random data
BoundingBall ddd
Display ddd[][1] vs ddd[][0]
ModifyGraph mode=3
Make/n=360 xxx,yyy
yyy=v_centerY+V_radius*cos(p*2*pi/360)
xxx=v_centerX+V_radius*sin(p*2*pi/360)
AppendToGraph yyy vs xxx
```

References

Glassner, Andrew S., (Ed.), *Graphics Gems*, 833 pp., Academic Press, San Diego, 1990.

break

`break`

The break flow control keyword immediately terminates execution of a loop, switch, or strswitch. Execution then continues with code following the loop, switch, or strswitch.

See Also

Break Statement on page IV-44, **Switch Statements** on page IV-41, and **Loops** on page IV-42 for usage details.

BrowseURL

`BrowseURL [/Z]` ***`urlStr`***

The BrowseURL operation opens the Web browser or FTP browser on your computer and asks it to display a particular Web page or to connect to an FTP server.

BrowseURL sets a variable named V_flag to zero if the operation succeeds and to nonzero if it fails. This, in conjunction with the /Z flag, can be used to allow procedures to continue to execute if an error occurs.

Parameters

urlStr specifies a Web page or FTP server directory to be browsed. It is constructed of a naming scheme (e.g., “http://” or “ftp://”), a computer name (e.g., “www.wavemetrics.com” or “ftp.wavemetrics.com” or “38.170.234.2”), and a path (e.g., “/Test/TestFile1.txt”). See **Examples** for sample usage.

Flags

/Z	Errors are not fatal. Will not abort procedure execution if the URL is bad or if the server is down. Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded or nonzero if it failed. Syntactic errors, such as omitting the URL altogether or omitting quotes, are still fatal.

Examples

```
// Browse a Web page.
   String url = "http://www.wavemetrics.com/News/index.html"
   BrowseURL url
// Browse an FTP server.
   String url = "ftp://ftp.wavemetrics.com/pub/test"
   BrowseURL url
```

See Also

URLRequest

BuildMenu

`BuildMenu` ***`menuNameStr`***

The BuildMenu operation rebuilds the user-defined menu items in the specified menu the next time the user clicks in the menu bar.

Parameters

menuNameStr is a string expression containing a menu name or "`All`".

Details

Call BuildMenu when you’ve defined a custom menu using string variables for the menu items. After you change the string variables, call BuildMenu to update the menu.

BuildMenu "`All`" rebuilds all the menu items and titles and updates the menu bar.

Under the current implementation, if menuNameStr is not "`All`", Igor will rebuild *all* user-defined menu items if BuildMenu is called for *any* user-defined menu.

See Also

Dynamic Menu Items on page IV-120.

Button

`Button [/Z]` ***`ctrlName`*** **`[`*****`keyword = value`*** **`[,`** ***`keyword = value`*** **…]]**

The Button operation creates or modifies the named button control.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

name is the name of the Button control to be created or changed.

appearance={*kind* [, *platform*]}	Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind=default:	Appearance determined by **DefaultGUIControls**.
kind=native:	Creates standard-looking controls for the current computer platform.
kind=os9:	Igor Pro 5 appearance (quasi-Macintosh OS 9 controls that look the same on Macintosh and Windows).
platform=Mac:	Changes the appearance of controls only on Macintosh; affects the experiment whenever it is used on Macintosh.
platform=Win:	Changes the appearance of controls only on Windows; affects the experiment whenever it is used on Windows.
platform=All:	Changes the appearance on both Macintosh and Windows computers.

disable=*d* — Sets the state of the control. *d* is a bit field: bit 0 (the least significant bit) is set when the control is hidden. Bit 1 is set when the control is disabled:

d=0:	Normal (visible), enabled.
d=1:	Hidden.
d=2:	Visible and disabled. Drawn in grayed state, also disables action procedure.
d=3:	Hidden and disabled.

See the **ModifyControl** example for setting the bits individually.

fColor=(*r*,*g*,*b*) — Sets color of the button. *r*, *g*, and *b* are integers from 0 to 65535. To set the color of the title text, use escape sequences as described below for title. fColor defaults to black (0,0,0). To set the color of the title text, see valueColor.

focusRing=*fr* — Enables or disables the drawing of a rectangle indicating keyboard focus:

fr=0:	Focus rectangle will not be drawn.
fr=1:	Focus rectangle will be drawn (default).

On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.

font="*fontName*" — Sets button font, e.g., `font="Helvetica"`.

fsize=*s* — Sets font size.

fstyle=*fs* — Specifies the font style. *fs* is a bitwise parameter with each bit controlling one aspect of the font style:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*} — Specifies the help for the control. Help text is limited to a total of 255 bytes. On Macintosh, help appears if you turn Igor Tips on. On Windows, help for the first 127 bytes or up to the first line break appears in the status line. If you press F1 while the cursor is over the control, you will see the entire help text. You can insert a line break by putting “`\r`” in a quoted string.

noproc — No procedure is executed when clicking the button.

picture= *pict*	Draws the button using the named picture. The picture is taken to be three side-by-side frames that show the control appearance in the normal state, when the mouse is down, and in the disabled state. The picture may be either a global (imported) picture or a Proc Picture (see **Proc Pictures** on page IV-53). In Igor6, the size keyword is ignored when a picture is used with a button control. To make it easier to size graphics for high-resolution screens, as of Igor7, the size keyword is respected in this case.
pos={*left*,*top*}	Sets the position of the button in pixels.
pos+={*dx*,*dy*}	Offsets the position of the button in pixels.
proc=*procName*	Names the procedure to execute when clicking the button.
rename=*newName*	Gives the button a new name.
size={*width*,*height*}	Sets *width* and *height* of button in pixels.
title=*titleStr*	Sets title of button (text that appears in the button) to the specified string expression. If not given then title will be “New”. If you use `""` the button will contain no text. Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.
userdata(*UDName*) =*UDStr*	Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.
userdata(*UDName*) +=*UDStr*	Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.
valueColor=(*r*,*g*,*b*)	Sets initial color of the button's text (title). *r*, *g*, and *b* range from 0 to 65535. valueColor defaults to black (0,0,0). To further change the color of the title text, use escape sequences as described for title=*titleStr*.
win=*winName*	Specifies which window or subwindow contains the named button control. If not given, then the top-most graph or panel window or subwindow is assumed. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The target window must be a graph or panel.

Button Action Procedure

The action procedure for a Button control takes a predefined structure `WMButtonAction` as a parameter to the function:

```
Function ActionProcName(B_Struct) : ButtonControl
   STRUCT WMButtonAction &B_Struct
   …
   return 0
End
```

The “`: ButtonControl`” designation tells Igor to include this procedure in the Procedure pop-up menu in the Button Control dialog.

See **WMButtonAction** for details on the WMButtonAction structure.

Although the return value is not currently used, action procedures should always return zero.

You may see an old format button action procedure in old code:

```
Function procName(ctrlName) : ButtonControl
   String ctrlName
   …
   return 0
End
```

This old format should not be used in new code.

See Also

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

ButtonControl

`ButtonControl`

ButtonControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined button control. See **Procedure Subtypes** on page IV-193 for details. See **Button** for details on creating a button control.

cabs

`cabs(`*`z`*`)`

The cabs function returns the real-valued absolute value of complex number *z*.

See Also

The **magsqr** function.

CameraWindow

`CameraWindow`

CameraWindow is a procedure subtype keyword that identifies a macro as being a camera window recreation macro. It is automatically used when Igor creates a window recreation macro for a camera window. See **Procedure Subtypes** on page IV-193 and **Saving and Recreating Graphs** on page II-261 for details.

CaptureHistory

`CaptureHistory(`*`refnum, stopCapturing`*`)`

The CaptureHistory function returns a string containing text from the history area of the command window since a matching call to the **CaptureHistoryStart** function.

Parameters

refnum is a number returned from a call to CaptureHistoryStart. It identifies the starting point in the history for the returned string.

Set *stopCapturing* to nonzero to indicate that no more history should be captured for the given refnum. Subsequent calls to CaptureHistory with the same *refnum* will result in an error.

Set *stopCapturing* to zero to retrieve history text captured so far. Further calls to CaptureHistory with the same reference number will return this text, plus any additional history text added subsequently.

Details

You can have multiple captures active at one time. Each call to CaptureHistoryStart will return a unique reference number identifying a start point in the history. The capture corresponding to each reference number can be terminated at any time, regardless of the order of the CaptureHistoryStart calls.

CaptureHistoryStart

`CaptureHistoryStart()`

The CaptureHistoryStart function returns a reference number to identify a starting point in the history area text. Subsequently, the CaptureHistory function can be used to retrieve captured history text. See **CaptureHistory** for details.

catch

`catch`

The catch flow control keyword marks the beginning of code in a try-catch-entry flow control construct for handling any abort conditions.

See Also

The **try-catch-endtry** flow control statement for details.

cd

`cd` *`dataFolderSpec`*

The cd operation sets the current data folder to the specified data folder. It is identical to the longer-named SetDataFolder operation.

cd is named after the UNIX "change directory" command.

See Also

SetDataFolder, **pwd**, **Dir**, **Data Folders** on page II-99

CDFFunc

`CDFFunc`

CDFFunc is a procedure subtype keyword that identifies a function as being suitable for calling from the **StatsKSTest** operation.

ceil

`ceil(`*`num`*`)`

The ceil function returns the closest integer greater than or equal to *num*.

See Also

The **round**, **floor**, and **trunc** functions.

cequal

`cequal(`*`z1, z2`*`)`

The cequal function determines the equality of two complex numbers *z1* and *z2*. It returns 1 if they are equal, or 0 if not.

This is in contrast to the == operator, which compares only the real components of *z1* and *z2*, ignoring the imaginary components.

Examples

```
Function TestComplexEqualities()
   Variable/C z1= cmplx(1,2), z2= cmplx(1,-2)
   // This test compares only the real parts of z1 and z2:
   if( z1 == z2 )
      Print "== match"
   else
      Print "no == match"
   endif
   // This test compares both real and imaginary parts of z1 and z2:
   if( cequal(z1,z2) )
      Print "cequal match"
   else
      Print "no cequal match"
   endif
End
```

```
•TestComplexEqualities()
  == match
  no cequal match
```

See Also

The **imag**, **real**, and **cmplx** functions.

char2num

`char2num(`*`str`*`)`

The char2num function returns a numeric code representing the first byte of *str* or the first character of *str*.

If *str* contains zero bytes, char2num returns NaN.

If *str* contains exactly one byte, char2num returns the value of that byte, treated as a signed byte. For backward compatibility with Igor6, if the input is a single byte in the range 0x80..0xFF, char2num returns a negative number.

If *str* contains more than one byte, char2num returns a number which is the Unicode code point for the first character in *str* treated as UTF-8 text. If *str* does not start with a valid UTF-8 character, char2num returns NaN.

Prior to Igor Pro 7.00, char2num always returned the value of the first byte, treated as a signed byte.

Examples

```
Function DemoChar2Num()
   String str

   str = "A"
   Printf "Single ASCII character: %02X\r", char2num(str)          // Prints 0x41

   str = "ABC"
   Printf "Multiple ASCII characters: %02X\r", char2num(str)       // Prints 0x41

   str = U+2022        // Bullet character
   Printf "Valid UTF-8 text: U+%04X\r", char2num(str)              // Prints U+2022
   Printf "First byte value: %02X\r", char2num(str[0]) & 0xFF      // Prints E2
   Printf "Second byte value: %02X\r", char2num(str[1]) & 0xFF     // Prints 80

   str = num2char(0xE2, 0) + num2char(0x41, 0)                     // Invalid UTF-8 text
   Printf "Invalid UTF-8 text: U+%04X\r", char2num(str)            // Prints NaN

   str = ""
   Printf "Empty string: %g\r", char2num(str)                      // Prints NaN
End
```

See Also

The **num2char**, **str2num** and **num2str** functions.

Text Encodings on page III-409.

Chart

`Chart` `[/Z]` ***`ctrlName`*** `[`***`keyword = value`*** `[,` ***`keyword = value`*** `…]]`

The Chart operation creates or modifies a chart control. Charts are generally used in conjunction with data acquisition. Charts do not have to be connected to a FIFO, but they are not useful until they are.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the Chart control to be created or changed.

The following keyword=value parameters are supported:

chans={*ch#*, *ch#*,…}	List of FIFO channel numbers that Chart is to monitor.
color(*ch#*)=(*r*,*g*,*b*)	Sets the color of the specified trace. *r, g,* and *b* specify the amount of red, green, and blue in the color as an integer from 0 to 65535.
ctab=*colortableName*	When a channel is connected to an image strip FIFO channel, the data is displayed as an image using this built-in color table. Valid names are the same as used in images. Invalid name will result in the default Grays color table being used.

disable=*d*	Sets user editability of the control. *d*=0: Normal. *d*=1: Hide. *d*=2: Disable user input. Charts do not change appearance because they are read-only. When disabled, the hand cursor is not shown.
fbkRGB=(*r*,*g*,*b*)	Sets frame background color. *r*, *g* and *b* are integers from 0 to 65535.
fgRGB=(*r*,*g*,*b*)	Sets foreground color (text, etc.). *r*, *g* and *b* are integers from 0 to 65535
fifo=*FIFOName*	Sets which named FIFO the chart will monitor. See the **NewFIFO** operation.
font="*fontName*"	Sets the font used in the chart, e.g., `font="Helvetica"`.
fsize=*s*	Sets font size for chart.
fstyle=*fs*	Specifies the font style. *fs* is a bitwise parameter with each bit controlling one aspect of the font style: Bit 0: Bold Bit 1: Italic Bit 2: Underline Bit 4: Strikethrough See **Setting Bit Parameters** on page IV-12 for details about bit settings.
gain(*ch#*)=*g*	Sets the display gain *g* of the specified channel relative to nominal. Values greater than unity expand the display.
gridRGB=(*r*,*g*,*b*)	Sets grid color. *r*, *g*, and *b* are integers from 0 to 65535.
help={*helpStr*}	Specifies help for the control. Help text is limited to a total of 255 bytes. On Macintosh, help appears if you turn Igor Tips on. On Windows, help for the first 127 bytes or up to the first line break appears in the status line. If you press F1 while the cursor is over the control, you will see the entire help text. You can insert a line break by putting "`\r`" in a quoted string.
jumpTo=*p*	Jumps to point number *p*. This works in review mode only.
lineMode(*ch#*)=*lm*	Sets the display line mode for the given channel. *lm*=0: Dots mode. Draws values as dots. However, if the number of dots in a strip exceeds maxDots then Igor draws a vertical line from the min to the max of the values packed into the strip. *lm*=1: Lines mode. Draws a vertical line encompassing the min and the max of the points in a given strip along with the last point of the preceding strip. Since which strip is the preceding strip depends on the direction of motion then the appearance may slightly shift depending on which direction the chart is moving. *lm*=2: Dots mode. Draws values as dots. However, if the number of dots in a strip exceeds maxDots then Igor draws a vertical line from the min to the max of the values packed into the strip.
mass=*m*	Sets the "feel" of the chart paper when you move it with the mouse. The larger the mass *m*, the slower the chart responds. Odd values cause the movement of the paper to stop the instant the mouse is clicked while even values continue with the illusion of mass.
maxDots=*md*	Controls whether points in a given vertical strip of the chart are displayed as dots or as a solid line. See lineMode above. Default is 20.

offset(*ch#*)=*o*	Sets the display offset of the specified channel. The offset value *o* is subtracted from the data before the gain is applied.
oMode=*om*	Chart operation mode. *om*=0: Live mode. *om*=1: Review mode.
pbkRGB=(*r*,*g*,*b*)	Sets plot area background color. *r*, *g*, and *b* are integers from 0 to 65535.
ppStrip=*pps*	Number of data points packed into each vertical strip of the chart.
rSize(*ch#*)=*rs*	Sets the relative vertical size allocated to the given channel. Nominal is unity. If the value of *rs* is zero then this channel shares space with the previous channel.
sMode=*sm*	Status line mode. *sm*=0: Turns off fancy status line and positioning bar. *sm*=1: Normal mode. *sm*=2: Uses alternate style for bar.
sRate=*sr*	Sets the scroll rate (vertical strips/second). If the chart control is in review mode negative numbers scroll in reverse.
title=*titleStr*	Specifies the chart title. Use `""` for no title.
uMode=*um*	Status line mode. *um*=1: Fast update with no bells and whistles. *um*=2: Status line and positioning bar. *um*=3: Status line, positioning bar, and animated pens.
win=*winName*	Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Flags

/Z	No error reporting.

Details

The target window must be a graph or panel.

The action of some of the Chart keywords depends on whether or not data acquisition is taking place. If the chart is in review mode then all keywords cause the chart to be redrawn. If data acquisition is taking place and the chart is in live mode then some keywords affect new data but do not attempt to update the part of the "paper" that has already been drawn. The following keywords affect only new data during live mode:

`ppStrip, maxDots, gain, offset, color, lineMode`

See Also

Charts on page III-368 and **FIFOs and Charts** on page IV-291.

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

chebyshev

`chebyshev(`*`n, x`*`)`

The chebyshev function returns the Chebyshev polynomial of the first kind and of degree *n*.

The Chebyshev polynomials satisfy the recurrence relation:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

with:
$$T_0(x) = 1$$
$$T_1(x) = x$$
$$T_2(x) = 2x^2 - 1.$$

The orthogonality of the polynomial is expressed by the integral:

$$\int_{-1}^{1} \frac{T_n(x)T_m(x)}{\sqrt{1-x^2}} dx = \begin{cases} 0 & m \neq n \\ \pi/2 & m = n \neq 0 \\ \pi & m = m = 0 \end{cases} .$$

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

See Also

chebyshevU.

chebyshevU

`chebyshevU(`*`n, x`*`)`

The chebyshevU function returns the Chebyshev polynomial of the second kind, degree *n* and argument *x*.

The Chebyshev polynomial of the second kind satisfies the recurrence relation

```
U(n+1,x)=2xU(n,x)-U(n-1,x)
```

which is also the recurrence relation of the Chebyshev polynomials of the first kind.

The first 10 polynomials of the second kind are:

```
U(0,x)=1
U(1,x)=2x
U(2,x)=4x^2-1
U(3,x)=8x^3-4x
U(4,x)=16x^4-12x^2+1
U(5,x)=32x^5-32x^3+6x
U(6,x)=64x^6-80x^4+24x-1
U(7,x)=128x^7-192x^5+80x^3-8x
U(8,x)=256x^8-448x^6+240x^4-40x^2+1
U(9,x)512x^9-1024x^^7+672x^5-160x^3+10x
```

See Also

The **chebyshev** function.

CheckBox

`CheckBox` `[/Z]` ***`ctrlName`*** `[`***`keyword = value`*** `[,` ***`keyword = value`*** `…]]`

The CheckBox operation creates or modifies a checkbox, radio button or disclosure triangle in the target or named window, which must be a graph or control panel.

ctrlName is the name of the checkbox.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the CheckBox control to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}	Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings. *kind* can be one of `default`, `native`, or `os9`. *platform* can be one of `Mac`, `Win`, or *All*. See **Button** and **DefaultGUIControls** for more appearance details.
disable=*d*	Sets user editability of the control. *d*=0: Normal. *d*=1: Hide. *d*=2: Disable user input.
fsize=*s*	Sets font size for checkbox.
fColor=(*r*,*g*,*b*)	Sets the initial color of the title. *r*, *g*, and *b* range from 0 to 65535. fColor defaults to black (0,0,0). To further change the color of the title text, use escape sequences as described for title=*titleStr*.
focusRing=*fr*	Enables or disables the drawing of a rectangle indicating keyboard focus: *fr*=0: Focus rectangle will not be drawn. *fr*=1: Focus rectangle will be drawn (default). On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.
help={*helpStr*}	Sets the help for the control. The help text is limited to a total of 255 bytes. You can insert a line break by putting "`\r`" in a quoted string.
mode=*m*	Specifies checkbox appearance. *m*=0: Default checkbox appearance. *m*=1: Display as a radio button control. *m*=2: Display as a disclosure triangle (*Macintosh*) or treeview expansion node (*Windows*).
noproc	Specifies that no procedure is to execute when clicking the checkbox.
picture= *pict*	Draws the checkbox using the named picture. The picture is taken to be six side-by-side frames which show the control appearance in the normal state, when the mouse is down, and in the disabled state. The first three frames are used when the checked state is false and the next three show the true state. The picture may be either a global (imported) picture or a Proc Picture (see **Proc Pictures** on page IV-53).
pos={*left*,*top*}	Sets the position of the checkbox in pixels.
pos+={*dx*,*dy*}	Offsets the position of the checkbox in pixels.
proc=*procName*	Specifies the procedure to execute when the checkbox is clicked.
rename=*newName*	Renames the checkbox to *newName*.
side=s	Sets the location of the title relative to the box: s =0: Checkbox is on the left, title is on the right (default). s =1: Checkbox is on the right, title is on the left.
size={*width*,*height*}	Sets checkbox size in pixels.

title=*titleStr*	Sets title of checkbox to the specified string expression. The title is the text that appears in the checkbox. If not given or if `""` then the title will be “New”. Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.
userdata(*UDName*)=*UDStr*	Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.
userdata(*UDName*)+=*UDStr*	Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.
value=*v*	Specifies whether the checkbox is selected (*v*=1) or not (*v*=0).
variable= *varName*	Specifies a global numeric variable to be set to the current state of a checkbox whenever it is clicked or when it is set by the value parameter. The variable is two-way: setting the variable also changes the state of the checkbox.
win=*winName*	Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Flags

/Z	No error reporting.

Details

The target window must be a graph or panel.

Checkbox Action Procedure

The action procedure for a CheckBox control can takes a predefined structure `WMCheckboxAction` as a parameter to the function:

```
Function ActionProcName(CB_Struct) : CheckBoxControl
   STRUCT WMCheckboxAction &CB_Struct
   …
   return 0
End
```

The “`: CheckboxControl`” designation tells Igor to include this procedure in the Procedure pop-up menu in the Checkbox Control dialog.

See **WMCheckboxAction** for details on the WMCheckboxAction structure.

Although the return value is not currently used, action procedures should always return zero.

You may see an old format checkbox action procedure in old code:

```
Function procName(ctrlName,checked) : CheckBoxControl
   String ctrlName
   Variable checked       // 1 if selected, 0 if not
   …
   return 0
End
```

This old format should not be used in new code.

When using radio button controls, it is the responsibility of the Igor programmer to turn off other radio buttons when one of a group of radio buttons is pressed.

Examples

The following code is an example of how to program such a group.

```
Window Panel0() : Panel
   PauseUpdate; Silent 1         // building window …
   NewPanel /W=(150,50,353,212)
   Variable/G gRadioVal= 1
   CheckBox check0,pos={52,25},size={78,15},title="Radio 1"
   CheckBox check0,value=1,mode=1,proc=MyCheckProc
   CheckBox check1,pos={52,45},size={78,15},title="Radio 2"
```

```
    CheckBox check1,value=0,mode=1,proc=MyCheckProc
    CheckBox check2,pos={52,65},size={78,15},title="Radio 3"
    CheckBox check2,value= 0,mode=1,proc=MyCheckProc
EndMacro

Function MyCheckProc(cb) : CheckBoxControl
    STRUCT WMCheckboxAction& cb

    NVAR gRadioVal= root:gRadioVal

    strswitch (cb.ctrlName)
        case "check0":
            gRadioVal= 1
            break
        case "check1":
            gRadioVal= 2
            break
        case "check2":
            gRadioVal= 3
            break
    endswitch
    CheckBox check0,value= gRadioVal==1
    CheckBox check1,value= gRadioVal==2
    CheckBox check2,value= gRadioVal==3

    return 0
End
```

See Also

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

CheckBoxControl

`CheckBoxControl`

CheckBoxControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined checkbox control. See **Procedure Subtypes** on page IV-193 for details. See **CheckBox** for details on creating a checkbox control.

CheckDisplayed

`CheckDisplayed [/A/W]` ***`waveName`*** **`[,`** ***`waveName`*** **`]`**…

The CheckDisplayed operation determines if named waves are displayed in a host window or subwindow.

Flags

/A	Checks all graph and table windows
/W=*winName*	Checks only the named graph or table window When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

If neither /A nor /W are used, CheckDisplayed checks only the top graph or table.

CheckDisplayed sets a bit in the variable V_flag for each wave that is displayed.

Examples

```
CheckDisplayed/W=Graph0 aWave,bWave,cWave
```

Checks Graph0 to see if aWave, bWave, and cWave are displayed in it. If aWave is displayed, CheckDisplayed sets bit 0 of V_flag (V_flag=1). If bWave is displayed, sets bit 1 (V_flag=2). If cWave is displayed, sets bit 2 (V_flag=4). If all three waves are displayed, V_flag=7.

See Also

Setting Bit Parameters on page IV-12 for information about bit settings.

CheckName

CheckName(*nameStr*, *objectType* [, *windowNameStr*])

The CheckName function returns a number which indicates if the specified name is legal and unique among objects in the namespace of the specified object type.

Waves, global numeric variables, and global string variables are all in the same namespace and need to be unique only within the data folder containing them. However, they also need to be distinct from names of Igor operations and functions and from names of user-defined procedures.

Data folders are in their own namespace and need to be unique only among other data folders at the same level of the data folder hierarchy.

windowNameStr is optional. If missing, it is taken to be the top graph, panel, layout, or notebook according to the value of *objectType*.

Details

A result of zero indicates that the name is legal and unique within its namespace. Any nonzero result indicates that the name is illegal or not unique. You can use the **CleanupName** and **UniqueName** functions to guarantee legality and uniqueness.

nameStr should contain an unquoted name (i.e., no single quotes for liberal names), such as you might receive from the user through a dialog or control panel.

objectType is one of the following:

1:	Wave.	9:	Control panel window.
2:	Reserved.	10:	Notebook window.
3:	Global numeric variable.	11:	Data folder.
4:	Global string variable.	12:	Symbolic path.
5:	XOP target window.	13:	Picture.
6:	Graph window.	14:	Annotation in the named or topmost graph or layout.
7:	Table window.	15:	Control in the named topmost graph or panel.
8:	Layout window.	16:	Notebook action character in the named or topmost notebook.

The *windowNameStr* argument is used only with *objectTypes* 14, 15, and 16. The *nameStr* is checked for uniqueness only within the named window (other windows might have objects with the given name). If a named window is given but does not exist, any valid *nameStr* is permitted

Examples

```
Variable waveNameIsOK = CheckName(proposedWaveName, 1) == 0
Variable annotationNameIsOK = CheckName("text0", 14, "Graph0") == 0


// Create a valid and unique wave name
Function/S CreateValidAndUniqueWaveName(proposedName)
   String proposedName

   String result = proposedName

   if (CheckName(result,1) != 0)           // 1 for waves
      result = CleanupName(result, 1)      // Make sure it's valid
      result = UniqueName(result, 1, 0)    // Make sure it's unique
   endif

   return result
End
```

See Also

CleanupName and **UniqueName** functions.

ChildWindowList

`ChildWindowList(`*`hostNameStr`*`)`

The ChildWindowList function returns a string containing a semicolon-separated list of immediate subwindow window names of the specified host window or subwindow.

Parameters

hostNameStr is a string or string expression containing the name of an existing host window or subwindow.

When identifying a subwindow with *hostNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

Error if the host does not exist or if it is not an allowed host type.

See Also

WinList and **WinType** functions.

ChooseColor

`ChooseColor [/A[=a]/C=(`*`r,g,b`*`)]`

The ChooseColor operation displays a dialog for choosing a color.

The color initially shown is black unless you specify a different color with /C.

Flags

/A[=*a*]	a=1 shows the alpha (opacity) channel. /A is the same as /A=1. a=0 hides the alpha channel. This is the default setting. The /A flag was added in Igor Pro 7.00.
/C=(*r*,*g*,*b*)	*r*, *g*, and *b* specify the amount of red, green, and blue in the color initially displayed in the dialog as an integer from 0 to 65535.

Details

ChooseColor sets the variable V_flag to 1 if the user clicks OK in the dialog or to 0 otherwise.

If V_flag is 1 then V_Red, V_Green, V_Blue, and V_Alpha are set to the selected color as integers from 0 to 65535.

A completely opaque color sets V_Alpha=65535. A fully transparent color sets V_Alpha=0.

See Also

ImageTransform **rgb2hsl** and **hsl2rgb**.

CleanupName

`CleanupName(`*`nameStr, beLiberal`*`)`

The CleanupName function returns the input name string, possibly altered to make it a legal object name.

Details

nameStr should contain an unquoted (i.e., no single quotes for liberal names) name, such as you might receive from the user through a dialog or control panel.

beLiberal is 0 to use strict name rules or 1 to use liberal name rules. Strict rules allow only letters, digits and the underscore character. Liberal rules allow other characters such as spaces and dots. Liberal rules were introduced with Igor Pro 3.0 and are allowed for names of waves and data folders only.

Note that a cleaned up name is not necessarily unique. Call **CheckName** to check for uniqueness or **UniqueName** to ensure uniqueness.

Examples

```
String cleanStrVarName = CleanupName(proposedStrVarName, 0)
```

See Also

CheckName and **UniqueName** functions.

Close

`Close [/A] fileRefNum`

The Close operation closes a file previously opened by the **Open** operation or closes all such files if /A is used.

Parameters

fileRefNum is the file reference number of the file to close. This number comes from the Open operation. If /A is used, *fileRefNum* should be omitted.

Flags

/A	Closes all files. Mainly useful for cleaning up after an error during procedure execution occurs so that the normal Close operation is never executed.

CloseHelp

`CloseHelp [ /FILE=fileNameStr /NAME=helpNameStr /P=pathName ]`

The CloseHelp operation closes a help window.

The CloseHelp operation was added in Igor Pro 7.00.

Flags

/FILE=*fileNameStr*	Identifies the help window to close using the help file's location on disk. The file is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.
/NAME=*helpNameStr*	Identifies the help window to close using the window's title as specified by *helpNameStr*. This is the text that appears in the help window title bar.
/P=*pathName*	Specifies the folder to look in for the file specified by /FILE. *pathName* is the name of an existing Igor symbolic path.

Details

You must provide either the /NAME or /FILE flag.

See Also

OpenHelp

CloseMovie

`CloseMovie`

The CloseMovie operation closes the currently open movie. You must close a movie before you can play it.

See Also

Movies on page IV-230.

The **NewMovie** operation.

CloseProc

`CloseProc /NAME=procNameStr [flags]`

`CloseProc /FILE=fileNameStr [flags]`

The CloseProc operation closes a procedure window. You cannot call CloseProc on the main Procedure window.

CloseProc

CloseProc provides a way to programmatically create and alter procedure files. You might do this in order to make a user-defined menu-bar menu with contents that change.

Note: CloseProc alters procedure windows so it cannot be called while functions or macros are running. If you want to call it from a function or macro, use **Execute/P**.

Warning: If you close a procedure window that has no source file or without specifying a destination file, the window contents will be permanently lost.

Flags

/COMP[=*compile*]	Specifies whether procedures should be compiled after closing the procedure window. *compile*=1: Compiles procedures (same as /COMP only). *compile*=0: Leaves procedures in an uncompiled state.
/D[=*delete*]	Specifies whether the procedure file should be deleted after closing the procedure window. *delete*=1: Deletes procedure file (same as /D only). **Warning**: You cannot recover any file deleted this way. *delete*=0: Leaves any associated file unaffected.
/FILE=*fileNameStr*	Identifies the procedure window to close using the file name and path to the file given by *fileNameStr*. The string can be just the file name if /P is used to specify a symbolic path name of the enclosing folder. It can be a partial path if /P points to a folder enclosing the start of the partial path. It can also be a full path the file.
/NAME=*procNameStr*	Identifies the procedure window to close with the string expression *procNameStr*. This is the same text that appears in the window title. If the procedure window is associated with a file, it will be the file name and extension. To close a procedure file that is part of an independent module, you must include the independent module name in *procNameStr*. For example: `CloseProc /NAME="GraphBrowser.ipf [WM_GrfBrowser]"` Note that there is a space after the file name followed by the independent module name in brackets.
/P=*pathName*	Specifies the folder to look in for the file specified by /FILE. *pathName* is the name of an existing symbolic path.
/SAVE[=*savePathStr*]	Saves the procedure before closing the window. If the flag is used with no argument, it saves any changes to the procedure window to its source file before closing it. If *savePathStr* is present, it must be a full path naming a file in which to save the procedure window contents. The /P flag is not used with *savePathStr* so it must be a full path.

Details

Specify which window to close using either the /NAME or /FILE flag. You must use one or the other. Usually you would use /NAME, as it is usually more convenient. If by some chance two procedures have the same name, /FILE can be used to distinguish between them.

You cannot call CloseProc on a nonmain procedure window that someone has had the bad taste to call “Procedure”.

See Also

Chapter III-13, **Procedure Windows**.

The **Execute/P** operation.

cmplx

`cmplx(`*`realPart, imagPart`*`)`

The cmplx function returns a complex number whose real component is *realPart* and whose imaginary component is *imagPart*.

Use this to assign a value to a complex variable or complex wave.

Examples

Assume wave1 is complex. Then:

```
wave1(0) = cmplx(1,2)
```

sets the Y value of wave1 at x=0 such that its real component is 1 and its imaginary component is 2.

Assuming wave2 and wave3 are real, then:

```
wave1 = cmplx(wave2,wave3)
```

sets the real component of wave1 equal to the contents of wave2 and the imaginary component of wave1 equal to the contents of wave3.

You may get unexpected results if the number of points in wave2 or wave3 differs from the number of points in wave1. If wave2 or wave3 are shorter than wave1, the last element of the short wave is copied repeatedly to fill wave1.

See Also

conj, **imag**, **magsqr**, **p2rect**, **r2polar**, and **real** functions.

cmpstr

`cmpstr(`*`str1, str2`*`[, `*`flags`*`])`

The cmpstr function returns -1, 0 or 1 depending on how string *str1* compares alphabetically to string *str2*.

Details

cmpstr returns the following values:

-1:	*str1* is alphabetically before *str2*.
0:	*str1* and *str2* are equal.
1:	*str1* is alphabetically after *str2*.

If *flags* is not present, or if *flags* is zero, case (upper or lower) is not significant. Set *flags* to 1 for a case-sensitive comparison.

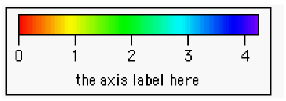
See Also

The **ReplaceString** function.

ColorScale

`ColorScale [`*`flags`*`] [, `*`keyword = value, …`*`] [`*`axisLabelStr`*`]`

The ColorScale operation puts a color scale (or "color legend") annotation on the top graph or page layout.



The ColorScale operation can be executed with no flags and no parameters.

When a graph is the top window the color scale represents the colors and values associated with the first image plot that was added to the graph.

If there is no image plot in the graph, the color scale represents the first contour plot or first f(z) trace added to the graph, one of which must exist for the command to execute without error when executed without parameters.

Executing ColorScale (with no parameters) when a layout is the top window displays a color bar as if the `ctab={0,100,Rainbow}` parameters had been specified.

Flags

Use the /W=*winName* flag to specify a specific graph or layout window. When used on the command line or in a Macro or Proc, /W must precede all other flags.

To change a color scale, use the /C/N=*name* flags. Annotations are automatically named "text0", "text1", etc. if no name is specified when it is created, and you must use that name to modify an existing annotation or else a new one will be created.

For explanations of all flags see the **TextBox** operation.

Parameters

The following keyword-value pairs are the important parameters to the ColorScale operation, because they specify what object the color scale is representing.

For use with page layouts, the keyword ={*graphName*,…} form is required.

For graphs it is simpler to use the image=*imageInstanceName* form (omitting graphName), though you can use `$""` to mean the top graph, or specify the name of another graph with the long form. See the **Examples**.

axisLabelStr Contains the text printed beside the color scale's main axis.

Using escape codes you can change the font, size, style and color of text, create superscripts and subscripts, create dynamically-updated text, insert legend symbols, and apply other effects. See **Annotation Escape Codes** on page III-53 for details.

You can use **AppendText** or **ReplaceText** to modify this axis label string. The default value for axisLabelStr is `""`.

cindex=*cindexMatrixWave*

The colors shown are those in the named color index wave with axis values derived from the wave's X (row) scaling and units.

The image colors are determined by doing a lookup in the specified matrix wave. See the ModifyImage cindex keyword.

contour=*contourInstanceName*

contour={*graphName*,*contourInstanceName*}

The colors show the named contour plot's colors and associated contour (Z) values and contour data units. All of the image plot's characteristics are represented, including color table, cindex, and fixed colors.

ctab={*zMin*,*zMax*,*ctName*, *reverse*}

The color table specified by `ctName` is drawn in the color legend.

zMin and *zMax* set the range of z values to map.

The color table name can be omitted if you want to leave it unchanged.

ctName can be any color table name returned by the CTabList function, such as Grays or Rainbow (see **Image Color Tables** on page II-305) or the name of a 3 column or 4 column color table wave (**Color Table Waves** on page II-311).

A color table wave name supplied to `ctab` must not be the name of a built-in color table (see **CTabList**). A 3 column or 4 column color table wave must have values that range between 0 and 65535. Column 0 is red, 1 is green, and 2 is blue. In column 3 a value of 65535 is opaque, and 0 is fully transparent.

Set *reverse* to 1 to reverse the color table. Setting it to 0 or omitting it leaves the color table unreversed.

image=*imageInstanceName*

image={*graphName*,*imageInstanceName*}

The colors show the named image plot's colors and associated image (Z) values and image data units. All of the image plot's characteristics are represented, including color table, cindex, lookup wave, eval colors, and NaN transparency. **Note**: only false-color image plots can be used with ColorScale (see **Indexed Color Details** on page II-312).

logLabel=*t*	*t* is the maximum number of decades before minor tick labels are suppressed. The default value is 3. The logLabel keyword was added in Igor Pro 7.00.
logTicks=*t*	*t*=0 means "auto". This is the default value. If *t* is not 0 then it represents the maximum number of decades before minor ticks are suppressed. The logLabel keyword was added in Igor Pro 7.00.
lookup=*waveName*	Specifies an optional 1D wave used to modify the mapping of scaled Z values into the color table specified with the ctab keyword. Values should range from 0.0 to 1.0. A linear ramp from 0 to 1 would have no effect while a ramp from 1 to 0 would reverse the image. Used to apply gamma correction to grayscale images or for special effects. Use `lookup=$""` to remove option. This keyword is not needed with the image keyword, even if the image plot uses a lookup wave. The image plot's lookup wave is used instead of the ColorScale lookup wave.

trace=*traceInstanceName*

trace={*graphName*,*traceInstanceName*}

The colors show the color(s) of the named trace. This is useful when the trace has its color set by a "Z wave" using the `ModifyGraph zColor(`*`traceName`*`)=`... feature. In the Modify Trace Appearance dialog this is selected in the "Set as f(z)" subdialog. The color scale's main axis shows the range of values in the Z wave, and displays any data units the wave may have.

Size Parameters

The following keyword-value parameters modify the size of the color scale annotation. These keywords are similar to those used by the **Slider** control. The size of the annotation is indirectly controlled by setting the size of the "color bar" and the various axis parameters. The annotation sizes itself to accommodate the color bar, tick labels, and axis labels.

height=*h*	Sets the height of the color bar in points, overriding any heightPct setting. The default height is 75% of the plot area height if the color scale is vertical, or a constant of 15 points if the color scale is horizontal. The default is restored by specifying `height=0`. Specifying a heightPct value resets height to this default.
heightPct=*hpct*	Sets height as a percentage of the graph's plot area, overriding any height setting. The default height is 75% of the plot area height if the color scale is vertical, or a constant of 15 points if the color scale is horizontal. The default height is restored by setting heightPct=0. Specifying a height value resets heightPct to this default.
side=*s*	Selects on which axis to draw main axis ticks. *s*=1: Right of the color bar if vert=1, or below if vert=0. *s*=2: Left of the color bar if vert=1, or above if vert=0.
vert=*v*	Specifies color scale orientation. *v*=0: Horizontal. *v*=1: Vertical (default).
width=*w*	Sets the width of the color bar in points, overriding any widthPct setting. The default width is a constant 15 points if the color scale is vertical, or 75% of the plot area width if the color scale is horizontal. The default is restored by specifying `width=0`. Specifying a widthPct value resets width to this default.
widthPct=*wpct*	Sets width as a percentage of the graph's plot area, overriding any width setting. The default width is a constant 15 points if the color scale is vertical, or 75% of the plot area width if the color scale is horizontal. The default is restored by setting `widthPct=0`. Specifying a width value resets widthPct to this default.

Color Bar Parameters

The following keyword-value parameters modify the appearance of the color scale color bar.

colorBoxesFrame=*on*	Draws frames surrounding up to 99 swatches of colors in the color bar (*on*=1). When specifying more than 99 colors in the color bar (such as the Rainbow color table, which has 100 colors), the boxes aren't framed. Framing color boxes is effective only for small numbers of colors. Set the width of the frame with the frame keyword. Use *on*=0 to turn off color box frames.
frame=*f*	Specifies the thickness of the frame drawn around the color bar in points (*f* can range from 0 to 5 points). The default is 1 point. Fractional values are permitted. Turn frames off with *f*=0. Values less that 0.5 do not display on screen, but the thin frame will print.
frameRGB=(*r*, *g*, *b*) or 0	Sets the color of the frame around the color bar. *r*, *g*, and *b* specify the amount of red, green, and blue as an integer from 0 to 65535. The frame includes the individual color bar colors when colorBoxesFrame=1. The frame will use the colorscale foreground color, as set by the /G flag, when frameRGB=0.

Axis Parameters

The following keyword-value parameters modify the appearance of the color scale axes. These keywords are based on the **ModifyGraph** Axis keywords because they modify the main or secondary color scale axes.

axisRange={*zMin*, *zMax*}

Sets the color bar axis range to values specified by *zMin* and *zMax*. Use `*` to use the default axis range for either or both values.

Omit *zMin* or *zMax* to leave that end of the range unchanged. For example, use {`zMin,` } to change *zMin* and leave *zMax* alone, or use { ,`*`} to set only the axis maximum value to the default value.

dateInfo={*sd*,*tm*,*dt*}

Controls formatting of date/time axes.

sd=0:	Show date in the date&time format.
sd=1:	Suppress date.
tm=0:	12 hour (AM/PM) time.
tm=1:	24 hour (military) time.
tm=2:	Elapsed time.
dt=0:	Short dates (2/22/90).
dt=1:	Long dates (Thursday, February 22, 1990).
dt=2:	Abbreviated dates (Thurs, Feb 22, 1990).

These have no effect unless the axis is controlled by a wave with 'dat' data units.

For an f(z) color scale:

```
SetScale d, 0,0, "dat", fOfZWave
```

For a contour plot or image plot color scale:

```
SetScale d, 0,0, "dat", ZorXYZorImageWave
```

See **Date/Time Axes** on page II-244 and **Date, Time, and Date&Time Units** on page II-64 for details on how date/time axes work.

font=*fontNameStr*

Name of font as string expression. If the font does not exist, the default font is used. Specifying "default" has the same effect. Unlike ModifyGraph, the *fontNameStr* is evaluated at runtime, and its absence from the system is not an error.

fsize=*s*	Sets the font size in points. *s*=0: Use the graph font size for tick labels and axis labels (default).
fstyle=*fs*	Sets the font style. *fs* is a bitwise parameter with each bit controlling one aspect of the font style for the tick mark labels: Bit 0: Bold Bit 1: Italic Bit 2: Underline Bit 4: Strikethrough See **Setting Bit Parameters** on page IV-12 for details about bit settings.
highTrip=*h*	If the extrema of an axis are between its lowTrip and its highTrip then tick mark labels use fixed point notation. Otherwise they use exponential (scientific or engineering) notation. The default highTrip is 100,000.
lblLatPos=*p*	Sets a lateral offset for the main axis label. This is an offset parallel to the axis. *p* is in points. Positive is down for vertical axes and to the right for horizontal axes. The default is 0.
lblMargin=*m*	Moves the main axis label by *m* points (default is 0) from the normal position. The default value is -5, which brings the axis label closer to the axis. Use more positive values to move the axis label away from the axis.
lblRot=*r*	Rotates the axis label by *r* degrees. *r* is a value from -360 to 360. Rotation is counterclockwise and starts from the label's normal orientation.
log=*l*	Specifies the axis type: *l*=0: Linear (default). *l*=1: Log base 10. *l*=2: Log base 2.
logHTrip=*h*	Same as highTrip but for log axes. The default is 10,000.
logLTrip=*l*	Same as lowTrip but for log axes. The default is 0.0001.
logTicks=*t*	Specifies the maximum number of decades in log axis before minor ticks are suppressed.
lowTrip=*l*	If the axis extrema are between its lowTrip and its highTrip, then tick mark labels use fixed point notation. Otherwise, they use exponential (scientific or engineering) notation. The default lowTrip is 0.1.
minor=*m*	Controls minor tick marks: *m*=0: Disables minor ticks (default). *m*=1: Enables minor ticks.
notation=*n*	Controls tick label notation: *n*=0: Engineering notation (default). *n*=1: Scientific notation.
nticks=*n*	Specifies the approximate number of ticks to be distributed along the main axis. Ticks are labelled using the same automatic algorithm used for graph axes. The default is 5. Set *n*=0 for no ticks.
prescaleExp=*exp*	Multiplies axis range by 10*exp* for tick labeling and *exp* is subtracted from the axis label exponent. In other words, the exponent is moved from the tick labels to the axis label. The default is 0 (no modification). See the discussion in the **ModifyGraph (axes) Details** section.

tickExp=*te*	Controls tick label exponential notation: *te*=1: Forces tick labels to exponential notation when labels have units with a prefix. *te*=0: Turns off exponential notation.
tickLen=*t*	Sets the length of the ticks. *t* is the major tick mark length in points. This value must be between -100 and 50. *t*= 0 to 50: Draws tick marks between the tick labels and the colors box. *t*= -1: Default; auto tick length, equal to 70% of the tick label font size. Draws tick marks between the tick labels and the colors box. *t*= -2 to -50: Draws tick marks crossing the edge of the colors box nearest the tick labels. The actual total tick mark length is -*t*. *t*= -100 to -51: Draws tick marks inside the edge of the colors box nearest the tick labels. Actual tick mark length is -(*t*+50). For example, -58 makes in an inside tick mark that is 8 points long.
tickThick=*t*	Sets the tick mark thickness in points (from 0 to 5 points). The default is 1 point. Fractional values are permitted. *t*=0: Turns tick marks off, but not the tick labels.
tickUnit=*tu*	Turns on (*tu*=0) or off (*tu*=1) units labels attached to tick marks.

userTicks={*tvWave*,*tlblWave*}

Supplies user defined tick positions and labels for the main axis. *tvWave* contains the numeric tick positions while text wave *tlblWave* contains the corresponding labels.

Overrides normal ticking specified by nticks.

See **User Ticks from Waves** on page II-241 for details.

The tick mark labels can be multiline and use styled text. For more details, see **Fancy Tick Mark Labels** on page II-270.

ZisZ=*z*	*z* =1 labels the zero tick mark (if any) with the single digit “0” regardless of the number of digits used for other labels. Default is *z*=0.

Secondary Axis Parameters

axisLabel2=*axisLabelString2*

Axis label for the secondary axis. This axis label is drawn only if userTicks2 is in effect. Text after any `\r` character is ignored, as is the `\r` character. The default is `""`.

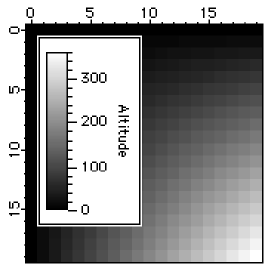
lblLatPos2=*p*	Sets lateral offset for secondary axis labels. This is an offset parallel to the axis. *p* is in points. Positive is down for vertical axes and to the right for horizontal axes. The default is 0.
lblMargin2=*m*	Specifies the distance in points (default 0) to move the secondary axis label from the position that would be normal for a graph. The default is value is -5, which brings the axis label closer to the axis. Use more positive values to move the axis label away from the axis.
lblRot2=*r*	Rotates the secondary axis label by *r* degrees counterclockwise starting from the normal label orientation. r is a value from -360 to 360.

userTicks2={*tvWave*,*tlblWave*}

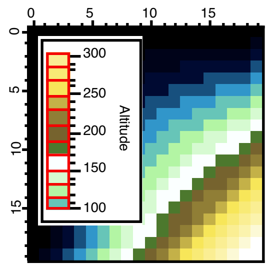
Supplies user defined tick positions and labels for a second axis which is always on the opposite side of the color bar from the main axis. The tick mark labels can be multiline and use styled text. For more details, see **Fancy Tick Mark Labels** on page II-270. This is the only way to draw a second axis.

Examples

```
Make/O/N=(20,20) img=p*q; NewImage img      // Make and display an image
ColorScale                                  // Create default color scale
// First annotation is text0
ColorScale/C/N=text0 nticks=3,minor=1,"Altitude"
```



```
ModifyImage img ctab= {*,*,Relief19,0}        // 19-color color table
ColorScale/C/N=text0 axisRange={100,300}      // Detail for 100-300 range
ColorScale/C/N=text0 colorBoxesFrame=1        // Frame the color boxes
ColorScale/C/N=text0 frameRGB=(65535,0,0)     // Red frame
```



See Also

For all other flags see the **TextBox** and **AppendText** operations.

AnnotationInfo, **AnnotationList**

ColorTab2Wave

`ColorTab2Wave` ***`colorTableName`***

The ColorTab2Wave operation extracts colors from the built-in color table and places them in an Nx3 matrix of red, green, and blue columns named M_colors. Values are unsigned 16-bit integers and range from 0 to 65535.

N will typically be 100 but may be as little as 9 and as large as 476. Use

```
Variable N= DimSize(M_colors,0)
```

to determine the actual number of colors.

The wave M_colors is created in the current data folder. Red is in column 0, green is in column 1, and blue in column 2.

Parameters

colorTableName can be any of those returned by **CTabList**, such as Grays or Rainbow.

colorTableName can also be `Igor` or `IgorRecent`, to return either the 128 standard or 0-32 user-selected colors from Igor's color menu.

Details

See **Image Color Tables** on page II-305.

Complex

`Complex` ***`localName`***

Declares a local complex 64-bit double-precision variable in a user-defined function or structure.

Complex is another name for Variable/C. It is available in Igor Pro 7 and later.

Concatenate

`Concatenate` `[`***`type flags`***`][`***`flags`***`]` ***`waveListStr, destWave`***

`Concatenate` `[`***`type flags`***`][`***`flags`***`]` ***`{wave1, wave2, wave3,…}, destWave`***

The Concatenate operation combines data from the source waves into *destWave*, which is created if it does not already exist. If *destWave* does exists and overwrite is not specified, the source waves' data is concatenated with the existing data in the destination wave.

By default the concatenation increases the dimensionality of the destination wave if possible. For example, if you concatenate two 1D waves of the same length you get a 2D wave with two columns. The destination wave is said to be "promoted" to a higher dimensionality.

If you use the /NP (no promotion) flag, the dimensionality of the destination wave is not changed. For example, if you concatenate two 1D waves of the same length using /NP you get a 1D wave whose length is the sum of the lengths of the source waves.

If the source waves are of different lengths, no promotion is done whether /NP is used or not.

Parameters

waveListStr is a string expression containing a list of wave names separated by semicolons. The list must be terminated with a semicolon. The alternate syntax using {*wave1*, *wave2*, …} is limited to 100 waves or less, but there is no limit when using *waveListStr*.

destWave is the name of a new or existing wave that will contain the concatenation result.

Flags

/DL	Sets dimension labels. For promotion, it uses source wave names as new dimension labels otherwise it uses existing labels.
/KILL	Kills source waves.
/NP	Prevents promotion to higher dimension.
/NP=*dim*	Prevents promotion and appends data along the specified dimension (0= rows, 1= columns, 2=layers, 3=chunks). All dimensions other than the one specified by *dim* must be the same in all waves.
/O	Overwrites *destWave*.

Type Flags *(used only in functions)*

Concatenate also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-67 and **WAVE Reference Type Flags** on page IV-68 for a complete list of type flags and further details.

Details

If *destWave* does not already exist or, if the /O flag is used, *destWave* is created by duplication of the first source wave. Waves are concatenated in order through the list of source waves. If *destWave* exists and the /O flag is not used, then the concatenation starts with *destWave*.

destWave cannot be used in the source wave list.

Source waves must be either all numeric or all text.

If promotion is allowed, the number of low-order dimensions that all waves share in common determines the dimensionality of *destWave* so that the dimensionality of *destWave* will then be one greater. The default behaviors will vary according to the source wave sizes. Concatenating 1D waves that are all the same length will produce a 2D wave, whereas concatenating 1D waves of differing lengths will produce a 1D wave.

Similarly, concatenating 2D waves of the same size will produce a 3D wave; but if the 2D source waves have differing numbers of columns then *destWave* will be a 2D wave, or if the 2D waves have differing numbers of rows then *destWave* will be a 1D wave. Concatenating 1D and 2D waves that have the same number of rows will produce a 2D wave, but when the number of rows differs, *destWave* will be a 1D wave. See the examples.

Use the /NP flag to suppress dimension promotion and keep the dimensionality of *destWave* the same as the input waves.

Examples

```
// Given the following waves:
Make/N=10 w1,w2,w3
Make/N=11 w4
Make/N=(10,7) m1,m2,m3
Make/N=(10,8) m4
Make/N=(9,8) m5

// Concatenate 1D waves
Concatenate/O {w1,w2,w3},wdest          // wdest is a 10x3 matrix
Concatenate {w1,w2,w3},wdest            // wdest is a 10x6 matrix
Concatenate/NP/O {w1,w2,w3},wdest       // wdest is a 30-point 1D wave
Concatenate/O {w1,w2,w3,w4},wdest       // wdest is a 41-point 1D wave

// Concatenate 2D waves
Concatenate/O {m1,m2,m3},wdest          // wdest is a 10x7x3 volume
Concatenate/NP/O {m1,m2,m3},wdest       // wdest is a 10x21 matrix
Concatenate/O {m1,m2,m3,m4},wdest       // wdest is a 10x29 matrix
Concatenate/O {m4,m5},wdest             // wdest is a 152-point 1D wave
Concatenate/O/NP=0 {m4,m5},wdest        // wdest is a 19x8 matrix

// Concatenate 1D and 2D waves
Concatenate/O {w1,m1},wdest             // wdest is a 10x8 matrix
Concatenate/O {w4,m1},wdest             // wdest is a 81-point 1D wave

// Append rows to 2D wave
Make/O/N=(3,2) m6, m7
Concatenate/NP=0 {m6}, m7               // m7 is a 6x2 matrix

// Append columns to 2D wave
Make/O/N=(3,2) m6, m7
Concatenate/NP=1 {m6}, m7               // m7 is a 3x4 matrix

// Append layer to 2D wave
Make/O/N=(3,2) m6, m7
Concatenate/NP=2 {m6}, m7               // m7 is a 3x2x2 volume
// The last command has the same effect as:
// Concatenate {m6}, m7
// Both versions extend add a third dimension to m7
```

See Also

Duplicate, **Redimension**, **SplitWave**

conj

`conj(`*`z`*`)`

The conj function returns the complex conjugate of the complex value *z*.

See Also

cmplx, **imag**, **magsqr**, **p2rect**, **r2polar**, and **real** functions.

Constant

`Constant` ***`kName = literalNumber`***

`Constant/C` ***`kName`*** **`= (`*`literalNumberReal, literalNumberImag`*`)`**

The Constant declaration defines the number *literalNumber* under the name *kName* for use by other code, such as in a switch construct.

The complex form, using the /C flag to create a complex constant, requires Igor Pro 7.00 or later.

See Also

The **Strconstant** keyword for string types, **Constants** on page IV-47 and **Switch Statements** on page IV-41.

continue

`continue`

The continue flow control keyword returns execution to the beginning of a loop, bypassing the remainder of the loop’s code.

See Also

Continue Statement on page IV-45 and **Loops** on page IV-42 for usage details.

ContourInfo

`ContourInfo(`*`graphNameStr, contourWaveNameStr, instanceNumber`*`)`

The ContourInfo function returns a string containing a semicolon-separated list of information about the specified contour plot in the named graph.

Parameters

graphNameStr can be `""` to refer to the top graph.

contourWaveNameStr is a string containing either the name of a wave displayed as a contour plot in the named graph, or a contour instance name (wave name with “#n” appended to distinguish the nth contour plot of the wave in the graph). You might get a contour instance name from the **ContourNameList** function.

If *contourWaveNameStr* contains a wave name, *instanceNumber* identifies which instance you want information about. *instanceNumber* is usually 0 because there is normally only one instance of a wave displayed as a contour plot in a graph. Set *instanceNumber* to 1 for information about the second contour plot of the wave, etc. If *contourWaveNameStr* is `""`, then information is returned on the *instanceNumber*th contour plot in the graph.

If *contourWaveNameStr* contains an instance name, and *instanceNumber* is zero, the instance is taken from *contourWaveNameStr*. If *instanceNumber* is greater than zero, the wave name is extracted from *contourWaveNameStr*, and information is returned concerning the *instanceNumber*th instance of the wave.

Details

The string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

Keyword	Information Following Keyword
AXISFLAGS	Flags used to specify the axes. Usually blank because /L and /B (left and bottom axes) are the defaults.
DATAFORMAT	Either XYZ or Matrix.
LEVELS	A comma-separated list of the contour levels, including the final automatic levels, (or manual or from-wave levels), and the “more levels”, all sorted into ascending Z order.
RECREATION	List of keyword commands as used by **ModifyContour** command. The format of these keyword commands is: ***keyword (x)=modifyParameters;***
TRACESFORMAT	The format string used to name the contour traces (see **AppendMatrixContour** or **AppendXYZContour**).
XAXIS	X axis name.
XWAVE	X wave name if any, else blank.
XWAVEDF	Full path to the data folder containing the X wave or blank if there is no X wave.
YAXIS	Y axis name.
YWAVE	Y wave name if any, else blank.

Keyword	Information Following Keyword
YWAVEDF	Full path to the data folder containing the Y wave or blank if there is no Y wave.
ZWAVE	Name of wave containing Z data from which the contour plot was calculated.
ZWAVEDF	Full path to the data folder containing the Z data wave.

The format of the RECREATION information is designed so that you can extract a keyword command from the keyword and colon up to the "`;`", prepend "`ModifyContour`", replace the "`x`" with the name of a contour plot ("`data#1`" for instance) and then **Execute** the resultant string as a command.

Examples

The following command lines create a very unlikely contour display. If you did this, you would most likely want to put each contour plot on different axes, and arrange the axes such that they don't overlap. That would greatly complicate the example.

```
Make/O/N=(20,20) jack
Display;AppendMatrixContour jack
AppendMatrixContour/T/R jack        // Second instance of jack
```

This example accesses the contour information for the second contour plot of the wave "jack" (which has an instance number of 1) displayed in the top graph:

```
Print StringByKey("ZWAVE", ContourInfo("","jack",1))     // prints jack
```

See Also

The **Execute** and **ModifyContour** operations.

ContourNameList

`ContourNameList(`*`graphNameStr, separatorStr`*`)`

The ContourNameList function returns a string containing a list of contours in the graph window or subwindow identified by *graphNameStr*.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

The parameter *separatorStr* should contain a single character such as "`,`" or "`;`" to separate the names.

A contour name is defined as the name of the wave containing the data from which a contour plot is calculated, with an optional #n suffix that distinguishes between two or more contour plots in the same graph window that have the same wave name. Since the contour name has to be parsed, it is quoted if necessary.

Examples

The following command lines create a very unlikely contour display. If you did this, you would most likely want to put each contour plot on different axes, and arrange the axes such that they don't overlap. That would greatly complicate the example.

```
Make/O/N=(20,20) jack,'jack # 2';
Display;AppendMatrixContour jack
AppendMatrixContour/T/R jack
AppendMatrixContour 'jack # 2'
AppendMatrixContour/T/R 'jack # 2'
Print ContourNameList("",";")
```

prints `jack;jack#1;'jack # 2';'jack # 2'#1;`

See Also

Another command related to contour plots and waves: **ContourNameToWaveRef**.

For commands referencing other waves in a graph: **TraceNameList**, **WaveRefIndexed**, **XWaveRefFromTrace**, **TraceNameToWaveRef**, **CsrWaveRef**, **CsrXWaveRef**, **ImageNameList**, and **ImageNameToWaveRef**.

ContourNameToWaveRef

`ContourNameToWaveRef(`*`graphNameStr`*`, `*`contourNameStr`*`)`

Returns a wave reference to the wave corresponding to the given contour name in the graph window or subwindow named by *graphNameStr*.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

The contour name is identified by the string in *contourNameStr*, which could be a string determined using ContourNameList. Note that the same contour name can refer to different waves in different graphs, if the waves are in different data folders.

See Also

The **ContourNameList** function.

For a discussion of wave reference functions, see **Wave Reference Functions** on page IV-186.

ContourZ

`ContourZ(`*`graphNameStr`*`, `*`contourInstanceNameStr`*`, `*`x`*`, `*`y`*` [,`*`pointFindingTolerance`*`])`

The ContourZ function returns the interpolated Z value of the named contour plot data displayed in the named graph.

For gridded contour data, ContourZ returns the bilinear interpolation of the four surrounding XYZ values.

For XYZ triplet contour data, ContourZ returns the value interpolated from the three surrounding XYZ values identified by the Delaunay triangulation.

Parameters

graphNameStr can be `""` to specify the topmost graph.

contourNameStr is a string containing either the name of the wave displayed as a contour plot in the named graph, or a contour instance name (wave name with "#n" appended to distinguish the nth contour plot of the wave in the graph). You might get a contour instance name from the **ContourNameList** function.

If *contourNameStr* contains a wave name, *instance* identifies which contour plot of *contourNameStr* you want information about. *instance* is usually 0 because there is normally only one instance of a wave displayed as a contour plot in a graph. Set *instance* to 1 for information about the second contour plot of *contourNameStr*, etc. If *contourNameStr* is `""`, then information is returned on the *instance*th contour plot in the graph.

If *contourNameStr* contains an instance name, and *instance* is zero, the instance is taken from *contourNameStr*. If *instance* is greater than zero, the wave name is extracted from *contourNameStr*, and information is returned concerning the *instance*th instance of the wave.

x and *y* specify the X and Y coordinates of the value to be returned. This may or may not be the location of a data point in the wave selected by *contourNameStr* and *instance*.

Set *pointFindingTolerance* =1e-5 to overcome the effects of perturbation (see the perturbation keyword of the **ModifyContour** operation).

The default value is 1e-15 to account for rounding errors created by the triangulation scaling (see ModifyContour's equalVoronoiDistances keyword), which works well ModifyContour perturbation=0.

A value of 0 would require an exact match between the scaled x/y coordinate and the scaled and possibly perturbed coordinates to return the original z value; that is an unlikely outcome.

Details

For gridded contour data, ContourZ returns NaN if *x* or *y* falls outside the XY domain of the contour data. If *x* and *y* fall on the contour data grid, the corresponding Z value is returned.

For XYZ triplet contour data, ContourZ returns the null value if *x* or *y* falls outside the XY domain of the contour data. You can set the null value to *v* with this command:

```
ModifyContour contourName nullValue=v
```

If *x* and *y* match one of the XYZ triplet values, the corresponding Z value from the triplet usually won't be returned because Igor uses the Watson contouring algorithm which perturbs the x and y values by a small random amount. This also means that normally x and y coordinates on the boundary will return a null value about half the time if perturbation is on and *pointFindingTolerance* is greater than 1e-5.

Examples

Because ContourZ can interpolate the Z value of the contour data at any X and Y coordinates, you can use ContourZ to convert XYZ triplet data into gridded data:

```
// Make example XYZ triplet contour data
Make/O/N=50 wx,wy,wz
wx= enoise(2)          // x = -2 to 2
wy= enoise(2)          // y = -2 to 2
wz= exp(-(wx[p]*wx[p] + wy[p]*wy[p]))       // XY gaussian, z= 0 to 1

// ContourZ requires a displayed contour data set
Display; AppendXYZContour wz vs {wx,wy};DelayUpdate
ModifyContour wz autolevels={*,*,0}         // no contour levels are needed
ModifyContour wz xymarkers=1                // show the X and Y locations

// Set the null (out-of-XY domain) value
ModifyContour wz nullValue=NaN              // default is min(wz) - 1

// Convert to grid: Make matrix that spans X and Y
Make/O/N=(30,30) matrix
SetScale/I x, -2, 2, "", matrix
SetScale/I y, -2, 2, "", matrix
matrix= ContourZ("","wz",0,x,y)             // or = ContourZ("","",0,x,y)
AppendImage matrix
```

See Also

AppendMatrixContour, **AppendXYZContour**, **ModifyContour**, **FindContour**, **zcsr**, **ContourInfo**

References

Watson, David F., *Contouring: A Guide To The Analysis and Display of Spatial Data*, Pergamon, 1992.

ControlBar

`ControlBar [`*`flags`*`]` ***`barHeight`***

The ControlBar operation sets the height and location of the control bar in a graph.

Parameters

barHeight is in points on Macintosh and pixels or points on Windows, depending on the screen resolution. See **Control Panel Resolution on Windows** on page III-405 for details.

Setting *barHeight* to zero removes the control bar.

Flags

/L/R/B/T	Designates whether to use the Left, Right, Bottom, or Top (default) window edge, respectively, for the control bar location.
/W=*graphName*	Specifies the name of a particular graph containing a control bar.

Details

The control bar is an area at the top of graphs reserved for controls such as buttons, checkboxes and pop-up menus. A line is drawn between this area and the graph area. The control bar may be assigned a separate background color by pressing Control (*Macintosh*) or Ctrl (*Windows*) and clicking in the area, by right-clicking it (*Windows*), or with the **ModifyGraph** operation. You can not use draw tools in this area.

For graphs with no controls you do not need to use this operation.

Examples

```
Display myData
ControlBar 35       // 35 pixels high
Button button0,pos={56,8},size={90,20},title="My Button"
```

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

ControlInfo

`ControlInfo [/W=winName] controlName`

The ControlInfo operation returns information about the state or status of the named control in a graph or control panel window or subwindow.

Flags

/G [=*doGlobal*]	If *doGlobal* is non-zero or absent, the position returned via V_top and V_left is in global screen coordinates rather relative to the window containing the control.
/W=*winName*	Looks for the control in the named graph or panel window or subwindow. If /W is omitted, ControlInfo looks in the top graph or panel window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Parameters

controlName is the name of a control in the window specified by /W or in the top graph or panel window. *controlName* may also be the keyword kwBackgroundColor to set V_Red, V_Green, V_Blue, and V_Alpha, the keyword kwControlBar or kwControlBarBottom to set V_Height, the keyword kwControlBarLeft or kwControlBarRight to set V_Width, or the keyword kwSelectedControl to set S_value and V_flag.

Details

Information for all controls is returned via the following string and numeric variables:

S_recreation	Commands to recreate the named control.
V_disable	Disable state of control: 0: Normal (enabled, visible). 1: Hidden. 2: Disabled, visible.
V_Height, V_Width, V_top, V_left	Dimensions and position of the named control in pixels.

The kind of control is returned in V_flag as a positive or negative integer. A negative value indicates the control is incomplete or not active. If V_flag is zero, then the named control does not exist. Information returned for specific control types is as follows:

Buttons

V_flag	1
V_value	Tick count of last mouse up.
S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation. See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left at the beginning of the Details section.

Chart

V_flag	6 or -6
V_value	Current point number.
S_UserData	Keyword-packed information string. See **S_value for Chart Details** for more keyword information. See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left at the beginning of the Details section.

Checkbox

V_flag	2
V_value	0 if it is deselected or 1 if it is selected.
S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
	See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left at the beginning of the Details section.

CustomControl

V_flag	12
V_value	Tick count of last mouse up.
S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
S_value	Name of the picture used to define the control appearance.
	See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left at the beginning of the Details section.

GroupBox

V_flag	9
S_value	Title text.
	See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left at the beginning of the Details section.

ListBox

V_flag	11
V_value	Currently selected row (valid for mode 1 or 2 or modes 5 and 6 when no selWave is used). If no list row is selected, then it is set to -1.
V_selCol	Currently selected column (valid for modes 5 and 6 when no selWave is used).
V_horizScroll	Number of pixels the list has been scrolled horizontally to the right.
V_vertScroll	Number of pixels the list has been scrolled vertically downwards.
V_rowHeight	Height of a row in pixels.
V_startRow	The current top visible row.
S_columnWidths	A comma-separated list of column widths in pixels.
S_dataFolder	Full path to listWave (if any).
S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
S_value	Name of listWave (if any).
	See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left at the beginning of the Details section.

PopupMenu

V_flag	3 or -3

V_Red, V_Green, V_Blue. V_Alpha	For color array pop-up menus, these are the encoded color values.
V_value	Current item number (counting from one).
S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
S_value	Text of the current item. If PopupMenu is a color array then it contains color values encoded as (*r*,*g*,*b*) where *r*, *g*, and *b* are integers from 0 to 65535.
	See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left at the beginning of the Details section.

SetVariable

V_flag	5 or -5
V_value	Value of the variable. If the SetVariable is used with a string variable, then it is the interpretation of the string as a number, which will be NaN if conversion fails.
S_dataFolder	Full path to the variable.
S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
S_value	Name of the variable or, if the value was set using _STR: syntax, the string value itself.
	See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left at the beginning of the Details section.

Slider

V_flag	7
V_value	Numeric value of the variable.
S_dataFolder	Full path to the variable.
S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
S_value	Name of the variable.
	See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left at the beginning of the Details section.

TabControl

V_flag	8
V_value	Number of the current tab.
S_UserData	Primary (unnamed) user data text. For retrieving any named user data, you must use the **GetUserData** operation.
S_value	Tab text.
	See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left at the beginning of the Details section.

TitleBox

V_flag	10
S_dataFolder	Full path if text is from a string variable.
S_value	Name if text is from a string variable.

See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left at the beginning of the Details section.

ValDisplay

V_flag	4 or -4
V_value	Displayed value.
S_value	Text of expression that ValDisplay evaluates.
	See also the descriptions of S_recreation, V_disable, V_Height, V_Width, V_top, V_left at the beginning of the Details section.

kwBackgroundColor

V_Red, V_Green, V_Blue, V_Alpha	If *controlName* is kwBackgroundColor then this is the color of the control panel background. This color is usually the default user interface background color, as set by the Appearance control panel on the Macintosh or by the Appearance tab of the Display Properties on Windows, until changed by `ModifyPanel cbRGB`.

kwControlBar or kwControlBarTop

V_Height	The height in pixels of the top control bar area in a graph as set by **ControlBar**.

kwControlBarBottom

V_Height	The height in pixels of the bottom control bar area in a graph as set by **ControlBar**/B.

kwControlBarLeft

V_Width	The width in pixels of the left control bar area in a graph as set by **ControlBar**/L.

kwControlBarRight

V_Width	The width in pixels of the right control bar area in a graph as set by **ControlBar**/R.

kwSelectedControl

V_flag	Set to 1 if a control is selected or 0 if not.
	SetVariable and **ListBox** controls can be selected, most other controls can not.
S_value	Name of selected control (if any) or `""`.

S_value for Chart Details

The following applies *only* to the keyword-packed information string returned in S_value for a chart. S_value will consist of a sequence of sections with the format: "*keyword*:*value*;" You can pick a value out of a keyword-packed string using the **NumberByKey** and **StringByKey** functions. Here are the S_value keywords:

Keyword	Type	Meaning
FNAME	string	Name of the FIFO chart is monitoring.
LHSAMP	number	Left hand sample number.
NCHANS	number	Number of channels displayed in chart.
PPSTRIP	number	The chart's points per strip value.
RHSAMP	number	Right hand sample number (same as V_value).

In addition, ControlInfo writes fields to S_value for each channel in the chart. The keyword for the field is a combination of a name and a number that identify the field and the channel to which it refers. For example, if channel 4 is named “Pressure” then the following would appear in the S_value string: “CHNAME4:Pressure”. In the following table, the channel’s number is represented by #:

Keyword	Type	Meaning
CHCTAB#	number	Channel’s color table value as set by Chart ctab keyword.
CHGAIN#	number	Channel’s gain value as set by Chart gain keyword.
CHNAME#	string	Name of channel defined by FIFO.
CHOFFSET#	number	Channel’s offset value as set by Chart offset keyword.

Examples

```
ControlInfo myChart; Print S_value
```

Prints the following to the history area:

```
FNAME:myFIFO;NCHANS:1;PPSTRIP:1100;RHSAMP:271;LHSAMP:-126229;
```

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

ControlNameList

ControlNameList(*winNameStr* [, *listSepStr* [, *matchStr*]])

The ControlNameList function returns a string containing a list of control names in the graph or panel window or subwindow identified by *winNameStr*.

Parameters

winNameStr can be `""` to refer to the top graph or panel window.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

The optional parameter *listSepStr* should contain a single character such as "," or ";" to separate the names; the default value is ";".

The optional parameter *matchStr* is some combination of normal characters and the asterisk wildcard character that matches anything. To use *matchStr*, *listSepStr* must also be used. See **StringMatch** for wildcard details.

Only control names that satisfy the match expression are returned. For example, "*_tab0" matches all control names that end with "_tab0". The default is "*", which matches all control names.

Examples

```
NewPanel
Button myButton
Checkbox myCheck
Print ControlNameList("")                 // prints "myButton;myCheck;"
Print ControlNameList("", ";", "*Check")   // prints "myCheck;"
```

See Also

The **ListMatch**, **StringFromList** and **StringMatch** functions, and the **ControlInfo** and **ModifyControlList** operations. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

ControlUpdate

`ControlUpdate` `[`***`/A/W=winName`***`]` `[`***`controlName`***`]`

The ControlUpdate operation updates the named control or all controls in a window, which can be the top graph or control panel or the named graph or control panel if you use /W.

Flags

/A	Updates all controls in the window. You must omit *controlName*.
/W=*winName*	Specifies the window or subwindow containing the control. If you omit *winName* it will use the top graph or control panel window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

ControlUpdate is useful for forcing a pop-up menu to rebuild, to update a ValDisplay control, or to forcibly accept a SetVariable's currently-being-edited value.

Normally, a pop-up menu rebuilds only when the user clicks on it. If you set up a pop-up menu so that its contents depend on a global string variable, on a user-defined string function or on an Igor function (e.g., **WaveList**), you may want to force the pop-up menu to be updated at your command.

Usually, a ValDisplay control displays the value of a global variable or of an expression involving a global variable. If the global variable changes, the ValDisplay will automatically update. However, you can create a ValDisplay that displays a value that does not depend on a global variable. For example, it might display the result of an external function. In a case like this, the ValDisplay will not automatically update. You can update it by calling ControlUpdate.

When a SetVariable control is being edited, the text the user types isn't "accepted" (or processed) until the user presses Return or Enter. ControlUpdate effectively causes the named control to act as though the user has pressed one of those keys. If /A is specified, the currently active SetVariable control (if any) is affected this way. The motivation here is that the user may have typed a new value without having yet pressed return, and then may click a button in a different panel which runs a routine that uses the SetVariable value as input. The user expected the typed value to have been accepted but the variable has not yet been set. Calling ControlUpdate/A on the first panel will read the typed value in the variable, avoiding a discrepancy between the visible value of the SetVariable control and the actual value of the variable.

Examples

```
NewPanel;DoWindow/C PanelX
String/G popupList="First;Second;Third"
PopupMenu oneOfThree value=popupList        // popup shows "First"
popupList="1;2;3"                           // popup is unchanged
ControlUpdate/W=PanelX oneOfThree           // popup shows "1"
```

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **ValDisplay** and **WaveList** operations.

ConvertGlobalStringTextEncoding

`ConvertGlobalStringTextEncoding [flags ] originalTextEncoding, newTextEncoding,`
`[string , string, ...]`

The ConvertGlobalStringTextEncoding operation converts the contents of the specified global string variables or the contents of all global string variables in the data folder specified with the /DF flag from one text encoding to another.

The ConvertGlobalStringTextEncoding operation was added in Igor Pro 7.00.

By default, the contents of global string variables containing control characters (codes less than 32) other than carriage return (13), linefeed (10), and tab (9) are not converted. To convert the contents of global string variables containing control characters, you must include the /SKIP=0 flag.

In Igor Pro 7, the contents of all global string variables are assumed to be encoded as UTF-8. You should convert the text encoding of a global string variable if, for example, you have global string variables in experiments created prior to Igor7 and they contain non-ASCII characters. For those strings to be displayed and interpreted correctly in Igor Pro 7, their contents need to be converted to UTF-8.

Most users will have no need to worry about the text encoding of Igor global string variables since most global string variables do not contain non-ASCII text. You should not use this operation unless you have a thorough understanding of text encoding issues or are instructed to use it by someone who has a thorough understanding.

See **String Variable Text Encodings** on page III-428 for essential background information.

ConvertGlobalStringTextEncoding can work on a list of specific global string variables or on all of the global string variables in a data folder (/DF flag). When working on a data folder, it can work on just the data folder itself or recursively on sub-data folders as well.

Conversion does not change the characters that make up text - it merely changes the numeric codes used to represent those characters.

Parameters

originalTextEncoding specifies the original (current) text encoding used by the specified global string variables. See **Text Encoding Names and Codes** on page III-434 for a list of codes. It will typically be 2 (MacRoman), 3 (Windows-1252) or 4 (Shift JIS), depending on the system on which the string variables were created.

newTextEncoding specifies the output text encoding. See **Text Encoding Names and Codes** on page III-434 for a list of codes. It will typically be 1 which stands for UTF-8.

string , *string* , ... is a list of targeted global string variables. Only the name of string variables, not a path specification plus the name, is allowed. Therefore, the string variables must be in the current data folder. The list is optional and must be omitted if you use the /DF flag. Using both the /DF flag and a list of string variables is treated as an error. Use of local string variables in this list is also an error. Use **ConvertTextEncoding** to convert local string variables.

Flags

/CONV={*errorMode* [, *diagnosticsFlags*]}

errorMode determines how ConvertGlobalStringTextEncoding behaves if the conversion can not be done because the text can not be mapped to the specified text encoding. This occurs if the string variable's original text is not valid in the specified *originalTextEncoding* or if it contains characters that can not be represented in the specified *newTextEncoding*.

errorMode takes one of these values:

1:	Generate error. SetWaveTextEncoding returns an error to Igor.
2:	Use a substitute character for any unmappable characters. The substitute character for most text encodings is either control-Z or a question mark.
3:	Skip unmappable input characters. Any unmappable characters will be missing in the output.
4:	Use escape sequences representing any unmappable characters or invalid source text. If the source text is valid in the source text encoding but can not be represented in the destination text encoding, unmappable characters are replaced with \uXXXX where XXXX specifies the UTF-16 code point of the unmappable character in hexadecimal. If the conversion can not be done because the source text is not valid in the source text encoding, invalid bytes are replaced with \xXX where XX specifies the value of the invalid byte in

diagnosticsFlags is an optional bitwise parameter defined as follows:

Bit 0:	Emit diagnostic message if text conversion succeeds.
Bit 1:	Emit diagnostic message if text conversion fails.
Bit 2:	Emit diagnostic message if text conversion is skipped.
Bit 3:	Emit summary diagnostic message.

All other bits are reserved for future use.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

diagnosticsFlags defaults to 6 (bits 1 and 2 set) if the /DF flag is not present and to 14 (bits 1, 2 and 3 set) if the /DF flag is present.

ConvertGlobalStringTextEncoding skips text conversion if the global string variable's contents are detected as binary and you omit /SKIP=0.

/DF={*dfr*, *recurse*, *excludedDFR*}

dfr is a reference to a data folder. ConvertGlobalStringTextEncoding operates on all global string variables in the specified data folder. If *dfr* is null ($"") ConvertGlobalStringTextEncoding acts as if /DF was omitted.

If you use the /DF flag, you must omit the optional string variable list.

If *recurse* is 1, ConvertGlobalStringTextEncoding works recursively on all sub-data folders. Otherwise it affects only the data folder referenced by *dfr*.

excludedDFR is an optional reference to a data folder to be skipped by ConvertGlobalStringTextEncoding. For example, this command converts the string data for all global string variables in all data folders except for root:Packages and its sub-data folders:

```
ConvertGlobalStringTextEncoding /DF={root:,1,root:Packages} 4, 1
```

If *excludedDFR* is null ($"") ConvertGlobalStringTextEncoding acts as if *excludedDFR* was omitted and no data folders are excluded.

/SKIP=*skip* Skips conversion of a string variable if the string contains control characters (codes less than 32) other than carriage return (13), linefeed (10), and tab (9).

skip=0:	Do not skip conversion of any string variables.
skip=1:	Skip conversion of string variables containing binary data.
	This is the default behavior if /SKIP is omitted.

/Z[=z] Prevents procedure execution from aborting if ConvertGlobalStringTextEncoding generates an error. Use /Z or the equivalent, /Z=1, if you want to handle errors in your procedures rather than having execution abort.

/Z does not suppress invalid parameter errors. It suppresses only errors in doing text encoding reinterpretation or conversion.

Details

For general background information on text encodings, see **Text Encodings** on page III-409.

For background information on string variable text encodings, see **String Variable Text Encodings** on page III-428.

ConvertGlobalStringTextEncoding is used to change the numeric codes representing the text - i.e., to convert the content to a different text encoding. Its main use is to convert Igor Pro 6 global string variables from whatever text encoding they use, which typically will be MacRoman, Windows-1252, or Shift JIS, to UTF-8. UTF-8 is a form of Unicode which is more modern but is not backward compatible with Igor Pro 6. Igor Pro 7 assumes that string variables are encoded as UTF-8.

For example, if you have global string variables from Igor Pro 6 that are encoded in Japanese (Shift JIS) and contain non-ASCII characters, you should convert them to UTF-8. Otherwise you will get errors or garbled text if you print the string variable or use it, for example, to label a graph. This also applies to western text encoded as MacRoman or Windows-1252 which contain non-ASCII characters. Conversion changes the underlying numeric codes but does not change the characters represented by the text.

Output Variables

The ConvertGlobalStringTextEncoding operation returns information in the following variables:

`V_numConversionsSucceeded`	V_numConversionsSucceeded is set to the number of successful text conversions.
`V_numConversionsFailed`	V_numConversionsFailed is set to the number of unsuccessful text conversions.
`V_numConversionsSkipped`	V_numConversionsSkipped is set to the number of skipped text conversions. Text conversion is skipped if the string variable contains binary data and the /SKIP=0 flag is omitted.

Examples

```
// In the following examples 1 means UTF-8, 4 means Shift JIS.

// Convert specific strings' content from Shift JIS to UTF-8
ConvertGlobalStringTextEncoding 4, 1, string0, string1

// Convert all strings' content from Shift JIS to UTF-8
ConvertGlobalStringTextEncoding /DF={root:,1} 4, 1

// Same as before but exclude the root:Packages data folder
ConvertGlobalStringTextEncoding /DF={root:,1,root:Packages} 4, 1

// Convert all strings’ content from Shift JIS to UTF-8 except strings containing binary
ConvertGlobalStringTextEncoding /DF={root:,1}/SKIP=0 4, 1
```

See Also

Text Encodings on page III-409, **String Variable Text Encodings** on page III-428, **Text Encoding Names and Codes** on page III-434

ConvertTextEncoding, **SetWaveTextEncoding**

ConvertTextEncoding

`ConvertTextEncoding(`*`sourceTextStr, sourceTextEncoding, destTextEncoding,`*
`                                   `*`mapErrorMode, options`*`)`

ConvertTextEncoding converts text from one text encoding to another.

The ConvertTextEncoding function was added in Igor Pro 7.00.

In Igor Pro 7, all text in memory is assumed to be in UTF-8 format except for text stored in waves which can be stored in any text encoding. You might want to convert text from UTF-8 to Windows-1252 (Windows Western European), for example, to export it to a program that expects Windows-1252.

You might have text already loaded into Igor that you know to be in Windows-1252. To display it correctly in Igor Pro 7 you need to convert it to UTF-8.

You can also use ConvertTextEncoding to test if text is valid in a given text encoding, by specifying the same text encoding for *sourceTextEncoding* and *destTextEncoding*.

Parameters

sourceTextStr is the text that you want to convert.

sourceTextEncoding specifies the source text encoding.

destTextEncoding specifies the output text encoding.

See **Text Encoding Names and Codes** on page III-434 for a list of acceptable text encoding codes.

mapErrorMode determines what happens if an input character can not be mapped to the output text encoding because the character does not exist in the output text encoding. It takes one of these values:

options is a bitwise parameter which defaults to 0 and with the bits defined as follows:

All other bits are reserved and must be cleared.

1:	Generate error. The function returns "" and generates an error.
2:	Return a substitute character for the unmappable character. The substitute character for most text encodings is either control-Z or a question mark.
3:	Skip unmappable input character.
4:	Return escape sequences representing unmappable characters and invalid source text. If the source text is valid in the source text encoding but can not be represented in the destination text encoding, unmappable characters are replaced with \uXXXX where XXXX specifies the UTF-16 code point of the unmappable character in hexadecimal. The DemoUnmappable example function below illustrates this. If the conversion can not be done because the source text is not valid in the source text encoding, invalid bytes are replaced with \xXX where XX specifies the value of the invalid byte in hexadecimal. The DemoInvalid example function below illustrates this. If mapErrorMode is 2, 3 or 4, the function does not return an error in the event of an unmappable character.

Bit 0:	If cleared, in the event of a text conversion error, a null string is returned and an error is generated. Use this if you want to abort procedure execution if an error occurs. If set, in the event of a text conversion error, a null string is returned but no error is generated. Use this if you want to detect and handle a text conversion error. You can test for null using strlen as shown in the example below.
Bit 1:	If cleared (default), null bytes in *sourceTextStr* are considered invalid and ConvertTextEncoding returns an error. If set, null bytes are considered valid.
Bit 2:	If cleared (default) and *sourceTextEncoding* and *destTextEncoding* are the same, ConvertTextEncoding attempts to do the conversion anyway. If *sourceTextStr* is invalid in the specified text encoding, the issue is handled according to *mapErrorMode*. This allows you to check the validity of text whose text encoding you think you know, by passing 1 for *mapErrorMode* and 5 for *options*. Use **strlen** to test if the returned string is null, indicating that *sourceTextStr* is not valid in the specified text encoding. If set and *sourceTextEncoding* and *destTextEncoding* are the same, ConvertTextEncoding merely returns *sourceTextStr* without doing any conversion.

Details

ConvertTextEncoding returns a null result string if *sourceTextEncoding* or *destTextEncoding* are not valid text encoding codes or if a text conversion error occurs. You can test for a null string using strlen which returns NaN if the string is null.

If bit 0 of the *options* parameter is cleared, Igor generates an error which halts procedure execution. If it is set, Igor generates no error and you should test for null and attempt to handle the error, as illustrated by the example below.

A text conversion error occurs if *mapErrorMode* is 1 and the source text contains one or more characters that are not mappable to the destination text encoding. A text conversion error also occurs if the source text contains a sequence of bytes that is not valid in the source text encoding.

The "binary" text encoding (255) is not a real text encoding. If either *sourceTextEncoding* or *destTextEncoding* are binary (255), ConvertTextEncoding does no conversion and just returns *sourceTextStr* unchanged.

See **Text Encodings** on page III-409 for further details.

Example

In reading these examples, keep in mind that Igor converts escape codes such as “\u8C4A”, when they appear in literal text, to the corresponding UTF-8 characters. See **Unicode Escape Sequences in Strings** on page IV-14 for details.

```
Function DemoConvertTextEncoding()
   // Get text encoding codes for text the text encodings used below
   Variable teUTF8 = TextEncodingCode("UTF-8")
   Variable teWindows1252 = TextEncodingCode("Windows-1252")
```

```
    Variable teShiftJIS = TextEncodingCode("ShiftJIS")

    // Convert from Windows-1252 to UTF-8
    String source = "Division sign: " + num2char(0xF7)
    String result = ConvertTextEncoding(source, teWindows1252, teUTF8, 1, 0)
    Print result

    // Convert unmappable character from UTF-8 to Windows-1252
    // \u8C4A is an escape sequence representing a Japanese character in Unicode
    // for which there is no corresponding character in Windows-1252

    // Demonstrate mapErrorMode = 1 (fail unmappable character)
    source = "Unmappable character causes failure: {\u8C4A}"
    // Pass 1 for options parameter to tell Igor to ignore error and let us handle it
    result = ConvertTextEncoding(source, teUTF8, teWindows1252, 1, 1)
    Variable len = strlen(result)     // Will be NaN if conversion failed
    if (NumType(len) == 2)
        Print "Conversion failed (as expected). Result is NULL."
        // You could cope with this error by trying again with the mapErrorMode
        // parameter set to 2, 3 or 4.
    else
        // We should not get here
        Print "Conversion succeeded (should not happen)."
        Print result
    endif

    // Demonstrate mapErrorMode = 2 (substitute for unmappable character)
    source = "Unmappable character replaced by question mark: {\u8C4A}"
    result = ConvertTextEncoding(source, teUTF8, teWindows1252, 2, 0)
    Print result                      // Prints "?" in place of unmappable character

    // Demonstrate mapErrorMode = 3 (skip unmappable character)
    source = "Unmappable character skipped: {\u8C4A}"
    result = ConvertTextEncoding(source, teUTF8, teWindows1252, 3, 0)
    Print result                      // Skips unmappable character

    // Demonstrate mapErrorMode = 4 (insert escape sequence for unmappable character)
    source = "Unmappable character replaced by escape sequence: {\u8C4A}"
    result = ConvertTextEncoding(source, teUTF8, teWindows1252, 4, 0)
    Print result        // Unmappable character represented as escape sequence

    // Demonstrate mapErrorMode = 4 (insert escape sequence for unmappable character)
    source = "Unmappable character replaced by escape sequence: {\u8C4A}"
    // First convert UTF-8 to Shift_JIS (Japanese). This will succeed.
    result = ConvertTextEncoding(source, teUTF8, teShiftJIS, 1, 0)
    // Next convert Shift_JIS (Japanese) to Windows-1252. The character can not
    // be mapped and is replaced by an escape sequence.
    result = ConvertTextEncoding(result, teShiftJIS, teWindows1252, 4, 0)
    Print result        // Unmappable character represented as escape sequence
End

// Demo unmappable character
// In this example, the source text is valid but not representable in destination
// text encoding. Because we pass 4 for the mapErrorMode parameter, ConvertTextEncoding
// uses an escape sequenceto represent the unmappable text.
Function DemoUnmappable()
    String input = "\u2135"// Alef symbol - available in UTF-8 but not in MacRoman
    String output = ConvertTextEncoding(input, 1, 2, 4, 0)
    Print output            // Prints "\u2135"
End

// Demo invalid input text
// In this example, the source text is invalid in the source text encoding.
// Because we pass 4 for the mapErrorMode parameter, ConvertTextEncoding uses an escape
    sequences
// to represent the invalid text.
Function DemoInvalidInput()
    String input = "\x8E"  // Represents "é" in MacRoman but is not valid in UTF-8
    String output = ConvertTextEncoding(input, 1, 2, 4, 0)
    Print output            // Prints "\x8E"
End
```

See Also

Text Encodings on page III-409, **Text Encoding Names and Codes** on page III-434

TextEncodingCode, **TextEncodingName**, **SetWaveTextEncoding**

ConvertGlobalStringTextEncoding, **String Variable Text Encoding Error Example** on page III-428

ConvexHull

`convexHull [`*`flags`*`]`*`xwave, ywave`*

`convexHull [`*`flags`*`]` ***`tripletWave`***

The ConvexHull operation calculates the convex hull in either 2 or 3 dimensions. The dimensionality is deduced from the input wave(s). If the input consists of two 1D waves of the same length, the number of dimensions is assumed to be 2. If the input consists of a single triplet wave (a wave of 3 columns), then the number of dimensions is 3.

In 2D cases the operation calculates the convex hull and produces the result in a pair of x and y waves, W_XHull and W_YHull.

In 3D cases the operation calculates the convex hull and stores it in a triplet wave M_Hull that describes ordered facets of the convex hull.

ConvexHull returns an error if the input waves have fewer than 3 data points.

Flags

/C	(2D convex hull only) adds the first point to the end of the W_XHull and W_YHull waves so that the first and the last points are the same.
/E	(3D case only) if you use this flag the operation also creates a wave that lists the indices of the vertices which are not part of the convex hull, i.e., vertices which are interior to the hull. The output is in the wave W_HullExcluded.
/I	(3D convex hull only) use this flag to get the corresponding index of the vertex as the fourth column in the M_Hull wave.
/S	(3D convex hull only) use this flag if you want the resulting M_Hull to have NaN lines separating each triangle.
/T=*tolerance*	(3D case only) default tolerance for measuring if a point is inside or outside the convex hull is $1.0x10^{-20}$. You can use any other positive value.
/V	(3D case only) if you use this flag the operation also creates a wave containing the output in a list of vertex indices. The wave M_HullVertices contains a row per triangle where each entry on a row corresponds to the index of the input vertex.
/Z	No error reporting.

Examples

```
Make/O/N=33 xxx=gnoise(5),yyy=gnoise(7)
Convexhull/c xxx,yyy
Display W_Yhull vs W_Xhull
Appendtograph yyy vs xxx
ModifyGraph mode(yyy)=3,marker(yyy)=8,rgb(W_YHull)=(0,15872,65280)
```

See Also

Triangulate3D

Convolve

`Convolve [/A/C]` ***`srcWaveName, destWaveName`*** **`[,`** ***`destWaveName`*****`]`**...

The Convolve operation convolves *srcWaveName* with each destination wave, putting the result of each convolution in the corresponding destination wave.

Convolve is not multidimensional aware. Some multidimensional convolutions are covered by the **MatrixConvolve**, **MatrixFilter**, and **MatrixOp** operations

Flags

/A	Acausal linear convolution.
/C	Circular convolution.

Details

Convolve performs linear convolution unless the /C or /A flag is used. See the diagrams in the examples below.

Depending on the type of convolution, the destination waves' lengths may increase. *srcWaveName* is not altered unless it also appears as a destination wave.

If *srcWaveName* is real-valued, each destination wave must be real-valued, and if *srcWaveName* is complex, each destination wave must be complex, too. Double and single precision waves may be freely intermixed; calculations are performed in the higher precision.

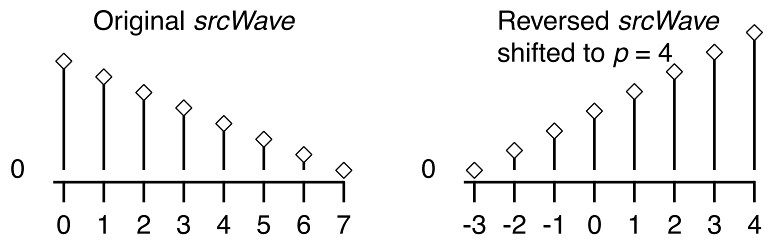
The linear convolution equation is:

$$destWaveOut[p] = \sum_{m=0}^{N-1} destWaveIn[m] \cdot srcWave[p-m]$$

where *N* is the number of points in the longer of *destWaveIn* and *srcWave*. For circular convolution, the index [*p* -*m*] is wrapped around when it exceeds the range of [0,numpnts(*srcWave*)-1]. For acausal convolution, when [*p* -*m*] exceeds the range a zero value is substituted for *srcWave* [*p* -*m*]. Similar operations are applied to *destWaveIn* [*m*].

Another way of looking at this equation is that, for all *p*, *destWaveOut*[*p*] equals the sum of the point-by-point products from 0 to *p* of the destination wave and an end-to-end reversed copy of the source wave that has been shifted to the right by *p*.

The following diagram shows the reversed/shifted *srcWave* that would be combined with *destWaveIn*. The points numbered 0 through 4 of the reversed *srcWave* would be multiplied with *destWaveIn*[0…4] and summed to produce *destWaveOut*[4]:



For linear and acausal convolution, the destination wave is first zero-padded by one less than the length of the source wave. This prevents the "wrap-around" effect that occurs in circular convolution. The zero-padded points are removed after acausal convolution, and retained after linear convolution. The X scaling of the waves is ignored.

The convolutions are performed by transforming the source and destination waves with the Fast Fourier Transform, multiplying them in the frequency domain, and then inverse-transforming them into the destination wave(s).

The convolution is performed in segments if the resulting wave has more than 256 points and the destination wave has twice as many points as the source wave. For acausal convolution, the length of the resulting wave is considered to be (numpnts(*srcWaveName*) +numpnts(*destWaveName*)-1) for this calculation.

Applications

The usual application of convolution is to compute the response of a linear system defined by its impulse response to an input signal. *srcWaveName* would contain the impulse response, and the destination wave would initially contain the input signal. After the Convolve operation has completed, the destination wave contains the output signal.

Use linear convolution when the source wave contains an impulse response (or filter coefficients) where the first point of *srcWave* corresponds to no delay (*t* = 0).

Use circular convolution for the case where the data in *srcWaveName* and *destWaveName* are considered to endlessly repeat (or “wrap around” from the end back to the start), which means no zero padding is needed.

Use acausal convolution when the source wave contains an impulse response where the middle point of *srcWave* corresponds to no delay (*t* = 0).

See Also

Convolution on page III-253 for illustrated examples. **MatrixOp**.

References

A very complete explanation of circular and linear convolution can be found in sections 2.23 and 2.24 of Rabiner and Gold, *Theory and Application of Digital Signal Processing*, Prentice Hall, 1975.

CopyFile

`CopyFile [`*`flags`*`][`*`srcFileStr`*`] [as `*`destFileOrFolderStr`*`]`

The CopyFile operation copies a file on disk.

Parameters

srcFileStr can be a full path to the file to be copied (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*.

If Igor can not determine the location of the source file from *srcFileStr* and *pathName*, it displays a dialog allowing you to specify the source file.

destFileOrFolderStr is interpreted as the name of (or path to) an existing folder when /D is specified, otherwise it is interpreted as the name of (or path to) a possibly existing file.

If *destFileOrFolderStr* is a partial path, it is relative to the folder associated with pathName.

If /D is specified, the source file is copied inside the folder using the source file’s name.

If Igor can not determine the location of the destination file from *pathName*, *srcFileStr*, and *destFileOrFolderStr*, it displays a Save File dialog allowing you to specify the destination file (and folder).

If you use a full or partial path for either *srcFileStr* or *destFileOrFolderStr*, see **Path Separators** on page III-401 for details on forming the path.

Folder paths should not end with single Path Separators. See the **Details** section for **MoveFolder**.

Flags

/D	Interprets *destFileOrFolderStr* as the name of (or path to) an existing folder (or “directory”). Without /D, *destFileOrFolderStr* is interpreted as the name of (or path to) a file. If *destFileOrFolderStr* is not a full path to a folder, it is relative to the folder associated with pathName.
/I [=*i*]	Specifies the level of user interactivity. /I=0: Interactive only if one or *srcFileStr* or *destFileOrFolderStr* is not specified or if the source file is missing. (Same as if /I was not specified.) /I=1: Interactive even if *srcFileStr* is specified and the source file exists. /I=2: Interactive even if *destFileOrFolderStr* is specified. /I=3: Interactive even if *srcFileStr* is specified, the source file exists, and *destFileOrFolderStr* is specified. Same as /I only.
/M=*messageStr*	Specifies the prompt message in the Open File dialog. If /S is not used, then *messageStr* will be used for both Open File and for Save File dialogs. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/O	Overwrites any existing destination file.

/P=*pathName*	Specifies the folder to look in for the source file, and the folder into which the file is copied. *pathName* is the name of an existing symbolic path. Using /P means that both *srcFileStr* and *destFileOrFolderStr* must be either simple file or folder names, or paths relative to the folder specified by *pathName*.
/S=*saveMessageStr*	Specifies the prompt message in the Save File dialog.
/Z [=z]	Prevents procedure execution from aborting if it attempts to copy a file that does not exist. Use /Z if you want to handle this case in your procedures rather than aborting execution. /Z=0: Same as no /Z. /Z=1: Copies a file only if it exists. /Z alone has the same effect as /Z=1. /Z=2: Copies a file if it exists or displays a dialog if it does not exist.

Variables

The CopyFile operation returns information in the following variables:

`V_flag`	Set to zero if the file was copied, to -1 if the user cancelled either the Open File or Save File dialogs, and to some nonzero value if an error occurred, such as the specified file does not exist.
`S_fileName`	Stores the full path to the file that was copied. If an error occurred or if the user cancelled, it is set to an empty string.
`S_path`	Stores the full path to the file copy. If an error occurred or if the user cancelled, it is set to an empty string.

Examples

Copy a file within the same folder using a new name:

```
CopyFile/P=myPath "afile.txt" as "destFile.txt"
```

Copy a file into subfolder using the original name (using /P):

```
CopyFile/D/P=myPath "afile.txt" as ":subfolder"
Print S_Path     // prints "Macintosh HD:folder:subfolder:afile.txt"
```

Copy file into subfolder using the original name (using full paths):

```
CopyFile/D "Macintosh HD:folder:afile.txt" as "Server:archive"
```

Copy a file from one folder to another, assigning the copy a new name:

```
CopyFile "Macintosh HD:folder:afile.txt" as "Server:archive:destFile.txt"
```

Copy user-selected file in any folder as destFile.txt in myPath folder (prompt to save even if destFile.txt doesn't exist):

```
CopyFile/I=2/P=myPath as "destFile.txt"
```

Copy user-selected file in any folder as destFile.txt in any folder:

```
CopyFile as "destFile.txt"
```

See Also

The **Open**, **MoveFile**, **DeleteFile**, and **CopyFolder** operations. The **IndexedFile** function. **Symbolic Paths** on page II-21.

CopyFolder

`CopyFolder [`*`flags`*`][`*`srcFolderStr`*`] [as `*`destFolderStr`*`]`

The CopyFolder operation copies a folder (and its contents) on disk.

Parameters

srcFolderStr can be a full path to the folder to be copied (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a folder inside the folder associated with *pathName*.

If Igor can not determine the location of the folder from *srcFolderStr* and *pathName*, it displays a dialog allowing you to specify the source folder.

Warning: *The CopyFolder command can destroy data* by overwriting another folder and contents!

When overwriting an existing folder on disk, CopyFolder will do so only if permission is granted by the user. The default behavior is to display a dialog asking for permission. The user can alter this behavior via the Miscellaneous Settings dialog's Misc category.

If permission is denied, the folder will not be copied and V_Flag will return 1088 (Command is disabled) or 1275 (You denied permission to overwrite a folder). Command execution will cease unless the /Z flag is specified.

If /P=*pathName* is given, but *srcFolderStr* is not, then the folder associated with *pathName* is copied.

destFolderStr can be a full path to the output (destination) folder (in which case /P is not needed), or a partial path relative to the folder associated with *pathName*.

An error is returned if the destination folder would be inside the source folder.

If Igor can not determine the location of the destination folder from *destFolderStr* and *pathName*, it displays a dialog allowing you to specify or create the destination folder.

If you use a full or partial path for either folder, see **Path Separators** on page III-401 for details on forming the path.

Flags

/D Interprets *destFolderStr* as the name of (or path to) an existing folder (or directory) to copy the source folder into. Without /D, *destFolderStr* is interpreted as the name of (or path to) the copied folder.

If *destFolderStr* is not a full path to a folder, it is relative to the folder associated with *pathName*.

/I [=*i*] Specifies the level of user interactivity.

/I=0:	Interactive only if the source or destination folder is not specified or if the source folder is missing. (Same as if /I was not specified.)
/I=1:	Interactive even if the source folder is specified and it exists.
/I=2:	Interactive even if *destFolderStr* is specified.
/I=3:	Interactive even if the source folder is specified, the source folder exists, and *destFolderStr* is specified. Same as /I only.

/M=*messageStr* Specifies the prompt message in the Select (source) Folder dialog. If /S is not used, then *messageStr* will be used for the Select Folder dialog and for the Create Folder dialog. But see **Prompt Does Not Work on Macintosh** on page IV-137.

/O Overwrite existing destination folder, if any.

On Macintosh, a Macintosh-style overwrite-move is performed in which the source folder completely replaces the destination folder.

On Windows, a Windows-style mix-in move is performed in which the contents of the source folder are moved into the destination folder, replacing any same-named files but leaving other files in place.

/P=*pathName* Specifies the folder to look in for the source folder. *pathName* is the name of an existing symbolic path.

If *srcFolderStr* is not specified, the folder associated with *pathName* is copied.

Using /P means that *srcFolderStr* (if specified) and *destFolderStr* must be either simple folder names or paths relative to the folder specified by *pathName*.

/S=*saveMessageStr* Specifies the prompt message in the Create Folder dialog.

/Z [=z]	Prevents procedure execution from aborting if it attempts to copy a file that does not exist. Use /Z if you want to handle this case in your procedures rather than aborting execution. /Z=0: Same as no /Z. /Z=1: Copies a folder only if it exists. /Z alone has the same effect as /Z=1. /Z=2: Copies a folder if it exists or displays a dialog if it does not exist.

Variables

The CopyFolder operation returns information in the following variables:

`V_flag`	Set to zero if the folder was copied, to -1 if the user cancelled either the Select Folder or Create Folder dialogs, and to some nonzero value if an error occurred, such as the specified file does not exist.
`S_fileName`	Stores the full path to the folder that was copied, with a trailing colon. If an error occurred or if the user cancelled, it is set to an empty string.
`S_path`	Stores the full path to the folder copy, with a trailing colon. If an error occurred or if the user cancelled, it is set to an empty string.

Details

You can use only /P=*pathName* (without *srcFolderStr*) to specify the source folder to be copied.

Folder paths should not end with single Path Separators. See the **Details** section for **MoveFolder**.

Examples

Copy the folder that the current experiment is stored in:

```
CopyFile/P=home as "HD:Copy Of Folder Experiment Is In"
```

Copy the Igor Extensions Folder to the Windows desktop:

```
CopyFile/D/P=Igor ":Igor Extensions" as "C:WINDOWS:Desktop"
```

Ask the user to select a folder, starting with the Igor folder, and then make a copy of that folder in the Igor Pro 7 folder:

```
CopyFile/I=2/P=Igor as "::Folder Copy"
```

Copy an entire disk inside a folder:

```
CopyFolder/O/D "Floppy" as "HD:Desktop Folder:Copy Into Here"
```

See Also

Open, **MoveFile**, **DeleteFile**, **MoveFolder**, **NewPath**, and **IndexedDir** operations, and **Symbolic Paths** on page II-21.

CopyScales

`CopyScales [/I/P] `*`srcWaveName, waveName`*` [, `*`waveName`*`]…`

The CopyScales operation copies the x, y, z, and t scaling, x, y, z, and t units, the data Full Scale and data units from *srcWaveName* to the other waves.

Flags

/I	Copies the x, y, z, and t scaling in inclusive format.
/P	Copies the x, y, z, and t scaling in slope/intercept format (x0, dx format).

Details

Normally the x, y, z, and t (dimension) scaling is copied in min/max format. However, if you use /P, the dimension scaling is copied in slope/intercept format so that if *srcWaveName* and the other waves have differing dimension size (number of points if the wave is a 1D wave), then their dimension values will still match for the points they have in common. Similarly, /I uses the inclusive variant of the min/max format. See **SetScale** for a discussion of these dimension scaling formats.

If a wave has only one point, /I mode reverts to /P mode.

CopyScales copies scales only for those dimensions that *srcWaveName* and *waveName* have in common.

See Also

x, **y**, **z**, and **t** scaling functions.

Correlate

```
Correlate [/AUTO/C/NODC] srcWaveName, destWaveName [, destWaveName]…
```

The Correlate operation correlates *srcWaveName* with each destination wave, putting the result of each correlation in the corresponding destination wave.

Flags

/AUTO	Auto-correlation scaling. This forces the X scaling of the destination wave's center point to be x=0, and divides the destination wave by the center point's value so that the center value is exactly 1.0. If srcWaveName and destWaveName do not have the same number of points, this flag is ignored. /AUTO is not compatible with /C.
/C	Circular correlation. (See **Compatibility Note**.)
/NODC	Removes the mean from the source and destination waves before computing the correlations. Removing the mean results in the un-normalized auto- or cross-covariance. "DC" is an abbrevation of "direct current", an electronics term for the non-varying average value component of a signal.

Details

Note: To compute a single-value correlation number use the **StatsCorrelation** function which returns the Pearson's correlation coefficient of two same-length waves.

Correlate performs linear correlation unless the /C flag is used.

Depending on the type of correlation, the length of the destination may increase. *srcWaveName* is not altered unless it also appears as a destination wave.

If the source wave is real-valued, each destination wave must be real-valued and if the source wave is complex, each destination wave must be complex, too. Double and single precision waves may be freely intermixed; calculations are performed in the higher precision.

The linear correlation equation is:

$$destWaveOut[p] = \sum_{m=0}^{N-1} srcWave[m] \cdot destWaveIn[p+m]$$

where N is the number of points in the longer of *destWaveIn* and *srcWave*.

For circular correlation, the index [*p* +*m*] is wrapped around when it exceeds the range of `[0,numpnts(destWaveIn)-1]`. For linear correlation, when [*p* +*m*] exceeds the range a zero value is substituted for *destWaveIn*[*p* +*m*]. When *m* exceeds `numpnts(srcWave)-1`, 0 is used instead of *srcWave*[*m*].

Comparing this with the **Convolve** operation, which is the linear convolution:

$$destWaveOut[p] = \sum_{m=0}^{N-1} destWaveIn[m] \cdot srcWave[p-m]$$

you can see that the only difference is that for correlation the source wave is *not* reversed before shifting and combining with the destination wave.

The Correlate operation is not multidimensional aware. For details, see **Analysis on Multidimensional Waves** on page II-86 and in particular **Analysis on Multidimensional Waves** on page II-86.

Compatibility Note

Prior to Igor Pro 5, Correlate/C scaled and rotated the results improperly (the result was often rotated left by one and the X scaling was entirely negative).

Now the destination wave's X scaling is unaltered and it does not rotate the result. You can force the old behavior for compatibility with old procedures that depend on the old behavior by setting `root:V_oldCorrelationScaling=1`.

A better way to get identical Correlate/C results with all versions of Igor Pro is to use this code, which rotates the result so that x=0 is always the first point in *destWave*, no matter which Igor Pro version runs this code (currently, it doesn't change anything and runs extremely quickly because it does no rotation):

```
Correlate/C srcWave, destWave
Variable pointAtXEqualZero= x2pnt(destWave,0)     // 0 for Igor Pro 5
Rotate -pointAtXEqualZero,destWave
SetScale/P x, 0, DimDelta(destWave,0), "", destWave
```

Applications

A common application of correlation is to measure the similarity of two input signals as they are shifted by one another.

Often it is desirable to normalize the correlation result to 1.0 at the maximum value where the two inputs are most similar. To normalize *destWaveOut*, compute the RMS values of the input waves and the number of points in each wave:

```
WaveStats/Q srcWave
Variable srcRMS = V_rms
Variable srcLen = numpnts(srcWave)

WaveStats/Q destWave
Variable destRMS = V_rms
Variable destLen = numpnts(destWave)

Correlate srcWave, destWave                 // overwrites destWave

// now normalize to max of 1.0
destWave /= (srcRMS * sqrt(srcLen) * destRMS * sqrt(destLen))
```

Another common application is using autocorrelation (where *srcWaveName* and *destWaveName* are the same) to determine Power Spectral Density. In this case it better to use the **DSPPeriodogram** operation which provides more options.

See Also

Convolution on page III-253 and **Correlation** on page III-255 for illustrated examples. See the **Convolve** operation for algorithm implementation details, which are identical except for the lack of source wave reversal, and the lack of the /A (acausal) flag.

The **MatrixOp**, **StatsCorrelation**, **StatsCircularCorrelationTest**, **StatsLinearCorrelationTest**, and **DSPPeriodogram** operations.

References

An explanation of autocorrelation and Power Spectral Density (PSD) can be found in Chapter 12 of Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

WaveMetrics provides Igor Technical Note 006, "DSP Support Macros" that computes the PSD with options such as windowing and segmenting. See the Technical Notes folder. Some of the techniques discussed there are available as Igor procedure files in the "WaveMetrics Procedures:Analysis:" folder.

Wikipedia: http://en.wikipedia.org/wiki/Correlation

Wikipedia: http://en.wikipedia.org/wiki/Cross_covariance

Wikipedia: http://en.wikipedia.org/wiki/Autocorrelation_function

cos

`cos(`*`angle`*`)`

The cos function returns the cosine of *angle* which is in radians.

In complex expressions, *angle* is complex, and `cos(`*angle*`)` returns a complex value:

$$\cos(x+iy)=\cos(x)\cosh(y)-i\sin(x)\sinh(y).$$

See Also

acos, **sin**, **tan**, **sec**, **csc**, **cot**

cosh

`cosh(num)`

The cosh function returns the hyperbolic cosine of *num*:

$$\cosh(x)=\frac{e^{x}+e^{-x}}{2}.$$

In complex expressions, *num* is complex, and `cosh(num)` returns a complex value.

See Also

sinh, **tanh**, **coth**

CosIntegral

`CosIntegral(z)`

The CosIntegral(*z*) function returns the cosine integral of *z*.

If *z* is real, a real value is returned. If *z* is complex then a complex value is returned.

The CosIntegral function was added in Igor Pro 7.00.

Details

The cosine integral is defined by

$$Ci(z)=\gamma+\ln(z)+\int_0^z\frac{\cos(t)-1}{t}dt, \qquad \left(|\arg(z)|<\pi\right)$$

where γ is the Euler-Mascheroni constant 0.5772156649015328606065.

IGOR computes the CosIntegral using the expression:

$$Ci(z)=-\frac{Z^2}{4}\,{}_2F_3\left(1,1;2,2,\frac{3}{2};-\frac{z^2}{4}\right)+\ln(z)+\gamma,$$

References

Abramowitz, M., and I.A. Stegun, "Handbook of Mathematical Functions", Dover, New York, 1972. Chapter 5.

See Also

SinIntegral, **ExpIntegralE1**, **hyperGPFQ**

cot

`cot(angle)`

The cot function returns the cotangent of *angle* which is in radians.

In complex expressions, *angle* is complex, and cot(*angle*) returns a complex value.

See Also

sin, **cos**, **tan**, **sec**, **csc**

coth

`coth(`***`num`***`)`

The coth function returns the hyperbolic cotangent of *num*:

$$\coth(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}}.$$

In complex expressions, *num* is complex, and coth(*num*) returns a complex value.

See Also

sinh, **cosh**, **tanh**

CountObjects

`CountObjects(`***`sourceFolderStr, objectType`***`)`

The CountObjects function returns the number of objects of the specified type in the data folder specified by the string expression.

CountObjectsDFR is preferred.

Parameters

sourceFolderStr can be either `":"` or `""` to specify the current data folder. You can also use a full or partial data folder path. *objectType* should be one of the following values:

1	Waves
2	Numeric variables
3	String variables
4	Data folders

See Also

Chapter II-8, **Data Folders**, and the **GetIndexedObjName** function.

CountObjectsDFR

`CountObjectsDFR(`***`dfr,objectType`***`)`

The CountObjectsDFR function returns the number of objects of the specified type in the data folder specified by the data folder reference *dfr*.

CountObjectsDFR is the same as CountObjects except the first parameter, *dfr*, is a data folder reference instead of a string containing a path.

Parameters

objectType is one of the following values:

1	Waves
2	Numeric variables
3	String variables
4	Data folders

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-72.

GetIndexedObjNameDFR

cpowi

`cpowi(`*`num, ipow`*`)`

This function is obsolete as the exponentiation operator ^ handles complex expressions with any combination of real, integer and complex arguments. See **Operators** on page IV-5. The cpowi function returns a complex number resulting from raising complex *num* to integer-valued power *ipow*. *ipow* can be positive or negative, but if it is not an integer cpowi returns (NaN, NaN).

CreateAliasShortcut

`CreateAliasShortcut` `[`***`flags`***`][`***`targetFileDirStr`***`] [`**`as`** ***`aliasFileStr`***`]`

The CreateAliasShortcut operation creates an alias (*Macintosh*) or shortcut (*Windows*) file on disk. The alias can point to either a file or a folder. The file or folder pointed to is called the “target” of the alias or shortcut.

Parameters

targetFileDirStr can be a full path to the file or folder to make an alias or shortcut for, a partial path relative to the folder associated with /P=*pathName*, or the name of a file or folder in the folder associated with *pathName*.

If Igor can not determine the location of the file or folder from *targetFileDirStr* and *pathName*, it displays a dialog allowing you to specify a target file. Use /D to select a folder as the alias target, instead.

aliasFileStr can be a full path to the created alias file, a partial path relative to the folder associated with *pathName* if specified, or the name of a file in the folder associated with *pathName*.

If Igor can not determine the location of the alias or shortcut file from *aliasFileStr* and *pathName*, it displays a File Save dialog allowing you to create the file.

If you use a full or partial path for either *targetFileDirStr* or *aliasFileStr*, see **Path Separators** on page III-401 for details on forming the path.

Folder paths should not end with single path separators. See the **MoveFolder Details** section.

Flags

/D	Uses the Select Folder dialog rather than Open File dialog when *targetFileDirStr* is not fully specified.
/I [=*i*]	Specifies the level of user interactivity. /I=0: Interactive only if one or *targetFileDirStr* or *aliasFileStr* is not specified or if the target file is missing. (Same as if /I was not specified.) /I=1: Interactive even if *targetFileDirStr* is fully specified and the target file exists. /I=2: Interactive even if *targetFileDirStr* is specified. /I=3: Interactive even if *targetFileDirStr* is specified and the target file exists. Same as /I only.
/M=*messageStr*	Specifies the prompt message in the Open File or Select Folder dialog. If /S is not specified, then *messageStr* will be used for Open File (or Select Folder) and for Save File dialogs. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/O	Overwrites any existing file with the alias or shortcut file.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/S=*saveMessageStr*	Specifies the prompt message in the Save File dialog when creating the alias or shortcut file.

/Z[=z]	Prevents procedure execution from aborting the procedure tries to create an alias or shortcut for a file or folder that does not exist. Use /Z if you want to handle this case in your procedures rather than aborting execution.
	/Z=0: Same as no /Z.
	/Z=1: Creates an alias to a file or folder only if it exists. /Z alone has the same effect as /Z=1.
	/Z=2: Creates an alias to a file or folder only if it exists and displays a dialog if it does not exist.

Variables

The CreateAliasShortcut operation returns information in the following variables:

`V_flag`	Status output:	
	0	Created an alias or shortcut file.
	1	User cancelled any of the Open File, Select Folder, or Save File dialogs.
	Other:	An error occurred, such as the target file does not exist.
`S_fileName`	Full path to the target file or folder. If an error occurred or if the user cancelled, it is an empty string.	
`S_path`	Full path to the created alias or shortcut file. If an error occurred or if the user cancelled, it is an empty string.	

Examples

Create a shortcut (*Windows*) to the current experiment, on the desktop:

```
String target= Igorinfo(1)+".pxp" // experiments are usually .pxp on Windows
CreateAliasShortcut/O/P=home target as "C:WINDOWS:Desktop:"+target
```

Create an alias (*Macintosh*) to the VDT XOP in the Igor Extensions folder:

```
String target= ":More Extensions:Data Acquisition:VDT"
CreateAliasShortcut/O/P=Igor target as ":Igor Extensions:VDT alias"
```

Create an alias to the “HD 2” disk. Put the alias on the desktop:

```
CreateAliasShortcut/D/O "HD 2" as "HD:Desktop Folder:Alias to HD 2"
```

See Also

Symbolic Paths on page II-21.

The **Open**, **MoveFile**, **DeleteFile**, and **GetFileFolderInfo** operations. The **IgorInfo** and **ParseFilePath** functions.

CreateBrowser

`CreateBrowser [/M]` `[`***`keyword = value`***` [, `***`keyword = value`***` …]]`

The CreateBrowser operation creates a data browser window.

Documentation for the CreateBrowser operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "CreateBrowser"
```

CreationDate

`CreationDate(`*`waveName`*`)`

Returns creation date of wave as an Igor date/time value, which is the number of seconds from 1/1/1904.

The returned value is valid for waves created with Igor Pro 3.0 or later. For waves created in earlier versions, it returns 0.

See Also

ModDate.

Cross

`Cross [/DEST=destWave /FREE /T /Z] vectorA, vectorB [, vectorC]`

The Cross operation computes the cross products *vectorA* x *vectorB* and *vectorA* x (*vectorB* x *vectorC*). Each vector is a 1D real wave containing 3 rows. Stores the result in the wave W_Cross in the current data folder.

Flags

/DEST=*destWave*	Stores the cross product in the wave specified by *destWave*. The destination wave is overwritten if it exists. The destination wave must be different from the input waves. The operation creates a wave reference for the destination wave if called in a user-defined function. See **Automatic Creation of WAVE References** on page IV-66 for details. If you omit /DEST, the operation stores the result in the wave W_Cross in the current data folder. Requires Igor7 or later.
/FREE	When used with /DEST, the destination wave is created as a free wave. See **Free Waves** on page IV-84 for details on free waves. /FREE is allowed in user-defined functions only. Requires Igor7 or later.
/T	Stores output in a row instead of a column in W_Cross.
/Z	Generates no errors for any unsuitable inputs.

csc

`csc(angle)`

The csc function returns the cosecant of *angle* which is in radians.

$$\csc(x) = \frac{1}{\sin(x)}.$$

In complex expressions, *angle* is complex, and csc(*angle*) returns a complex value.

See Also

sin, **cos**, **tan**, **sec**, **cot**

csch

`csch(x)`

The csch function returns the hyperbolic cosecant of *x*.

$$\mathrm{csch}(x) = \frac{1}{\sinh(x)} = \frac{2}{e^{x} - e^{-x}}.$$

In complex expressions, *x* is complex, and csch(*x*) returns a complex value.

See Also

cosh, **tanh**, **coth**, **sech**

CsrInfo

`CsrInfo(`*`cursorName`*` [, `*`graphNameStr`*`])`

The CsrInfo function returns a keyword-value pair list of information about the specified cursor (*cursorName* is A through J) in the top graph or graph specified by *graphNameStr*. It returns `""` if the cursor is not in the graph.

Details

The returned string contains information about the cursor in the following format:

```
TNAME:traceName; ISFREE:freeNum;POINT:xPointNumber;[YPOINT:yPointNumber;]
   RECREATION:command;
```

The *traceName* value is the name of the graph trace or image to which it is attached or which supplies the x (and y) values even if the cursor isn't attached to it.

If TNAME is empty, fields POINT, ISFREE, and YPOINT are not present.

The *freeNum* value is 1 if the cursor is not attached to anything, 0 if attached to a trace or image.

The POINT value is the same value **pcsr** returns.

The YPOINT keyword and value are present only when the cursor is attached to a two-dimensional item such as an image, contour, or waterfall plot or when the cursor is free. Its value is the same as returned by **qcsr**.

If cursor is free, POINT and YPOINT values are fractional relative positions (see description in the **Cursor** command).

The RECREATION keyword contains the Cursor commands (including /W) necessary to regenerate the current settings.

Examples

```
Variable aExists= strlen(CsrInfo(A)) > 0    // A is a name, not a string
Variable bIsFree= NumberByKey("ISFREE",CsrInfo(B,"Graph0"))
```

See Also

Programming With Cursors on page II-249.

Cursors — Moving Cursor Calls Function on page IV-316.

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

CsrWave

`CsrWave(`*`cursorName`*` [, `*`graphNameStr`*` [, `*`wantTraceName`*`]])`

The CsrWave function returns a string containing the name of the wave the specified cursor (A through J) is on in the top (or named) graph. If the optional *wantTraceName* is nonzero, the trace name is returned. A trace name is the wave name with optional instance notation (see **ModifyGraph (traces)**).

Details

The name of a wave by itself is not sufficient to identify the wave because it does not specify what data folder contains the wave. Thus, if you are calling CsrWave for the purpose of passing the wave name to other procedures, you should use the **CsrWaveRef** function instead. Use CsrWave if you want the name of the wave to use in an annotation or a notebook.

Examples

```
String waveCursorAIsOn = CsrWave(A)            // not CsrWave("A")
String waveCursorBIsOn = CsrWave(B,"Graph0")   // in specified graph
String traceCursorBIsOn = CsrWave(B,"",1)      // trace name in top graph
```

See Also

Programming With Cursors on page II-249.

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

CsrWaveRef

`CsrWaveRef(`*`cursorName`*` [, `*`graphNameStr`*`])`

The CsrWaveRef function returns a wave reference to the wave the specified cursor (A through J) is on in the top (or named) graph.

Details

The wave reference can be used anywhere Igor is expecting the name of a wave (not a string containing the name of a wave).

CsrWaveRef should be used in place of the CsrWave() string function to work properly with data folders.

Examples

```
Print CsrWaveRef(A)[50]              // not CsrWaveRef("A")
Print CsrWaveRef(B,"Graph0")[50]     // in specified graph
```

See Also

Programming With Cursors on page II-249.

Wave Reference Functions on page IV-186.

CsrXWave

`CsrXWave(`*`cursorName`*` [, `*`graphNameStr`*`])`

The CsrXWave function returns a string containing the name of the wave supplying the X coordinates for an XY plot of the Y wave the specified cursor (A through J) is attached to in the top (or named) graph.

Details

CsrXWave returns an empty string (`""`) if the wave the cursor is on is not plotted versus another wave providing the X coordinates (that is, if the wave was not plotted with a command such as `Display theWave vs anotherWave`).

The name of a wave by itself is not sufficient to identify the wave because it does not specify what data folder contains the wave. Thus, if you are calling CsrXWave for the purpose of passing the wave name to other Igor procedures, you should use the **CsrXWaveRef** function instead. Use CsrXWave if you want the name of the wave to use in an annotation or a notebook.

Examples

```
Display ywave vs xwave
```

ywave supplies the Y coordinates and xwave supplies the X coordinates for this XY plot.

```
Cursor A ywave,0
Print CsrXWave(A)      // prints xwave
```

See Also

Programming With Cursors on page II-249.

CsrXWaveRef

`CsrXWaveRef(`*`cursorName`*` [, `*`graphNameStr`*`])`

The CsrXWaveRef function returns a wave reference to the wave supplying the X coordinates for an XY plot of the Y wave the specified cursor (A through J) is attached to in the top (or named) graph.

Details

The wave reference can be used anywhere Igor is expecting the name of a wave (not a string containing the name of a wave).

CsrXWaveRef returns a null reference (see **WaveExists**) if the wave the cursor is on is not plotted versus another wave providing the X coordinates (that is, if the wave was not plotted with a command such as `Display theWave vs anotherWave`). CsrXWaveRef should be used in place of the CsrXWave string function to work properly with data folders.

Examples

```
Display ywave vs xwave
```

ywave supplies the Y coordinates and xwave supplies the X coordinates for this XY plot.

```
Cursor A ywave,0
Print CsrXWaveRef(A)[50]      // prints value of xwave at point #50
```

See Also

Programming With Cursors on page II-249.

Wave Reference Functions on page IV-186.

CTabList

`CTabList()`

The CTabList string function returns a semicolon-separated list of the names of built-in color tables. This can be useful when creating pop-up menus in control panels.

Color tables available through version 4:

Grays	Rainbow	YellowHot	BlueHot	BlueRedGreen
RedWhiteBlue	PlanetEarth	Terrain		

Additional color tables added for version 5:

Grays256	Rainbow256	YellowHot256	BlueHot256	BlueRedGreen256
RedWhiteBlue256	PlanetEarth256	Terrain256	Grays16	Rainbow16
Red	Green	Blue	Cyan	Magenta
Yellow	Copper	Gold	CyanMagenta	RedWhiteGreen
BlueBlackRed	Geo	Geo32	LandAndSea	LandAndSea8
Relief	Relief19	PastelsMap	PastelsMap20	Bathymetry9
BlackBody	Spectrum	SpectrumBlack	Cycles	Fiddle
Pastels				

Additional color tables added for version 6:

RainbowCycle	Rainbow4Cycles	GreenMagenta16	dBZ14	dBZ21
Web216	BlueGreenOrange	BrownViolet	ColdWarm	Mocha
VioletOrangeYellow	SeaLandAndFire			

Additional color tables added for version 6.2:

Mud	Classification

See Also

See **Image Color Tables** on page II-305 and **ColorTab2Wave**.

CtrlBackground

`CtrlBackground` `[`***`key`*** `[=` ***`value`***`]]…`

The CtrlBackground operation controls the unnamed background task.

CtrlBackground works only with the unnamed background task. New code should used named background tasks instead. See **Background Tasks** on page IV-298 for details.

Parameters

dialogsOK=1 *or* 0	If 1, your task will be allowed to run while an Igor dialog is present. This can potentially cause crashes unless your task is well-behaved.
noBurst=1 *or* 0	Normally (or noBurst=0), your task will be called at maximum rate if a delay causes normal run times to be missed. Using noBurst=1, will suppress this burst catch up mode.
period=*deltaTicks*	Sets the minimum number of ticks that must pass between invocations of the background task.

start[=*startTicks*]	Starts the background task (designated by SetBackground) when the tick count reaches *startTicks*. If you omit *startTicks* the task starts immediately.
stop	Stops the background task.

See Also

The **BackgroundInfo**, **SetBackground**, **CtrlNamedBackground**, **KillBackground**, and **SetProcessSleep** operations, and **Background Tasks** on page IV-298.

CtrlNamedBackground

CtrlNamedBackground taskName, keyword = value [***, keyword = value*** …]

The CtrlNamedBackground operation creates and controls named background tasks.

We recommend that you see **Background Tasks** on page IV-298 for an orientation before working with background tasks.

Important: Unlike the unnamed background task, by default named tasks run when a dialog window is active. This can cause a crash if the background task does things the dialog does not expect. See **Background Tasks and Dialogs** on page IV-300 for details.

Parameters

taskName	*taskName* is the name of the background task or _all_ to control all named background tasks. You can use any valid standard Igor object name as the background task name.
burst [= *b*]	Enable burst catch up mode (off by default, *b*=0). When on (*b*=1), the task is called at the maximum rate if a delay misses normal run times.
dialogsOK [= *d*]	Use dialogsOK=0 to prevent the background task from running when a dialog window is active. By default, dialogsOK=1 is in effect. See **Background Tasks and Dialogs** on page IV-300 for details.
kill [= *k*]	Stops and releases task memory for reuse (*k*=1; default) or continues (*k*=0).
period=*deltaTicks*	Sets the minimum number of ticks (*deltaTicks*) that must pass between background task invocations. *deltaTicks* is truncated to an integer and clipped to a value greater than zero. See **Background Task Period** on page IV-299 for details.
proc=*funcName*	Specifies name of a background user function (see **Details**).
start [=*startTicks*]	Starts when the tick count reaches *startTicks*. A task starts immediately without *startTicks*.
status	Returns background task information in the S_info string variable.
stop [= *s*]	Stops the background task (*s*=1; default) or continues (*s*=0).

Details

The user function you specify via the proc keyword must have the following format:

```
Function myFunc(s)
   STRUCT WMBackgroundStruct &s
   …
```

The members of the `WMBackgroundStruct` are:

Base `WMBackgroundStruct` Structure Members

Member	Description
`char name[MAX_OBJ_NAME+1]`	Background task name.
`uint32 curRunTicks`	Tick count when task was called.
`int32 started`	TRUE when CtrlNamedBackground start is issued. You may clear or set to desired value.
`uint32 nextRunTicks`	Precomputed value for next run but user functions may change this.

You may also specify a user function that takes a user-defined STRUCT as long as the first elements of the structure match the `WMBackgroundStruct` or, preferably, if the first element is an instance of `WMBackgroundStruct`. Use the started field to determine when to initialize the additional fields. Your structure may not include any String, WAVE, NVAR, DFREF or other fields that reference memory that is not part of the structure itself.

If you specify a user-defined structure that matches the first fields rather than containing an instance of `WMBackgroundStruct`, then your function will fail if, in the future, the size of the built-in structure changes. The value of `MAX_OBJ_NAME` is 31 but this may also change.

Your function should return zero unless it wants to stop in which case it should return 1.

You can call CtrlNamedBackground within your background function. You can even switch to a different function if desired.

Use the status keyword to obtain background task information via the S_info variable, which has the format:

`NAME:`*`name`*`;PROC:`*`fname`*`;RUN:`*`r`*`;PERIOD:`*`p`*`;NEXT:`*`n`*`;QUIT:`*`q`*`;FUNCERR:`*`e`*`;`

When parsing S_info, do not rely on the number of key-value pairs or their order. RUN, QUIT, and FUNCERR values are 1 or 0, NEXT is the tick count for the next firing of the task. QUIT is set to 1 when your function returns a nonzero value and FUNCERR is set to 1 if your function could not be used for some reason.

See Also

See **Background Tasks** on page IV-298 for examples.

Demos

Choose File→Example Experiments→Programming→ Background Task Demo.

CtrlFIFO

`CtrlFIFO` ***`FIFOName`*** **`[, `*`key`*` = `*`value`*`]…`**

The CtrlFIFO operation controls various aspects of the named FIFO.

Parameters

close	Closes the FIFO's output or review file (if any).
deltaT=*dt*	Documents the data acquisition rate.
doffset=*dataOffset*	Used only with rdfile. Offset to data. If not provided offset is zero.
dsize=*dataSize*	Used only with rdfile. Size of data in bytes. If not provided, then data size is assumed to be the remainder of file. If this assumption is not valid then unexpected results may be observed.
flush	New data in FIFO is flushed to disk immediately.
file=*oRefNum*	File reference number for the FIFO's output file. You obtain this reference number from the **Open** operation used to create the file.
note=*noteStr*	Stores the note string in the file header. It is limited to 255 bytes.
rdfile=*rRefNum*	Like rfile but for review of raw data (use Open/R command). Channel data must match raw data in file. Offset from start of file to start of data can be provided using doffset given in same command. If data does not extend all the way to the end of the file, then the number of bytes of data can be provided using dsize in the same command.
rfile=*rRefNum*	File reference number for the FIFO's review file. Use a review file when you are using a FIFO to review existing data. Obtain the reference number from the Open/R operation used to open the file. File may be either unified header/data or a split format where the header contains the name of a file containing the raw data.
size=*s*	Sets number of chunks in the FIFO. The default is 10000. A chunk of data consists of a single data point from each of the FIFO's channels.

start	Starts the FIFO running by setting the time/date in the FIFO header, writing the header to the output file and marking the FIFO active.
stop	Stops the FIFO by flushing data to disk and marking the FIFO as inactive.
swap	Used only with rdfile. Indicates that the raw data file requires byte-swapping when it is read. This would be the case if you are running on a Macintosh, reading a binary file from a PC, or vice versa.

Details

Once start has been issued, the FIFO can accept no further commands except stop.

The FIFO must be in the valid state for you to access its data (using a chart control or using the **FIFO2Wave** operation). When you create a FIFO, using **NewFIFO**, it is initially invalid. It becomes valid when you issue the start command via the CtrlFIFO operation. It remains valid until you change a FIFO parameter using CtrlFIFO.

FIFOs are used for data acquisition.

See Also

The **NewFIFO** and **FIFO2Wave** operations, and **FIFOs and Charts** on page IV-291.

Cursor

```
Cursor [flags] cursorName traceName x_value
Cursor /F[flags] cursorName traceName x_value, y_value
Cursor /K[/W=graphName] cursorName
Cursor /I[/F][flags] cursorName imageName x_value, y_value
Cursor /M[flags] cursorName
```

The Cursor operation moves the cursor specified by *cursorName* onto the named trace at the point whose X value is *x_value*. or the coordinates of an image pixel or free cursor position at *x_value* and *y_value*.

Parameters

cursorName is one of ten cursors A through J.

Flags

/A=*a*	Activates (*a*=1) or deactivates (*a*=0) the cursor. Active cursors move with arrow keys or the cursor panel.
/C=(*r*,*g*,*b*[,*a*])	Sets the cursor color (default is black). *r*, *g*, and *b* specify the amount of red, green, and blue in the color of the waves as an integer from 0 to 65535. In Igor Pro 7.00 or later, optionally provide a to set transparency, with 0 being fully transparent (invisible) and 65535 being fully opaque (default).
/F	Cursor roams free. The trace or image provides the axis pair that defines x and y coordinates for the setting and readout. Use /P to set in relative coordinates, where 0,0 is the top left corner of the rectangle defined by the axes and 1,1 is the right bottom corner.
/H=*h*	Specifies crosshairs on cursors. *h* =0: Full crosshairs off. *h* =1: Full crosshairs on. *h* =2: Vertical hairline. *h* =3: Horizontal hairline.
/I	Places cursor on specified image.
/K	Removes the named cursor from the top graph.
/L=*lStyle*	Line style for crosshairs (full or small). *lStyle*=0: Solid lines. *lStyle*=1: Alternating color dash.

/M	Modifies properties without having to specify trace or image coordinates. Does not work with the /F or /I flags.
/N=*noKill*	Determines if the cursor is removed ("killed") if the user drags it outside of the plot area: *noKill*=0: Remove the cursor (default). *noKill*=1: Do not remove the cursor.
/NUML=*n*	Used in conjunction with /H when *h* is non-zero. Sets the number of crosshair lines to draw. *n* must be between 1 and 3. When *n* is greater than 1, the line separation is set by the /T=*t* flag. If *n* = 2 or 3 and *t* is less than 3, the line appears as if *n* is 1. If *n* = 3 and *t* is less than 5, the appearance reverts to *n* = 2. Lines are symmetrically disposed around the cursor position. When *n* = 3, *t* sets the separation of the outer pair of lines. /NUML was added in Igor Pro 7.00.
/P	Interpret *xNum* as a point number rather than an X value.
/S=*s*	Sets cursor style. *s*=0: Original square or circle. *s*=1: Small crosshair with letter. *s*=2: Small crosshair without letter.
/T=*t*	Sets the thickness of crosshair lines for /H when *h* is non-zero. If /NUML sets the number of lines greater than 1 then /T sets the separation of the outer pair of lines. *t* is the line thickness or separation distance in units of pixels. The default is /T=1. The form /T={*mode*, *t1*, *t2*} provides finer control. /T was added in Igor Pro 7.00.
/T={*mode*,*t1*,*t2*}	Sets the thickness of crosshair lines for /H when h is non-zero. If /NUML sets the number of lines greater than 1 then /T sets the separation of the outer pair of lines. If *mode*=1 then *t1* and *t2* are in units of screen pixels. *t1* is the vertical line thickness or separation distance and *t2* is the horizontal line thickness or separation distance. The default crosshair appearance is equivalent to /T={1,1,1}. If *mode*=0 then *t1* and *t2* are in units of axis coordinates and consequently track changes in axis range and graph size. Normally *t1* is the vertical line thickness or separation distance and *t2* is the horizontal line thickness or separation distance but they are swapped if the trace or graph is in swap XY mode. /T was added in Igor Pro 7.00.
/W=*graphName*	Specifies a particular named graph window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *graphName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

Usually *traceName* is the same as the name of the wave displayed by that trace, but it could be a name in instance notation. See **ModifyGraph (traces)** and **Instance Notation** on page IV-19 for discussions of trace names and instance notation.

A string containing *traceName* can be used with the $ operator to specify the trace name.

x_value is an X value in terms of the X scaling of the wave displayed by *traceName*. If *traceName* is graphed as an XY pair, then *x_value* is *not* the same as the X axis coordinate. Since the X scaling is ignored when displaying an XY pair in a graph, we recommend you use the /P flag and use a point number for *x_value*.

cursorName is a name, *not* a string.

To get a cursor readout, choose ShowInfo from the Graph menu.

Moving a cursor in a macro or function does not immediately erase the old cursor. DoUpdate has to be explicitly called.

Examples

```
Display myWave                       // X coordinates from X scaling of myWave
Cursor A, myWave, leftx(myWave)      //cursor A on first point of myWave

AppendToGraph yWave vs xWave         //X coordinates from xWave, not X scaling
Cursor/P B,yWave,numpnts(yWave)-1    //cursor B on last point of yWave
DoUpdate                             // erase any old A or B cursors
```

See Also

Programming With Cursors on page II-249 and the **DoUpdate** operation.

CursorStyle

`CursorStyle`

CursorStyle is a procedure subtype keyword that puts the name of the procedure in the “Style function” submenu of the Cursor Info pop-up menu. It is automatically used when Igor creates a cursor style function. To create a cursor style function, choose “Save style function” in the “Style function” submenu of the Cursor Info pop-up menu.

See also **Programming With Cursors** on page II-249.

CurveFit

`CurveFit` `[`***`flags`***`]` ***`fitType,`*** `[`**`kwCWave=`*****`coefWaveName`***`,]` ***`waveName`*** `[`***`flag parameters`***`]`

The CurveFit operation fits one of several built-in functions to your data (for user-defined fits, see the **FuncFit** operation). When with CurveFit and built-in fit functions, automatic initial guesses will provide a good starting point in most cases.

The results of the fit are returned in a wave, by default W_coef. In addition, the results are put into the system variables K0, K1 … K*n* but the use of the system variables is limited and considered obsolete

You can specify your own wave for the coefficient wave instead of W_coef using the kwCWave keyword.

Virtually all waves specified to the CurveFit operation can be a sub-range of a larger wave using the same sub-range syntax as the Display operation uses for graphing. See **Wave Subrange Details** on page V-112.

See Chapter III-8, **Curve Fitting** for detailed information including the use of the Curve Fit dialog.

CurveFit operation parameters are grouped in the following categories: flags, fit type, parameters (kwCWave=*coefWaveName* and *waveName*), and flag parameters. The sections below correspond to these categories. Note that flags must precede the fit type and flag parameters must follow *waveName*.

Flags

/B=*pointsPerCycle*	Used when *type* is sin; *pointsPerCycle* is the estimated number of data points per sine wave cycle. This helps provide initial guesses for the fit. You may need to try a few different values on either side of your estimated points/cycle.
/C	Makes constraint matrix and vector. This only applies if you use the /C=*constraintSpec* parameter to specify constraints (see below). Creates the M_FitConstraint matrix and the W_FitConstraint vector. For more information, see **Fitting with Constraints** on page III-198.
/G	Use values in variables K0, K1 … K*n* as starting guesses for a fit. If you specify a coefficient wave with the kwCWave keyword, the starting guesses will be read from the coefficient wave.
/H="*hhh…*"	Specifies coefficients to hold constant. *h* is 1 for coefficients to hold, 0 for coefficients vary. For example, `/H="100"` holds K0 constant, varies K1 and K2.

/K={*constants*}	Sets values of constants (not fit coefficients) in certain fitting functions. For instance, the exp_XOffset function contains an X offset constant. Built-in functions will set the constant automatically, but the automatic value can be overridden using this flag. *constants* is a list of constant values, e.g., /K={1,2,3}. The length of the list must match the number of constants used by the chosen fit function. This flag is not currently supported by the Curve Fit dialog. Use the To Cmd button and add the flag on the command line.
/L=*destLen*	Sets the length of the wave created by the AutoTrace feature, that is, /D without destination wave (see the /D parameter above). The length of the wave fit_*waveName* will be set to *destLen*. This keyword also sets the lengths of waves created for confidence and prediction bands.
/M	Generates the covariance matrix, the waves CM_K*n*, where *n* is from 0 (for K0) to the number of coefficients minus one.
/M=*doMat*	Generates the covariance matrix. If *doMat* =2, the covariance matrix is put into a 2D matrix wave called M_Covar. If *doMat* =1 or is missing, the covariance matrix is generated as the 1D waves CM_K*n*, where *n* is from 0 (for K0) to the number of coefficients minus one. If *doMat* =0, the covariance matrix is not generated. *doMat* =1 is included for compatibility with previous versions; it is better to use *doMat* =2.
/N[=*dontUpdate*]	If *dontUpdate* = 1, suppresses updates during the fit. This can make the curve fit go much faster; all graphs, tables, etc. will be updated when the fit finishes. /N is the same as /N=1. The default is /N=1. Prior to Igor Pro 7.00 it was /N=0. To update every iteration, use /N=0.
/NTHR = *nthreads*	This flag is accepted but is obsolete and does nothing. See **Curve Fitting with Multiple Processors** on page III-218 for further information.
/O	Only generates initial guesses; doesn't actually do the fit. Unless /ODR=2, this flag is ignored when used with linear fit types (line, poly, poly_XOffset, and poly2D). The **FuncFit** operation also ignores this flag.
/ODR=*fitMethod*	Selects a fitting method. Values for *fitMethod* are: 0: Default Levenberg-Marquardt least-squares method using old code. 1: Trust-region Levenberg-Marquardt ordinary least-squares method implemented using ODRPACK95 code. See **Curve Fitting References** on page III-236. 2: Trust-region Levenberg-Marquardt least orthogonal distance method implemented using ODRPACK95 code. This method is appropriate for fitting when there are measurement errors in the independent variables, sometimes called "errors in variables fitting", "random regressor models," or "measurement error models". 3: Implicit fit. No dependent variable is specified; instead fitting attempts to adjust the fit coefficients such that the fit function returns zero for all dependent variables. Implicit fitting will be of almost no use with the built-in fitting functions. Instead, use **FuncFit** and a user-defined fit function designed for an implicit fit. Note that fitting with non-zero *fitMethod* is not threadsafe. Since the basic curve fitting operations are threadsafe, using /ODR=<nonzero> in a threadsafe user function will compile, but will result in a run-time error.
/Q[=*quiet*]	If quiet = 1, prevents results from being printed in history. /Q is the same as /Q=1.

/TBOX = *textboxSpec* — Adds an annotation to the graph containing the fit data (see the **TextBox** operation, or Chapter III-2, **Annotations**). The textbox contains a customizable set of information about the fit. The argument *textboxSpec* is a bitfield to select various elements to be included in the textbox:

1	Title “Curve Fit Results”
2	Date
4	Time
8	Fit Type (Least Squares, ODR, etc.)
16	Fit function name
32	Model Wave, the autodestination wave (includes a symbol for the trace if appropriate)
64	Y Wave, with trace symbol
128	X Wave
256	Coefficient value report
512	Include errors in the coefficient value report

Request inclusion of various parts by adding up the values for each part you want.

Setting *textboxSpec* to zero will remove the textbox. Default is *textboxSpec* = 0.

/X — Sets the X scaling of the auto-trace destination wave to match the appropriate X axis on the graph when the Y data wave is on the top graph. This is useful when you want to extrapolate the curve outside the range of X data being fit.

/W=*wait* — Specifies behavior for the curve fit results window.

wait=1:	Wait till user clicks OK button before dismissing curve fit results window. This is the default behavior from the command line or dialog.
wait=0:	Display the curve fit window but do not wait for the user to click the OK button.
wait=2:	Do not display the curve fit results window at all. This is the default for fits run from a procedure.

Fit Types

fitType is one of the built-in curve fit function types:

gauss — Gaussian peak: $y = K_0 + K_1 \exp\left[-\left(\frac{x-K_2}{K_3}\right)^2\right]$.

lor — Lorentzian peak: $y = K_0 + \frac{K_1}{(x-K_2)^2 + K_3}$.

exp — Exponential: $y = K_0 + K_1 \exp(-K_2 x)$.

dblexp — Double exponential: $y = K_0 + K_1 \exp(-K_2 x) + K_3 \exp(-K_4 x)$.

sin — Sinusoid: $y = K_0 + K_1 \sin(K_2 x + K_3)$.

line — Line: $y = K_0 + K_1 x$.

poly *n* — Polynomial: $y = k_0 + K_1 x + K_2 x^2 + \dots$.

n is from 3 to 20. *n* is the number of terms or the degree plus one.

poly_XOffset n — Polynomial: y = K0+K1*(x-x_0)+K2*(x-x_0)^2+...

	n is from 3 to 20. *n* is the number of terms or the degree plus one. x_0 is a constant; by default it is set to the minimum X value involved in the fit. Inclusion of x_0 prevents problems with floating-point roundoff errors when you have large values of X in your data set.
hillequation	Hill's Equation: $y = K_0 + \frac{(K_1 - K_0)}{1 + (K_3/x)^{K_2}}$. This is a sigmoidal function. Note that X values must be greater than 0.
sigmoid	$y = K_0 + \frac{K_1}{1 + \exp(K_2 - x/K_3)}$.
power	Power law: $y = K_0 + K_1 x^{K_2}$. Note that X values must be greater than 0.
lognormal	Log normal: $y = K_0 + K_1 \exp\left[-\left(\frac{\ln(x/K_2)}{K_3}\right)^2\right]$. X values must be greater than 0.
gauss2D	2D Gaussian: $z = K_0 + K_1 \exp\left[\frac{-1}{2(1 - K_6^2)}\left(\left(\frac{x - K_2}{K_3}\right)^2 + \left(\frac{y - K_4}{K_5}\right)^2 - \frac{2K_6(x - K_2)(y - K_4)}{K_3 K_5}\right)\right]$. The cross-correlation coefficient (K_6) must be between -1 and 1. This coefficient is automatically constrained to lie in that range. If you are confident that the correlation is zero, it may greatly speed the fit to hold it at zero.
poly2D *n*	Two-dimensional polynomial: $z = K_0 + K_1 x + K_2 y + K_3 x^2 + K_4 xy + K_5 y^2 + \ldots$. where *n* is the degree of the polynomial. All terms up to degree *n* are included, including cross terms. For instance, degree 3 terms are x^3, x^2y, xy^2, and y^3.
exp_XOffset	Exponential: $y = K_0 + K_1 \exp(-(x - x_0)/K_2)$. x_0 is a constant; by default it is set to the minimum *x* value involved in the fit. Inclusion of x_0 prevents problems with floating-point roundoff errors that can afflict the exp function.
dblexp_XOffset	Double exponential: $y = K_0 + K_1 \exp(-(x - x_0)/K_2) + K_3 \exp(-(x - x_0)/K_{4(2)})$. x_0 is a constant; by default it is set to the minimum *x* value involved in the fit. Inclusion of x_0 prevents problems with floating-point roundoff errors that can afflict the exp function.

Parameters

kwCWave=*coefWaveName* specifies an optional coefficient wave. If present, the specified coefficient wave is set to the final coefficients determined by the curve fit. If absent, a wave named W_coef is created and is set to the final coefficients determined by the curve fit.

If you use kwCWave=*coefWaveName* and you include the /G flag, initial guesses are taken from the specified coefficient wave.

waveName is the wave containing the Y data to be fit to the selected function *type*. You can fit to a subrange of the wave by supplying (*startX*,*endX*) after the wave name. Though not shown in the syntax description, you can also specify the subrange in points by supplying [*startP*,*endP*] after the wave name. See **Wave Subrange Details** on page V-112 for more information on subranges of waves in curve fitting.

If you are using one of the two-dimensional fit functions (gauss2D or poly2D) either *waveName* must name a matrix wave or you must supply a list of X waves via the /X flag.

Flag Parameters

These flag parameters must follow *waveName*.

/A=*appendResid*

appendResid =1 (default) appends the automatically-generated residual to the graph and *appendResid* =0 prevents appending (see /R[=*residwaveName*]). With *appendResid* =0, the wave is generated and filled with residual values, but not appended to the graph.

/AD[=*doAutoDest*]

If *doAutoDest* is 1, it is the same as /D alone. /AD is the same as /AD=1.

/C=*constraintSpec*

Applies linear constraints during curve fitting. Constraints can be in the form of a text wave containing constraint expressions (/C=*textWaveName*) or a suitable matrix and vector (/C={*constraintMatrix*, *constraintVector*}). See **Fitting with Constraints** on page III-198. **Note**: Constraints are not available for the built-in line, poly and poly2D fit functions. To apply constraints to these fit functions you must create a user-defined fit function.

/D [=*destwaveName*]

destwaveName is evaluated based on the equation resulting from the fit. *destwaveName* must have the same length as *waveName*.

If only /D is specified, an automatically named wave is created. The name is based on the *waveName* with "fit_" as a prefix. This automatically named wave will be appended (if necessary) to the top graph if *waveName* is graphed there. The X scaling of the fit_ wave is set from the range of x data used during the fit.

By default the length of the automatically-created wave is 200 points (or 2 points for a straight line fit). This can be changed with the /L flag.

If *waveName* is a 1D wave displayed on a logarithmic X axis, Igor also creates an X wave with values exponentially spaced. The name is based on *waveName* with "fitX_" as a prefix.

/F={*confLevel*, *confType* [, *confStyleKey* [, *waveName*…]]}

Calculates confidence intervals for a confidence level of *confLevel*. The value of *confLevel* must be between 0 and 1 corresponding to confidence levels of 0 to 100 per cent.

confType selects what to calculate:

1:	Confidence bands for the model.
2:	Prediction bands for the model.
4:	Confidence intervals for the fit coefficients.

These values can be added together to select multiple options. That is, to select both a confidence band and fit coefficient confidence intervals, set *confType* to 5.

Confidence and prediction bands can be shown as waves contouring a given confidence level (use "Contour" for *confStyleKey*) or as error bars (use "ErrorBar" for *confStyleKey*). The default is Contour.

If no waves are specified, waves to contain the results are automatically generated and appended to the top graph (if the top graph contains the fitted data). See **Confidence Band Details** for details on the waves for confidence bands.

Note: Confidence bands and prediction bands are not available for multivariate curve fits.

/I [=*weightType*]

If *weightType* is 1, the weighting wave (see /W parameter) contains standard deviations. If *weightType* is 0, the weighting wave contains reciprocal of the standard deviation. If the /I parameter is not present, the default is /I=0.

/M=*maskWaveName*

Specifies that you want to use the wave named *maskWaveName* to select points to be fit. The mask wave must match the dependent variable wave in number of points and dimensions. Setting a point in the mask wave to zero or NaN (blank in a table) eliminates that point from the fit.

/R [=*residwaveName*]	Calculates elements of *residwaveName* by subtracting model values from the data values. *residwaveName* must have the same length as *waveName*.
	If only /R is specified, an automatically named wave is created with the same number of points as *waveName*. The name is based on *waveName* with "Res_" as a prefix.
	The automatically created residual wave will be appended (if necessary) to the top graph if *waveName* is graphed there. The residual wave is appended to a new free axis named by prepending "Res_" to the name of the vertical axis used for plotting *waveName*. To the extent possible, the new free axis is formatted nicely.
	If the graph containing the data to be fit has very complex formatting, you may not wish to automatically append the residual to the graph. In this case, use /A=0.
/AR=*doAutoResid*	If *doAutoResid* is 1, it is the same as /R alone. /AR is the same as /AR=1.
/W=*wghtwaveName*	*wghtwaveName* contains weighting values applied during the fit, and must have the same length as *waveName*. These weighting values can be either the reciprocal of the standard errors, or the standard errors. See the /I parameter above for details.
/X=*xwaveName*	The X values for the data to fit come from *xwaveName*, which must have the same length and type as *waveName*.
	If you are fitting to one of the two-dimensional fit functions and *waveName* is a matrix wave, *xwaveName* supplies independent variable data for the X dimension. In this case, *xwaveName* must name a 1D wave with the same number of rows as *waveName*.
/X={*xwave1*, *xwave2*}	For fitting to one of the two-dimensional fit functions when *waveName* is a 1D wave. *xwave1* and *xwave2* must have the same length as *waveName*.
/Y=*ywaveName*	For fitting using one of the 2D fit functions if *waveName* is a matrix wave. *ywaveName* must be a 1D wave with length equal to the number of columns in *waveName*.
/NWOK	Allowed in user-defined functions only. When present, certain waves may be set to null wave references. Passing a null wave reference to CurveFit is normally treated as an error. By using /NWOK, you are telling CurveFit that a null wave reference is not an error but rather signifies that the corresponding flag should be ignored. This makes it easier to write function code that calls CurveFit with optional waves.
	The waves affected are the X wave or waves (/X), weight wave (/W), mask wave (/M) and constraint text wave (/C). The destination wave (/D=wave) and residual wave (/R=wave) are also affected, but the situation is more complicated because of the dual use of /D and /R to mean "do autodestination" and "do autoresidual". See /AR and /AD.
	If you don't need the choice, it is better not to include this flag, as it disables useful error messages when a mistake or run-time situation causes a wave to be missing unexpectedly.
	Note: To work properly this flag must be the last one in the command.

Flag Parameters for Nonzero /ODR

/XW=*xWeightWave*

/XW={*xWeight1*, *xWeight2*}

/ODR=2 or 3 only.

Specifies weighting values for the independent variables using *xWeightWave*, which must have the same length as *waveName*. When fitting to one of the multivariate fit functions such as poly2D or Gauss2D, you must supply a weight wave for each independent variable using the second form.

Weighting values can be either the reciprocal of the standard errors, or the standard errors. The choice of standard error or reciprocal standard error must be the same for both /W and /XW. See /I for details.

/XHLD=*holdWave*

/XHLD={*holdWave1*, *holdWave2*}

/ODR=2 or 3 only.

Specifies a wave or waves to hold the values of the independent variables fixed during orthogonal distance regression. The waves must match the input X data; a one in a wave element fixes the value of the corresponding X value.

/CMAG=*scaleWave* Specifies a wave that indicates the expected scale of the fit coefficients at the solution. If different coefficients have very different orders of magnitude of expected values, this can improve the efficiency and accuracy of the fit.

/XD=*xDestWave*

/XD={*xDestWave1*, *xDestWave2*}

/ODR=2 or 3 only.

Specifies a wave or waves to receive the fitted values of the independent variables during a least orthogonal distance regression.

/XR=*xResidWave*

/XR={*xResidWave1*, *xResidWave2*}

/ODR=2 or 3 only.

Specifies a wave or waves to receive the differences between fitted values of the independent variables and the starting values during a least orthogonal distance regression. That is, they will be filled with the X residuals.

Details

CurveFit gets initial guesses from the K*n* system variables when user guesses (/G) are specified, unless a coefficient wave is specified using the kwCWave keyword. Final curve fit parameters are written into a wave name W_coef, unless you specify a coefficient wave with the kwCWave keyword.

Other output waves are M_Covar (see the /M flag), M_FitConstraint and W_FitConstraint (see /C parameter and **Fitting with Constraints** on page III-198) and W_sigma.

For compatibility with earlier versions of Igor, the parameters are also stored in the system variables Kn. This can be a source of confusion. We suggest you think of W_coef as the **output** coefficients and Kn as **input** coefficients that get overwritten.

Other output waves are M_Covar (see the /M flag), M_FitConstraint and W_FitConstraint (see /C parameter and **Fitting with Constraints** on page III-198), W_sigma. If you have selected coefficient confidence limits using the /F parameter, a wave called W_ParamConfidenceInterval is created with the confidence intervals for the fit coefficients.

CurveFit stores other curve fitting statistics in variables whose names begin with "V_". CurveFit also looks for certain V_ variables which you can use to modify its behavior. These are discussed in **Special Variables for Curve Fitting** on page III-202.

When fitting with /ODR=nonzero, fitting with constraints is limited to simple "bound constraints." That is, you can constrain a fit coefficient to be greater than or less than some value. Constraints involving combinations of fit coefficients are supported only with /ODR=0. The constraints are entered in the same way, using an expression like `K0>1`.

Wave Subrange Details

Almost any wave you specify to CurveFit can be a subrange of a wave. The syntax for wave subranges is the same as for the Display command (see **Subrange Display Syntax** on page II-250 for details). However, the Display command allows only one dimension to have a range (multiple elements from the dimension); if a multidimensional wave is appropriate for CurveFit, you may use a range for more than one dimension.

Some waves must have the same number of points as other waves. For instance, a one-dimensional Y wave must have the same number of points as any X waves. Thus, if you use a subrange for an X wave, the number of points in the subrange must match the number of points being used in the Y wave (but see **Subrange Backward Compatibility** on page V-113 for a complication to this rule).

A common use of wave subranges might be to package all your data into a single multicolumn wave, along with the residuals and model values. For a univariate fit, you might need X and Y waves, plus a destination (model) wave and a residual wave. You can achieve all of that using a 4 column wave. For example:

```
Make/D/N=(100, 4) Data
... fill column zero with X data and column one with Y data ...
CurveFit poly 3, Data[][1] /X=Data[][0]/D=Data[][2]/R=Data[][3]
```

Note that because all the waves are full columns from a single multicolumn wave, the number of points is guaranteed to be the same.

The number of points used for X waves (*xwaveName* or {*xwave1*, *xwave2*, …}), weighting wave (*wghtwaveName*), mask wave (`maskWaveName`), destination wave (*destwaveName*) and residual wave (*residwaveName*) must be the same as the number of points used for the Y wave (*waveName*). If you specify your own confidence band waves (/F flag) they must match the Y wave; you cannot use subranges with confidence band waves. If you set /ODR = nonzero, the X weight, hold, destination and residuals waves must match the Y wave.

The total number of points in each wave does not need to match other waves, just the number of points in the specified subrange.

When fitting to a univariate fit function (that includes almost all the fit types) the Y wave must have effectively one dimension. That means the Y wave must either be a 1D wave, or it must have a subrange that makes the data being used one dimensional. For instance:

```
Make/N=(100,100) Ydata           // 2D wave
CurveFit gauss Ydata[][0]        // OK- a single column is one-dimensional
CurveFit gauss Ydata[2][]        // OK- s single row is one-dimensional
CurveFit gauss Ydata             // not OK- Ydata is two-dimensional
CurveFit gauss Ydata[][0,1]      // not OK- two columns makes 2D subrange
```

When fitting a multivariate function (**poly2D** or **Gauss2D**) you have the choice of making the Y data either one-dimensional or two-dimensional. If it is one-dimensional, then you must be fitting XYZ (or Y,X1,X2) triplets. In that case, you must provide a one-dimensional Y wave and two one-dimensional X waves, or 2 columns from a multicolumn wave. For instance:

These are OK:

```
Make/N=(100,3) myData
CurveFit Gauss2D myData[][0] /X={myData[][1],myData[][2]}
CurveFit Gauss2D myData[][0] /X=myData[][1,2]
```

These are not OK:

```
CurveFit Gauss2D myData /X={myData[][1],myData[][2]}// 2D Y wave with 1D X waves
CurveFit Gauss2D myData[][0] /X=myData            // too many X columns
```

If you use a 2D Y wave, the X1 and X2 data can come from the grid positions and the Y wave's X and Y index scaling, or you can use one-dimensional waves or wave subranges to specify the X1 and X2 positions of the grid:

```
Make/N=(20,30) yData
CurveFit Gauss2D yData        //OK- 2D Y data, X1 and X2 from scaling
Make/N=20 x1Data
Make/N=30 x2Data
// OK: 2D effective Y data, matching 1D X and Y flags
CurveFit Gauss2D yData[0,9][0,19] /X=x1Data[0,9]/Y=x2data[10,29]
// OK: effective 2D Y data
Make/N=(10,20,3) Y data
CurveFit Gauss2D yData[][][0]
```

There are, of course, lots of possible combinations, too numerous to enumerate.

Subrange Backward Compatibility

Historically, a Y wave could have a subrange. The same subrange applied to all other waves. For backward compatibility, if you use a subrange with the Y wave only, and other waves lack a subrange, these other waves must have either: 1) The same total number of points as the total number of points in the Y wave in which case the Y wave subrange will be applied; or 2) The same total number of points as the Y wave's subrange.

In addition, the Y wave can take a subrange in parentheses to indicate that the subrange refers to the Y wave's scaled indices (X scaling). If you use parentheses to specify an X range, you must satisfy the old subrange rules: All waves must have the same number of points. Subrange is allowed for the Y wave only. The Y wave subrange is applied to all other waves.

Confidence Band Details

Automatic generation of confidence and prediction bands occurs if the /F={…} parameter is used with no wave names. One to four waves are generated, or you can specify one to four wave names yourself depending on the *confKind* and *confStyle* settings.

Waves auto-generated by /F={*confLevel*, *confKind*, *confStyle*}:

confKind	*confStyle*	**What You Get**	**Auto Wave Names**
1	"Contour"	upper and lower confidence contours	UC_*dataName*, LC_*dataName*
2	"Contour"	upper and lower prediction contours	UP_*dataName*, LP_*dataName*
3	"Contour"	upper and lower confidence contours and prediction contours	UC_*dataName*, LC_*dataName*, UP_*dataName*, LP_*dataName*
1	"ErrorBar"	confidence interval wave	CI_*dataName*
2	"ErrorBar"	prediction interval wave	PI_*dataName*
3	"ErrorBar"	confidence and prediction interval waves	CI_*dataName*, PI_*dataName*

Note that *confKind* may have 4 added to it if you want coefficient confidence limits calculated as well.

The contour waves are appended to the top graph as traces if the data wave is displayed in the top graph. The wave names have *dataName* replaced with the name of the wave containing the Y data for the fit.

Waves you must supply for /F={*confLevel*, *confKind*, *confStyle*, *wave*, *wave…*}:

confKind	*confStyle*	**You Supply**
1	"Contour"	2 waves to receive upper and lower confidence contours.
2	"Contour"	2 waves to receive upper and lower prediction contours.
3	"Contour"	4 waves to receive upper and lower confidence and upper and lower prediction contours.
1	"ErrorBar"	1 wave to receive values of confidence band width.
2	"ErrorBar"	1 wave to receive values of prediction band width.
3	"ErrorBar"	2 waves to receive values of confidence and prediction band widths.

The waves you supply must have the same number of points as the dependent variable data wave. The band intervals will be calculated at the X values of the input data. These waves are not automatically appended to a graph; it is expected that you will display the contour waves as traces or use the error bar waves to make error bars on the model fit wave.

Residual Details

Residuals are calculated only for elements corresponding to elements of waveName that are included in the fit. Thus, you can calculate residuals automatically for a piecewise fit done in several steps.

The automatic residual wave will be appended to the top graph if the graph displays the Y data. It is appended to a new free axis positioned directly above the axis used to display the Y data, making a stacked graph. Other axes are shortened as necessary to make room for the new axis. You can alter the axis formatting later. See **Creating Stacked Plots** on page II-253 for details.

While Igor will go to some lengths to make a nicely formatted stacked graph, the changes made to the graph formatting may be undesirable in certain cases. Use /A=0 to suppress the automatic append to the graph. The automatic residual wave will be created and filled with residual values, but not appended to the graph.

See Also

Inputs and Outputs for Built-In Fits on page III-184 and **Special Variables for Curve Fitting** on page III-202 as well as **Accessing Variables Used by Igor Operations** on page IV-115.

When fitting to a user-specified function, see **FuncFit**. For multivariate user-specified fitting functions, see **FuncFit** and **FuncFitMD**. See **Confidence Bands and Coefficient Confidence Intervals** on page III-193 for a detailed description of confidence and prediction bands.

References

An explanation of the Levenberg-Marquardt nonlinear least squares optimization can be found in Chapter 14.4 of Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

CustomControl

`CustomControl [/Z] ctrlName [keyword = value [, keyword = value …]]`

The CustomControl operation creates or modifies a custom control in the target window. A CustomControl starts out as a generic button, but you can customize both its appearance and its action.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the CustomControl to be created or changed. See **Button** for standard default parameters.

The following keyword=value parameters are supported:

fColor=(*r*,*g*,*b*)	Sets color of the button only when picture is not used and frame=1. *r*, *g*, and *b* can range from 0 to 65535.
focusRing=*fr*	Enables or disables the drawing of a rectangle indicating keyboard focus: *fr*=0: Focus rectangle will not be drawn. *fr*=1: Focus rectangle will be drawn (default). On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.
frame=*f*	Sets frame style used only when picture is not used: *f*=0: No frame (only the title is drawn). *f*=1: Default, a button is drawn with a centered title. Set fColor to something other than black to colorize the button. *f*=2: Simple box. *f*=3: 3D sunken frame. On Macintosh, when "native GUI appearance" is enabled for the control, the frame is filled with the proper operating system color. *f*=4: 3D raised frame. *f*=5: Text well.
labelBack=(*r*,*g*,*b*) or 0	Sets background color for the control only when a picture is not used and frame is not 1 and is not 3 on Macintosh. *r*, *g* and *b* specify the amount of red, green and blue in the color as an integer from 0 to 65535. If not set (or labelBack=0), then background is transparent (not erased).

mode=*m*	Notifies the control that something has happened. Can be used for any purpose. See **Details** discussion of the `kCCE_mode` event.
noproc	Specifies that no procedure will execute when clicking the custom control.
picture= *pict*	Uses the named Proc Pictures to draw the control. The picture is taken to be three side-by-side frames, which show the control appearance in the normal state, when the mouse is down, and in the disabled state. The control action function can overwrite the picture number using the picture={*pict*,*n*} syntax. The picture size overrides the size keyword.
picture={*pict*,*n*}	Uses the specified Proc Picture to draw the control. The picture is *n* side-by-side frames instead of the default three frames.
pos={*left*,*top*}	Sets the postion of the control in pixels.
pos+={*dx*,*dy*}	Offsets the position of the control in pixels.
proc=*procName*	Specifies the name of the action function for the control. The function must not kill the control or the window.
size={*width*,*height*}	Sets size of the control in pixels but only when not using a Proc Picture.
title=*titleStr*	Specifies text that appears in the control. Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.
userdata(*UDName*)=*UDStr*	Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.
userdata(*UDName*)+=*UDStr*	Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.
value=*varName*	Sets the numeric variable, string variable, or wave that is associated with the control. With a wave, specify a point using the standard bracket notation with either a point number (`value=awave[4]`) or a row label (`value=awave[%alabel]`).
valueColor=(*r*,*g*,*b*)	Sets initial color of the title for the button drawn only when picture is not used and frame=1. *r*, *g*, and *b* range from 0 to 65535. *valueColor* defaults to black (0,0,0). To further change the color of the title text, use escape sequences as described for title=*titleStr*.

Flags

/Z	No error reporting.

Details

When you create a custom control, your action procedure gets information about the state of the control using the `WMCustomControlAction` structure. See **WMCustomControlAction** for details on the `WMCustomControlAction` structure.

Although the return value is not currently used, action procedures should always return zero.

When you call a function with the `kCCE_draw` event, the basic button picture (custom or default) will already have been drawn. You can use standard draw commands such as **DrawLine** to draw on top of the basic picture. Unlike the normal situation when draw commands merely add to a draw list, which only later is drawn, `kCCE_draw` event draw commands are executed directly. The coordinate system, which you can not change, is pixels with (0,0) being the top left corner of the control. Most drawing commands are legal but because of the immediate nature of drawing, the /A (append) flag of **DrawPoly** is not allowed.

The `kCCE_mode` event can be used for any purpose, but it mainly serves as a notification to the control that something has happened. For example, to send information to a control, you can set a named (or the unnamed) userdata and then set the `mode` to indicate that the control should examine the userdata. For this signaling purpose, you should use a `mode` value of 0 because this value will not become part of the recreation macro.

The `kCCE_frame` event is sent just before drawing one of the *pict* frames, as set by the picture parameter. On input, the `curFrame` field is set to 0 (normal or mouse down outside button), to 1 (mouse down in button), or to 2 (disable). You may modify `curFrame` as desired but your value will be clipped to a valid value.

When you specify a *pict* with the picture parameter, you will get a `kCCE_drawOSBM` event when that *pict* is drawn into an offscreen bitmap. Once it is created, all updates use the offscreen bitmap until you specify a new picture parameter. Thus the custom drawing done at this event is static, unlike drawing done during the `kCCE_draw` event, which can be different each time the control is drawn. Because the *pict* can be contain multiple side-by-side frames, the width of the offscreen bitmap is the width derived from the `ctrlRect` field multiplied by the number of frames.

Because the action function is called in the middle of various control events, it must not kill the control or the window. Doing so will almost certainly cause a crash.

Examples

See **Creating Custom Controls** on page III-377 for some examples of custom controls.

For a demonstration of custom controls, see the Custom Control Demo.pxp example experiment, which is located in your Igor Pro 7 Folder in the Examples:Feature Demos 2: folder.

See Also

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Proc Pictures on page IV-53.

The **TextBox**, **DrawPoly** and **DefaultGUIControls** operations.

CWT

`CWT` `[`***`flags`***`]` ***`srcWave`***

The CWT operation computes the continuous wavelet transform (CWT) of a 1D real-valued input wave (*srcWave*). The input can be of any numeric type. The computed CWT is stored in the wave M_CWT in the current data folder. M_CWT is a double precision 2D wave which, depending on your choice of mother wavelet and output format, may also be complex. The dimensionality of M_CWT is determined by the specifications of offsets and scales. The operation sets the variable V_flag to zero if successful or to a nonzero number if it fails for any reason.

Flags

/ENDM=*method*	Selects the method used to handle the two ends of the data array with direct integration (/M=1).
	method=0: Padded on both sides by zeros.
	method=1: Reflected at both the start and end.
	method=2: Entered with cyclical repetition.
/FSCL	Use correction factor to the wave scaling of the second dimension of the output wave so that the numbers are more closely related to Fourier wavelength. See **References** for more information on the calculation of these correction factors. This flag does not affect the output from the Haar wavelet.
/M=*method*	Specifies the CWT computation method.
	method=0: Fast method uses FFT (default).
	method=1: Slower method using direct integration.

You should mostly use the more efficient FFT method. The direct method should be reserved to situations where the FFT is not producing optimal results. Theoretically, when the FFT method fails, the direct method should also be fairly inaccurate, e.g., in the case of undersampled signal. The main advantage in the direct method is that you can use it to investigate edge effects.

/OUT=*format* Sets the format of the output wave M_CWT:

format=1:	Complex.
format=2:	Real valued.
format=4:	Real and contains the magnitude.

Depending on the method of calculation and the choice of mother wavelet, the "native" output of the transform may be real or complex. You can force the output to have a desired format using this flag.

/Q Quiet mode; no results printed to the history.

/R1={*startOffset*, *delta1*, *numOffsets*

Specifies offsets for the CWT. Offsets are the first dimension in a CWT. Normally you will calculate the CWT for the full range of offsets implied by *srcWave* so you will not need to use this flag. However, when using the slow method, this flag restricts the output range of offsets and save some computation time. *startOffset* (integer) is the point number of the first offset in *srcWave*. *delta1* is the interval between two consecutive CWT offsets. It is expressed in terms of the number *srcWave* points. *numOffsets* is the number of offsets for which the CWT is computed.

By default *startOffset*=0, *delta1*=1, and *numOffsets* is the number of points in *srcWave*. If you want to specify just the *startOffset* and *delta1*, you can set *numOffsets*=0 to use the same number of points as the source wave.

/R2={*startScale*, *scaleStepSize*, *numScales*}

Specifies the range of scales for the CWT is computation. Scales are the second dimension in the output wave. Note however that there are limitations on the minimum and maximum scales having to do with the sampling of your data. Because there is a rough correspondence between a Fourier spatial frequency and CWT scale it should be understood that there is also a maximum theoretical scale. This is obvious if you compute the CWT using an FFT but it also applies to the slow method. If you specify a range outside the allowed limits, the corresponding CWT values are set to NaN.

Use NaN if you want to use the default value for any parameter.

The default value for *startScale* is determined by sampling of the source wave and the wavelet parameter or order.

At a minimum you must specify either *scaleStepSize* or *numScales*.

/SMP1=*offsetMode* Determines computation of consecutive offsets. Currently supporting only *offsetMode*=1 for linear, user-provided partial offset limits (see /R1 flag): `val1=startOffset+numOffsets*delta1`.

/SMP2=*scaleMode* Determines computation of consecutive scales. *scaleMode* is 1 by default if you specify the /R2 flag.

format=1:	Linear: `theScale=startScale+index*scaleStepSize`
format=2:	User-provided scaling wave.
format=4:	Power of 2 scaling interval: `theScale=startScale*2.^(index*scaleStepSize)`

When using *scaleMode*=4 the operation saves the consecutive scale values in the wave W_CWTScaling. Note also that if you use *scaleMode*=4 without specifying a corresponding /R2 flag, the default *scaleStepSize* of 1 and 64 scale values gives rise to scale values that quickly exceed the allowed limits.

(See /R2 flag for details about the different parameters used in the equations above.)

/SW2=*sWave* Provides specific scale values at which the transform is evaluated. Use instead of /R2 flag. It is your responsibility to make sure that the entries in the wave are appropriate for the sampling density of *srcWave*.

/WBI1={*Wavelet* [, *order*]}

Specifies the built-in wavelet (mother) function. *Wavelet* is the name of a wavelet function: Morlet (default), MorletC (complex), Haar, MexHat, DOG, and Paul.

Morlet:
$$\Psi_0(x)=\frac{1}{\pi^{1/4}}\cos(\omega x)\exp\left(-\frac{x^2}{2}\right).$$

By default, ω=5. Use the /WPR1 flag to specify other values for ω.

MorletC:
$$\Psi_0(x)=\frac{1}{\pi^{1/4}}\exp(i\omega x)\exp\left(-\frac{x^2}{2}\right).$$

By default, ω=5. Use the /WPR1 flag to specify other values for ω.

Haar:
$$\Psi_0(x)=\begin{cases} 1 & 0\le x<0.5 \\ -1 & 0.5\le x<1 \end{cases}.$$

DOG:
$$\Psi_0(x)=\frac{(-1)^{m+1}}{\sqrt{\Gamma\left(m+\frac{1}{2}\right)}}\frac{d^m}{dx^m}\left(\exp\left(-\frac{x^2}{2}\right)\right).$$

MexHat: Special case of DOG with m=2.

Paul:
$$\Psi_0(m,x)=\frac{2^m i^m m!}{\sqrt{\pi(2m)!}}(1-ix)^{-(m+1)}.$$

order applies to DOG and Paul wavelets only and specifies *m*, the particular member of the wavelet family.

The default wavelet is the Morlet.

/WPR1={*param1*} *param1* is a wavelet-specific parameter for the wavelet function selected by /WBI1. For example, use /WPR1={6} to change the Morlet frequency from the default (5).

/Z	No error reporting. If an error occurs, sets V_flag to -1 but does not halt function execution.

Details

The CWT can be computed directly from its defining integral or by taking advantage of the fact that the integral represents a convolution which in turn can be calculated efficiently using the fast Fourier transform (FFT).

When using the FFT method one encounters the typical sampling problems and edge effects. Edge effects are also evident when using the slow method but they only significant in high scales.

From sampling considerations it can be shown that the maximum frequency of a discrete input signal is 1/2dt where dt is the time interval between two samples. It follows that the smallest CWT scale is 2dt and the largest scale is Ndt where N is the total number of samples in the input wave.

The transform in M_CWT is saved with the wave scaling. *startOffset* and *delta1* are used for the X-scaling. Both *startOffset* and *delta1* are either specified by the /R1 flag or copied from *srcWave*. The Y-scaling of M_CWT depends on your choice of /SMP2. If the CWT scaling is linear then the wave scaling is based on *startScale* and *scaleStepSize*. If you are using power of 2 scaling interval then the Y wave scaling of M_CWT has a *start*=0 and *delta*=1 and the wave W_CWTScaling contains the actual scale values for each column of M_CWT. Note that W_CWTScaling has one extra data point to make it suitable for display using an operation like:

```
AppendImage M_CWT vs {*, W_CWTScaling}
```

We have encountered two different definitions for the Morlet wavelet in the literature. The first is a complex function (MorletC) and the second is real (Morlet). Instead of choosing one of these definitions we implemented both so you may choose the appropriate wavelet.

See Also

For discrete wavelet transforms use the **DWT** operation. The **WignerTransform** and **FFT** operations.

For further discussion and examples see **Continuous Wavelet Transform** on page III-251.

References

Torrence, C., and G.P. Compo, A Practical Guide to Wavelet Analysis, *Bulletin of the American Meteorological Society*, *79*, 61-78, 1998.

The Torrence and Compo paper is also online at:
`<http://paos.colorado.edu/research/wavelets/>`.

DataFolderDir

`DataFolderDir(`*`mode`*` [, `*`dfr`*` ])`

The DataFolderDir function returns a string containing a listing of some or all of the objects contained in the current data folder or in the data folder referenced by *dfr*.

Parameters

mode is a bitwise flag for each type of object. Use -1 for all types. Use a sum of the bit values for multiple types.

Desired Type	Bit Number	Bit Value
All		-1
Data folders	0	1
Waves	1	2
Numeric variables	2	4
String variables	3	8

dfr is a data folder reference.

Details

The returned string has the following format:

1. FOLDERS:*name*,*name*,…;<CR>
2. WAVES:*name*,*name*,…;<CR>
3. VARIABLES:*name*,*name*,…;<CR>
4. STRINGS:*name*,*name*,…;<CR>

Where <CR> represents the carriage return character.

Tip

This function is mostly useful during debugging, used in a **Print** command. For finding the contents of a data folder programmatically, it will be easier to use the functions **CountObjects** and **GetIndexedObjName**.

Examples

```
Print DataFolderDir(8+4)       // prints variables and strings
Print DataFolderDir(-1)        // prints all objects
```

See Also

Chapter II-8, **Data Folders**.

Setting Bit Parameters on page IV-12 for information about bit settings.

DataFolderExists

`DataFolderExists(`*`dataFolderNameStr`*`)`

The DataFolderExists function returns the truth that the specified data folder exists.

dataFolderNameStr can bea a full path or partial path relative to the current data folder.

If *dataFolderNameStr* is null DataFolderExists returns 1 because, for historical reasons, a null data folder path is taken to refer to the current data folder.

See Also

Chapter II-8, **Data Folders**.

DataFolderRefsEqual

`DataFolderRefsEqual(`*`dfr1, dfr2`*`)`

The DataFolderRefsEqual function returns the truth the two data folder references are the same.

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-72.

The **DataFolderRefStatus** function.

DataFolderRefStatus

`DataFolderRefStatus(`*`dfr`*`)`

The DataFolderRefStatus function returns the status of a data folder reference.

Details

DataFolderRefStatus returns zero if the data folder reference is invalid or non-zero if it is valid.

DataFolderRefStatus returns a bitwise result with bit 0 indicating if the reference is valid and bit 1 indicating if the reference data folder is free. Therefore the returned values are:

0:	The data folder reference is invalid.
1:	The data folder reference refers to a regular global data folder.
3:	The data folder reference refers to a free data folder.

A data folder reference is invalid if it was never assigned a value or if it is assigned an invalid value. For example:

```
DFREF dfr                                   // dfr is invalid
DFREF dfr = root:                           // dfr is valid
DFREF dfr = root:NonExistentDataFolder      // dfr is invalid
```

A data folder reference can be valid and yet point to a non-existent data folder:

```
NewDataFolder/O root:MyDataFolder
DFREF dfr = root:MyDataFolder               // dfr is valid
KillDataFolder root:MyDataFolder            // dfr is still valid
```

After the KillDataFolder, dfr is still a valid data folder reference but points to a non-existent data folder.

You should use DataFolderRefStatus to test any DFREF variables that might not be valid, such as after assigning a reference when you are not sure that the referenced data folder exists. For historical reasons, an invalid DFREF variable will often act like root.

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-72.

dateToJulian

`dateToJulian(`*`year, month, day`*`)`

The dateToJulian function returns the Julian day number for the specified date. The Julian day starts at noon. Use negative number for BC years and positive numbers for AD years. To exclude any ambiguity, there is no year zero in this calendar. For general orientation, Julian day 2450000 corresponds to October 9, 1995.

See Also

The **JulianToDate** function.

For more information about the Julian calendar see:
`<http://www.tondering.dk/claus/calendar.html>`.

date

`date()`

The date function returns a string containing the current date.

Formatting of dates depends on your operating system and on your preferences entered in the Date & Time control panel (*Macintosh*) or the Regional Settings control panel (*Windows*).

Examples

```
Print date()        // Prints Mon, Mar 15, 1993
```

See Also

The **Secs2Date**, **Secs2Time**, and **time** functions.

date2secs

`date2secs(`*`year, month, day`*`)`

The date2secs function returns the number of seconds from midnight on 1/1/1904 to the specified date.

Note that the month and day parameters are one-based, so these series start at one.

If *year*, *month*, and *day* are all -1 then date2secs returns the offset in seconds from the local time to the UTC (Universal Time Coordinate) time.

Examples

```
Print Secs2Date(date2secs(1993,3,15),1)        // Ides of March, 1993
```

Prints the following, depending on your system's date settings, in the history area:

```
  Monday, March 15, 1993
```

This next example sets the X scaling of a wave to 1 day per point, starting January 1, 1993:

```
Make/N=125 myData = 100 + gnoise(50)
SetScale/P x,date2secs(1993,1,1),24*60*60,"dat",myData
Display myData;ModifyGraph mode=5
```

See Also

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-78.

The **Secs2Date**, **Secs2Time**, and **time** functions.

DateTime

```
DateTime
```

The DateTime function returns number of seconds from 1/1/1904 to current local date and time.

To get the UTC date and time, subtract Date2Secs(-1,-1,-1) from the value returned by DateTime.

Unlike most Igor functions, DateTime is used without parentheses.

Examples

```
Variable localNow = DateTime
```

See Also

The **Secs2Date**, **Secs2Time** and **time** functions.

dawson

dawson(*x*)

The dawson function returns the value of the Dawson integral:

$$F(x) = \exp\left(-x^2\right)\int_0^x \exp\left(t^2\right)dt.$$

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 298 pp., Dover, New York, 1972.

DDEExecute

DDEExecute(*refNum*, *cmdStr* [, *timeout*])

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDEExecute function sends a string of commands, *cmdStr*, to the server for execution. The format of the commands depends on the server application.

refNum is a DDE session reference number returned by DDEInitiate to start a particular session.

It returns an error code from the server or zero if there was no error.

DDEExecute returns -1 if the server gave an error but the error code was zero. Returns -2 if the server did not reply before the timeout period. Returns -3 if the *refNum* is invalid and -4 for other errors.

The optional *timeout* value is in seconds. The default *timeout* is 60 sec.

In some situations, you may want to use a zero *timeout* and then use DDEStatus to monitor when the server is finished. You would do this if the server might take a long time to accomplish the commands and you want Igor to continue working at the same time.

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDEInitiate

DDEInitiate(*serverName*, *topicName*)

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDEInitiate function opens a DDE session and returns a reference number for use by the rest of the DDE routines. Returns zero if failure.

If the desired application is not running, DDEInitiate will return zero. If this occurs, you can use the ExecuteScriptText operation to start the server application.

Refer to your application's documentation for the *serverName* and *topicName*. See the following example for usage with Excel.

Examples

```
Variable ch= DDEInitiate("excel","book1")
```

See Also

The **ExecuteScriptText** operation. For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDEPokeString

`DDEPokeString(`*`refNum`*`, `*`itemString`*`, `*`string`*` [, `*`timeout`*`])`

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDEPokeString function sends *string* to the server.

itemString is a string specifying the server application's DDE item name for the location into which to store the string. For example, `"R1C1"` specifies the first cell in an Excel spreadsheet.

refNum is the DDE session reference number returned by DDEInitiate to start a particular session.

It returns an error code from the server or zero if there was no error.

DDEPokeString returns -1 if the server gave an error but the error code was zero. Returns -2 if the server did not reply before the timeout period. Returns -3 if the *refNum* is invalid and -4 for other errors.

The optional *timeout* value is in seconds. The default *timeout* is 60 sec.

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDEPokeWave

`DDEPokeWave(`*`refNum`*`, `*`itemString`*`, `*`wave`*` [, `*`timeout`*` [, `*`format`*`]])`

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDEPokeWave function sends data from a *wave* to the server.

itemString is a string specifying the server application's DDE item name for the location into which to store the string. For example, `"R1C1:R10C10"` specifies a 10x10 block of cells in an Excel spreadsheet.

refNum is a DDE session reference number returned by DDEInitiate to start a particular session.

It returns an error code from the server or zero if there was no error.

DDEPokeWave returns -1 if the server gave an error but the error code was zero. Returns -2 if the server did not reply before the *timeout* period. Returns -3 if the *refNum* is invalid and -4 for other errors.

The optional *format* parameter can be 0 to send the data as tab-delimited text (default) or can be 1 to specify Microsoft's XLTable (excel) format. To specify *format* without specifying *timeout,* the latter may be completely missing (`,,`) or a `*` symbol may be used:

```
err= DDEPokeWave(ch,"R1C1:R10C10",*,1)
```

The optional *timeout* value is in seconds. The default *timeout* is 60 sec.

Examples

A session using Microsoft Excel:

```
Variable ch,err1,err2
ch= DDEInitiate("excel","book1")
Make/O/N=(5,5) jack= P+10*Q
err1= DDEPokeWave(ch,"R1C1:R5C5",jack)
err2= DDETerminate(ch)
```

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDERequestString

`DDERequestString(`***`refNum, itemString`*** `[,` ***`timeout`***`])`

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDERequestString string function returns a string of requested data or null handle in case of failure.

itemString is a string specifying the server application's DDE item name for the data being requested. For example, `"R1C1"` specifies the first cell in an Excel spreadsheet.

refNum is a DDE session reference number returned by DDEInitiate to start a particular session.

The optional *timeout* value is in seconds. The default *timeout* is 60 sec.

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDERequestWave

`DDERequestWave(`***`refNum, itemString, destWave`*** `[,` ***`timeout`***`])`

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDERequestWave function loads a preexisting wave with data from the server. The provided destination wave, *destWave*, can be either text or numeric. Data from the server must be a tab delimited array. It is analyzed to determine the dimensions of the wave but the numeric type (or string type) of the wave is not changed.

itemString is a string specifying the server application's DDE item name for the requested data. For example, `"R1C1:R10C10"` specifies a 10x10 block of cells in an Excel spreadsheet.

refNum is a DDE session reference number returned by DDEInitiate to start a particular session.

It returns an error code from the server or zero if there was no error.

DDERequestWave returns -1 if the server gave an error but the error code was zero. Returns -2 if the server did not reply before the timeout period. Returns -3 if the *refNum* is invalid and -4 for other errors.

The optional *timeout* value is in seconds. The default *timeout* is 60 sec.

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDEStatus

`DDEStatus(`***`refNum`***`)`

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDEStatus function returns the status of the DDE session defined by *refNum* from a previous DDEInitiate.

Returns zero if *refNum* is not valid or if the session has been closed.

Returns nonzero if session is valid where the individual bits have the following meanings:

Bit 0:	Set if the session is valid and not busy.
Bit 1:	Set if waiting for an ack from the server for a previous DDEPoke or DDEExecute that timed out. (bit 0 and 1 are exclusive).
Bit 2:	Set if the ack from the server for a previous poke or execute command was negative. Only applies to timed out commands.

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

DDETerminate

`DDETerminate(`*`refNum`*`)`

This is a *Windows-only* function; it will return an error on the Macintosh.

The DDETerminate function closes the DDE session defined by *refNum* from a previous DDEInitiate. Returns truth session was valid.

Pass zero to terminate all client sessions.

See Also

For further information refer to the other DDE functions and to the DDE Server and DDE Client sections in the Obsolete Topics help file.

Debugger

`Debugger`

The Debugger operation breaks into the debugger if it is enabled.

See Also

The Debugger on page IV-198 and the **DebuggerOptions** operation.

DebuggerOptions

`DebuggerOptions` `[`**`enable=`*`en`*`, debugOnError=`*`doe`*`, NVAR_SVAR_WAVE_Checking=`*`nvwc`***`]`

The DebuggerOptions operation programmatically changes the user-level debugger settings. These are the same three settings that are available in the Procedure menu (and the debugger source pane contextual menu)

Parameters

All parameters are optional. If none are specified, no action is taken, but the output variables are still set.

enable=*en*	Turns the debugger on (*en*=1) or off (*en*=0). If the debugger is disabled then the other settings are cleared even if other settings are on.
debugOnError=*doe*	Turns Debugging On Error on or off. *doe*=0: Disables Debugging On Error (see **Debugging on Error** on page IV-199). *doe*=1: Enables Debugging On Error and also enables the debugger (implies enable=1).
NVAR_SVAR_WAVE_Checking=*nvwc*	Turns NVAR, SVAR, and WAVE checking on or off. *nvwc*=0: Disables “NVAR SVAR WAVE Checking”. See **Accessing Global Variables and Waves** on page IV-59 for more details. *nvwc*=1: Enables this checking and also enables the debugger (implies enable=1).

Details

DebuggerOptions sets the following variables to indicate the Debugger settings that are in effect *after* the command is executed. A value of zero means the setting is off, nonzero means the setting is on.

```
V_enable             V_debugOnError       V_NVAR_SVAR_WAVE_Checking
```

See Also

The Debugger on page IV-198 and the **Debugger** operation.

default

`default:`

The default flow control keyword is used in switch and strswitch statements. When none of the case labels in the switch or strswitch match the evaluation expression, execution will continue with code following the default label, if it is present.

See Also

Switch Statements on page IV-41.

DefaultFont

`DefaultFont [/U] "`*`fontName`*`"`

The DefaultFont operation sets the default font to be used in graphs for axis labels, tick mark labels and annotations, and in page layouts for annotations.

Parameters

"fontName" should be a font name, optionally in quotes. The quotes are not required if the font name is one word.

Flags

/U	Updates existing graphs and page layouts immediately to use the new default font.

DefaultGUIControls

`DefaultGUIControls [/Mac/W=`*`winName`*`/Win] [`*`appearance`*`]`

The DefaultGUIControls operation changes the appearance of user-defined controls.

Note: The recommended way to change the appearance of user-defined controls is to use the Miscellaneous Settings dialog's Native GUI Appearance for Controls checkbox in the Compatibility tab, which is equivalent to `DefaultGUIControls native` when checked, and to `DefaultGUIControls os9` when unchecked.

Use `DefaultGUIControls/W=`*`winName`* to override that setting for individual windows.

Parameters

appearance may be one of the following:

native	Creates standard-looking controls for the current computer platform. This is the default value.
os9	Igor Pro 5 appearance (quasi-Macintosh OS 9 controls that look the same on Macintosh and Windows).
default	Inherits the window appearance from either a parent window or the experiment-wide default (only valid with /W).

Flags

/Mac	Changes the appearance of controls only on Macintosh, and it affects the experiment whenever it is used on Macintosh.
/W=*winName*	Affects the named window or subwindow. When omitted, sets an experiment-wide default. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/Win	Changes the appearance of controls only on Windows, and it affects the experiment whenever it is used on Windows.

Details

If *appearance* is not specified, nothing is changed. The current value for appearance is returned in S_value.

If *appearance* is specified the previous appearance value for the window- or experiment-wide default is returned in S_value.

With /W, the control appearance applies only to the specified window (Graph or Panel). If it is not used, then the settings are global to experiments on the current computer. **Tip**: Use `/W=#` to refer to the current active subwindow.

The /Mac and /Win flags specify the affected computer platform. If the current platform other than specified, then the settings are not used, but (if native or OS9) are remembered for use in window recreation macros or experiment recreation. This means you can create an experiment that with different appearances depending on the current platform.

If neither /Mac nor /Win are used, it is implied by the current platform. To set native appearance on both platforms, use two commands:

```
DefaultGUIControls/W=Panel0/Mac native
DefaultGUIControls/W=Panel0/Win native
```

Note: The setting for DefaultGUIControls without /W is not stored in the experiment file; it is a user preference set by the Miscellaneous Settings dialog's Native GUI Appearance for Controls checkbox in the Compatibility tab. If you use DefaultGUIControls native or DefaultGUIControls os9 commands, the checkbox will not show the current state of the experiment-wide setting. Clicking Save Settings in the Miscellaneous Settings dialog will overwrite the DefaultGUIControls setting (but not the per-window settings).

In addition to the experiment-wide appearance setting and the window-specific appearance setting, an individual control's appearance can be set with the appropriate control command's appearance keyword (or a ModifyControl appearance keyword). A control-specific appearance setting overrides a window-specific appearance, which in turn overrides the experiment-wide appearance setting.

Although meant to be used before controls are created, calling DefaultGUIControls will update all open windows.

DefaultGUIControls does not change control fonts or font sizes, which means you can create controls that look "native-ish" without having to readjust their positions to avoid avoid shifting or overlap. However, the smooth font rendering that the Native GUI uses on Macintosh does change the length of text slightly, so some shifting will occur that affects mostly controls that were aligned on their right sides.

The native appearance affects the way that controls are drawn in **TabControl** and **GroupBox** controls.

TabControl Background Details

Unlike the os9 appearance which draws only an outline to define the tab region (leaving the center alone) the native tab appearance fills the tab region. Fortunately, TabControls are drawn before all other kinds of controls which allows enclosed controls to be drawn on top of a tab control regardless of the order in which the buttons are defined in the window recreation macro.

However the drawing order of native TabControls does matter: the top-most TabControls draws over other TabControls. (The top-most TabControl is listed last in the window recreation macro.) The os9 appearance allows a smaller (nested) TabControl to be underneath the later (enclosing) TabControl because tabs normally aren't filled. Converting these tabs to native appearance will cause nested tab to be hidden.

To fix the drawing order problem in an existing panel, turn on the drawing tools, select the arrow tool, right-click the enclosing TabControl, and choose Send to Back to correct this situation. If the TabControl itself is inside another TabControl, select that enclosing TabControl and also choose Send to Back, etc.

To fix the window recreation macro or function that created the panel, arrange the enclosing TabControl commands to execute before the commands that create the enclosed TabControls.

A natively-drawn TabControl draws any drawing objects that are entirely enclosed by the tab region so that it behaves the same as an os9 unfilled TabControl with drawing objects inside.

Groupbox Control Background Details

GroupBox controls, unlike TabControls, are not drawn before all other controls, so the drawing order always matters if the GroupBox specifies a background (fill) color and it contains other controls.

You may find that enabling native appearance hides some controls inside the GroupBox. They are probably underneath (before) the GroupBox in the drawing order.

To fix this in an existing panel, turn on the drawing tools, right-click on the GroupBox and choose Send to Back. To fix the window recreation macro or function that created the panel, arrange the GroupBox commands to execute before the commands that create the enclosed controls.

A natively-drawn GroupBox draws any drawing objects that are entirely enclosed by the box; an os9 filled GroupBox does not.

See Also

The **DefaultGUIFont**, **ModifyControl**, **Button**, **GroupBox**, and **TabControl** operations.

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

DefaultGUIFont

`DefaultGUIFont [/W=`*`winName`*` /Mac/Win] `*`group`*` = {`*`fNameStr`*`,`*`fSize`*`,`*`fStyle`*`} [,…]`

The DefaultGUIFont operation changes the default font for user-defined controls and other Graphical User Interface elements.

Parameters

fNameStr is the name of a font, *fSize* is the font size, and *fStyle* is a bitwise parameter with each bit controlling one aspect of the font style. See **Button** for details about these parameters.

group may be one of the following:

all	All controls
button	Button and default CustomControl
checkbox	CheckBox controls
tabcontrol	TabControl controls
popup	Affects the icon (not the title) of a PopupMenu control. The text in the popped state is set by the system and can not be changed. The title of a PopupMenu is affected by the all group but the icon text is not.
panel	Draw text in a panel.
graph	Overlay graphs. Size is used only if `ModifyGraph gfSize= -1`; style is not used.
table	Overlay tables.

Flags

/Mac	Changes control fonts only on Macintosh, and it affects the experiment whenever it is used on Macintosh.
/W=*winName*	Affects the named window or subwindow. When omitted, sets an experiment-wide default. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/Win	Changes control fonts only on Windows, and it affects the experiment whenever it is used on Windows.

Details

Although designed to be used before controls are created, calling DefaultGUIFont will update all affected windows with controls. This makes it easy to experiment with fonts. Keep in mind that fonts can cause compatibility problems when moving between machines or platforms.

The /Mac and /Win flags indicate the platform on which the fonts are to be used. If the current platform is not the one specified then the settings are not used but are remembered for use in window recreation macros or experiment recreation. This allows a user to create an experiment that will use different fonts depending on the current platform.

If the /W flag is used then the font settings apply only to the specified window (Graph or Panel.) If the /W flag is not used, then the settings are global to the experiment. Tip: Use `/W=#` to refer to the current active subwindow.

fNameStr may be an empty string (`""`) to clear a group. Setting the font name to `"_IgorSmall"`, `"_IgorMedium"`, or `"_IgorLarge"` will use Igor's own defaults. The standard defaults for controls are the equivalent to setting all to `"_IgorSmall"`, tabcontrol to `"_IgorMedium"`, and button to `"_IgorLarge"`. Use a *fSize* of zero to also get the standard default for size. On Windows, the three default fonts and sizes are all the same.

Although designed to be used before controls are created, calling DefaultGUIFont will update all affected windows with controls. This makes it easy to experiment with fonts. Keep in mind that fonts can cause compatibility problems when moving between machines or platforms.

To read back settings, use `DefaultGUIFont [/W=`*`winName`*`/Mac/Win/OVR]` *`group`* to return the current font name in S_name, the size in V_value, and the style in V_flag. With /OVR or if /Mac or /Win is not current, it returns only override values. Otherwise, values include Igor built-in defaults. If S_name is zero length, values are not defined.

Default Fonts and Sizes

The standard defaults for controls is the equivalent to setting all to `"_IgorSmall"`, tabcontrol to `"_IgorMedium"`, and button to `"_IgorLarge"`. Use a *fSize* of zero to also get the standard default for size. On Windows, the three default fonts and sizes are all the same.

	Macintosh		**Windows**	
Control	**Font**	**Font Size**	**Font**	**Font Size**
Button	Lucida Grande	13	MS Shell Dlg*	12
Checkbox	Geneva	9	MS Shell Dlg	12
GroupBox	Geneva	9	MS Shell Dlg	12
ListBox	Geneva	9	MS Shell Dlg	12
PopupMenu†	Geneva	9	MS Shell Dlg	12
SetVariable	Geneva	9	MS Shell Dlg	12
Slider	Geneva	9	MS Shell Dlg	12
TabControl	Geneva	12	MS Shell Dlg	12
TitleBox	Geneva	9	MS Shell Dlg	12
ValDisplay	Geneva	9	MS Shell Dlg	12

* MS Shell Dlg is a "virtual font name" which maps to Tahoma on Windows XP, to MS Sans Serif on Windows 7, and to Segoe UI on Windows 8 and Windows 10.

† On Macintosh, the PopupMenu font is Geneva 9 for the title and Lucida Grande 12 for the popup menu itself. On Windows, both fonts are MS Shell Dlg 12.

Examples

```
DefaultGUIFont/Mac all={"Zapf Chancery",12,0},panel={"geneva",12,3}
DefaultGUIFont/Win all={"Century Gothic",12,0},panel={"arial",12,3}
NewPanel
Button b0
DrawText 40,43,"Some text"
```

See Also

The **DefaultGUIControls** operation. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Window Position Coordinates on page III-405 and **Points Versus Pixels** on page III-404 for explanations of how font sizes in panels are interpreted for various screen resolutions.

Demos

Choose File→Example Experiments→Feature Demos 2→All Controls Demo.

DefaultTextEncoding

`DefaultTextEncoding [encoding=`*`textEncoding`*`, overrideDefault=`*`override`*`]`

The DefaultTextEncoding operation programmatically changes the default text encoding and experiment text encoding override settings. These settings, which are discussed under **The Default Text Encoding** on page III-415, are also accessible via the Misc→Text Encoding→Default Text Encoding menu.

DefaultTextEncoding is rarely needed because typically you will change the default text encoding manually using the menu, if at all.

The DefaultTextEncoding operation was added in Igor Pro 7.00.

Parameters

All parameters are optional. If none are specified, no action is taken, but the output variables are still set.

encoding=*textEncoding*

textEncoding specifies the new default text encoding. See **Text Encoding Names and Codes** on page III-434 for a list of codes.

Pass 0 to set the default text encoding to the equivalent of selecting "Western" from the Default Text Encoding submenu.

The value 255, corresponding to the binary text encoding type, is treated as an invalid value for the *textEncoding* parameter.

overrideDefault=*override*

Turns overriding of the experiment's text encoding off or on.

0: Turns override off

1: Turns override on

Details

The default text encoding affects Igor's behavior when opening a file whose text encoding is unknown. See **The Default Text Encoding** on page III-415 for details.

The experiment text encoding affects how files are loaded during experiment loading only. The override setting allows you to override the experiment text encoding stored in the experiment file. Normally you will not need to do this. See **The Default Text Encoding** on page III-415 for further discussion.

You may occasionally find it necessary to change the default text encoding because an Igor operation lacks a /ENCG flag that allows you to specify the text encoding and instead uses the current default text encoding. In such cases it is a good idea to save the original default text encoding, change it as necessary, and then change it back to the original text encoding. The example below demonstrates this technique.

Output Variables

DefaultTextEncoding sets the following output variables to indicate the settings that are in effect after the command executes:

`V_defaultTextEncoding`	A text encoding code.
`V_overrideExperiment`	A value of zero means the setting is off, nonzero means the setting is on.

Example

```
Function DemoDefaultTextEncoding()
   // Store the original default text encoding
   DefaultTextEncoding
   Variable originalTextEncoding = V_defaultTextEncoding

   // Set new default text encoding
   DefaultTextEncoding encoding = 3     // 3= Windows-1252

   [ Do something that depends on the default text encoding ]

   // Restore the original default text encoding
   DefaultTextEncoding encoding = originalTextEncoding
End
```

See Also

The Default Text Encoding on page III-415, **Text Encoding Names and Codes** on page III-434

defined

`defined(`*`symbol`*`)`

The defined function returns 1 if the symbol is defined 0 if the symbol is not defined.

symbol is a symbol possibly created by a #define statement or by SetIgorOption poundDefine=*symbol*.

symbol is a name, not a string. However you can use $ to convert a string expression to a name.

Details

The defined function can be used in three ways:

Outside of a procedure using a #if statement

Inside a procedure using a #if statement

Inside a procedure using an if statement

For example:

```
#define DEBUG

#if defined(DEBUG)                    // Outside of a function with #if
   Constant kSomeConstant = 100
#else
   Constant kSomeConstant = 50
#endif

Function Test1()                      // Inside a function with #if
   #if defined(DEBUG)
      Print "Debugging"
   #else
      Print "Not debugging"
   #endif
End

Function Test1()                      // Inside a function with if
   if (defined(DEBUG))
      Print "Debugging"
   else
      Print "Not debugging"
   endif
End
```

In these examples, we could have just as well used #ifdef instead of the defined function. For logical combinations of conditions however, only defined will do:

```
#if (defined(SYMBOL1) && defined(SYMBOL2)
   . . .
#endif
```

When used in a procedure window, defined(symbol) returns 1 if symbol is defined at the time the line is compiled. In a given procedure file, only the following symbols are visible:

Symbols defined earlier in that procedure file *

Symbols defined in the built-in procedure window †

Predefined symbols (see **Predefined Global Symbols** on page IV-101)

Symbols defined by **SetIgorOption** poundDefine=*symbol*

* When used in the body of a procedure, as opposed to outside of a procedure, a symbol defined anywhere in a given procedure window is visible. However, to avoid depending on this confusing exception, you should define all symbols before they are referenced in a procedure file.

† Symbols defined in the built-in procedure window are not available to independent modules.

When the defined function is used from the command line, only symbols defined in the built-in procedure window, predefined symbols, and symbols defined using SetIgorOption are visible.

See Also

#**define**, **Conditional Compilation** on page IV-100, **Predefined Global Symbols** on page IV-101

DefineGuide

`DefineGuide [/W= winName] newGuideName = {[guideName1, val [, guideName2]]} [,…]`

The DefineGuide operation creates or overwrites a user-defined guide line in the target or named window or subwindow. Guide lines help with the positioning of subwindows in a host window.

Parameters

newGuideName is the name for the newly created guide. When it is the name of an existing guide, the guide will be moved to the new position.

guideName1, *guideName2*, etc., must be the names of existing guides.

The meaning of *val* depends on the form of the command syntax. When using only one guide name, *val* is an absolute distance offset from to the guide. The directionality of *val* is to the right or below the guide for positive values. The units of measure are points except in panels where they are in pixels. When using two guide names, *val* is the fractional distance between the two guides.

Flags

/W=*winName*	Defines guides in the named window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The names for the built-in guides are as defined in the following table:

	Left	**Right**	**Top**	**Bottom**
Host Window Frame	FL	FR	FT	FB
Host Graph Rectangle	GL	GR	GT	GB
Inner Graph Plot Rectangle	PL	PR	PT	PB

The frame guides apply to all window and subwindow types. The graph rectangle and plot rectangle guide types apply only to graph windows and subwindows.

To delete a guide use `guideName={}`.

See Also

The **Display**, **Edit**, **NewPanel**, **NewImage**, and **NewWaterfall** operations.

The **GuideInfo** function.

DelayUpdate

`DelayUpdate`

The DelayUpdate operation delays the updating of graphs and tables while executing a macro.

Details

Use DelayUpdate at the end of a line in a macro if you want the next line in the macro to run before graphs or tables are updated.

This has no effect in user-defined functions. During execution of a user-defined function, windows update only when you explicitly call the **DoUpdate** operation.

See Also

The **DoUpdate**, **PauseUpdate**, and **ResumeUpdate** operations.

DeleteAnnotations

`DeleteAnnotations [`*`flags`*`] [tagOffscreen, tagTraceHidden, invisible,`
`offsetOffscreen, tooSmall[=`*`size`*`]]`

The DeleteAnnotations operation lists, in the S_name output variable, and optionally deletes annotations that are hidden for reasons specified by the flags and keywords.

The operation affects the window or subwindow specified by the /W flag or, if /W is omitted, the active window or subwindow.

Do not use DeleteAnnotations to progammatically delete a specific, single annotation. Instead use:

```
TextBox/W=winName/K/N=annotationName
```

The /LIST flag limits the action to only listing, instead of deleting, the annotations.

The DeleteAnnotations operation was added in Igor Pro 7.00.

Keywords

The keywords identify annotations based on the reasons for their being hidden:

invisible	Deletes or lists annotations hidden with /V=0.
offsetOffscreen	Deletes or lists annotations that are offscreen, usually because of excessive /X and /Y offsets.
tagOffscreen	Deletes or lists tags hidden because they are attached to trace points that are offscreen. This affects trace tags, axis tags, and image tags if their "if offscreen" setting, as set in the Position tab of the Modify Annotation dialog, is set to "hide the tag".
tagTraceHidden	Deletes or lists tags hidden because the tagged trace is hidden.
tooSmall [=*size*]	Deletes or lists annotations whose height or width is *size* points or smaller. *size* is expressed in points and defaults to 8. This is useful for deleting annotations that are too small to see or to double-click.

Flags

/A	All annotations, whether hidden or not, are listed or deleted. All keywords are ignored.
/LIST	Specifies that annotations identified by the other parameters are to be listed in the S_name output variable but not deleted.
/W=*winName*	Annotations in the named window or subwindow are considered. When omitted, annotations in the active window or subwindow are considered. When identifying a subwindow with winName, see Subwindow Syntax for details on forming the window hierarchy.

Output Variables

`S_name`	A semicolon-separated list of the annotations that match the criteria set by the keywords and flags.
`V_flag`	Set to the number of annotations deleted or listed.

Examples

```
Function DeleteAnnotationsInWin(win)
   String win          // Specifies a top-level window or a subwindow

   // Handle specified top-level window or subwindow
   DeleteAnnotations/W=$win/A
   Variable numDeleted = V_Flag

   // Now handle subwindows, if any
   String children = ChildWindowList(win)
   Variable n = ItemsInList(children)
   Variable i
   for(i=0; i<n; i+=1)
      String child = StringFromList(i, children)
      numDeleted += DeleteAnnotationsInWin(child) // Recurse
   endfor

   return numDeleted
End
```

See Also

TextBox, **StringFromList**, **AnnotationList**

DeleteFile

`DeleteFile` `[`***`flags`***`] [`***`fileNameStr`***`]`

The DeleteFile operation deletes a file on disk.

Parameters

fileNameStr can be a full path to the file to be deleted (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*.

If Igor can not locate the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file to be deleted.

If you use a full or partial path for either file, see **Path Separators** on page III-401 for details on forming the path.

Flags

/I	Interactive mode displays the Open File dialog even if *fileNameStr* is specified and the file exists.
/M=*messageStr*	Specifies the prompt message for the Open File dialog. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/Z[=z]	Prevents procedure execution from aborting if it attempts to delete a file that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

/Z=0:	Same as no /Z.
/Z=1:	Deletes a file only if it exists. /Z alone has the same effect as /Z=1.
/Z=2:	Deletes a file if it exists or displays a dialog if it does not exist.

Variables

The DeleteFile operation returns information in the following variables:

`V_flag`	Set to zero if the file was deleted, to -1 if the user cancelled the Open File dialog, and to some nonzero value if an error occurred, such as the specified file does not exist.
`S_path`	Stores the full path to the file that was deleted. If an error occurred or if the user cancelled, it is set to an empty string.

See Also

DeleteFolder, **MoveFile**, **CopyFile**, **NewPath**, and **Symbolic Paths** on page II-21.

DeleteFolder

`DeleteFolder [`*`flags`*`] [`*`folderNameStr`*`]`

The DeleteFolder operation deletes a disk folder and all of its contents.

Warning: *The DeleteFolder command destroys data!* The deleted folder and the contents are not moved to the Trash or Recycle Bin.

DeleteFolder will delete a folder only if permission is granted by the user. The default behavior is to display a dialog asking for permission. The user can alter this behavior via the Miscellaneous Settings dialog's Misc category.

If permission is denied, the folder will not be deleted and V_Flag will return 1088 (Command is disabled) or 1276 (You denied permission to delete a folder). Command execution will cease unless the /Z flag is specified.

Parameters

folderNameStr can be a full path to the folder to be deleted, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a folder within the folder associated with *pathName*.

If Igor can not determine the location of the folder from *folderNameStr* and *pathName*, it displays a Select Folder dialog allowing you to specify the folder to be deleted.

If /P=*pathName* is given, but *folderNameStr* is not, then the folder associated with *pathName* is deleted.

If you use a full or partial path for either folder, see **Path Separators** on page III-401 for details on forming the path.

Folder paths should not end with single Path Separators. See the **MoveFolder Details** section.

Flags

/I	Interactive mode displays a Select Folder dialog even if *folderNameStr* is specified and the folder exists.
/M=*messageStr*	Specifies the prompt message for the Select Folder dialog. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/P=*pathName*	Specifies the folder to look in for the folder. *pathName* is the name of an existing symbolic path.
/Z[=z]	Prevents procedure execution from aborting if it attempts to delete a folder that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

/Z=0:	Same as no /Z.
/Z=1:	Deletes a folder only if it exists. /Z alone has the same effect as /Z=1.
/Z=2:	Deletes a folder if it exists or displays a dialog if it does not exist.

Variables

The DeleteFolder operation returns information in the following variables:

`V_flag`	Set to zero if the folder was deleted, to -1 if the user cancelled the Select Folder dialog, and to some nonzero value if an error occurred, such as the specified folder does not exist.
S_path	Stores the full path to the folder that was deleted, with a trailing colon. If an error occurred or if the user cancelled, it is set to an empty string.

Details

You can use only /P=*pathName* (without *folderNameStr*) to specify the source folder to be deleted.

Folder paths should not end with single Path Separators. See the **Details** section for **MoveFolder**.

See Also

The **DeleteFile**, **MoveFolder**, **CopyFolder**, **NewPath**, and **IndexedDir** operations. **Symbolic Paths** on page II-21.

DeletePoints

`DeletePoints [/M=`*`dim`*`]` ***`startElement, numElements, waveName`***
   `[,` ***`waveName`*** **`]…`**

The DeletePoints operation deletes *numElements* elements from the named waves starting from element *startElement*.

Flags

/M=*dim*	*dim* specifies the dimension from which elements are to be deleted. Values are:
	0: Rows.
	1: Columns.
	2: Layers.
	3: Chunks.
	If /M is omitted, DeletePoints deletes from the rows dimension.

Details

A wave may have any number of points, including zero. Removing all elements from any dimension removes all points from the wave, leaving a 1D wave with zero points.

Except for the case of removing all elements, DeletePoints does not change the dimensionality of a wave. Use **Redimension** for that.

See Also

The **Redimension** operation.

deltax

`deltax(`*`waveName`*`)`

The deltax function returns the named wave's dx value. deltax works with 1D waves only.

Details

This is equal to the difference of the X value of point 1 minus the X value of point 0.

See Also

The **leftx** and **rightx** functions.

When working with multidimensional waves, use the **DimDelta** function.

For an explanation of waves and wave scaling, see **Changing Dimension and Data Scaling** on page II-63.

DFREF

`DFREF` ***`localName`*** **`[=`** ***`path`*** **`or`** ***`dfr`*** **`], [`*`localName1`*** **`[=`** ***`path`*** **`or`** ***`dfr`*** **`]]`**

DFREF is used to define a local data folder reference variable or input parameter in a user-defined function.

The syntax of the DFREF is:

```
DFREF localName [= path or dfr ][, localName1 [= path or dfr ]]...
```

where *dfr* stands for "data folder reference". The optional assignment part is used only in the body of a function, not in a parameter declaration.

Unlike the **WAVE** reference, a DFREF in the body without the assignment part does not do any lookup. It simply creates a variable whose value is null.

Examples

```
Function Test(dfr)
   DFREF dfr

   Variable dfrStatus = DataFolderRefStatus(dfr)
   if (dfrStatus == 0)
      Print "Invalid data folder reference"
      return -1
   endif

   if (dfrStatus & 2)            // Bit 1 set means free data folder
      Print "Data folder reference refers to a free data folder"
   endif

   if (dfrStatus == 1)
      Print "Data folder reference refers a global data folder"
      DFREF dfSav = GetDataFolderDFR()
      Print GetDataFolder(1)     // Print data folder path
      SetDataFolder dfSav
   endif

   Make/O dfr:jack=sin(x/8)      // Make a wave in the referenced data folder

   return 0
End
```

See Also

For information on programming with data folder references, see **Data Folder References** on page IV-72.

Differentiate

Differentiate [***type flags***][***flags***] ***yWaveA*** [**/X =** ***xWaveA***]
   [**/D =** ***destWaveA***][, ***yWaveB*** [**/X =** ***xWaveB***][**/D =** ***destWaveB***][, …]]

The Differentiate operation calculates the 1D numerical derivative of a wave.

Differentiate is multi-dimension-aware in the sense that it computes a 1D differentiation along the dimension specified by the /DIM flag or along the rows dimension if you omit /DIM.

Complex waves have their real and imaginary components differentiated individually.

Flags

/DIM=*d*	Specifies the wave dimension along which to differentiate when *yWave* is multi-dimensional. *d*=-1: Treats entire wave as 1D (default). *d*=0: Differentiates along rows. *d*=1: Differentiates along columns. *d*=2: Differentiates along layers. *d*=3: Differentiates along rows. For example, for a 2D wave, /DIM=0 differentiates each row and /DIM=1 differentiates each column.
/EP=*e*	Controls end point handling. *e*=0: Replaces undefined points with an approximation (default). *e*=1: Deletes the point(s).
/METH=*m*	Sets the differentiation method. *m*=0: Central difference (default). *m*=1: Forward difference. *m*=2: Backward difference.
/P	Forces point scaling.

Type Flags (*used only in functions*)

Differentiate also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-67 and **WAVE Reference Type Flags** on page IV-68 for a complete list of type flags and further details.

For example, when the input (and output) waves are complex, the output wave will be complex. To get the Igor compiler to create a complex output wave reference, use the /C type flag with /D=destwave:

```
Make/O/C cInput=cmplx(sin(p/8), cos(p/8))
Make/O/C/N=0 cOutput
Differentiate/C cInput /D=cOutput
```

Wave Parameters

Note:	***All*** wave parameters must follow *yWave* in the command. All wave parameter flags and type flags must appear immediately after the operation name.
/D=*destWave*	Specifies the name of the wave to hold the differentiated data. It creates *destWave* if it does not already exist or overwrites it if it exists.
/X=*xWave*	Specifies the name of the corresponding X wave.

Details

If the optional /D = *destWave* flag is omitted, then the wave is differentiated in place overwriting the original data.

When using a method that deletes points (/EP=1) with a multidimensional wave, deletion is not done if no dimension is specified.

When using an X wave, the X wave must match the Y wave data type (excluding the complex type flag) and it must be 1D with the number points matching the size of the dimension being differentiated. X waves are not used with integer source waves.

`Differentiate/METH=1/EP=1` is the inverse of `Integrate/METH=2`, but `Integrate/METH=2` is the inverse of `Differentiate/METH=1/EP=1` only if the original first data point is added to the output wave.

Differentiate applied to an XY pair of waves does not check the ordering of the X values and doesn't care about it. However, it is usually the case that your X values should be monotonic. If your X values are not monotonic, you should be aware that the X values will be taken from your X wave in the order they are found, which will result in random X intervals for the X differences. It is usually best to sort the X and Y waves using **Sort**.

See Also

The **Integrate** operation.

digamma

`digamma(x)`

The digamma function returns the digamma, or psi function of *x*. This is the logarithmic derivative of the gamma function:

$$\Psi(z) \equiv \frac{d}{dz}\ln\left(\Gamma(z)\right) = \frac{\Gamma'(z)}{\Gamma(z)}.$$

In complex expressions, *x* is complex, and `digamma(x)` returns a complex value.

Limited testing indicates that the accuracy is approximately 1 part in 10^{16}, at least for moderately-sized values of *x*.

Dilogarithm

`Dilogarithm(z)`

Returns the Dilogarithm function for real or complex argument *z*. The dilogarithm is a special case of the polylogarithm defined by

$$Li_2(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^2}.$$

The dilogarithm function was added in Igor Pro 7.00.

See Also

zeta

Reference

Wood, D.C. (June 1992). "The Computation of Polylogarithms. Technical Report 15-92". Canterbury, UK: University of Kent Computing Laboratory.

The function based on an algorithm by Didier Clamond.

DimDelta

`DimDelta(`***`waveName, dimNumber`***`)`

The DimDelta function returns the scale factor delta of the given dimension.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers and 3 for chunks. If *dimNumber*=0 this is identical to `deltax(`*`waveName`*`)`.

See Also

DimOffset, **DimSize**, **SetScale**, **WaveUnits**, **ScaleToIndex**

For an explanation of waves and wave scaling, see **Changing Dimension and Data Scaling** on page II-63.

DimOffset

`DimOffset(`***`waveName, dimNumber`***`)`

The DimOffset function returns the scaling offset of the given dimension.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers, and 3 for chunks. If *dimNumber*=0 this is identical to `leftx(`*`waveName`*`)`.

See Also

DimDelta, **DimSize**, **SetScale**, **WaveUnits**, **ScaleToIndex**

For an explanation of waves and wave scaling, see **Changing Dimension and Data Scaling** on page II-63.

DimSize

`DimSize(`***`waveName, dimNumber`***`)`

The DimSize function returns the size of the given dimension.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers, and 3 for chunks. For a 1D wave, `DimSize(`*`waveName`*`,0)` is identical to `numpnts(`*`waveName`*`)`.

See Also

DimDelta, **DimOffset**, **SetScale**, **WaveUnits**

Dir

`Dir [`***`dataFolderSpec`***`]`

The Dir operation returns a listing of all the objects in the specified data folder.

Parameters

If you omit *dataFolderSpec* then the current data folder is used.

If present, *dataFolderSpec* can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

Details

The format of the printed information is the same as the format used by the string function **DataFolderDir**. Igor programmers may find it more convenient to use **CountObjects** and **GetIndexedObjName**.

Usually it is easier to use the Data Browser (Data menu). However, **Dir** is useful when you want to copy a name into the command line or when you want to document the current state of the folder in the history.

See Also

Chapter II-8, **Data Folders**.

Display

```
Display [flags] [waveName [, waveName ]…[vs xwaveName]]
   [as titleStr]
```

The Display operation creates a new graph window or subwindow, and appends the named waves, if any. Waves are displayed as 1D traces.

By default, waves are plotted versus the left and bottom axes. Use the /L, /B, /R, and /T flags to plot the waves against other axes.

Parameters

Up to 100 *waveName*s may be specified, subject to the 1000 byte command line length limit. If no wave names are specified, a blank graph is created and the axis flags are ignored.

If you specify "vs *xwaveName*", the Y values of the named waves are plotted versus the Y values of *xwaveName*. If you don't specify "vs *xwaveName*", the Y values of each *waveName* are plotted versus its own X values.

If *xwaveName* is a text wave, the resulting plot is a category plot. Each element of *waveName* is plotted by default in bars mode (`ModifyGraph mode=5`) against a category labeled with the text of the corresponding element of *xwaveName*.

The Y waves for a category plot should have point scaling (see **Changing Dimension and Data Scaling** on page II-63); this is how category plots were intended to work. However, if all the Y waves have the same scaling, it will work correctly.

titleStr is a string expression containing the graph's title. If not specified, Igor will provide one which identifies the waves displayed in the graph.

Subsets of data, including individual rows or columns from a matrix, may be specified using **Subrange Display Syntax** on page II-250.

You can provide a custom name for a trace by appending /TN=traceName to the waveName specification. This is useful when displaying waves with the same name but from different data folders. See **User-defined Trace Names** on page IV-82 for more information.

Flags

/B[=*axisName*]	Plots X coordinates versus the standard or named bottom axis.
/FG=(*gLeft*, *gTop*, *gRight*, *gBottom*)	
	Specifies the frame guide to which the outer frame of the subwindow is attached inside the host window. The standard frame guide names are `FL`, `FR`, `FT`, and `FB`, for the left, right, top, and bottom frame guides, respectively, or user-defined guide names as defined by the host. Use `*` to specify a default guide name. Guides may override the numeric positioning set by /W.
/HIDE=*h*	Hides (h = 1) or shows (h = 0, default) the window.
/HOST=*hcSpec*	Embeds the new graph in the host window or subwindow specified by *hcSpec*. When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/I	Specifies that /W coordinates are in inches.

/K=*k*	Specifies window behavior when the user attempts to close it. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. *k*=3: Hides the window. If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.
/L[=*axisName*]	Plots Y coordinates versus the standard or named left axis.
/M	Specifies that /W coordinates are in centimeters.
/N=*name*	Requests that the created graph have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen graph name. Use `DoWindow/K` *`name`* to ensure that *name* is available.
/NCAT	In Igor Pro 6.37 or later, allows subsequent appending of a category trace to a numeric plot. See **Combining Numeric and Category Traces** on page II-322 for details.
/PG=(*gLeft*, *gTop*, *gRight*, *gBottom*)	Specifies the inner plot rectangle of the graph subwindow inside its host window. The standard plot rectangle guide names are `PL`, `PR`, `PT`, and `PB`, for the left, right, top, and bottom plot rectangle guides, respectively, or user-defined guide names as defined by the host. Use `*` to specify a default guide name. Guides may override the numeric positioning set by /W.
/R[=*axisName*]	Plots Y coordinates versus the standard or named right axis.
/T[=*axisName*]	Plots Y coordinates versus the standard or named top axis.
/TN=*traceName*	Allows you to provide a custom trace name for a trace. This is useful when displaying waves with the same name but from different data folders. See **User-defined Trace Names** on page IV-82 for details.
/W=(*left*,*top*,*right*,*bottom*)	Gives the graph a specific location and size on the screen. Coordinates for /W are in points unless /I or /M are specified before /W. When used with the /HOST flag, the specified location coordinates of the sides can have one of two possible meanings: 1: When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size. 2: When any value is greater than 1, coordinates are taken to be fixed locations measured in points relative to the top left corner of the host frame. When the subwindow position is fully specified using guides (using the /HOST, /FG, or /PG flags), the /W flag may still be used although it is not needed.

Details

If /N is not used, Display automatically assigns to the graph a name of the form "Graph*n*", where *n* is some integer. In a function or macro, the assigned name is stored in the S_name string. This is the name you can use to refer to the graph from a procedure. Use the **RenameWindow** operation to rename the graph.

Examples

To make a contour plot, use:

```
Display; AppendMatrixContour waveName
```

or

```
Display; AppendXYZContour waveName
```

To display an image, use:

```
Display; AppendImage waveName
```

or

```
NewImage waveName
```

See Also

The **AppendToGraph** operation.

The operations **AppendImage**, **AppendMatrixContour**, **AppendXYZContour**, and **NewImage**. For more information on Category Plots, see Chapter II-13, **Category Plots**.

The operations **ModifyGraph**, **ModifyContour**, and **ModifyImage** for changing the characteristics of graphs.

The **DoWindow** operation for changing aspects of the graph window.

DisplayHelpTopic

`DisplayHelpTopic [/K=`*`k`*` /Z]` ***`TopicString`***

The DisplayHelpTopic operation displays a help topic as if a help link had been clicked in an Igor help file.

Parameters

TopicString is string expression containing the topic. It may be in one of three forms: <topic name>, <subtopic name>, <topic name>[<subtopic name>]. These forms are illustrated by the examples.

Make sure that your topic string is specific to minimize the likelihood that Igor will find the topic in a help file other than the one you intended. To avoid this problem, it is best to use the <topic name>[<subtopic name>] form if possible.

Flags

/K=*k*	Determines when the help file is closed.	
	k=0:	Leaves the help file open indefinitely (default). Use this if the help topic may be of interest in any experiment.
	k=1:	If the found topic is in a closed help file, the help file closes with the current experiment. Use this if the help topic is tightly associated with the current experiment.
/Z	Ignore errors. If /Z is used, DisplayHelpTopic sets V_flag to 0 if the help topic was found or to a nonzero error code if it was not found. V_flag is set only when /Z is used.	

Details

DisplayHelpTopic first searches for the specified topic in the open help files. If the topic is not found, it then searches all help files in the Igor Pro 7 folder and subfolders.

If the topic is still not found, it then searches all help files in the current experiment's home folder, but not in subfolders. This puts a help file that is specific to a particular experiment in the experiment's home folder.

If the topic is still not found and if DisplayHelpTopic was called from a procedure and if the procedure resides in a stand-alone file on disk (i.e., it is not in the built-in procedure window or in a packed procedure file), Igor then searches all help files in the procedure file's folder, but not in subfolders. This puts a help file that is specific to a particular set of procedures in the same folder as the procedure file.

If Igor finds the topic, it displays it. If Igor can not find the topic, it displays an error message, unless /Z is used.

Examples

```
// This example uses the topic only.
DisplayHelpTopic "Waves"

// This example uses the subtopic only.
DisplayHelpTopic "Waveform Arithmetic and Assignment"

// This example uses the topic[subtopic] form.
DisplayHelpTopic "Waves[Waveform Arithmetic and Assignment]"
```

See Also

Chapter II-1, **Getting Help** for information about Igor help files and formats.

DisplayProcedure

`DisplayProcedure [flags] [`*`functionOrMacroNameStr`*`]`

The DisplayProcedure operation displays the named function, macro or line by bringing the procedure window it is defined in to the front with the function, macro or line highlighted.

Parameters

functionOrMacroNameStr is a string expression containing the name of the function or macro to display. If you omit *functionOrMacroNameStr* then you must use /W or /L.

functionOrMacroNameStr may be a simple name or may include independent module and/or module name prefixes to display static functions.

If you use /L to display a particular line then you must omit *functionOrMacroNameStr*.

To display a procedure window without changing its scrolling or selection, use /W and omit *functionOrMacroNameStr*.

Flags

/B=*winTitleOrName*	Brings up the procedure window just behind the window with this name or title.
/L=*lineNum*	If /W is specified, *lineNum* is a zero-based line number in the specified window. If /W is not specified, *lineNum* is a “global” line number. Each procedure window line has a unique global line number as if all of the procedure files were concatenated into one big file. The order of concatenation of files can change when procedures are recompiled. If you use /L then you must omit *functionOrMacroNameStr*.
/W=*procWinTitle*	Searches in the procedure window with this title. *procWinTitle* is a name, not a string, so you construct /W like this: `/W=$"New Polar Graph.ipf"` If you omit /W, DisplayProcedure searches all open (nonindependent module) procedure windows.

Details

If a procedure window has syntax errors that prevent Igor from determining where functions and macros start and end, then DisplayProcedure may not be able to locate the procedure.

winTitleOrName is not a string; it is a name. To position the found procedure window behind a window whose title has a space in the name, use the $ operator as in the second example, below.

If *winTitleOrName* does not match any window, then the found procedure window is placed behind the top target window.

lineNum is a zero-based line number: 0 is the first line of the window. Because each line of a procedure window is a paragraph, line numbers and paragraph numbers are the same. You can use the Procedure→Info menu item to show a selection's starting and ending paragraph/line number.

procWinTitle is also a name. Use `/W=$"New Polar Graph.ipf"` to search for the function or macro in only that procedure file.

Don’t specify both *functionOrMacroNameStr* and /L=*lineNum* as this is ambiguous and not allowed.

Advanced Details

If `SetIgorOption IndependentModuleDev=1`, *procWinTitle* can also be a title followed by a space and, in brackets, an independent module name. In such cases searches for the function or macro are in the specified procedure window and independent module. (See **Independent Modules** on page IV-224 for independent module details.)

For example, if any procedure file contains these statements:

```
#pragma IndependentModule=myIM
#include <Axis Utilities>
```

The command

```
DisplayProcedure/W=$"Axis Utilities.ipf [myIM]" "HVAxisList"
```

opens the procedure window that contains the `HVAxisList` function, which is in the Axis Utilities.ipf file and the independent module `myIM`. The command uses the `$""` syntax because space and bracket characters interfere with command parsing.

Similarly, if `SetIgorOption IndependentModuleDev=1` then *functionOrMacroNameStr* may also contain an independent module prefix followed by the # character. The preceding command can be rewritten as:

```
DisplayProcedure/W=$"Axis Utilities.ipf" "myIM#HVAxisList"
```

or more simply

```
DisplayProcedure "myIM#HVAxisList"
```

You can use the same syntax to display a static function in a non-independent module procedure file using a module name instead of (or in addition to) the independent module name.s

procWinTitle can also be just an independent module name in brackets to retrieve the text from *any* procedure window that belongs to named independent module:

```
DisplayProcedure/W=$"[myIM]" "HVAxisList"
```

Examples

```
DisplayProcedure "Graph0"
DisplayProcedure/B=Panel0 "MyOwnUserFunction"
DisplayProcedure/W=Procedure         // Shows the main Procedure window
DisplayProcedure/W=Procedure/L=5     // Shows line 5 (the sixth line)
DisplayProcedure/W=$"Wave Lists.ipf"
DisplayProcedure "moduleName#myStaticFunctionName"
SetIgorOption IndependentModuleDev=1
DisplayProcedure "WMGP#GizmoBoxAxes#DrawAxis"
```

See Also

Independent Modules on page IV-224.

MacroList, **FunctionList**, and **ProcedureText**, **HideProcedures**, **DoWindow**.

do-while

```
do
   <loop body>
while(<expression>)
```

A do-while loop executes *loop body* until *expression* evaluates as FALSE (zero) or until a break statement is executed.

See Also

Do-While Loop on page IV-42 and **break** for more usage details.

DoAlert

`DoAlert [/T=`*`titleStr`*`] `*`alertType`*`, `*`promptStr`*

The DoAlert operation displays an alert dialog and waits for user to click button.

Parameters

alertType=*t*	Controls the type of alert dialog:	
	t=0:	Dialog with an OK button.
	t=1:	Dialog with Yes button and No buttons.
	t=2:	Dialog with Yes, No, and Cancel buttons.
promptStr	Specifies the text that is displayed in the alert dialog.	

Flags

/T=*titleStr*	Changes the title of the dialog window from the default title.

Details

DoAlert sets the variable V_flag as follows:

1:	Yes clicked.
2:	No clicked.
3:	Cancel clicked.

See Also

The **Abort** operation.

DoIgorMenu

`DoIgorMenu [/C /OVRD]` ***`MenuNameStr, MenuItemStr`***

The DoIgorMenu operation allows an Igor programmer to invoke Igor's built-in menu items. This is useful for bringing up Igor's built-in dialogs under program control.

Parameters

MenuNameStr	The name of an Igor menu, like "File", "Graph", or "Load Waves".
MenuItemStr	The text of an Igor menu item, like "Copy" (in the Edit menu) or "New Graph" (in the Windows menu).

Flags

Using both the /C and the /OVRD flag in one command is not permitted.

/C	Just Checking. The menu item is not invoked, but V_flag is set to 1 if the item was enabled or to 0 if it was not enabled.
/OVRD	Tells Igor to skip checks that it normally does before executing the menu command specified by *MenuNameStr* and *MenuItemStr*. You are responsible for ensuring that the menu command you are invoking is appropriate under conditions existing at runtime. The main use for the /OVRD flag is to allow an advanced programmer to invoke a menu command for a menu that is currently hidden when dealing with subwindows. For example, if you have a graph subwindow in a control panel which is in operate mode, the Graph menu is not visible in the menu bar. Normally the user could not invoke an item, such as Modify Trace Appearance. /OVRD allows you to invoke the menu command, but it is up to you to verify that it is appropriate. In the Modify Trace Appearance example, you should invoke the menu command only if the active window or subwindow is a graph that contains at least one trace. /OVRD was added in Igor Pro 7.00.

Details

All menu names and menu item text are in English to ensure that code developed for a localized version of Igor Pro will run on all versions. Note that no trailing "…" is used in *MenuItemStr*.

V_flag is set to 1 if the corresponding menu item was enabled, which usually means the menu item was successfully selected. Otherwise V_flag is 0. V_flag does not reflect the success or failure of the resulting dialog, if any.

If the menu item selection displays a dialog that generates a command, clicking the Do It button executes the command immediately without using the command line as if Execute/Z operation had been used. Clicking the To Cmd Line button appends the command to the command line rather than inserting the command at the front.

The DoIgorMenu operation will not attempt to select a menu during curve fitting. Doubtless there are other times during which using DoIgorMenu would be unwise.

The text of some items in the File menu changes depending on the type of the active window. In these cases you must pass generic text as the *MenuItemStr* parameter. Use “Save Window”, “Save Window As”, “Save Window Copy”, “Adopt Window”, and “Revert Window” instead of “Save Notebook” or “Save Procedure”, etc. Use “Page Setup” instead of “Page Setup For All Graphs”, etc. Use “Print” instead of “Print Graph”, etc.

See Also

The **SetIgorMenuMode** and **Execute** operations.

DoPrompt

`DoPrompt [/HELP=helpStr] dialogTitleStr, variable [, variable]…`

The DoPrompt statement in a function invokes the simple input dialog. A DoPrompt specifies the title for the simple input dialog and which input variables are to be included in the dialog.

Flags

/HELP=*helpStr*	Sets the help topic or help text that appears when the dialog’s Help button is pressed. *helpStr* can be a help topic and subtopic such as is used by `DisplayHelpTopic/K=1 helpStr`, or it can be text (255 characters max) that is displayed in a subdialog just as if `DoAlert 0, helpStr` had been called, or *helpStr* can be `""` to remove the Help button.

Parameters

variable is the name of a dialog input variable, which can be real or complex numeric local variable or local string variable, defined by a Prompt statement. You can specify as many as 10 variables.

dialogTitleStr is a string or string expression containing the text for the title of the simple input dialog.

Details

Prompt statements are required to define what variables are to be used and the text for any string expression to accompany or describe the input variable in the dialog. When a DoPrompt variable is missing a Prompt statement, you will get a compilation error. Pop-up string data can not be continued across multiple lines as can be done using Prompt in macros. See **Prompt** for further usage details.

Prompt statements for the input variables used by DoPrompt must come before the DoPrompt statement itself, otherwise, they may be used anywhere within the body of a function. The variables are not required to be input parameters for the function (as is the case for Prompt in macros) and they may be declared within the function body. DoPrompt can accept as many as 10 variables.

Functions can use multiple DoPrompt statements, and Prompt statements can be reused or redefined.

When the user clicks the Cancel button, any new input parameter values are not stored in the variables.

DoPrompt sets the variable V_flag as follows:

0:	Continue button clicked.
1:	Cancel button clicked.

See Also

The Simple Input Dialog on page IV-132, the **Prompt** keyword, and **DisplayHelpTopic**.

Double

`double localName`

Declares a local 64-bit double-precision variable in a user-defined function or structure.

Double is another name for Variable. It is available in Igor Pro 7 and later.

DoUpdate

`DoUpdate [/E=`*`e`*` /W=`*`targWin`*` /SPIN=`*`ticks`*` ]`

The DoUpdate operation updates windows and dependent objects.

Flags

/E=*e*	Used with /W, /E=1 marks window as a progress window that can accept mouse events while user code is executing. Currently, only control panel windows can be used as a progress window.
/W=*targWin*	Updates only the specified window. Does not update dependencies or do any other updating. Currently, only graph and panel windows honor the /W flag. V_Flag is set to the truth the window exists. See **Progress Windows** on page IV-144 for other values for V_Flag.
/SPIN=*ticks*	Sets the delay between the start of a control procedure and the spinning beachball. *ticks* is the delay in ticks (60th of a second.) Unless used with the /W flag, /SPIN just sets the delay and an update is not done.

Details

Call DoUpdate from an Igor procedure to force Igor to update any objects that need updating. Igor updates any graphs, tables or page layouts that need to be updated and also any objects (string variables, numeric variables, waves, controls) that depend on other objects that have changed since the last update.

Igor performs updates automatically when:

- No user-procedure is running.
- An interpreted procedure (Macro, Proc, Window type procedures) is running and PauseUpdate or DelayUpdate is not in effect.

An automatic DoUpdate is not done while a user-defined function is running. You can call DoUpdate from a user-defined function to force an update.

See Also

The **DelayUpdate**, **PauseUpdate**, and **ResumeUpdate** operations, **Progress Windows** on page IV-144.

DoWindow

`DoWindow [`*`flags`*`] [`*`windowName`*`]`

The DoWindow operation controls various window parameters and aspects. There are additional forms for DoWindow when the /S or /T flags are used; see the following DoWindow entries.

Parameters

windowName is the name of a graph, table, page layout, notebook, panel, Gizmo, camera, or XOP target window.

A window's name is *not* the same as its title. The title is shown in the window's title bar. The name is used to manipulate the window from Igor commands. You can check both the name and the title using the Window Control dialog (in the Arrange submenu of the Window menu).

Flags

/B[=*bname*]	Moves the specified window to the back (to the bottom of desktop) or behind window *bname*.
/C	Changes the name of the target window to the specified name. The specified name must not be used for any other object except that it can be the name of an existing window macro.
/C/N	Changes the target window name and creates a new window macro for it. However, /N does nothing if a macro or function is running. /N is not applicable to notebooks.
/D	Deletes the file associated with window, if any (for notebooks only).

/F	Brings the window with the given name to the front (top of desktop).
/H	Specifies the command window as the target of the operation. When using /H, *windowName* must not be specified and only the /B and /HIDE flags are honored. Use /H to bring the command window to the front (top of desktop). Use /H/B to send the command window to the bottom of the desktop. Use /H/HIDE to hide or show the command window.
/HIDE=*h*	Sets hidden state of a window. *h*=0: Visible. *h*=1: Hidden. *h*=?: Sets the variable V_flag as follows: 0: The window does not exist. 1: The window is visible. 2: The window is hidden. You can also read the hidden state using **GetWindow** and set it using **SetWindow**.
/K	Kills the window with the given name. **KillWindow** is preferred.
/N	Creates a new window macro for the window with the given name. However, /N does nothing if a macro or function is running. /N is not applicable to notebooks.
/R	Replaces (updates) the window macro for the named window or creates it if it does not yet exist. However, /R does nothing if a macro or function is running. /R is not applicable to notebooks.
/R/K	Replaces (updates) the window macro for the named window or creates it if it does not yet exist and then kills the window. However, /R does nothing if a macro or function is running. /R is not applicable to notebooks.
/W=*targWin*	Designates *targWin* as the target window; it also requires that you specify *windowName*. Use this mainly with floating panels, which are always on top. You can use a subwindow specification of an external subwindow only with the /T flag or without any flags.

Details

DoWindow sets the variable V_flag to 1 if there was a window with the specified name after DoWindow executed, to 0 if there was no such window, or to 2 if the window is hidden.

Call DoWindow with no flags to check if a window exists.

When used with the /N flag, *windowName* must not conflict with the name of any other object. When used with the /C flag, *windowName* must not conflict with the name of any other object except that it can be the name of an existing window macro.

The /R and /N flags do nothing when executed while a macro or function is running. This is necessary because changing procedures while they are executing causes unpredictable and undesirable results. However you can use the Execute/P operation to cause the DoWindow command to be executed after procedures are finished running. For example:

```
Function SaveWindowMacro(windowName)
   String windowName                   // "" for top graph or table

   if (strlen(windowName) == 0)
      windowName = WinName(0, 3)       // Name of top graph or table
   endif

   String cmd
   sprintf cmd, "DoWindow/R %s", windowName
   Execute/P cmd
End
```

You can use the /D flag in conjunction with the /K flag to kill a notebook window and delete its associated file, if any. /D has no effect on any other type of window and has no effect if the /K flag is not present.

Examples

```
DoWindow Graph0           // Set V_flag to 1 if Graph0 window exists.
DoWindow/F Graph0         // Make Graph0 the top/target window.
DoWindow/C MyGraph        // Target window (Graph0) renamed MyGraph.
DoWindow/K Panel0         // Kill the Panel0 window.
DoWindow/H/B              // Put the command window in back.
DoWindow/D/K Notebook2    // Kill Notebook2, delete its file.
```

See Also

RenameWindow, **MoveWindow**, **MoveSubwindow**, **SetActiveSubwindow**, **KillWindow**

HideProcedures, **IgorInfo**

DoWindow/T

`DoWindow /T` ***`windowName, windowTitleStr`***

The DoWindow/T operation sets the window title for the named window to the specified title.

Details

The title is shown in the window's title bar, and listed in the appropriate Windows submenu. The window *name* is still used to manipulate the window, so, for example, the window name (if *windowName* is a graph or table) is listed in the New Layout dialog; not the title.

You can check both the name and the title using the Window Control dialog (in the Control submenu of the Windows menu).

windowName is the name of the window or a special keyword, kwTopWin or kwFrame.

If *windowName* is kwTopWin, DoWindow retitles the top target window.

If *windowName* is kwFrame, DoWindow retitles the "frame" or "application" window that Igor has only under Windows. This is the window that contains Igor's menus and status bar. On Macintosh, kwFrame is allowed, but the command does nothing.

The Window Control dialog does not support kwFrame. The frame title persists until Igor quits or until it is restored as shown in the example. Setting *windowTitleStr* to `""` will restore the normal frame title.

Examples

```
DoWindow/T MyGraph, "My Really Neat Graph"
DoWindow/T kwFrame, "My Igor-based Application"
DoWindow/T kwFrame, ""          // restore normal frame title
```

DoWindow/S

`DoWindow /N/S=`*`styleMacroName windowName`*

`DoWindow /R/S=`*`styleMacroName windowName`*

The DoWindow/S operation creates a new "style macro" for the named window, using the specified style macro name. Does not create or replace the window macro for the specified window.

Flags

/N/S=*styleMacroName*	Creates a new style macro with the given name based on the named window.
/R/S=*styleMacroName*	Creates or replaces the style macro with the given name based on the named window.

Details

The /R or /N flag must appear before the /S flag.

If the /S flag is present, the DoWindow operations does *not* create or replace the window macro for the specified window.

The /R and /N flags do nothing when executed while a macro or function is running. This is necessary because changing procedures while they are executing causes unpredictable and undesirable results.

DoXOPIdle

`DoXOPIdle`

The DoXOPIdle operation sends an IDLE event to all open XOPs. This operation is very specialized. Generally, only the author of an XOP will need to use this operation.

Details

Some XOPs (External OPeration code modules) require IDLE events to perform certain tasks.

Igor does not automatically send IDLE events to XOPs while an Igor program is running. You can call DoXOPIdle from a user-defined program to force Igor to send the event.

DPSS

`DPSS [`*`flags`*`]` ***`numPoints, numWindows`***

The DPSS operation generates Slepian's Discrete Prolate Spheroidal Sequences.

The DPSS operation was added in Igor Pro 7.00.

Flags

/DEST=*destWave*	Saves the DPSS in a wave specified by *destWave*. The destination wave is overwritten if it exists. Creates a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-66 for details. If you omit /DEST the operation saves the result in the wave M_DPSS in the current data folder.
/EV=*evWave*	Saves the first numWindows eigenvalues in a wave specified by *evWave*. The eigenvalues are computed for a symmetric tridiagonal matrix. They are real, positive and close to 1. They can be used to estimate bias in multitaper calculations.
/FREE	Creates output waves as free waves. /FREE is permitted in user-defined functions only, not from the command line or in macros. If you use /FREE then *destWave*, *evWave* and *sumsWave* must be simple names, not paths. See **Free Waves** on page IV-84 for details on free waves.
/NW=nw	Specifies the time-bandwidth product. This value should typically be in the range [2,6]. Given a time-bandwidth product nw it is recommended to use no more than 2*nw tapers in order to maximize variance efficiency. The default value of the time-bandwidth product is 3.
/DTPS=*sumsWave*	Saves the sums of the generated DPSS windows in a wave specified by *sumsWave*.
/Q	Suppress printing information in the history.
/Z	Suppress errors. The variable V_Flag is set to 0 if successful and to -1 otherwise.

Details

DPSS generates Slepian's Discrete Prolate Spheroidal Sequences in a 2D double-precision wave of dimensions *numPoints* by *numWindows.*

If you do omit /DEST the operation creates the output wave M_DPSS in the current data folder. The sequences/tapers are arranged as columns in the output wave.

Examples

```
DPSS/DEST=dpss5 1000,5
Display dpss5[][0],dpss5[][1],dpss5[][2],dpss5[][3],dpss5[][4]
ModifyGraph rgb(dpss5#1)=(0,65535,0),rgb(dpss5#2)=(1,16019,65535)
ModifyGraph rgb(dpss5#3)=(65535,0,52428),rgb(dpss5#4)=(0,0,0)

// Different sequences are orthogonal
```

```
MatrixOp/o aa=col(dpss5,1)*col(dpss5,4)
Integrate/METH=1 aa/D=W_INT
Print W_INT[numpnts(W_INT)-1]
```

See Also

MultiTaperPSD, **WindowFunction**, **ImageWindow**, **Hanning**

References

D. Slepian, "Prolate spheroidal wave functions, Fourier analysis and uncertainty -- V: The discrete case.", Bell Syst. Tech J., vol 57 pp. 1317-1430, May 1978.

DrawAction

`DrawAction` `[`**`/L=`*`layerName`*`/W=`*`winName`***`]` ***`keyword = value`*** `[`***`, keyword = value`*** `…]`

The DrawAction operation deletes, inserts, and reads back a named drawing object group or the entire draw layer.

Parameters

DrawAction accepts multiple *keyword* = *value* parameters on one line.

beginInsert [=*index*]	Inserts draw commands before or at *index* position or at position specified by getgroup or delete parameters; position otherwise is zero.
commands [=*start*,*stop*]	Stores commands in S_recreation for draw objects between *start* and *stop* index values, range defined by getgroup, or entire layer otherwise.
delete [=*start*,*stop*]	Deletes draw objects between *start* and *stop* index values, range defined by getgroup, or entire layer otherwise.
extractOutline [=*start*,*stop*]	Stores polygon outline between *start* and *stop* index values, range defined by getgroup, or entire layer otherwise. Waves W_PolyX and W_PolyY contain coordinates with NaN separators. V_npnts contains the number of objects. Coordinates are for the first object encountered.
endInsert	Terminates insert mode.
getgroup=*name*	Stores first and last index of named group in V_startPos and V_endPos. Use `_all_` to specify the entire layer. Sets V_flag to truth group exists.

Flags

/L=*layerName*	Specifies the drawing layer on which to act. *layerName* is one of the drawing layers as specified in **SetDrawLayer**.
/W=*winName*	Sets the named window or subwindow for drawing. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

Commands stored in S_recreation are the same as those that would be generated for the range of objects in the recreation macro for the window but also have comment lines preceding each object of the form:

`// ;ITEMNO:`*`n`*`;`

where *n* is the item number of the draw object.

Examples

Create a drawing with a named group:

```
NewPanel /W=(455,124,936,413)
SetDrawEnv fillfgc= (65535,0,0)
DrawRect 58,45,132,103
SetDrawEnv gstart,gname= fred
SetDrawEnv fillfgc= (65535,43690,0)
DrawRect 79,62,154,120
```

```
SetDrawEnv arrow= 1
DrawLine 139,70,219,70
SetDrawEnv gstop
SetDrawEnv fillfgc= (0,65535,65535)
DrawRect 95,77,175,138
SetDrawEnv fillfgc= (0,0,65535)
DrawRect 111,91,191,156
```

Get and print commands for the “fred” group:

```
DrawAction getgroup=fred,commands
Print S_recreation
```

prints:

```
// ;ITEMNO:3;
SetDrawEnv gstart,gname= fred
// ;ITEMNO:4;
SetDrawEnv fillfgc= (65535,43690,0)
// ;ITEMNO:5;
DrawRect 79,62,154,120
// ;ITEMNO:6;
SetDrawEnv arrow= 1
// ;ITEMNO:7;
DrawLine 139,70,219,70
// ;ITEMNO:8;
SetDrawEnv gstop
```

Replace group fred (the orange rectangle and the arrow) with a different object. First delete the group and enter insert mode:

```
DrawAction getgroup=fred, delete, begininsert
```

Next draw the replacement:

```
SetDrawEnv gstart,gname= fred
SetDrawEnv fillfgc= (65535,65535,0)
DrawOval 82,62,161,123
SetDrawEnv gstop
```

Lastly exit insert mode:

```
DrawAction endinsert
```

See Also

The **SetDrawEnv** operation and Chapter III-3, **Drawing**.

DrawArc

DrawArc [**/W=*winName*/X/Y**] ***xOrg*, *yOrg*, *arcRadius*, *startAngle*, *stopAngle***

The DrawArc operation draws a circular counterclockwise arc with center at *xOrg* and *yOrg*.

Parameters

(*xOrg*, *yOrg*) defines the center point for the arc in the currently active coordinate system.

Angles are measured in degrees increasing in a counterclockwise direction. The *startAngle* specifies the starting angle for the arc and *stopAngle* specifies the end. If *stopAngle* is equal to *startAngle*, 360° is added to *stopAngle*. Thus, a circle can be drawn using *startAngle* = *stopAngle*.

The *arcRadius* is the radial distance measured in points from (*xOrg*, *yOrg*).

Flags

/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/X	Measures *arcRadius* using the current X-coordinate system. If /Y is also used, the arc may be elliptical.
/Y	Measures *arcRadius* using the current Y-coordinate system. If /X is also used, the arc may be elliptical.

Details

Arcs honor the current dash pattern and arrowhead setting in the same way as polygons and Beziers. In fact, arcs are implemented using Bezier curves.

Normally, you would create arcs programmatically. If you need to sketch an arc-like object, you should probably use a Bezier curve because it is more flexible and easier to adjust. However, there is one handy feature of arcs that make them useful for manual drawing: the origin can be in any of the supported coordinate systems and the radius is in points.

To draw an arc interactively, see **Arcs and Circles** on page III-63 for instructions.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv** and **SetDrawLayer** operations.

The **DrawBezier**, **DrawOval** and **DrawAction** operations.

DrawBezier

DrawBezier [**/W=*winName* /ABS**] ***xOrg*, *yOrg*, *hScaling*, *vScaling*, {x_0,y_0,x_1,y_1 …}**

DrawBezier [**/W=*winName* /ABS**] ***xOrg*, *yOrg*, *hScaling*, *vScaling*, *xWaveName*, *yWaveName***

DrawBezier/A [**/W=*winName***] **{x_n, y_n, x_{n+1}, y_{n+1} …}**

The DrawBezier operation draws a Bezier curve with first point of the curve positioned at *xOrg* and *yOrg*.

Parameters

(*xOrg*, *yOrg*) defines the starting point for the Bezier curve in the currently active coordinate system.

hScaling and *vScaling* set the horizontal and vertical scale factors about the origin, with 1 meaning 100%.

The *xWaveName*, *yWaveName* version of DrawBezier gets data from the named X and Y waves. This connection is maintained so that any changes to either wave will result in updates to the Bezier curve.

To use the version of DrawBezier that takes a literal list of vertices, you place as many vertices as you like on the first line and then use as many /A versions as necessary to define all the vertices.

Flags

/A	Appends the given vertices to the currently open Bezier (freshly drawn or current selection).
/ABS	Suppresses the default subtraction of the first point from the rest of the data.
/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

Data waves defining Bezier curves must have 1+3**n* data points. Every third data point is an anchor point and lies on the curve; intervening points are control points that define the direction of the curve relative to the adjacent anchor point.

Normally, you should create and edit a Bezier curve using drawing tools, and not calculate values. See **Polygon Tool** on page III-63 for instructions.

You can include the /ABS flag to suppress the default subtraction of the first point. Also, you can now insert NaN values to break a bezier into pieces.

To change just the origin and scale without respecifying the data use:

```
DrawBezier xOrg, yOrg, hScaling, vScaling,{}
```

To delete an anchor point, press Option (*Macintosh*) or Alt (*Windows*) and then click on the anchor. To insert a new anchor, click on the curve between anchor points.

Example

Create a half circle approximation from three anchor points starting at the 12 o'clock position, with an anchor at the 3 o'clock position, and the last at the 6 o'clock position using explicit values:

```
// Set plot relative coords, 0-1, no fill
SetDrawEnv xcoord=prel, ycoord=prel, fillpat= 0, save

Variable len= 0.275        // control point length = 0.55 * radius for a circle
// Starting anchor point has only a trailing control point
Variable anchor0x= 0.5, anchor0y=1          // starting point at 6 o'clock
Variable t0x= 0.5+len, t0y= 1               // trailing control point

// second anchor point has both leading and trailing control points
Variable l1x=1, l1y = 0.5+len               // leading control point
Variable anchor1x= 1, anchor1y= 0.5         // 3 o'clock anchor
Variable t1x=1, t1y = 0.5-len               // trailing control point

// Last (3rd) anchor point has only a leading control point
Variable l2x=0.5+len, l2y = 0               // leading control point
Variable anchor2x= 0.5, anchor2y= 0         // 6 o'clock

// One command per anchor for clarity
DrawBezier anchor0x, anchor0y, 1,1, {anchor0x, anchor0y, t0x, t0y}
DrawBezier/A {l1x, l1y, anchor1x, anchor1y, t1x, t1y}
DrawBezier/A {l2x, l2y, anchor2x, anchor2y}
```

To draw using waves:

```
Make/O bezierX= {anchor0x, t0x, l1x, anchor1x, t1x, l2x, anchor2x }
Make/O bezierY= {anchor0y, t0y, l1y, anchor1y, t1y, l2y, anchor2y }
DrawBezier anchor0x, anchor0y, 1,1, bezierX, bezierY
```

See Also

Chapter III-3, **Drawing**.

Polygon Tool on page III-63 for discussion on creating Beziers. **DrawPoly and DrawBezier Operations** on page III-71 and the **SetDrawEnv** and **SetDrawLayer** operations.

The **DrawArc**, **DrawPoly** and **DrawAction** operations.

DrawLine

DrawLine [**/W=*winName***] ***x0*, *y0*, *x1*, *y1***

The DrawLine operation draws a line in the target graph, layout or control panel from (*x0*,*y0*) to (*x1*,*y1*).

Flags

/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The coordinate system as well as the line's thickness, color, dash pattern and other properties are determined by the current drawing environment. The line is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawOval

`DrawOval` `[`**`/W=`*`winName`***`]` ***`left, top, right, bottom`***

The DrawOval operation draws an oval in the target graph, layout or control panel within the rectangle defined by *left*, *top*, *right,* and *bottom*.

Flags

/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The coordinate system as well as the oval's thickness, color, dash pattern and other properties are determined by the current drawing environment (note that you cannot draw dashed ovals). The oval is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawPICT

`DrawPICT` `[`**`/W=`*`winName`***`]``[`**`/RABS`**`]` ***`left, top, hScaling, vScaling, pictName`***

The DrawPICT operation draws the named picture in the target graph, layout or control panel. The *left* and *top* parameters set the position of the top/left corner of the picture. *hScaling* and *vScaling* set the horizontal and vertical scale factors with 1 meaning 100%.

Flags

/RABS	Draws the named picture using absolute scaling. In this mode, it draws the picture in the rectangle defined by *left* and *top* for point (x0,y0), and by *hScaling* and *vScaling* for point (x1,y1), respectively.
/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The coordinate system for the left and top parameters is determined by the current drawing environment. The PICT is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawPoly

`DrawPoly` `[`**`/W=`*`winName`*` /ABS`**`]` ***`xorg, yorg, hScaling, vScaling, xWaveName, yWaveName`***

`DrawPoly` `[`**`/W=`*`winName`*` /ABS`**`]` ***`xorg, yorg, hScaling, vScaling,`*** **{**x_0, y_0, x_1, y_1 …**}**

`DrawPoly/A` `[`**`/W=`*`winName`***`]` **{**x_n, y_n, x_{n+1}, y_{n+1} …**}**

The DrawPoly operation draws a polygon in the target graph, layout or control panel.

Parameters

(*xorg*, *yorg*) defines the starting point for the polygon in the currently active coordinate system.

hScaling and *vScaling* set the horizontal and vertical scale factors, with 1 meaning 100%.

The *xWaveName*, *yWaveName* version of DrawPoly gets data from those X and Y waves. This connection is maintained so that changes to either wave will update the polygon.

The DrawPoly operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

To use the version of DrawPoly that takes a literal list of vertices, you place as many vertices as you like on the first line and then use as many /A versions as necessary to define all the vertices.

Flags

/A	Appends the given vertices to the currently open polygon (freshly drawn or current selection).
/ABS	Suppresses the default subtraction of the first point from the rest of the data.
/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

Because *xorg* and *yorg* define the location of the starting vertex of the poly, adding or subtracting a constant from the vertices will have no effect. The first XY pair in the {*x0, y0, x1, y1,…*} vertex list will appear at (*xorg,yorg*) *regardless* of the value of *x0* and *y0*. *x0* and *y0* merely serve to set a reference point for the list of vertices. Subsequent vertices are relative to (*x0,y0*).

To keep your mental health intact, we recommend that you specify (*x0,y0*) as (0,0) so that all the following vertices are offsets from that origin. Then (*xorg,yorg*) sets the position of the polygon and all of the vertices in the list are relative to that origin.

An alternate method is to use the same values for (*x0,y0*) as for (*xorg,yorg*) if you consider the vertices to be “absolute” coordinates.

You can include the /ABS flag to suppress the subtraction of the first point. Also, you can now insert NaN values to break a polygon into pieces.

To change just the origin and scale of the currently open polygon — without having to respecify the data — use:

```
DrawPoly xorg, yorg, hScaling, vScaling,{}
```

The coordinate system as well as the polygon’s thickness, color, dash pattern and other properties are determined by the current drawing environment. The polygon is drawn in the current draw layer for the window, as determined by SetDrawLayer.

Examples

Here are some commands to draw some small triangles using absolute drawing coordinates (see **SetDrawEnv**).

```
Display              // make a new empty graph
//Draw one triangle, starting at 50,50 at 100% scaling
SetDrawEnv xcoord= abs,ycoord= abs
DrawPoly 50,50,1,1, {0,0,10,10,-10,10,0,0}
//Draw second triangle below and to the right, same size and shape
SetDrawEnv xcoord= abs,ycoord= abs
DrawPoly 100,100,1,1, {0,0,10,10,-10,10,0,0}
```

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv** and **SetDrawLayer** operations, **DrawPoly and DrawBezier Operations** on page III-71, and Chapter III-3, **Drawing** and **DrawAction**.

DrawRect

`DrawRect` `[`**`/W=`*`winName`***`]` ***`left, top, right, bottom`***

The DrawRect operation draws a rectangle in the target graph, layout or control panel within the rectangle defined by *left*, *top*, *right*, and *bottom*.

Flags

/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The coordinate system as well as the rectangle's thickness, color, dash pattern and other properties are determined by the current drawing environment. The rectangle is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawRRect

`DrawRRect` `[`**`/W=`*`winName`***`]` ***`left, top, right, bottom`***

The DrawRRect operation draws a rounded rectangle in the target graph, layout or control panel within the rectangle defined by *left*, *top*, *right*, and *bottom*.

Flags

/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The coordinate system as well as the rectangle's rounding, thickness, color, dash pattern and other properties are determined by the current drawing environment. The rounded rectangle is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawText

`DrawText` `[`**`/W=`*`winName`***`]` x_0, y_0, ***`textStr`***

The DrawText operation draws the specified text in the target graph, layout or control panel. The position of the text is determined by (*x0*, *y0*) along with the current textxjust, textyjust and textrot settings as set by **SetDrawEnv**.

Flags

/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The coordinate system as well as the text's font, size, style and other properties are determined by the current drawing environment. The text is drawn in the current draw layer for the window, as determined by SetDrawLayer.

See Also

Chapter III-3, **Drawing**.

The **SetDrawEnv**, **SetDrawLayer** and **DrawAction** operations.

DrawUserShape

`DrawUserShape [/W=`*`winName`*` /MO=`*`options`*`]` *`x0, y0, x1, y1, userFuncName, textString, privateString`*

The DrawUserShape operation is similar to built-in drawing operations except the shape is defined by a user-defined function which is executed when the shape is drawn.

DrawUserShape was added in Igor Pro 7.00.

Parameters

For rectangular shapes (*x0*,*y0*) defines the top left corner while (*x1*,*y1*) defines the lower right corner in the currently active coordinate system. For line-like shapes, (*x0*,*y0*) specifies the start of the line while(*x1*,*y1*) specifies the end.

userFuncName specifies your user-defined function that uses built-in drawing operations to define the shape and provides information about it to Igor.

textString specifies text to be drawn over the shape by Igor. Pass "" if you do not need text. Using escape codes you can change the font, size, style, and color of the text. See **Annotation Escape Codes** on page III-53 or details.

privateString is text or binary data used by the user-defined function for any purpose. It is typically used to maintain state information and is saved in recreation macros using a method that supports binary. Pass "" if you do not need this.

To support modifying an existing shape, each of the last three parameters above can be `_NoChange_`. The *x0*, *y0*, *x1*, *y1* parameters can be all zero for no change. Using `_NoChange_` when there is no existing shape is an error. To target an existing shape, it must be selected by the drawing tools.

Flags

/W=*winName*	Draws to the named window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/M=*options*	An integer that Igor passes to your user-defined drawing function for use for any purpose.

Details

The user-defined function must have the following form and structure parameter.

```
Function MyUserShape(s) : DrawUserShape // Optional ": DrawUserShape" indicates
                                        // that this function should be added
   STRUCT WMDrawUserShapeStruct &s      // to the menu of shapes in the drawing palette

   Variable returnValue= 1
```

```
	StrSwitch(s.action)
		case "draw":
			Print "Put DrawXXX commands here"
			break
		case "drawSelected":
			Print "Put Draw commands when selected (may be same as normalDraw) here"
			break
		case "hitTest":
			Print "Put code to test if mouse is over a control point here"
			// Set doSetCursor and optionally cursorCode if so"
			// Set returnValue to zero if not over control point or, if in operate
			// mode and you don't want to start a button operation.
			break
		case "mouseDown":
			if( s.operateMode )
				Print "Mouse down in button mode."
			else
				// Code similar or same as hitTest.
				Print "User is starting a drag of a control point."
			endif
			// Set returnValue to zero if if no action or redraw needed
			break
		case "mouseMove":
			if(s.operateMode)
				Print "Roll-over mode and mouse is inside shape."
			else
				Print "Mouse has been captured and user is dragging a control point."
			endif
			// Set returnValue to zero if if no action or redraw needed
			break
		case "mouseUp":
			if(s.operateMode)
				Print "Mouse button released."
			else
				Print "User finished dragging control point."
			endif
			break
		case "getInfo":
			Print "Igor is requesting info about this shape."
			s.textString= "category for shape"
			s.privateString= "shape name"
			s.options= 0// Or set bits 0, 1 and/or 2
			break
	EndSwitch

	return returnValue
End
```

WMDrawUserShapeStruct is a built-in structure with the following members:

`WMDrawUserShapeStruct` Structure Members

Member	Description
`char action[32]`	Input: Specifies what action is requested.
`SInt32 options`	Input: Value from /MO flag.
	Output: When action is `getInfo`, set bits as follows:
	Set bit 0 if the shape should behave like a simple line. When resizing end-points, you will get live updates.
	Set bit 1 if the shape is to act like a button; you will get mouse down in normal operate mode.
	Set bit 2 to get roll-over action. You will get `hitTest` action and if 1 is returned, the mouse will be captured.
`SInt32 operateMode`	Input: If 0, the shape is being edited; if 1, normal operate mode (only if options bit 1 or 2 was set during `getInfo`).
`PointF mouseLoc`	Input: The location of the mouse in normalized coordinates.

`WMDrawUserShapeStruct` Structure Members

Member	Description
`SInt32 doSetCursor`	Output: If action is `hitTest`, set true to use the following cursor number. Also used for `mouseMoved` in rollover mode.
`SInt32 cursorCode`	Output: If action is `hitTest` and `doSetCursor` is set, then set this to the desired Igor cursor number.
`double x0,y0,x1,y1`	Input: Coordinates of the enclosing rectangle of the shape.
`RectF objectR`	Input: Coordinates of the enclosing rectangle of the shape in device units.
`char winName[MAX_HostChildSpec+1]`	Full path to host subwindow
`// Information about the coordinate system`	
`Rect drawRect`	Draw rect in device coordinates
`Rect plotRect`	In a graph, this is the plot area
`char xcName[MAX_OBJ_NAME+1]`	Name of X coordinate system, may be axis name
`char ycName[MAX_OBJ_NAME+1]`	Name of Y coordinate system, may be axis name
`double angle`	Input: Rotation angle, use when displaying text
`String textString`	Input: Use or ignore; special output for `getInfo`
`String privateString`	Input and output: Maintained by Igor but defined by user function; may be binary; special output for `getInfo`

The constants used to size the char arrays, `MAX_HostChildSpec` and `MAX_OBJ_NAME`, are defined internally in Igor and are subject to change in future versions.

The drawing commands you provide for the `draw` and `drawSelected` actions must use normalized coordinates ranging from (0,0) to (1,1). For example, to draw a rectangle you would use

```
DrawRect 0,0,1,1
```

Generally, you will not set drawing environment parameters such as color or fill mode but will leave that to the user of your shape just as they would for built-in shapes such as a Rectangle.

You will get the `drawSelected` action when the user has selected your shape with the arrow draw tool. For simple shapes, this can use the same code as the draw action but if you want the shape to be editable, you can draw control points such as a yellow dot. You would then provide code in the `hitTest` and `mouseDown` actions that determines if the user has moved over or clicked on a control point. For an example of this, see the FatArrows procedure file in the User Draw Shapes demo experiment.

Your function should respond to the `getInfo` action by setting the textString field to a category name and the privateString field to a shape name. A category name might be "Arrows" and a shape name might be "Right Arrow". These names are used to populate the menus you get when right-clicking the user shapes icon in the drawing tools palette. You can also set bits in the options field to enable certain special actions. For examples of these, see the LineCallout and button procedure windows in the User Draw Shapes demo experiment.

Note: To support button-like shapes in graphs and layouts, an additional drawing layer named Overlay was added in Igor Pro 7.00. This layer is drawn over the top of everything else and is not printed or exported. Objects in the Overlay draw layer can update without the need to redraw the entire window as would be the case if one of the previously existing layers were used. For consistency, this layer is also available in control panels.

See Also

Chapter III-3, **Drawing**.

SetDrawEnv, **SetDrawLayer**, **DrawBezier**, **DrawPoly**, **DrawAction**

DSPDetrend

`DSPDetrend [`*`flags`*`] `*`srcWave`*

The DSPDetrend operation removes from *srcWave* a trend defined by the best fit of the specified function to the data in *srcWave*.

Flags

/A	Subtracts the average of *srcWave* before performing any fitting. Added in Igor Pro 7.00.
/F= *function*	*function* is the name of a built-in curve fitting function: gauss, lor, exp, dblexp, sin, line, poly (requires /P flag), hillEquation, sigmoid, power, lognormal, poly2d (requires /P flag), gauss2d. If *function* is unspecified, the defaults are line if *srcWave* is 1D or poly2d if *srcWave* is 2D.
/M=*maskWave*	Detrending will only affect points that are nonzero in *maskWave*. Note that *maskWave* must have the same dimensionality as *srcWave*.
/P=*n*	Specifies polynomial order for poly or poly2d functions (see **CurveFit** for details). When used with the 1D poly function *n* specifies the number of terms in the polynomial. By default *n*=3 for the 1D case and *n*=1 for poly2d.
/Q	Quiet mode; no error reporting.

Details

DSPDetrend sets V_flag to zero when the operation succeeds, otherwise it will be set to -1 or will contain an error code from the curve fitting routines. Results are saved in the wave W_Detrend (for 1D input) or M_Detrend (for 2D input) in the current data folder. If a wave by that name already exists in the current data folder it will be overwritten.

See Also

CurveFit for more information about V_FitQuitReason and the built-in fitting functions.

DSPPeriodogram

`DSPPeriodogram [`*`flags`*`] `*`srcWave`*` [`*`srcWave2`*`]`

The DSPPeriodogram operation calculates the periodogram, cross-spectral density or the degree of coherence of the input waves. The result of the operation is stored in the wave W_Periodogram in the current data folder.

To compute the cross-spectral density or the degree of coherence, you need to specify the second wave using the optional *srcWave2* parameter. In this case, W_Periodogram will be complex and the /dB and /dBR flags do not apply.

Flags

/dB	Expresses results in dB using the maximum value as reference.
/dBR=*ref*	Express the results in dB using the specified *ref* value.
/COHR	Computes the degree of coherence. This flag applies when the input consists of two waves.
/DLSG	When computing the periodogram, cross-spectral density or the degree of coherence using multiple segments the operation by default pads the last segment with zeros as necessary. If you specify this flag, an incomplete last segment is dropped and not included in the calculation.

/NODC=*val*	Suppresses the DC term:
	val=1: Removes the DC by subtracting the average value of the signal before processing and before applying any window function (see /Win below).
	val=2: Suppresses the DC term by setting it equal to the second term in the FFT array.
	val=0: Computes the DC term using the FFT (default).
/NOR=*N*	Sets the normalization, N, in the periodogram equation. By default, it is the number of data points times the square norm of the window function (if any).
	N=0 or 1: Skips default normalization.
	Any other value of *N* is used as the only normalization.
/Q	Quiet mode; suppresses printing in the history area.
/SEGN={*ptsPerSegment*, *overlapPts*}	
	Use this flag to compute the periodogram, cross-spectral density or degree of coherence by averaging over multiple segments taken from the input waves. The size of each interval is *ptsPerSegment*. *overlapPts* determines the number of points at the end of each interval that are included in the next segment.
/R=[*startPt*, *endPt*]	Calculates the periodogram for a limited range of the wave. *startPt* and *endPt* are expressed in terms of point numbers in *srcWave*.
/R=(*startX*, *endX*)	Calculates the periodogram for a limited range of the wave. *startX* and *endX* are expressed in terms of x-values. Note that this option will convert your x-specifications to point numbers and some roundoff may occur.
/WIN=*windowKind*	Specifies the window type. If you omit the /W flag, DSPPeriodogram uses a rectangular window for the full wave or the range of data selected by the /R flag.
	Choices for *windowKind* are:
	Bartlett, Blackman367, Blackman361, Blackman492, Blackman474, Cos1, Cos2, Cos3, Cos4, Hamming, Hanning, KaiserBessel20, KaiserBessel25, KaiserBessel30, Parzen, Poisson2, Poisson3, Poisson4, and Riemann.
	See **FFT** for window equations and details.
/Z	Do not report errors. When an error occurs, V_flag is set to -1.

Details

The default periodogram is defined as

$$Periodogram = \frac{|F(s)|^2}{N},$$

where F (s) is the Fourier transform of the signal s and N is the number of points.

In most practical situations you need to account for using a window function (when computing the Fourier transform) which takes the form

$$Periodogram = \frac{|F(s \cdot w)|^2}{N_p N_w},$$

where w is the window function, N_p is the number of points and N_w is the normalization of the window function.

If you compute the periodogram by subdividing the signal into multiple segments (with any overlap) and averaging the results over all segments, the expression for the periodogram is

$$Periodogram = \frac{\sum_{i=1}^{M} \left|F(s_i \cdot w)\right|^2}{MN_s N_w},$$

where si is the ith segment s, Ns is the number of points per segment and M is the number of segments.

When calculating the cross-spectral density (csd) of two waves s_1 and s_2, the operation results in a complex valued wave

$$csd = \frac{F(s_A)[F(s_B)]^*}{N},$$

which contains the normalized product of the Fourier transform of the first wave S_A with the complex conjugate of the Fourier transform of the second wave S_B. The extension of the csd calculation to segment averaging has the form

$$csd = \frac{\sum_{i=0}^{M} F(s_{Ai})[F(s_{Bi})]^*}{MN_s N_w},$$

where S_{Ai} is the ith segment of the first wave, M is the number of segments and N_s is the number of points in a segment.

The degree of coherence is a normalized version of the cross-spectral density. It is given by

$$\gamma = \frac{\sum_{i=0}^{M} F(s_{Ai})[F(s_{Bi})]^*}{\sqrt{\sum_{i=0}^{M} F(s_{Ai})[F(s_{Ai})]^* \sum_{i=0}^{M} F(s_{Bi})[F(s_{Bi})]^*}}.$$

The bias in the degree of coherence is calculated using the approximation

$$B = \frac{1}{M}\left[1 - |\gamma|^2\right]^2.$$

The bias is stored in the wave W_Bias.

If you use the /SEGN flag the actual number of segments is reported in the variable V_numSegments.

Note that DSPPeriodogram does not test the dimensionality of the wave; it treats the wave as 1D. When you compute the cross-spectral density or the degree of coherence the number-type, dimensionality and the scaling of the two waves must agree.

See Also

The **ImageWindow** operation for 2D windowing applications. **FFT** for window equations and details.

The **Hanning**, **LombPeriodogram** and **MatrixOp** operations.

References

For more information about the use of window functions see:

Harris, F.J., On the use of windows for harmonic analysis with the discrete Fourier Transform, *Proc, IEEE, 66*, 51-83, 1978.

G.C. Carter, C.H. Knapp and A.H. Nuttall, The Estimation of the Magnitude-squared Coherence Function Via Overlapped Fast Fourier Transform Processing, *IEEE Trans. Audio and Electroacoustics*, V. AU-21, (4) 1973.

Duplicate

`Duplicate [`*`flags`*`][`*`type flags`*`] `*`srcWaveName, destWaveName`*` [, `*`destWaveName`*`]…`

The Duplicate operation creates new waves, the names of which are specified by *destWaveName*s and the contents, data type and scaling of which are identical to *srcWaveName*.

Parameters

srcWaveName must be the name of an existing wave.

The *destWaveName*s should be wave names not currently in use unless the /O flag is used to overwrite existing waves.

Flags

/FREE	Creates a free wave (see **Free Waves** on page IV-84). /FREE is allowed only in functions and only if a simple name or wave reference structure field is specified as the destination wave name.
/O	Overwrites existing waves with the same name as *destWaveName*.
/R=(*startX*,*endX*)	Specifies an X range in the source wave from which the destination wave is created. See Details for further discussion of /R.
/R=(*startX*,*endX*)(*startY*,*endY*)	Specifies both X and Y range. Further dimensions are constructed analogously. See Details for further discussion of /R.
/R=[*startP*,*endP*]	Specifies a row range in the source wave from which the destination wave is created. Further dimensions are constructed just like the scaled dimension ranges. See Details for further discussion of /R.

/RMD=[*firstRow*,*lastRow*][*firstColumn*,*lastColumn*][*firstLayer*,*lastlayer*][*firstChunk*,*lastChunk*]	Designates a contiguous range of data in the source wave to which the operation is to be applied. This flag was added in Igor Pro 7.00.
	You can include all higher dimensions by leaving off the corresponding brackets. For example:
	`/RMD=[firstRow,lastRow]`
	includes all available columns, layers and chunks.
	You can use empty brackets to include all of a given dimension. For example:
	`/RMD=[][firstColumn,lastColumn]`
	means "all rows from column A to column B".
	You can use a * to specify the end of any dimension. For example:
	`/RMD=[firstRow,*]`
	means "from firstRow through the last row".

Type Flags *(used only in functions)*

When used in user-defined functions, Duplicate can also take the /B, /C, /D, /I, /S, /U, /W, /T, /DF and /WAVE flags. This does not affect the result of the Duplicate operation - these flags are used only to identify what kind of wave is expected at runtime.

This information is used if, later in the function, you create a wave assignment statement using a duplicated wave as the destination:

```
Function DupIt(wv)
   Wave/C wv                   //complex wave

   Duplicate/O/C wv,dupWv      //tell Igor that dupWv is complex
   dupWv[0]=cmplx(5.0,1.0)     //no error, because dupWv known complex
   …
```

If Duplicate did not have the /C flag, Igor would complain with a "function not available for this number type" message when it tried to compile the assignment of dupWv to the result of the cmplx function.

These type flags do not need to be used except when it needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-67 and **WAVE Reference Type Flags** on page IV-68 for a complete list of type flags and further details.

Details

If /R is omitted, the entire wave is duplicated.

In the range specifications used with /R, a `*` for the end means duplicate to the end. You can also simply leave out the end specification. To include all of a given dimension, use `/R=[]`. If you leave off higher dimensions, all those dimensions are duplicated. That is, `/R=[1,5]` for a 2D wave is equivalent to `/R=[1,5][]`.

The destination wave will always be unlocked, even if the source wave was locked.

Warning:

Under some circumstances, such as in loops in user-defined functions, Duplicate may exhibit undesired behavior. When you use

```
Duplicate/O srcWave, DestWaveName
```

in a user-defined function, it creates a local WAVE variable named *DestWaveName* at compile time. At runtime, if the WAVE variable is NULL, it creates a wave of the same name in the current data folder. If, however, the WAVE variable is not NULL, as it would be in a loop, then the referenced wave will be overwritten no matter where it is located. If the desired behavior is to create (or overwrite) a wave in the current data folder, you should use one of the following two methods:

```
Duplicate/O srcWave, $"DestWaveName"
WAVE DestWaveName   // only if you need to reference dest wave
```

or

```
Duplicate/O srcWave, DestWaveName
// then after you are finished using DestWaveName…
WAVE DestWaveName=$""
```

See Also

Rename, **Concatenate**, **SplitWave**

DuplicateDataFolder

`DuplicateDataFolder` ***`sourceDataFolderSpec, destDataFolderSpec`***

The DuplicateDataFolder operation makes a copy of the source data folder and everything in it and places the copy at the specified location with the specified name.

Parameters

sourceDataFolderSpec can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

destDataFolderSpec can be just a data folder name, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name. If just a data folder name is used then the new data folder is created in the current data folder.

Details

An error is issued if the destination is contained within the source data folder.

Examples

Create a copy of foo named foo2 in bar:

```
DuplicateDataFolder foo,:bar:foo2
```

See Also

See the **MoveDataFolder** operation. Chapter II-8, **Data Folders**.

DWT

`DWT [`*`flags`*`] `*`srcWaveName, destWaveName`*

The DWT operation performs discrete wavelet transform on the input wave *srcWaveName*. The operation works on one or more dimensions only as long as the number of elements in each dimension is a power of 2 or when the /P flag is specified

Flags

/D	Denoises the source wave. Performs the specified wavelet transform in the forward direction. It then zeros all transform coefficients whose magnitude fall below a given percentage (specified by the /V flag) of the maximum magnitude of the transform. It then performs the inverse transform placing the result in *destWaveName*. The /I flag is incompatible with the /D flag.
/I	Perform the inverse wavelet transform. The /S and /D flags are incompatible with the /I flag.
/N=*num*	Specifies the number of wavelet coefficients. See /T flag for supported combinations.
/P=*num*	Controls padding: *num*=1: Adds zero padding to the end of the dimension up to nearest power of 2 when the number of data elements in a given dimension of *srcWaveName* is not a power of 2. *num*=2: Uses zero padding to compute the transform, but the resulting wave is truncated to the length of the input wave.
/S	Smooths the source wave. This performs the specified wavelet transform in the forward direction. It then zeros all transform coefficients except those between 0 and the cut-off value (specified in % by /V flag). It then performs the inverse transform placing the result in *destWaveName*. The /I flag is incompatible with the /S flag.
/T=*type*	Performs the wavelet transform specified by *type*. The following table gives the transform name with the *type* code for the transform and the allowed values of the *num* parameter used with the /N flag. "NA" means that the /N flag is not applicable to the corresponding transform.

Wavelet Transform	***type***	***num***
Daubechies	1 (default)	4, 6, 8, 10, 12, 20
Haar	2	NA
Battle-Lemarie	4	NA
Burt-Adelson	8	NA
Coifman	16	2, 4, 6
Pseudo-Coifman	32	NA
splines	64	1 (2-2), 2 (2-4), 3 (3-3), 4 (3-7)

/V=*value*	Specifies the degree of smoothing with the /S and /D flags only. For /S, *value* gives the cutoff as a percentage of data points above which coefficients are set to zero. For /D, *value* specifies the percentage of the maximum magnitude of the transform such that coefficients smaller than this value are set to zero.

Details

If *destWaveName* exists, DWT overwrites it; if it does not exist, DWT creates it.

When used in a function, the DWT operation automatically creates a wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-66 for details.

If *destWaveName* is not specified, the DWT operation stores the results in W_DWT for 1D waves and M_DWT for higher dimensions.

When working with 1D waves, the transform results are packed such that the higher half of each array contains the detail components and the lower half contains the smooth components and each successive scale is packed in the lower elements. For example, if the source wave contains 128 points then the lowest scale results are stored in elements 64-127, the next scale (power of 2) are stored from 32-63, the following scale from 16-31 etc.

Example

```
Make/O/N=1024 testData=sin(x/100)+gnoise(0.05)
DWT /S/N=20/V=25 testData, smoothedData
```

See Also

For continuous wavelet transforms use the **CWT** operation. See the **FFT** operation.

For further discussion and examples see **Discrete Wavelet Transform** on page III-252.

e

`e`

The e function returns the base of the natural logarithm system (2.7182818…).

EdgeStats

`EdgeStats [`*`flags`*`]` ***`waveName`***

The EdgeStats operation produces simple statistics on a region of a wave that is expected to contain a single edge. If more than one edge exists, EdgeStats works on the first one found.

Flags

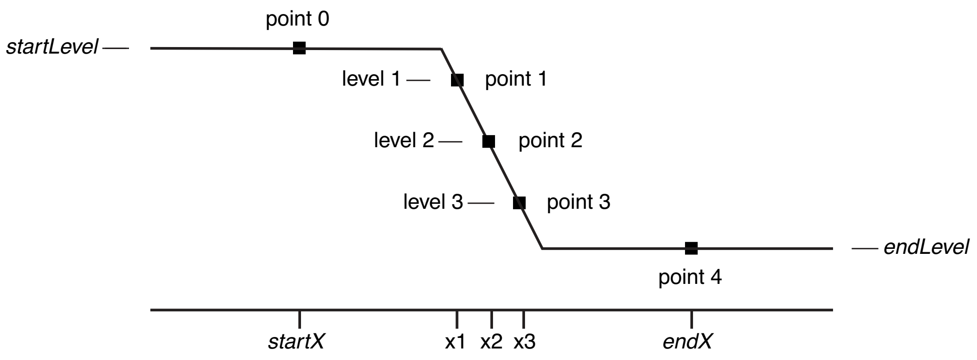
/A=*avgPts*	Determines *startLevel* and *endLevel* automatically by averaging *avgPts* points at centered at *startX* and *endX*. Default is /A=1.
/B=*box*	Sets box size for sliding average. This should be an odd number. If /B=*box* is omitted or *box* equals 1, no averaging is done.

/F=*frac*	Specifies levels 1, 2 and 3 as a fraction of (*endLevel*-*startLevel*):
	level1 = *frac** (*endLevel*-*startLevel*) + *startLevel*
	level2 = 0.5 * (*endLevel*-*startLevel*) + *startLevel*
	level3 = (1-*frac*) * (*endLevel*-*startLevel*) + *startLevel*
	The default value for *frac* is 0.1 which makes level1 the 10% level, level2 the 50% level and level3 the 90% level.
	frac must be between 0 and 0.5.

/L=(*startLevel*, *endLevel*)	Sets *startLevel* and *endLevel* explicitly. If omitted, they are determined automatically. See /A.
/P	Output edge locations (see **Details**) are returned as point numbers. If /P is omitted, edge locations are returned as X values.
/Q	Prevents results from being printed in history and prevents error if edge is not found.
/R=(*startX*,*endX*)	Specifies an X range of the wave to search. You may exchange *startX* and *endX* to reverse the search direction.
/R=[*startP*,*endP*]	Specifies a point range of the wave to search. You may exchange *startP* and *endP* to reverse the search direction. If /R is omitted, the entire wave is searched.
/T=*dx*	Forces search in two directions for a possibly more accurate result. *dx* controls where the second search starts.

Details

The /B=*box*, /T=*dx*, /P, and /Q flags behave the same as for the **FindLevel** operation.



EdgeStats considers a region of the input wave between two X locations, called *startX* and *endX*. *startX* and *endX* are set by the /R=(*startX*,*endX*) flag. If this flag is missing, *startX* and *endX* default to the start and end of the entire wave. *startX* can be greater than *endX* so that the search for an edge can proceed from the "right" to the "left".

The diagram above shows the default search direction, from the "left" (lower point numbers) of the wave toward the "right" (higher point numbers).

The *startLevel* and *endLevel* values define the base levels of the edge. You can explicitly set these levels with the /L=(*startLevel*, *endLevel*) flag or you can let EdgeStats find the base levels for you by using the /A=*avgPts* flag which averages points around *startX* and *endX*.

Given *startLevel* and *endLevel* and a *frac* value (see the /F=*frac* flag) EdgeStats defines level1, level2 and level3 as shown in the diagram above. With the default *frac* value of 0.1, level1 is the 10% point, level2 is the 50% point and level3 is the 90% point.

With these levels defined, EdgeStats searches the wave from *startX* to *endX* looking for level2. Having found it, it then searches for level1 and level3. It returns results via variables described below.

EdgeStats sets the following variables:

`V_flag`	0: All three level crossings were found. 1: One or two level crossings were found. 2: No level crossings were found.
`V_EdgeLoc1`	X location of level1.
`V_EdgeLoc2`	X location of level2.
`V_EdgeLoc3`	X location of level3.
`V_EdgeLvl0`	*startLevel* value.
`V_EdgeLvl1`	level1 value.
`V_EdgeLvl2`	level2 value.
`V_EdgeLvl3`	level3 value.
`V_EdgeLvl4`	*endLevel* value.
`V_EdgeAmp4_0`	Edge amplitude (*endLevel* - *startLevel*).
`V_EdgeDLoc3_1`	Edge width (x distance between point 1 and point 3).
`V_EdgeSlope3_1`	Edge slope (straight line slope from point 1 and point 3).

These X locations and distances are in terms of the X scaling of the named wave unless you use the /P flag, in which case they are in terms of point number.

The EdgeStats operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

See Also

The **FindLevel** operation for use of the /B=*box*, /T=*dx*, /P and /Q flags, and **PulseStats**.

Edit

`Edit [`*`flags`*`] [`*`columnSpec`*` [, `*`columnSpec`*`]…][as `*`titleStr`*`]`

The Edit operation creates a table window or subwindow containing the specified columns.

Parameters

columnSpec is usually just the name of a wave. If no *columnSpec*s are given, Edit creates an empty table.

Column specifications are wave names optionally followed by one of the suffixes:

Suffix	Meaning
.i	Index values.
.l	Dimension labels.
.d	Data values.
.id	Index and data values.
.ld	Dimension labels and data values.

If the wave is complex, the wave names may be followed by .real or .imag suffixes. However, as of Igor Pro 3.0, both the real and imaginary columns are added to the table together — you can not add one without the other — so using these suffixes is discouraged.

Historical Note: Prior to Igor Pro 3.0, only 1D waves were supported. We called index values "X values" and used the suffix ".x" instead of ".i". We called data values "Y values" and used the suffix ".y" instead of ".d". For backward compatibility, Igor accepts ".x" in place of ".i" and ".y" in place of ".d".

titleStr is a string expression containing the table's title. If not specified, Igor will provide one which identifies the columns displayed in the table.

Flags

/HIDE=*h* Hides (h = 1) or shows (h = 0, default) the window.

/HOST=*hcSpec* Embeds the new table in the host window or subwindow specified by *hcSpec*.

When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

/I Specifies that /W coordinates are in inches.

/K=*k* Specifies window behavior when the user attempts to close it.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.
k=3:	Hides the window.

If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.

/M Specifies that /W coordinates are in centimeters.

/N=*name* Requests that the created table have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen table name. Use `DoWindow/K` *`name`* to ensure that *name* is available.

/W=(*left*,*top*,*right*,*bottom*)

Gives the table a specific location and size on the screen. Coordinates for /W are in points unless /I or /M are specified before /W.

When used with the /HOST flag, the specified location coordinates of the sides can have one of two possible meanings:

When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.

When any value is greater than 1, coordinates are taken to be fixed locations measured in points relative to the top left corner of the host frame.

Details

You can not change dimension index values shown in a table. Use the Change Wave Scaling dialog or the **SetScale** operation.

If /N is not used, Edit automatically assigns to the table window a name of the form "Table*n*", where *n* is some integer. In a function or macro, the assigned name is stored in the S_name string. This is the name you can use to refer to the table from a procedure. Use the **RenameWindow** operation to rename the graph.

Examples

These examples assume that the waves are 1D.

```
Edit myWave,otherWave      // 2 columns: data values from each wave
Edit myWave.id             // 2 columns: x and data values
Edit cmplxWave             // 2 columns: real and imaginary data values
Edit cmplxWave.i           // One column: x values
```

The following examples illustrates the use of column name suffixes in procedures when the name of the wave is in a string variable.

```
Macro TestEdit()
   String w = "wave0"
   Edit $w                 // edit data values
   Edit $w.i               // show index values
   Edit $w.id              // index and data values
End
```

Note that the suffix, if any, must not be stored in the string. In a user-defined function, the syntax would be slightly different:

```
Function TestEditFunction()
   Wave w = $"wave0"
   Edit w                  // no $, because w is name, not string
   Edit w.i                // show index values
   Edit w.id               // index and data values
End
```

See Also

The **DoWindow** operation. For a description of how tables are used, see Chapter II-11, **Tables**.

ei

`ei(x)`

The ei function returns the value of the exponential integral *Ei*(*x*):

$$Ei(x) = \ P\int_{-\infty}^{x} \frac{e^t}{t} dt \qquad (x > 0),$$

where *P* denotes the principal value of the integral.

See Also

The **expInt** function.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 228 pp., Dover, New York, 1972.

End

`End`

The End keyword marks the end of a macro, user function, or user menu definition.

See Also

The **Function** and **Macro** keywords.

EndMacro

`EndMacro`

The EndMacro keyword marks the end of a macro. You can also use End to end a macro.

See Also

The **Macro** and **Window** keywords.

EndStructure

`EndStructure`

The EndStructure keyword marks the end of a Structure definition.

See Also

The **Structure** keyword.

endtry

`endtry`

The endtry flow control keyword marks the end of a try-catch-entry flow control construct.

See Also

The **try-catch-endtry** flow control statement for details.

enoise

`enoise(`*`num`*` [, `*`RNG`*`])`

The enoise function returns a random value drawn from a uniform distribution having a range of [-*num*, *num*).

The random number generator is initialized using the system clock when you start Igor, virtually guaranteeing that you will never repeat the same sequence. If you want repeatable "random" numbers, use **SetRandomSeed**.

The optional parameter *RNG* selects one of two different pseudo-random number generators. If omitted, the default is 1. The *RNG*'s are:

RNG	**Description**
1	Linear Congruential generator by L'Ecuyer with added Bayes-Durham shuffle. The algorithm is described in *Numerical Recipes* as the function `ran2()`. This option has nearly 23^2 distinct values and the sequence of random numbers has a period in excess of 10^{18}.
2	Mersenne Twister by Matsumoto and Nishimura. It is claimed to have better distribution properties and period of 2^{19937}-1.

In a complex expression, the enoise function returns a complex value, as if you had called:

```
cmplx(enoise(num), enoise(num))
```

Except for gnoise, other noise functions do not have complex implementations.

Example

```
// Generate uniformly-distributed integers on the interval [from,to] with from<to
Function IntNoise(from, to)
   Variable from, to
   Variable amp = to - from
   return floor(from + mod(abs(enoise(100*amp)),amp+1))
End
```

See Also

The **SetRandomSeed** operation and the **gnoise** function.

Noise Functions on page III-344.

References

Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

Details about the Mersene Twister are in:

Matsumoto, M., and T. Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Trans. on Modeling and Computer Simulation*, *8*, 3-30, 1998.

More information is available online at: <`http://en.wikipedia.org/wiki/Mersenne_twister`>

EqualWaves

`EqualWaves(`*`waveA, waveB, selector`*` [, `*`tolerance`*`])`

TheEqualWaves function compares *waveA* to *waveB*. Each wave can be of any data type. It returns 1 for equality and zero otherwise.

Use the *selector* parameter to determine which aspects of the wave are compared. You can add *selector* values to test more than one field at a time or pass -1 to compare all aspects.

selector	**Field Compared**
1	Wave data
2	Wave data type
4	Wave scaling
8	Data units
16	Dimension units
32	Dimension labels
64	Wave note
128	Wave lock state
256	Data full scale
512	Dimension sizes

If you use the selectors for wave data, wave scaling, dimension units, dimension labels or dimension sizes, EqualWaves will return zero if the waves have unequal dimension sizes. The other selectors do not require equal dimension sizes.

Details

If you are testing for equality of wave data and if the *tolerance* is specified, it must be a positive number. The function returns 1 for equality if the data satisfies:

$$\sum_i \left(waveA[i] - waveB[i]\right)^2 < tolerance.$$

If *tolerance* is not specified, it defaults to 10^{-8}.

If *tolerance* is set to zero and *selector* is set to 1 then the data in the two waves undergo a binary comparison (byte-by-byte).

If *tolerance* is non-zero then the presence of NaNs at a given point in both waves does not contribute to the sum shown in the equation above when both waves contain NaNs at the same point. A NaN entry that is present in only one of the waves is sufficient to flag inequality. Similarly, INF entries are excluded from the tolerance calculation when they appear in both waves at the same position and have the same signs.

If you are comparing wave data (selector =1) and both waves contain zero points the function returns 1.

See Also
The **MatrixOp** operation equal keyword.

erf

```
erf(num [, accuracy])
```

The erf function returns the error function of *num*.

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt.$$

Optionally, *accuracy* can be used to specify the desired fractional accuracy.

In complex expressions the error function is

$$erf(z) = \frac{2z}{\sqrt{\pi}}\, {}_1F_1\left(\frac{1}{2};\frac{3}{2};-z^2\right),$$

where

$${}_1F_1\left(\frac{1}{2};\frac{3}{2};-z^2\right)$$

is the confluent hypergeometric function of the first kind HyperG1F1. In this case the accuracy parameter is ignored.

Details
The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of (f_{actual} - $f_{returned}$)/f_{actual} be less than 10^{-7}.

For backwards compatibility, in the absence of *accuracy* an alternate calculation method is used that achieves fractional accuracy better than about 2x10^{-7}.

If *accuracy* is present, erf can achieve fractional accuracy better than 8x10^{-16} for *num* as small as 10^{-3}. For smaller *num* fractional accuracy is better than 5x10^{-15}.

Higher accuracy takes somewhat longer to calculate. With *accuracy* set to 10^{-16} erfc takes about 50% more time than with *accuracy* set to 10^{-7}.

See Also
The **erfc**, **erfcw**, **dawson**, **inverseErf**, and **inverseErfc** functions.

erfc

```
erfc(num [, accuracy])
```

The erfc function returns the complementary error function of *num* (erfc(x) = 1 - erf(x)). Optionally, *accuracy* can be used to specify the desired fractional accuracy.

In complex expressions the complementary error function is

$$erfc(z) = 1 - erfc(z) = 1 - \frac{2z}{\sqrt{\pi}}{}_1F_1(\frac{1}{2},\frac{3}{2},-z^2) \text{ where } {}_1F_1(\frac{1}{2},\frac{3}{2},-z^2)$$

is the confluent hypergeometric function of the first kind HyperG1F1. In this case the accuracy parameter is ignored.

Details
The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of (f_{actual} - $f_{returned}$)/f_{actual} be less than 10^{-7}.

For backwards compatibility, in the absence of *accuracy* an alternate calculation method is used that achieves fractional accuracy better than 2x10^{-7}.

If *accuracy* is present, erfc can achieve fractional accuracy better than $2\text{x}10^{-16}$ for *num* up to 1. From *num* = 1 to 10 fractional accuracy is better than $2\text{x}10^{-15}$.

Higher accuracy takes somewhat longer to calculate. With *accuracy* set to 10^{-16} erfc takes about 50% more time than with *accuracy* set to 10^{-7}.

See Also

The **erf**, **erfcw**, **inverseErfc**, and **dawson** functions.

erfcw

`erfcw(z)`

The erfcw is a complex form of the error function defined by

$$\mathrm{erfcw}(z) = \exp[-z^2]\,\mathrm{erfc}(-iz),$$

where

$$\mathrm{erfc}(z) = \frac{2}{\sqrt{\pi}}\int_z^\infty \exp[-t^2]\,dt.$$

The function is computed with accuracy of 0.5e-10. It is particularly useful for large |z| where the computation of erfc(*z*) starts encountering numerical instability.

References

1. http://en.wikipedia.org/wiki/Error_function

2. W. Gautschi, "*Efficient Computation of the Complex Error Function*", SIAM J. Numer. Anal. Vol. 7, No. 1, March 1970.

See Also

The **erf**, **erfc**, **inverseErfc**, and **dawson** functions.

ErrorBars

`ErrorBars [`*`flags`*`] `*`traceName`*`, `*`mode`*` [`*`errorSpecification`*`]`

The ErrorBars operation adds or removes error bars to or from the named trace in the specified graph.

The “error bars” are lines that extend from each data point to “caps”. The length of the line (or “bar”) is usually used to bracket a measured value by the amount of uncertainty, or “error” in the measurement.

Parameters

traceName is usually the name of a wave. If a wave is displayed more than once in a graph, the instance number can be appended to identify which instance to apply error bars to. For instance, wave0#2 refers to the third instance of wave0 displayed in the top graph (wave0, or wave0#0, is the first instance).

A string containing *traceName* can be used with the $ operator to specify *traceName*.

mode is one of the following keywords:

OFF	No error bars.
X	Horizontal error bars only.
Y	Vertical error bars only.
XY	Horizontal and vertical error bars.
BOX	Box error bars.

SHADE={*options*, *fillMode*, *fgColor*, *bkColor* [, *negFillMode*, *negFgColor*, *negBkColor*]}

SHADE was added in Igor Pro 7.00.

options is reserved for future use and must be zero *fillMode* sets the fill pattern.

n=0:	No fill.
n=1:	Erase.
n=2:	Solid black.
n=3:	75% gray.
n=4:	50% gray.
n=5:	25% gray.
n>=6:	See **Fill Patterns** on page III-441.

fgColor is (r,g,b) or (r,g,b,a). If all zeros including alpha, i.e., (0,0,0,0), then the actual color will be the trace color with an alpha of 20000.

bkColor is used for patterns only and can be simply (0,0,0) for solid fills.

negFillMode is the same as *fillMode* but for negative error shading.

negFgColor is the same as fgColor but for negative error shading.

negBkColor is the same as bkColor but for negative error shading.

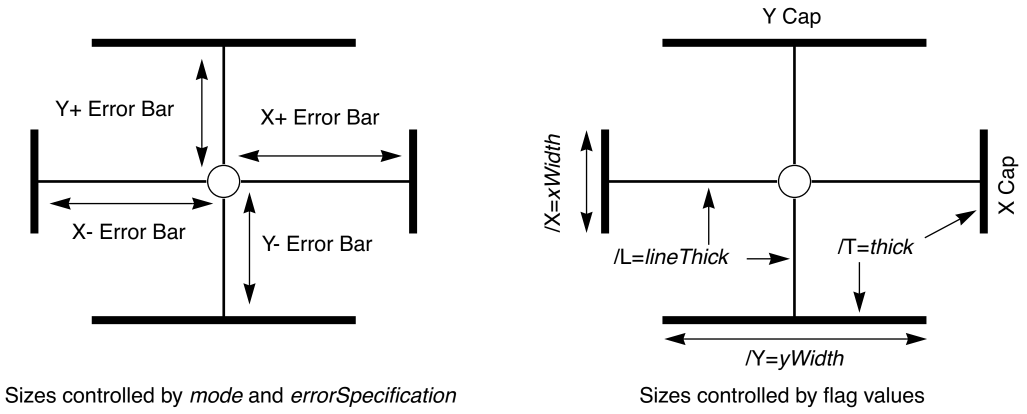
The *errorSpecification* , described below, affects only the Y amplitude of error shading.

See **Error Shading** on page II-234 for more information and examples. See **Color Blending** on page III-440 for information on the alpha color parameter.

For any *mode* other than `OFF`, there is an *errorSpecification* whose format is ***keyword*** [**=** ***value***]. The *errorSpecification* keywords are:

pct	Percent.
sqrt	Square root.
const	Constant.
wave	Arbitrary error values. You can use subranges; see **Subrange Display Syntax** on page II-250.

See the examples for the values that correspond to these *mode* and *errorSpecification* keywords. See the diagram below. *mode* and *errorSpecification* control only the lengths of the horizontal and vertical lines (the “bars”) to the “caps”. All other sizes and thicknesses are controlled by the flags.



XY *mode* Error Bars

Flags

/L=*lineThick*	Specifies the thickness of both the X and Y error bars drawn from the point on the wave to the caps.
/T=*thick*	Specifies the thickness of both the X and Y error bar “caps”.

/W=*winName*	Changes error bars in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/X=*xWidth*	Specifies the width (height, actually) of the caps to the left or right of the point.
/Y=*yWidth*	Specifies the width of the caps above or below the point.

The thicknesses and widths are in units of points. The thickness parameters need not be integers. Although only integral thicknesses can be displayed exactly on the screen, nonintegral thicknesses are produced properly on high resolution hard copy devices. Use /T=0 to completely suppress the caps and /L=0 to completely suppress the lines to the caps.

Details

If a point in *traceName* is not within the graph's axes (because the graph has been expanded) then that point's error bars are not shown. If a wave specifying error values for *traceName* is shorter than the wave displayed by *traceName* then the last value of the error wave is used for the unavailable points. If a point in an error wave contains NaN (Not a Number) then the half-bar associated with that point is not shown.

Examples

`ErrorBars wave1,XY pct=10,pct=5`	X and Y error bars, X is 10% of `wave1`, Y is 5%
`ErrorBars wave1,X sqrt`	X error bars only, square root of `wave1`
`ErrorBars wave1,Y const=4.3`	Y error bars only, constant error value = 4.3
`ErrorBars wave1,BOX pct=10,pct=5`	error box, 10% in horizontal direction 5% in vertical direction
`ErrorBars wave1,Y wave=(w1,w2)`	Y error bars only, arbitrary error values wave `w1` = errors for upper (Y+) bars wave `w2` = errors for lower (Y-) bars
`ErrorBars wave1,Y wave=(,w2)`	Y error bars only, no upper (Y+) error bars wave `w2` = errors for lower bars
`ErrorBars wave1,OFF`	turns error bars for wave1 off

See Also

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

Execute

`Execute` `[`**`/Q/Z`**`]` ***`cmdStr`***

The Execute operation executes the contents of *cmdStr* as if it had been typed in the command line.

The most common use of Execute is to call a macro or an external operation from a user-defined function. This is necessary because Igor does not allow you to make such calls directly.

When the /Z flag is used, an error code is placed in V_flag.The error code will be -1 if a missing parameter style macro is called and the user clicks Quit Macro, or zero if there was no error.

Flags

/Q	Command is not printed in the command line or history area.
/Z	Errors are not fatal and error dialogs are suppressed.

Details

Because the command line and command buffer are limited to 1000 bytes on a single line, *cmdStr* is likewise limited to a maximum of 1000 executable bytes.

Do not reference local variables in *cmdStr*. The command is not executed in the local environment provided by a macro or user-defined function.

Execute can accept a string expression containing a macro. The string must start with Macro, Proc, or Window, and must follow the normal rules for macros. All lines must be terminated with carriage returns including the last line. The name of the macro is not important but must exist. Errors will be reported except when using the /Z flag, which will assign V_Flag a nonzero number in an error condition.

Examples

It is a good idea to compose the command to be executed in a local string variable and then pass that string to the Execute operation. This prints the string to the history for debugging:

```
String cmd
sprintf cmd, "GBLoadWave/P=%s/S=%d \"%s\"", pathName, skipCount, fileName
Print cmd       // For debugging
Execute cmd
```

```
// Execute with a macro:
Execute "Macro junk(a,b)\rvariable a=1,b=2\r\rprint \"hello from macro\",a,b\rEnd\r"
```

See Also

The Execute Operation on page IV-190 for other uses.

Execute/P

`Execute/P [/Q/Z] `*`cmdStr`*

Execute/P is similar to Execute except the command string, *cmdStr*, is not immediately executed but rather is posted to an operation queue. Items in the operation queue execute only when nothing else is happening. Macros and functions must not be running and the command line must be empty.

Flags

/Q	Command is not printed in the command line or history area.
/Z	No error reporting.

See Also

Operation Queue on page IV-263 for more details on using Execute/P with the operation queue.

ExecuteScriptText

`ExecuteScriptText [`*`flags`*`] `*`textStr`*

The ExecuteScriptText operation passes your text to Apple's scripting subsystem for compilation and execution or to the Windows command line.

If the /Z flag is used then a variable named V_flag is created and is set to a nonzero value if an error was generated by the script or zero if no error. The error is not reported to Igor if the /Z flag is used.

On Macintosh:	Any results, including error messages, are placed in a string variable named S_value. String values returned in S_value often have double-quote characters at the start and end. Use /UNQ to remove them. The /B flag is ignored. The /W flag is ignored and ExecuteScriptText does not return until the script is finished. The current directory will be / and the PATH environment variable will be the system default, not the user default. Usually PATH=/usr/bin:/bin:/usr/sbin:/sbin.

On Windows: *textStr* contains the name of the executable file with optional Windows-style path and optional arguments:

[*path*]*executableName* [.exe] [*arg1*]…

If *path* is not given and *executableName* ends with ".exe" or with no extension, Igor locates the executable by searching first the registry and then along the PATH environment variable. If not found, then the Igor Pro 7 Folder directory is assumed.

```
ExecuteScriptText "calc"  // calc.exe, calculator
```

When calling a batch file or other non-*.exe file, supply the full path. If the path (or file name) contains spaces put quotes in the string:

```
ExecuteScriptText "\"C:\\Program Files\\my.bat\""
```

If you use the /W=*waitTime* flag with a positive value for *waitTime*, ExecuteScriptText waits up to that many seconds after submitting the command for the process started by the command to terminate. If the process fails to terminate within that period of time, ExecuteScriptText returns an error.

If you omit the /W flag or if you pass zero for *waitTime*, for GUI programs, ExecuteScriptText returns when the program begins processing messages. For non-GUI programs, ExecuteScriptText returns as soon as the OS returns control to Igor after Igor submits the script text to the OS.

Igor currently can not write to a console application's standard input nor read from its standard output.

Use the /B flag to run the command in the background, keeping Igor as the active application.

S_value is always set to `""`.

Parameters

textStr must contain a valid AppleScript program or Windows command line.

Flags

/B	Execute Windows command line as a background task.
/W=*waitTime*	This flag is accepted on any platform but has an effect only on Windows. See the description above.
/UNQ	Removes any leading and trailing double-quote characters from S_value. This is useful on Macintosh only and has no effect on Windows. The /UNQ flag was added in Igor Pro 7.00.
/Z	Script errors are not fatal.

Examples

```
// Macintosh: Convert file.PICT to file.GIF:
String ae = "tell application \"clip2gif\" "
ae += "to save file \"HD:file.PICT\"\r"
ExecuteScriptText/Z ae

// Macintosh: Execute a Unix shell command:
Function/S DemoUnixShellCommand()
    // Paths must be POSIX paths (using /).
    // Paths containing spaces or other nonstandard characters must be single-quoted.
    // See Apple Techical Note TN2065 for more on shell scripting via AppleScript.
    String unixCmd
    unixCmd = "ls '/Applications/Igor Pro 7 Folder'"

    String igorCmd

    sprintf igorCmd, "do shell script \"%s\"", unixCmd
    Print igorCmd                       // For debugging only.
    ExecuteScriptText/UNQ igorCmd
    Print S_value                     // For debugging only.
    return S_value
End

// Windows: Open MatLab in the background:
ExecuteScriptText/B "C:\\Matlab\\bin\\matlab.exe myFile.m"

// Windows: Pass a script to Windows Script Host:
ExecuteScriptText/W=5 "WScript.exe \"C:\\Test Script.vbs\""

// Windows: Execute a batch file and leave the command window open
ExecuteScriptText "cmd.exe /K \"C:\\mybatch.bat\""

// Windows: Execute a DOS command and get output, if any
// ExecuteDOSCommand(command, maxSecondsToWait)
// Executes a DOS command and returns any output text as the function result.
// Returns "" if the DOS command returns no text.
// maxSecondsToWait is the maximum number of seconds to wait for DOS to finish
// the command. If it takes longer than that, an error is generated.
// This function creates files in the Igor Pro User Folder:
//     igorBatch.bat           Holds command that DOS is to execute
//     igorBatchOutput.txt     Holds output generated by DOS command, if any
// Example:
//     Print ExecuteDOSCommand("echo %PATH%", 3)
Function/S ExecuteDOSCommand(command, maxSecondsToWait)
    String command                 // e.g., "echo %PATH%"
    Variable maxSecondsToWait      // Error if DOS takes longer than this

    String quoteStr = "\""

    // Get path to batch file in "Igor Pro User Files"
    String dirPath = SpecialDirPath("Igor Pro User Files", 0, 0, 0)
    dirPath = ParseFilePath(5, dirPath, "\\", 0, 0)       // Convert to Windows path
    String batchFilePath = dirPath + "igorBatch.bat"
    String batchOutputFilePath = dirPath + "igorBatchOutput.txt"

    DeleteFile/Z batchOutputFilePath

    // Write DOS command to batch file
    String dosCommand = command + " > " + quoteStr + batchOutputFilePath + quoteStr
    Variable refNum
    Open refNum as batchFilePath
    FBinWrite refNum, dosCommand
    Close refNum

    // Execute batch file
    // The DOS command must complete in the number of seconds specified via /W
    // /C means cmd.exe quits after executing the command
    String text
    sprintf text, "cmd.exe /C \"%s\"", batchFilePath
    ExecuteScriptText/W=(maxSecondsToWait) text

    // Get output
    String result = ""
    Open/R/Z refNum as batchOutputFilePath
    if (V_flag != 0)
        result = ""
        // result = "<No output was generated by batch file>"    // For debugging
    else
        // Read contents of batch file into string
        FStatus refNum
        Variable numBytesInFile = V_logEOF
        result = PadString("", numBytesInFile, 0x20)
        FBinRead refNum, result
        Close refNum
    endif

    return result
End
```

See Also

See **AppleScript** on page IV-249.

exists

`exists(`*`objNameStr`*`)`

The exists function returns a number which indicates if *objNameStr* contains the name of an Igor object, function or operation.

Details

objNameStr can optionally include a full or partial path to the object. If the name does not include a path, exists checks for waves, strings and variables in the current data folder.

objNameStr can optionally include a module name or independent module name prefix such as "ProcGlobal#" to check for the existence of functions. This works for macros as well.

The return values are:

0:	Name not in use, or does not conflict with a wave, numeric variable or string variable in the specified data folder.
1:	Name of a wave in the specified data folder.
2:	Name of a numeric or string variable in the specified data folder.
3:	Function name.
4:	Operation name.
5:	Macro name.
6:	User-defined function name.

exists is not aware of local variables or parameters in user-defined functions, however it is aware of local variables and parameters in macros.

objNameStr is a string or string expression, *not* a name.

Examples

```
// Prints 2 if V_flag exists as a global variable in the current data folder:
Print exists("V_Flag")

// Prints 5 if a macro named Graph0 exists.
Print exists("ProcGlobal#Graph0")
```

See Also

The **DataFolderExists** and **WaveExists** functions and the **WinType** operation.

exp

`exp(`*`num`*`)`

The exp function returns $e^{num.}$ In complex expressions, *num* is complex, and exp(*num*) returns a complex value.

ExperimentModified

`ExperimentModified  [`*`newModifiedState`*`]`

The ExperimentModified operation gets and optionally sets the modified (save) state of the current experiment.

Use this command to prevent Igor from asking you to save the current experiment after you have made changes you do not need to save or, conversely, to force Igor to ask about saving the experiment even though Igor would not normally do so.

The variable V_flag is always set to the experiment-modified state that was in effect before the ExperimentModified command executed: 1 for modified, 0 for not modified.

Parameters

If *newModifiedState* is present, it sets the experiment-modified state as follows:

newModifiedState = 0:	Igor will not ask to save the experiment before quitting or opening another experiment, and the Save Experiment menu item will be disabled.

newModifiedState = 1: Igor will ask to save the experiment before quitting or opening another experiment, and the Save Experiment menu item will be enabled.

If *newModifiedState* is omitted, the state of experiment-modified state is not changed.

Details

Executing ExperimentModified 0 on the command line will not work because the command will be echoed to the history area, marking the experiment as modifed. Use the command in a function or macro that does not echo text to the history area.

Examples

The /Q flag is vital: it suppresses printing into the history area which would mark the experiment as modified again.

```
Menu "File"
	"Mark Experiment Modified",/Q,ExperimentModified 1	// Enables "Save Experiment"
	"Mark Experiment Saved",/Q,ExperimentModified 0		// Disables "Save Experiment"
End
```

See Also

The **SaveExperiment** operation, **Menu Definition Syntax** on page IV-118.

expInt

`expInt(`*`n, x`*`)`

The expInt function returns the value of the exponential integral $E_n(x)$:

$$E_n(x) = P\int_1^\infty \frac{e^{-xt}}{t^n}dt \qquad (x > 0; n = 0, 1, 2\ldots).$$

See Also

ei, **ExpIntegralE1**

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

ExpIntegralE1

`ExpIntegralE1(z)`

The ExpIntegralE1(*z*) function returns the exponential integral of *z*.

If *z* is real, a real value is returned. If *z* is complex then a complex value is returned.

The ExpIntegralE1 function was added in Igor Pro 7.00.

Details

The exponential integral is defined by

$$E_1(z) = \int_z^\infty \frac{e^{-t}}{t}dt, \qquad \left(|\arg(z)| < \pi\right).$$

References

Abramowitz, M., and I.A. Stegun, "Handbook of Mathematical Functions", Dover, New York, 1972. Chapter 5.

See Also

expInt, **CosIntegral**, **SinIntegral**, **hyperGPFQ**

expnoise

`expnoise(`*`b`*`)`

The expnoise function returns a pseudo-random value from an exponential distribution whose average and standard deviation are *b* and the probability distribution function is

$$f(x) = \frac{1}{b}\exp\left(-\frac{x}{b}\right).$$

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-344.

Chapter III-12, **Statistics** for a function and operation overview.

ExportGizmo

`ExportGizmo [`*`flags`*`]` ***`keyword`*** **`[=`*`value`*`]`**

The ExportGizmo operation is obsolete but is still partially supported for partial backward compatibility.

As of Igor7 you can export Gizmo graphics using File→Save Graphics which generates a **SavePICT** command. The ExportGizmo operation is only partially supported. It can export to the clipboard or to an Igor wave and it can print but it can no longer export to a file. Use SavePICT instead.

Documentation for the ExportGizmo operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "ExportGizmo"
```

Extract

`Extract [`*`type flags`*`][/INDX/O]` ***`srcWave, destWave, LogicalExpression`***

The Extract operation finds data in *srcWave* wherever *LogicalExpression* evaluates to TRUE and stores the matching data sequentially in *destWave*, which will be created if it does not already exist.

Parameters

srcWave is the name of a wave.

destWave is the name of a new or existing wave that will contain the result.

LogicalExpression can use any comparison or logical operator in the expression.

Flags

/FREE	Creates a free destWave (see **Free Waves** on page IV-84). /FREE is allowed only in functions and only if a simple name or wave reference structure field is specified for *destWave*.
/INDX	Stores the index in *destWave* instead of data from *srcWave*.
/O	Allows *destWave* to be the same as *srcWave* (overwrite source).

Type Flags *(used only in functions)*

Extract also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when it needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-67 and **WAVE Reference Type Flags** on page IV-68 for a complete list of type flags and further details.

Details

srcWave may be of any type including text.

destWave has the same type as *srcWave*, but it is always one dimensional. With /INDX, the *destWave* type is set to unsigned 32-bit integer and the values represent a linear index into *srcWave* regardless of its dimensionality.

Example

```
Make/O source= x
Extract/O source,dest,source>10 && source<20
print dest
```

Prints the following to the history area:

```
  dest[0]= {11,12,13,14,15,16,17,18,19}
```

See Also

The **Duplicate** operation.

factorial

`factorial(`*`n`*`)`

The factorial function returns *n*!, where *n* is assumed to be a positive integer.

Note that while factorial is an integer-valued function, a double-precision number has 53 bits for the mantissa. This means that numbers over 2^{52} will be accurate to about one part in about $2\text{x}10^{16}$. Values of *n* greater than 170 result in overflow and return Inf.

FakeData

`FakeData(`*`waveName`*`)`

The FakeData function puts fake data in the named wave, which must be single-precision float. This is useful for testing things that require changing data before you have the source for the eventual real data. FakeData can be useful in a background task expression.

The FakeData function is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

Examples

```
Make/N=200 wave0; Display wave0
SetBackground FakeData(wave0)         // define background task
CtrlBackground period=60, start       // start background task
// observe the graph for a while
CtrlBackground stop                   // stop the background task
```

FastGaussTransform

`FastGaussTransform [`*`flags`*`] `*`srcLocationsWave, srcWeightsWave`*

The FastGaussTransform operation implements an efficient algorithm for evaluating the discrete Gauss transform, which is given by

$$G(y_j)=\sum_{i=0}^{N-1} q_i \exp\left(-\frac{\left\|y_j-x_i\right\|^2}{h}\right),$$

where $\boldsymbol{G}$ is an M-dimensional vector, $\boldsymbol{y}$ is an N-dimensional vector representing the observation position, $\{\boldsymbol{q}_i\}$ are the M-dimensional weights, $\{\boldsymbol{x}_i\}$ are N-dimensional vectors representing source locations, and *h* is the Gaussian width. The wave M_FGT contains the output in the current data folder.

Flags

/AERR=*aprxErr*	Sets the approximate error, which determines how many terms of the Taylor expansion of the Gaussian are used by the calculation. Default value is 1e-5.
/WDTH=*h*	Sets the Gaussian width. Default value is 1.
/OUTW=*locWave*	Specifies the locations at which the output is computed. *locWave* must have the same number of columns as *srcLocationsWave*. The other /OUT flags are mutually exclusive; you should use only one at any time.

/OUT1={*x1*,*nx*,*x2*}

/OUT2={*x1*,*nx*,*x2*,*y1*,*ny*,*y2*}

/OUT3={*x1*,*nx*,*x2*,*y1*,*ny*,*y2*,*z1*,*nz*,*z2*}

Specifies gridded output of the required dimension. In each case you set the starting and ending values together with the number of intervals in that dimension. You cannot specify an output that does not match the dimensions of the input source.

/Q	No results printed in the history area.
/RX=*rx*	Sets the maximum radius of any cluster. The clustering algorithm terminates when the maximum radius is less than *rx*. Without /RX, the maximum radius is the same as the maximum radius encountered.
/RY=*ry*	Sets the upper bound for the distance between an observation point and a cluster center for which the cluster contributes to the transform value. Default is 5*h*.
/TET=*nTerms*	Sets the number of terms in the Taylor expansion. Use /TET to set the number of terms and bypass the default error estimate, which is estimated from the approximate error value (/AERR).
/Z	No error reporting.

Details

The discrete Gauss transform can be computed as a direct sum. An exact calculation is practical only for moderate number of sources and observation points and for low spatial dimensionality. With increasing dimensionality and increasing number of sources it is more efficient to take advantage of some properties of the Gaussian function. The FastGaussianTransform operation does so in two ways: It first arranges the sources in N-dimensional spatial clusters so that it is not necessary to compute the contributions of all source points that belong to remote clusters (see **FPClustering**). The second component of the algorithm is an approximation that factorizes the sum into a factor that depends only on source points and a factor that depends only on observation points. The factor that depends only on source points is computed only once while the factor that depends on observation points is evaluated once for each observation point.

The trade-off between computation efficiency and accuracy can be adjusted using multiple parameters. By default, the operation calculates the number of terms it needs to use in the Taylor expansion of the Gaussian. You can modify the default approximate error value using /AERR or you can directly set the number of terms in the expansion using /TET.

FastGaussianTransform supports calculations in dimensions that may exceed the maximum allowed wave dimensionality. *srcLocationsWave* must be a 2D, real-valued single- or double-precision wave in which each row corresponds to a single source position and columns represent the components in each dimension (e.g., a triplet wave would represent 3D source locations). *srcWeightsWave* must have the same number of rows as *srcLocationsWave* and it must be a real-valued single- or double-precision wave. In most applications *srcWeightsWave* will have a single column so that the output ***G*** will be scalar. However, if *srcWeightsWave* has multiple columns than ***G*** is a vector. This can be handy if you need to test multiple sets of coefficients at one time. If you specify observation points using /OUTW then *locWave* must have the same number of columns as *srcLocationsWave* (the number of rows in the output is arbitrary). The operation does not support wave scaling.

See Also

The **CWT**, **FFT**, **ImageInterpolate**, **Loess**, and **FPClustering** operations.

References

Yang, C., R. Duraiswami, and L. Davis, Efficient Kernel Machines Using the Improved Fast Gauss Transform, *Advances in Neural Information Processing Systems 16*, 2004.

FastOp

`FastOp` `[/C]` ***`destwave = prod1`*** `[`***`± prod2`*** `[`***`± prod3`***`]]`

The FastOp operation can be used to get improved speed out of certain wave assignment statements. The syntax was designed so that you can simply insert `FastOp` in front of any wave assignment statement that meets the syntax requirements.

Parameters

destWave	An existing destination wave for the assignment expression. An error will be reported at runtime if the waves are not all the same length or number type.
prod1, *prod2*, *prod3*	Products with the following formats: *constexpr*wave1*wave2* or *constexpr*/wave2* *constexpr* may be a literal numeric constant or a constant expression in parentheses. Such expressions are evaluated only once. Any component in a prod expression may be omitted.

Flags

/C	Specifies a complex expression. Only applicable to floating point waves.

Details

Certain combinations are evaluated using faster specific code rather than more general but slower generic code. The following specific formats are given special consideration:

Single or Double Precision Real	Integer
waved = C0	*waved = C0*
*waved = C0 *waveA +C1*	*waved = waveA +C1*
waved = waveA +C1	*waved = waveA +waveB +C2*

In the above, pluses may be minuses and the trailing constant (*C0, C1, C2*) may be omitted.

Note: Integer waves are evaluated using double precision intermediate values except for the aforementioned special cases which are evaluated using the native format.

Typically, FastOp will improve performance by 10 to 40 times. The speed increase will be dependent on the computer and on the length of the waves, with the greatest improvement for waves having 1000 to 100,000 points.

This operation replaces the obsolete FastWaveOps XOP. It has all the capabilities of the XOP and then some and has an easier to read syntax.

Examples

Valid expressions:

```
FastOp waved= 3
FastOp waved= waveA + waveB
FastOp waved= 0.5*waveA + 0.5*waveB
FastOp waved= waveA*waveB
FastOp waved= (2*3)*waveA + 6
FastOp waved= (locvar)*waveA
```

Expressions that are **not** valid:

```
FastOp waved= 3*4
FastOp waved= (waveA + waveB)
FastOp waved= waveA*0.5 + 0.5*waveB
FastOp waved= waveA*waveB/2
FastOp waved= 2*3*waveA + 6
FastOp waved= locvar*waveA
```

See Also

The **MatrixOp** operation for more efficient matrix operations.

faverage

`faverage(`*`waveName`*` [, `*`x1, x2`*`])`

The faverage function returns the trapezoidal average value of the named wave from x=*x1* to x=*x2*.

If your data are in the form of an XY pair of waves, see **faverageXY**.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If *x1* or *x2* are not within the X range of *waveName*, faverage limits them to the nearest X range limit of *waveName*.

faverage returns the area divided by (*x2*-*x1*). In other words, the X scaling of *waveName* is eliminated when computing the average.

If any Y values in the specified X range are NaN, faverage returns NaN.

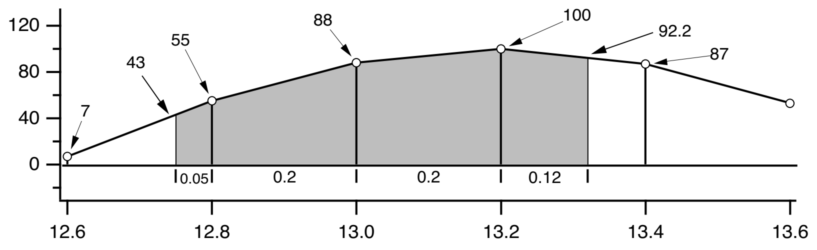
Unlike the **area** function, reversing the order of *x1* and *x2* does *not* change the sign of the returned value.

The faverage function is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

The faverage function returns a complex result for a complex inpt wave. The real part of the result is the average of the real components in the input wave and the imaginary part of the result is the average of the imaginary components.

Examples

Comparison of area, faverage and mean functions over interval (12.75,13.32)



area(wave,12.75,13.32)	= 0.05 · (43+55) / 2	// first trapezoid
	+ 0.20 · (55+88) / 2	// second trapezoid
	+ 0.20 · (88+100) / 2	// third trapezoid
	+ 0.12 · (100+92.2) / 2	// fourth trapezoid
	= 47.082	
faverage(wave,12.75,13.32)	= area(wave,12.75,13.32) / (13.32-12.75)	
	= 47.082/0.57 = 82.6	
mean(wave,12.75,13.32)	= (55+88+100+87)/4 = 82.5	

See Also

Integrate, **area**, **areaXY**, **faverageXY** and **PolygonArea**

faverageXY

`faverageXY(`*`XWaveName, YWaveName`*` [, `*`x1, x2`*`])`

The faverageXY function returns the trapezoidal average value of *YWaveName* from x=*x1* to x=*x2*, using X values from *XWaveName*.

This function operates identically to faverage, except that it uses an XY pair of waves for X and Y values and it does not work with complex waves.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If *x1* or *x2* are not within the X range of *XWaveName*, faverageXY limits them to the nearest X range limit of *XWaveName*.

faverageXY returns the area divided by (*x2* -*x1*).

If any values in the X range are NaN, faverageXY returns NaN.

Reversing the order of *x1* and *x2* does not change the sign of the returned value.

The values in *XWaveName* may be increasing or decreasing. faverageXY assumes that the values in *XWaveName* are monotonic. If they are not monotonic, Igor does not complain, but the result is not meaningful. If any X values are NaN, the result is NaN.

The faverageXY function is not multidimensional aware. See Chapter II-6, **Multidimensional Waves** for details on multidimensional waves, particularly **Analysis on Multidimensional Waves** on page II-86.

See Also

Integrate, **area**, **areaXY**, **faverage** and **PolygonArea**

FBinRead

`FBinRead [flags] refNum, objectName`

The FBinRead operation reads binary data from the file specified by *refNum* into the named object.

For simple applications of loading binary data into numeric waves you may find the GBLoadWave operation simpler to implement.

Parameters

refNum is a file reference number from the Open operation used to open the file.

objectName is the name of a wave, numeric variable, string variable, or structure.

Flags

/B[=*b*]	Specifies file byte ordering.
	b=0: Native (same as no /B).
	b=1: Reversed (same as /B).
	b=2: Big-endian (Motorola).
	b=3: Little-endian (Intel).
/F=*f*	Controls the number of bytes read and how the bytes are interpreted.
	f=0: Native binary format of the object (default).
	f=1: Signed byte; one byte.
	f=2: Signed 16-bit word; two bytes.
	f=3: Signed 32-bit word; four bytes.
	f=4: 32-bit IEEE floating point; four bytes.
	f=5: 64-bit IEEE floating point; eight bytes.
	f=6: 64-bit integer; eight bytes. Requires Igor Pro 7.00 or later.
/U	Integer formats (/F=1, 2, or 3) are unsigned. If /U is omitted, integers are signed.

Details

If *objectName* is the name of a string variable then /F doesn't apply. The number of bytes read is the number of bytes in the string *before* the FBinRead operation is called. You can use the **PadString** function to set the size of a string.

The binary format that FBinRead uses for numeric variables or waves depends on the /F flag. If no /F flag is present, the native binary format of the named object is used.

Byte ordering refers to the order in which a multibyte datum is read from a file. For example, a 16-bit word (sometimes called a "short") consists of a high-order byte and a low-order byte. Under big-endian byte ordering, which is commonly used on Macintosh, the high-order byte is read from the file first. Under little-endian byte ordering, which is commonly used on Windows, the low-order byte is read from the file first.

FBinRead will read an entire structure from a disk file. The individual fields of the structure will be byte-swapped if the /B flag is designated.

The FBinRead operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

See Also

FBinWrite, **Open**, **FGetPos**, **FSetPos**, **FStatus**, **GBLoadWave**

FBinWrite

`FBinWrite [flags] refNum, objectName`

The FBinWrite operation writes the named object in binary to a file.

Parameters

refNum is a file reference number from the **Open** operation used to open the file.

objectName is the name of a wave, numeric variable, string variable, or structure.

Flags

/B[=*b*]	Specifies file byte ordering. *b*=0: Native (same as no /B). *b*=1: Reversed (same as /B). *b*=2: Big-endian (Motorola). *b*=3: Little-endian (Intel).
/F=*f*	Controls the number of bytes written and how the bytes are formatted. *f*=0: Native binary format of the object (default). *f*=1: Signed byte; one byte. *f*=2: Signed 16-bit word; two bytes. *f*=3: Signed 32-bit word; four bytes. *f*=4: 32-bit IEEE floating point; four bytes. *f*=5: 64-bit IEEE floating point; eight bytes. *f*=6: 64-bit integer; eight bytes. Requires Igor Pro 7.00 or later.
/P	Adds an IgorBinPacket to the data. This is used for PPC or Apple event result packets (*refNum* = 0) and is not normally of use when writing to a file.
/U	Integer formats (/F=1, 2, or 3) are unsigned. If /U is omitted, integers are signed.

Details

A zero value of *refNum* is used in conjunction with Program-to-Program Communication (PPC) or Apple events (*Macintosh*) or DDE (*Windows*). The data that would normally be written to a file is appended to the PPC or Apple event or DDE result packet.

If the object is a string variable then /F doesn’t apply. The number of bytes written is the number of bytes in the string.

The binary format that FBinWrite uses for numeric variables or waves depends on the /F flag. If no /F flag is present, FBinWrite uses the native binary format of the named object.

Byte ordering refers to the order in which a multibyte datum is written to a file. For example, a 16-bit word (sometimes called a “short”) consists of a high-order byte and a low-order byte. Under big-endian byte ordering, which is commonly used on Macintosh, the high-order byte is written to the file first. Under little-endian byte ordering, which is commonly used on Windows, the low-order byte is written to the file first.

FBinWrite will write an entire structure to a disk file. The individual fields of the structure will be byte-swapped if the /B flag is designated.

The FBinWrite operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

See Also

FBinRead, **Open**, **FGetPos**, **FSetPos**, **FStatus**, **GBLoadWave**

FetchURL

`FetchURL(`*`urlStr`*`)`

The FetchURL function returns a string containing the server's response to a request to get the contents of the URL specified by *urlStr*. If *urlStr* contains a URL that uses the file:// scheme, the contents of the local file is returned.

Parameters

urlStr is a string containing the URL to retrieve. You can include a username, password, and server port number as part of the URL.

FetchURL expects that *urlStr* has been percent-encoded if it contains reserved characters. See **Percent Encoding** on page IV-253 for additional information on when percent-encoding is necessary and how to do it.

See **URLs** on page IV-252 for details about how to correctly specify the URL.

FetchURL supports only the http://, https://, ftp://, and file:// schemes. See **Supported Network Schemes** on page IV-253 for details.

There are two special values of *urlStr* that can be used to get information about the network library that Igor uses. The keyword=value pairs returned when *urlStr* is "curl_version_info" may be useful to programmers in that the features and protocols available in the library are specified.

```
FetchURL("curl_version")
FetchURL("curl_version_info")
```

Details

If FetchURL encounters an error, it returns a NULL string. You should check for errors before using the returned string. In a user-defined function, use the **GetRTError** function.

```
String urlStr = "http://www.badserver"
String response = FetchURL(urlStr)
Variable error = GetRTError(1)       // Check for error before using response
if (error != 0)
   // FetchURL produced an error
   // so don't try to use the response.
endif
```

Limitations

It is possible for FetchURL to return a valid server response even though the URL you requested does not exist on the server or requires a username and password that you did not provide. In this situation, the response returned by the server will usually be a web page stating that the page was not found or another error message. You can check for this kind of error in your own code by examining the response.

FetchURL does not support advanced features such as network proxies, file or data uploads, setting the timeout period, or saving the server's response directly to a file. When using the http:// scheme, only the GET method is supported. This means that you cannot use FetchURL to submit form data to a web server that requires using the http POST method. Use the **URLRequest** operation if you need any of these features.

Igor Pro is not capable of displaying the contents of a URL in a rendered form like a web browser.

Examples

```
// Retrieve the contents of the WaveMetrics home page.
String response
response = FetchURL("http://www.wavemetrics.com")

// Get a binary image file from a web server and then
// save the image to a file on the desktop.
String url = "http://www.wavemetrics.net/images/tbg.gif"
String imageBytes = FetchURL(url)
Variable error = GetRTError(1)
if (error != 0)
   Print "Error downloading image."
else
   Variable refNum
   String localPath = SpecialDirPath("Desktop", 0, 0, 0) + "tbg.gif"
   Open/T=".gif" refNum as localPath
   FBinWrite refNum, imageBytes
   Close refNum
endif
```

See Also

FTPDownload, **URLEncode**, **URLRequest**

Network Communication on page IV-252, **Network Connections From Multiple Threads** on page IV-255.

FFT

`FFT [flags] srcWave`

The FFT operation computes the Discrete Fourier Transform of *srcWave* using a multidimensional prime factor decomposition algorithm. By default, *srcWave* is overwritten by the FFT.

Output Wave Name

For compatibility with earlier versions of Igor, if you use FFT with no flags or with just the /Z flag, the operation overwrites *srcWave*.

If you use any flag other than /Z, FFT uses default output wave names: W_FFT for a 1D FFT and M_FFT for a multidimensional FFT.

We recommend that you use the /DEST flag to make the output wave explicit and to prevent overwriting *srcWave*.

Flags

/COLS	Computes the 1D FFT of 2D *srcWave* one column at a time, storing the results in the destination wave. $$I[t_1][n]=\sum_{k=0}^{N-1} f[t_1][k]\exp\left(i2\pi kn/N\right).$$ You must specify a destination wave using the /DEST flag. No other flags are allowed with this flag. The number of rows must be even. If *srcWave* is a real (NxM) wave, the output matrix will be (1+N/2,M) in analogy with 1D FFT. To avoid changes in the number of points you can convert *srcWave* to complex data type. This flag applies only to 2D source waves. See also the /ROWS flag.
/DEST=*destWave*	Specifies the output wave created by the FFT operation. It is an error to attempt specify the same wave as both *srcWave* and *destWave*. The default output wave name is W_FFT for a 1D FFT and M_FFT for a multidimensional FFT. When used in a function, the FFT operation by default creates a complex wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-66 for details.
/FREE	Creates *destWave* as a free wave. /FREE is allowed only in functions and only if *destWave*, as specified by /DEST, is a simple name or wave reference structure field. See **Free Waves** on page IV-84 for more discussion. The /FREE flag was added in Igor Pro 7.00.
/HCC	Hypercomplex transform (cosine). Computes the integral $$I_c(\omega_1,\omega_2)=\int\int_{-\infty}^{\infty} f(t_1,t_2)\cos(t_1\omega_1)\exp(it_2\omega_2)dt_1dt_2$$ using the 2D FFT (see **Details**).

/HCS	Hypercomplex transform (sine). Computes the integral

$$I_s(\omega_1,\omega_2)=\int\int_{-\infty}^{\infty} f(t_1,t_2)\sin(t_1\omega_1)\exp(it_2\omega_2)dt_1dt_2$$

using the 2D FFT (see **Details**).

/MAG Saves just the magnitude of the FFT in the output wave. See comments under /OUT.

/MAGS Saves the squared magnitude of the FFT in the output wave. See comments under /OUT.

/OUT=*mode* Sets the output wave format.

mode=1:	Default for complex output.
mode=2:	Real output.
mode=3:	Magnitude.
mode=4:	Magnitude square.
mode=5:	Phase.
mode=6:	Scaled magnitude.
mode=7:	Scaled magnitude squared.

You can also identify modes 2-4 using the convenience flags /REAL, /MAG, and /MAGS. The convenience flags are mutually exclusive and are overridden by the /OUT flag.

The scaled quantities apply to transforms of real valued inputs where the output is normally folded in the first dimension (because of symmetry). The scaling applies a factor of 2 to the squared magnitude of all components except the DC. The scaled transforms should be used whenever Parseval's relation is expected to hold.

/PAD={*dim1* [, *dim2*, *dim3*, *dim4*]}

Converts *srcWave* into a padded wave of dimensions *dim1*, *dim2*…. The padded wave contains the original data at the start of the dimension and adds zero entries to each dimension up to the specified dimension size. The *dim1*… values must be greater than or equal to the corresponding dimension size of *srcWave*. If you need to pad just the lowest dimension(s) you can omit the remaining dimensions; for example, /Pad=*dim1* will set *dim2* and above to match the dimensions in *srcWave*.

/REAL Saves just the real part of the transform in the output wave. See comments under /OUT.

/ROWS Calculates the FFT of only the first dimension of a 2D *srcWave*. It thus computes the 1D FFT of one row at a time, storing the results in the destination wave.

$$N[n][t_2]=\sum_{k=0}^{M-1} f[k][t_2]\exp(i2\pi kn/M)$$

You must specify a destination wave using the /DEST flag. No other flags are allowed with this flag. The number of columns must be even. If *srcWave* is a real (NxM) wave, the output matrix will be (N,1+M/2) in analogy with 1D FFT. To avoid changes in the number of points you can convert *srcWave* to complex data type. See also /COLS flag.

/RP=[*startPoint*, *endPoint*]

/RX=(*startX*, *endX*) Defines a segment of a 1D *srcWave* that will be transformed. By default the operation transforms the whole wave. It is sometimes useful to take advantage of this feature in order to transform just the defined interval, which includes both end points. You can define the interval using wave point indexing with the /RP flag or using the X-values with the /RX flag. The interval must include at least four data points and the total number of points must be an even number.

/WINF=*windowKind*

Premultiplies a 1D *srcWave* with the selected window function.

If you include the /PAD flag, the window function is applied to the pre-padded data.

See **Window Functions** below for details.

/Z Disables rotation of the FFT of a complex wave. Igor normally rotates the FFT result (which is also complex) by N/2 so that x=0 is at the center point (N/2). When /Z is specified, Igor does not perform this rotation and leaves x=0 at the first point (0).

Details

The data type of *srcWave* is arbitrary. The first dimension of *srcWave* must be an even number and the minimum length of *srcWave* is four points. When *srcWave* is a double precision wave, the FFT is computed in double precision. All other data types are transformed using single precision calculations. The result of the FFT operation is always a floating point number (single or double precision).

Depending on your choice of outputs, you may not be able to invert the transform in order to obtain the original *srcWave*.

srcWave or any of its intervals must have at least four data points and must not contain NaNs or INFs.

The FFT algorithm is based on prime number decomposition, which decomposes the number of points in each dimension of the wave into a product of prime numbers. The FFT is optimized for primes < 5. In time consuming applications it is frequently worthwhile to pad the data so that the total number of points factors into small prime numbers.

The hypercomplex transforms are computed by writing the sine and cosine as a sum of two exponentials. Let the 2D Fourier transform of the input signal be

$$F[n_1][n_2] = \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} f[k_1][k_2] \exp\left(i2\pi k_1 \frac{n_1}{N_1}\right) \exp\left(i2\pi k_2 \frac{n_2}{N_2}\right)$$

then the two hypercomplex transforms are given by

$$I_c[n_1][n_2] = \frac{1}{2}\left(F[n_1][n2] + F[-n_1][n2]\right)$$

and

$$I_s[n_1][n_2] = \frac{1}{2i}\left(F[n_1][n2] - F[-n_1][n2]\right)$$

Window Functions

The /F=*windowKind* flag premultiplies a 1D *srcWave* with the selected window function.

In the following window definitions, *w*(*n*) is the value of the window function that multiplies the signal, *N* is the number of points in the signal wave (or range if /R is specified), and *n* is the wave point index. With /R, *n*=0 for the first datum in the range.

Choices for *windowKind* are in bold.

Bartlet:

A synonym for Bartlett.

Bartlett:

$$w(n) = \begin{cases} \dfrac{2n}{N} & n = 0,1,...\dfrac{N}{2} \\ 2 - \dfrac{2n}{N} & n = \dfrac{N}{2},...N-1 \end{cases}$$

Blackman367, **Blackman361**, **Blackman492**, **Blackman474:**

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi}{N} n\right) + a_2 \cos\left(\frac{2\pi}{N} 2n\right) - a_3 \cos\left(\frac{2\pi}{N} 3n\right) .$$

$$n = 0,1,2...N-1.$$

windowKind	a_0	a_1	a_2	a_3
Blackman367	0.42323	0.49755	0.07922	
Blackman361	0.44959	0.49364	0.05677	
Blackman492	0.35875	0.48829	0.14128	0.01168
Blackman474	0.40217	0.49703	0.09392	0.00183

Cos1, **Cos2**, **Cos3**, **Cos4:**

$$w(n) = \cos\left(\frac{n}{N}\pi\right)^{\alpha},$$

$$n = -\frac{N}{2},...,-1,0,1,...,\frac{N}{2}.$$

windowKind	α
Cos1:	$\alpha = 1$
Cos2:	$\alpha = 2$
Cos3:	$\alpha = 3$
Cos4:	$\alpha = 4$

Hamming:

$$w(n) = \begin{cases} 0.54 + 0.46\cos\left(\dfrac{2\pi n}{N}\right) & n = -\dfrac{N}{2},...,-1,0,1,...,\dfrac{N}{2} \\ 0.54 - 0.46\cos\left(\dfrac{2\pi n}{N}\right) & n = 0,1,2,...,N-1 \end{cases}$$

Hanning:

$$w(n) = \begin{cases} \frac{1}{2}\left[1+\cos\left(\frac{2\pi n}{N}\right)\right] & n = -\frac{N}{2},\ldots,-1,0,1,\ldots,\frac{N}{2} \\ \frac{1}{2}\left[1-\cos\left(\frac{2\pi n}{N}\right)\right] & n = 0,1,2,\ldots,N-1 \end{cases}$$

KaiserBessel20, KaiserBessel25, KaiserBessel30:

$$w(n) = \frac{I_0\left(\pi\alpha\sqrt{1-\left(\frac{2n}{N}\right)^2}\right)}{I_0(\pi\alpha)} \qquad 0 \le |n| \le \frac{N}{2}.$$

where I_0 is the zero-order modified Bessel function of the first kind.

windowKind	***α***
KaiserBessel20:	α = 2.
KaiserBessel25:	α = 2.5.
KaiserBessel30:	α = 3.

Parzen:

$$w(n) = 1-\left|\frac{2n}{N}\right|^2 \qquad 0 \le |n| \le \frac{N}{2}.$$

Poisson2, Poisson3, Poisson4:

$$w(n) = \exp\left(-\alpha\frac{2|n|}{N}\right) \qquad 0 \le |n| \le \frac{N}{2}.$$

windowKind	***α***
Poisson2:	α = 2.
Poisson3:	α = 3.
Poisson4:	α = 4.

Riemann:

$$w(n) = \frac{\sin\left(\frac{2\pi n}{N}\right)}{\left(\frac{2\pi n}{N}\right)} \qquad 0 \le |n| \le \frac{N}{2}.$$

See Also

See **Fourier Transforms** on page III-239 for discussion. The inverse operation is **IFFT**.

Spectral Windowing on page III-244. For 2D windowing see **ImageWindow**. Also the **Hanning** window operation.

Also see the **DWT** operation for the discrete wavelet transform and the **CWT** operation for the continuous wavelet transform. The **HilbertTransform** and **WignerTransform** operations.

The **Unwrap**, **MatrixOp**, **DSPPeriodogram** and **LombPeriodogram** operations.

References

For more information about the use of window functions see:

Harris, F.J., On the use of windows for harmonic analysis with the discrete Fourier Transform, *Proc, IEEE*, *66*, 51-83, 1978.

FGetPos

`FGetPos` ***`refNum`***

The FGetPos operation returns the file position for a file.

FGetPos is a faster alternative to FStatus if the only thing you are interested in is the file position.

The FGetPos operation was added in Igor Pro 7.00.

Parameters

refNum is a file reference number obtained from the **Open** operation.

Details

FGetPos supports very big files theoretically up to about 4.5E15 bytes in length.

FGetPos sets the following variables:

`V_flag`	Nonzero (true) if *refNum* is valid.
`V_filePos`	Current file position for the file in bytes from the start.

See Also

Open, **FSetPos**, **FStatus**

FIFO2Wave

`FIFO2Wave [/R/S]` ***`FIFOName, channelName, waveName`***

The FIFO2Wave operation copies FIFO data from the specified channel of the named FIFO into the named wave. FIFOs are used for data acquisition.

Flags

/R=[*startPoint*,*endPoint*]	Dumps the specified FIFO points into the wave.	
/S=*s*	Controls the wave's X scaling and number type:	
	s=0:	Same as no /S.
	s=1:	Sets the wave's X scaling x_0 value to the number of the first sample in the FIFO.
	s=2:	Changes the wave's number type to match the FIFO channel's type.
	s=3:	Combination of *s*=1 and *s*=2.

Details

The FIFO must be in the valid state for FIFO2Wave to work. When you create a FIFO, using NewFIFO, it is initially invalid. It becomes valid when you issue the start command via the CtrlFIFO operation. It remains valid until you change a FIFO parameter using CtrlFIFO.

If you specify a range of FIFO data points, using /R=[*startPoint*,*endPoint*] then FIFO2Wave dumps the specified FIFO points into the wave after clipping *startPoint* and *endPoint* to valid point numbers.

The valid point numbers depend on whether the FIFO is running and on whether or not it is attached to a file. If the FIFO is running then *startPoint* and *endPoint* are truncated to number of points in the FIFO. If the FIFO is buffering a file then the range can include the full extent of the file.

If you specify no range then FIFO2Wave transfers the most recently acquired FIFO data to the wave. The number of points transferred is the smaller of the number of points in the FIFO and number of points in the wave.

FIFO2Wave may or may not change the wave's X scaling and number type, depending on the current X scaling and on the /S flag.

Think of the wave's X scaling as being controlled by two values, x_0 and dx, where the X value of point p is x_0 + p*dx. FIFO2Wave always sets the wave's dx value equal to the FIFO's deltaT value (as set by the CtrlFIFO operation). If you use no /S flag, FIFO2Wave does not set the wave's x_0 value nor does it set the wave's number type.

If you are using FIFO2Wave to update a wave in a graph as quickly as possible, the /S=0 flag gives the highest update rate. The other /S values trigger more recalculation and slow down the updating.

If the wave's number type (possibly changed to match the FIFO channel) is a floating point type, FIFO2Wave scales the FIFO data before transferring it to the wave as follows:

```
scaled_value = (FIFO_value - offset) * gain
```

If the FIFO channel's gain is one and its offset is zero, the scaling would have no effect so FIFO2Wave skips it.

If the specified FIFO channel is an image strip channel (one defined using the optional vectPnts parameter to NewFIFOChan), then the resultant wave will be a matrix with the number of rows set by vectPnts and the number of columns set by the number of points described above for one-dimensional waves. To create an image plot that looks the same as the corresponding channel in a Chart, you will need to transpose the wave using **MatrixTranspose**.

See Also

The **NewFIFO** and **CtrlFIFO** operations, and **FIFOs and Charts** on page IV-291 for more information on FIFOs and data acquisition. For an explanation of waves and wave scaling, see **Changing Dimension and Data Scaling** on page II-63.

FIFOStatus

`FIFOStatus [/Q]` ***`FIFOName`***

The FIFOStatus operation returns miscellaneous information about a FIFO and its channels. FIFOs are used for data acquisition.

Flags

/Q	Doesn't print in the history area.

Details

FIFOStatus sets the variable V_flag to nonzero if a FIFO of the given name exists. If the named FIFO does exist then FIFOStatus stores information about the FIFO in the following variables:

`V_FIFORunning`	Nonzero if FIFO is running.
`V_FIFOChunks`	Number of chunks of data placed in FIFO so far.
`V_FIFOnchans`	Number of channels in the FIFO.
`S_Info`	Keyword-packed information string.

The keyword-packed information string consists of a sequence of sections with the following form: *keyword*:*value*;

You can pick a value out of a keyword-packed string using the **NumberByKey** and **StringByKey** functions. Here are the keywords for S_Info:

In addition, FIFOStatus writes fields to S_Info for each channel in the FIFO. The keyword for the field is a combination of a name and a number that identify the field and the channel to which it refers. For example, if channel 4 is named "Pressure" then the following would appear in the S_Info string: NAME4:Pressure.

In the following table, the channel's number is represented by "#".

Keyword	Type	Meaning
DATE	Number	The date/time when start was issued via CtrlFIFO.
DELTAT	Number	The FIFO's deltaT value as set by CtrlFIFO.
DISKTOT	Number	Current number of chunks written to the FIFO's file.
FILENUM	Number	The output file refNum or review file refNum as set by CtrlFIFO. This will be zero if the FIFO is connected to no file.
NOTE	String	The FIFO's note string as set by CtrlFIFO.
VALID	Number	Zero if FIFO is not valid.

Keyword	Type	Meaning
FSMINUS#	Number	Channel's minus full scale value as set by NewFIFOChan.
FSPLUS#	Number	Channel's plus full scale value as set by NewFIFOChan.
GAIN#	Number	Channel's gain value as set by NewFIFOChan.
NAME#	String	Name of channel.
OFFSET#	Number	Channel's offset value as set by NewFIFOChan.
UNITS#	String	Channel's units as set by NewFIFOChan.

See Also

The **NewFIFO**, **CtrlFIFO**, and **NewFIFOChan** operations, **FIFOs and Charts** on page IV-291 for more information on FIFOs and data acquisition.

The **NumberByKey** and **StringByKey** functions for parsing keyword-value strings.

FilterFIR

`FilterFIR [`*`flags`*`]`*`waveName`*` [, `*`waveName`*`]…`

The FilterFIR operation convolves each *waveName* with automatically-designed filter coefficients or with *coefsWaveName* using time-domain methods.

The automatically-designed filter coefficients are simple lowpass and highpass window-based filters or a maximally-flat notch filter. Multiple filter designs are combined into a composite filter. The filter can be optionally placed into the first *waveName* or just used to filter the data in *waveName*.

FilterFIR filters data faster than **Convolve** when there are many fewer filter coefficient values than data points in *waveName*.

Note: FilterFIR replaces the obsolete **SmoothCustom** operation.

Parameters

waveName is a destination wave that is overwritten by the convolution of itself and the filter.

waveName may be multidimensional, but only one dimension selected by /DIM is filtered (for two-dimensional filtering, see **MatrixFilter**).

If *waveName* is complex, the real and imaginary parts are filtered independently.

Flags

/COEF [=*coefsWaveName*]

Replaces the first output *waveName* by the filter coefficients instead of the filtered results or, when *coefsWaveName* is specified, replaces the output wave(s) by the result of convolving *waveName* with coefficients in *coefsWaveName*.

coefsWaveName must not be one of the destination *waveName*s. It must be single- or double-precision numeric and one-dimensional.

To avoid shifting the output with respect to the input, *coefsWaveName* must have an odd length with the "center" coefficient in the middle of the wave.

The coefficients are usually symmetrical about the middle point, but FilterFIR does not enforce this.

/DIM=*d*

Specifies the wave dimension to filter.

d=-1:	Treats entire wave as 1D (default).
d=0:	Operates along rows.
d=1:	Operates along columns.
d=2:	Operates along layers.
d=3:	Operates along chunks.

Use /DIM=0 to apply the filter to each individual column (each one a channel, say left and right) in a multidimensional *waveName* where each row comprises all of the sound samples at a particular time.

/E=*endEffect*

Determines how the ends of the wave (*w*) are handled when fabricating missing neighbor values. *endEffect* has values:

0:	Bounce method (default). Uses w[*i*] in place of the missing w[-*i*] and w[*n*-*i*] in place of the missing w[*n*+*i*].
1:	Wrap method. Uses w[*n*-*i*] in place of the missing w[-*i*] and vice versa.
2:	Zero method. Uses 0 for any missing value.
3:	Fill method. Uses w[0] in place of the missing w[-*i*] and w[*n*] in place of the missing w[*n*+*i*].

/HI={*f1*, *f2*, *n*}

Creates a high-pass filter based on the windowing method, using the Hanning window unless another window is specified by /WINF.

f1 and *f2* are filter design frequencies measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).

f1 is the end of the reject band, and *f2* is the start of the pass band:

$0 < f1 < f2 < 0.5$

n is the number of FIR filter coefficients to generate. A larger number gives better stop-band rejection. A good number to start with is 101.

Use both /HI and /LO to create a bandpass filter.

/LO={*f1*, *f2*, *n*}

Creates a low-pass filter. *f1* is the end of the pass band, *f2* is the start of the reject band, and *n* is the number of FIR filter coefficients. See /HI for more details.

/NMF={*fc*, *fw* [, *eps*, *nMult*]}

Creates a maximally-flat notch filter centered at *fc* with a -3dB width of *fw*. *fc* and *fw* are filter design frequencies measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).

The longest filter length allowed is 4001 points, which requires *fw* >= 0.0079 (1.58% of the sampling frequency).

The longest filter length allowed is 2,147,483,647 points, which requires *fw* >= 1.07644e-05 (0.00107644 % of the sampling frequency).

Coefficients at the ends that are smaller than the optional *eps* parameter are removed, making the filter shorter (and faster), though less accurate. The default is 2^{-40}. Use 0 to retain all coefficients, no matter how small, even zero coefficients.

nMult specifies how much longer the filter may be to obtain the most accurate notch frequency. The default is 2 (potentially twice as many coefficients). Set *nMult* <= 1 to generate the shortest possible filter.

The maximally flat notch filter design is based on Zahradník and Vlcek, and uses arbitrary precision math (see **APMath**) to compute the coefficients.

/WINF=*windowKind*

Applies the named “window” to the filter coefficients. Windows alter the frequency response of the filter in obvious and subtle ways, enhancing the stop-band rejection or steepening the transition region between passed and rejected frequencies. They matter less when many filter coefficients are used.

If /WINF is not specified, the Hanning window is used. For no coefficient filtering, use /WINF=None.

Choices for *windowKind* are:

Bartlett, Blackman367, Blackman361, Blackman492, Blackman474, Cos1, Cos2, Cos3, Cos4, Hamming, Hanning, KaiserBessel20, KaiserBessel25, KaiserBessel30, Parzen, Poisson2, Poisson3, Poisson4, and Riemann.

See **FFT** for window equations and details.

Details

If *coefsWaveName* is specified, then /HI, /LO, and /NMF are ignored.

If more than one of /HI, /LO, and /NMF are specified, the filters are combined using linear convolution. The length of the combined filter is slightly less than the sum of the individual filter lengths.

The filtering convolution is performed in the time-domain. That is, the FFT is not employed to filter the data. For this reason the coefficients length should be small in comparison to the destination waves.

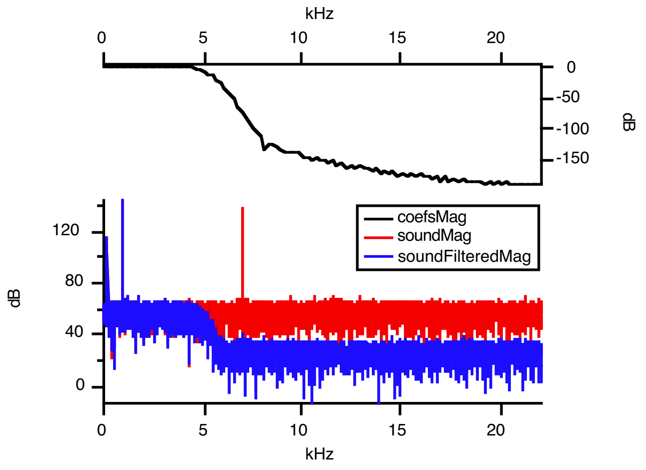
FilterFIR assumes that the middle point of *coefsWaveName* corresponds to the delay = 0 point. The “middle” point number = trunc(numpnts(*coefsWaveName* -1)/2). *coefsWaveName* usually contains the two-sided impulse response of a filter, and usually contains an odd number of points. This is the kind of coefficients data generated by /HI, /LO, and /NMF.

FilterFIR ignores the X scaling of all waves, except when /COEF creates a coefficients wave, which preserves the X scale deltax and alters the leftx value so that the zero-phase (center) coefficient is located at x=0.

Examples

```
// Make test sound from three sine waves
Variable/G fs= 44100                              // Sampling frequency
Variable/G seconds= 0.5                           // Duration
Variable/G n= 2*round(seconds*fs/2)
Make/O/W/N=(n) sound                              // 16-bit integer sound wave
SetScale/p x, 0, 1/fs, "s", sound
Variable/G f1= 200, f2= 1000, f3= 7000
Variable/G a1=100, a2=3000,a3=1500
sound= a1*sin(2*pi*f1*x)
sound += a2*sin(2*pi*f2*x)
sound += a3*sin(2*pi*f3*x)+gnoise(10)             // Add a noise floor
```

```
// Compute the sound's spectrum in dB
FFT/MAG/WINF=Hanning/DEST=soundMag sound
soundMag= 20*log(soundMag)
SetScale d, 0, 0, "dB", soundMag

// Apply a 5 kHz low-pass filter to the sound wave
Duplicate/O sound, soundFiltered
FilterFIR/E=3/LO={4000/fs, 6000/fs, 101} soundFiltered

// Compute the filtered sound's spectrum in dB
FFT/MAG/WINF=Hanning/DEST=soundFilteredMag soundFiltered
soundFilteredMag= 20*log(soundFilteredMag)
SetScale d, 0, 0, "dB", soundFilteredMag

// Compute the filter's frequency response in dB
Make/O/D/N=0 coefs                                  // Double precision is recommended
SetScale/p x, 0, 1/fs, "s", coefs
FilterFIR/COEF/LO={4000/fs, 6000/fs, 101} coefs
FFT/MAG/WINF=Hanning/PAD={(2*numpnts(coefs))}/DEST=coefsMag coefs
coefsMag= 20*log(coefsMag)
SetScale d, 0, 0, "dB", coefsMag

// Graph the frequency responses
Display/R/T coefsMag as "FIR Lowpass Example";DelayUpdate
AppendToGraph soundMag, soundFilteredMag;DelayUpdate
ModifyGraph axisEnab(left)={0,0.6}, axisEnab(right)={0.65,1}
ModifyGraph rgb(soundFilteredMag)=(0,0,65535), rgb(coefsMag)=(0,0,0)
Legend
```



```
// Graph the unfiltered and filtered sound time responses
Display/L=leftSound sound as "FIR Filtered Sound";DelayUpdate
AppendToGraph/L=leftFiltered soundFiltered;DelayUpdate
ModifyGraph axisEnab(leftSound)={0,0.45}, axisEnab(leftFiltered)={0.55,1}
ModifyGraph rgb(soundFiltered)=(0,0,65535)

// Listen to the sounds
PlaySound sound                // This has a very high frequency tone
PlaySound soundFiltered        // This doesn't
```

References

Zahradník, P., and M. Vlcek, Fast Analytical Design Algorithms for FIR Notch Filters, *IEEE Trans. on Circuits and Systems*, *51*, 608 - 623, 2004.

<http://euler.fd.cvut.cz/publikace/files/vlcek/notch.pdf>

See Also

Smoothing on page III-261; the **Smooth**, **Convolve**, **MatrixConvolve**, and **MatrixFilter** operations.

FilterIIR

`FilterIIR` [***`flags`***] [***`waveName`***,…]

The FilterIIR operation applies to each *waveName* either the automatically-designed IIR filter coefficients or the IIR filter coefficients in *coefsWaveName*. Multiple filter designs are combined into a composite filter. The filter can be optionally placed into the first *waveName* or just used to filter the data in *waveName*.

The automatically-designed filter coefficients are bilinear transforms of the Butterworth analog prototype with an optional variable-width notch filter.

To design more advanced IIR filters, see **Designing the IIR Coefficients**.

Parameters

waveName may be multidimensional, but only the one dimension selected by /DIM is filtered (for two-dimensional filtering, see **MatrixFilter**).

waveName may be omitted for the purpose of checking the format of *coefsWaveName*. If the format is detectably incorrect an error code will be returned in V_flag. Use /Z to prevent command execution from stopping.

Flags

/CASC	Specifies that *coefsWaveName* contains cascaded bi-quad filter coefficients. The cascade implementation is more stable and numerically accurate for high-order IIR filtering than Direct Form 1 filtering. See **Cascade Details**.
/COEF [=*coefsWaveName*]	
	Replaces the first output *waveName* by the filter coefficients instead of the filtered results or, when *coefsWaveName* is specified, replaces the output wave(s) by the result of filtering *waveName* with the IIR coefficients in *coefsWaveName*. *coefsWaveName* must not be one of the destination *waveName*s. It must be single- or double-precision numeric and two-dimensional. When used with /CASC, *coefsWaveName* must have 6 columns, containing real-valued coefficients for a product of ratios of second-order polynomials (cascaded bi-quad sections). If /ZP is specified, it must be complex, otherwise it must be real. See **Details** for the format of the coefficients in *coefsWaveName*.
/DIM=*d*	Specifies the wave dimension to filter. *d*=-1: Treats entire wave as 1D (default). *d*=0: Operates along rows. *d*=1: Operates along columns. *d*=2: Operates along layers. *d*=3: Operates along chunks. Use /DIM=0 to apply the filter to each individual column (each one a channel, say left and right) in a multidimensional *waveName* where each row comprises all of the sound samples at a particular time.
/HI=*fHigh*	Creates a high-pass Butterworth filter with the -3dB corner at *fHigh*. The order of the filter is controlled by the /ORD flag. *fHigh* is a filter design frequency measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency).
/LO=*fLow*	Creates a low-pass Butterworth filter with the -3dB corner at *fLow*. The /ORD flag controls the order of the filter. *fLow* is a filter design frequency measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency). Create bandpass and bandreject filters by specifying both /HI and /LO. For a bandpass filter, set *fLow* > *fHigh*, and for a band reject filter, set *fLow* < *fHigh*.

/N={*fNotch*, *notchQ*}	Creates a notch filter with the center frequency at *fNotch* and a -3dB width of *fNotch*/*notchQ*. *fNotch* is a filter design frequency measured in fractions of the sampling frequency, and may not exceed 0.5 (the normalized Nyquist frequency). *notchQ* is a number greater than 1, typically 10 to 100. Large values produce a filter that "rings" a lot.
/ORD=*order*	Sets the order of the Butterworth filter(s) created by /HI and /LO. The default is 2 (second order), and the maximum is 100.
/Z=*z*	Prevents procedure execution from aborting when an error occurs. Use /Z=1 to handle this case in your procedures using **GetRTError**(1) rather than having execution abort. /Z=0 is the same as no /Z at all.
/ZP	Specifies that *coefsWaveName* contains complex z-domain zeros (in column 0) and poles (in column 1) or, if *coefsWaveName* is not specified, that the first output *waveName* is to be replaced by filter coefficients in the zero-pole format. See **Zeros and Poles Details**.

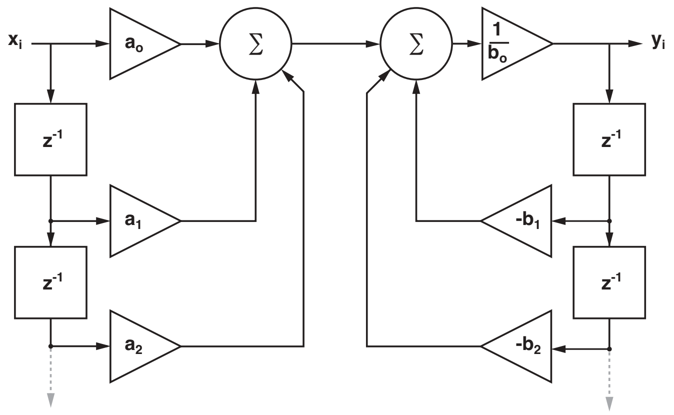
Details

FilterIIR sets V_flag to 0 on success or to an error code if an error occurred. Command execution stops if an error occurs unless the /Z flag is set. Omit /Z and call **GetRTError** and **GetRTErrMessage** under similar circumstances to see what the error code means.

Direct Form 1 Details

Unless /CASC or /ZP are specified, the coefficients in *coefsWaveName* describe a ratio of two polynomials of the Z transform:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2} + \ldots}{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots}$$



Direct Form I Implementation

$$y_i = \frac{a_0 x_i + a_1 x_{i-1} + a_2 x_{i-2} + \ldots - b_1 y_{i-1} - b_2 y_{i-2} + \ldots}{b_0}$$

where x is the input wave *waveName* and y is the output wave (either *waveName* again or *destWaveName*).

FilterIIR computes the filtered result using the Direct Form I implementation of $H(z)$.

The rational polynomial numerator (a_i) coefficients in are column 0 and denominator (b_i) coefficients in column 1 of *coefsWaveName*.

The coefficients in row 0 are the nondelayed coefficients a_0 (in column 0) and b_0 (in column 1).

The coefficients in row 1 are the z^{-1} coefficients, a_1 and b_1.

The coefficients in row n are the z^{-n} coefficients, a_n and b_n.

The number of coefficients for the numerator can differ from the number of coefficients for the denominator. In this case, specify 0 for unused coefficients.

Note: If all the coefficients of the denominator are 0 (b_i= 0 except b_0 = 1), then the filter is actually a causal FIR filter (Finite Impulse Response filter with delay of *n*-1). In this sense, FilterIIR implements a superset of the FilterFIR operation.

Alternate Direct Form 1 Notation

The designation of a_i, etc. as the numerator is at odds with many textbooks such as *Digital Signal Processing*, which uses *b* for the numerator coefficients of the rational function, *a* for the denominator coefficients with an implicit a_0 = 1, in addition to reversing the signs of the remaining denominator coefficients so that they can write *H*(*z*) as:

$$H(z)=\frac{Y(z)}{X(z)}=\frac{\sum_{i=0}^{n} b_i z^{-i}}{1-\sum_{i=1}^{n} a_i z^{-i}}.$$

Coefficients derived using this notation need their denominator coefficients sign-reversed before putting them into rows 1 through n of column 1 (the second column), and the "missing" nondelayed denominator coefficient of 1.0 placed in row 0, column 1.

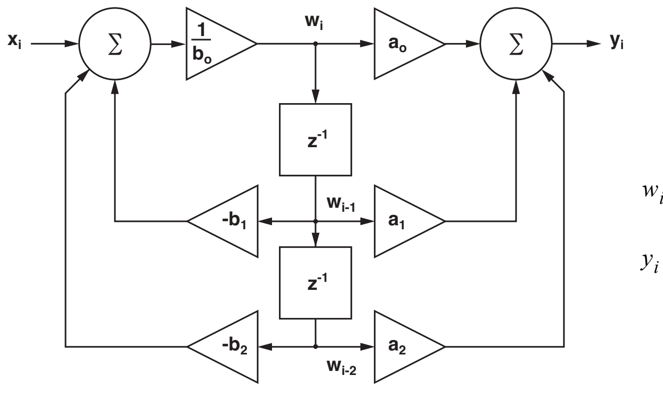
Cascade Details

When using /CASC, coefficients in *coefsWaveName* describe the product of one or more ratios of two quadratic polynomials of the Z transform:

$$H(z)=\frac{Y(z)}{X(z)}=\prod_{k=1}^{K}\frac{a_{0_k}+a_{1_k}z^{-1}+a_{2_k}z^{-2}}{b_{0_k}+b_{1_k}z^{-1}+b_{2_k}z^{-2}}.$$

Each product term implements a "cascaded bi-quad section", and *H*(*z*) can be realized by feeding the output of one section to the next one.

The cascade coefficients filter the data using a Direct Form II cascade implementation:



$$w_i = \frac{x_i - b_1 w_{i-1} - b_2 w_{i-2}}{b_0}$$

$$y_i = a_0 w_i + a_1 w_{i-1} + a_2 w_{i-2}$$

Cascaded Bi-Quad Direct Form II Implementation

The cascade implementation is more stable and numerically accurate for high-order IIR filtering than Direct Form I filtering. Cascade IIR filtering is recommended when the filter order exceeds 16 (a 16th-order Direct Form I filter has 17 numerator coefficients and 17 denominator coefficients).

coefsWaveName must be a six-column real-valued numeric wave. Each row describes one bi-quad section. The coefficients for the second term (or "section") of the product (k=2) are in the following row, etc.:

k	Row	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5
1	0	a_{0_1}	a_{1_1}	a_{2_1}	b_{0_1}	b_{1_1}	b_{2_1}
2	1	a_{0_2}	a_{1_2}	a_{2_2}	b_{0_2}	b_{1_2}	b_{2_2}
	…						

The number of coefficients for the numerator (a's) is allowed to differ from the number of coefficients for the denominator (b's). In this case, specify 0 for unused coefficients.

For example, a third order filter (three poles and three zeros) cascade implementation is a single-order section combined with a second order section. The values for a_{2_k}, b_{2_k} for that section (k) would be 0. Here the second section is specified as the first-order section:

k	Row	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5
1	0	a_{0_1}	a_{1_1}	a_{2_1}	b_{0_1}	b_{1_1}	b_{2_1}
2	1	a_{0_2}	a_{1_2}	0	b_{0_2}	b_{1_2}	0

Alternate Cascade Notation

In the DSP literature, the b_{0_k} gain values are typically one and the $H(z)$ expression contains an overall gain value, usually K. Here each product term (or "section") has a user-settable gain value. Computing the correct gain values to control overflow in integer implementations is the responsibility of the user. For floating implementations, you might as well set all b_{0_k} values to one except, say, b_{0_1}, to control the overall gain.

Zeros and Poles Details

When using /ZP, coefficients in *coefsWaveName* contains complex zeros and poles in the (also complex) Z transform domain:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{(z-z_0)(z-z_1)(z-z_2)\ldots}{(z-p_0)(z-p_1)(z-p_2)\ldots}$$

coefsWaveName must be a two-column complex wave with zero0, zero1,… zeroN in the first column of N+1 rows, and pole0, pole1,… poleN in the second column of those same rows:

k	Row	Col 0	Col 1
1	0	(zero0Real, zero0Imag)	(pole0Real, pole0Imag)
2	1	(zero1Real, zero1Imag)	(pole1Real, pole1Imag)
3	2	(zero2Real, zero2Imag)	(pole2Real, pole2Imag)
	…		

If a zero or pole has a nonzero imaginary component, the conjugate zero or pole must be included in *coefsWaveName*. For example, if a zero is placed at (0.7, 0.5), the conjugate is (0.7, -0.5), and that value must also appear in column 0. These two zeros form what is known as a "conjugate pair". The conjugate values must match within the greater of 1.0e-6 or 1.0e-6 * |zeroOrPole|.

Use (0,0) for unused poles or zeros, as a zero or pole at z= (0,0) has no effect on the filter frequency response.

The /ZP format for the coefficients is internally converted into the Direct Form 1 implementation, or into the Cascade Direct Form 2 implementation if /CASC is specified. There is no option for returning these implementation-dependent coefficients in a wave.

Designing the IIR Coefficients

Simple IIR filters can be used or created by specifying the /LO, /HI, /ORD, /N, /CASC, and /ZP flags. Use /COEF without *coefsWaveName* to put these simple IIR filter coefficients into the first *waveName*.

More advanced IIR filters (Bessel, Chebyshev) can be designed using the separate IFDL package. IFDL is a suite of extensions and macros that you use to design FIR (Finite Impulse Response) and IIR (Infinite Impulse Response) filters and to apply them to your data. The IIR design software creates IIR coefficients based on bilinear transforms of analog prototype filters such as Bessel, Butterworth, and Chebyshev. See the WaveMetrics web site for more about IFDL.

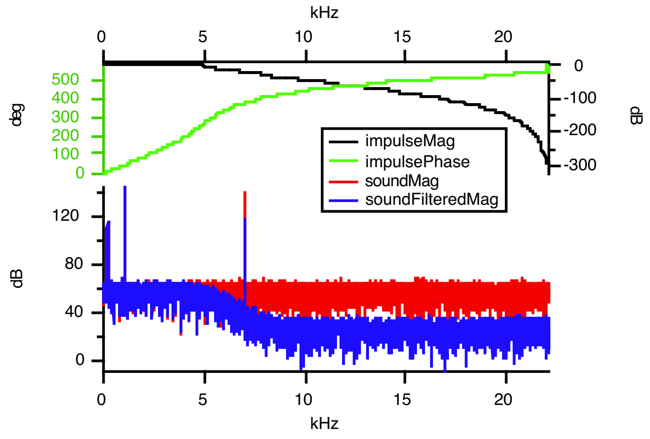
Even without IFDL, you can create custom IIR filters by manually placing poles and zeros in the Z plane using the Pole and Zero Filter Design procedures. Copy the following line to your Procedure window and click the Compile button at the bottom of the procedure window:

```
#include <Pole And Zero Filter Design>
```

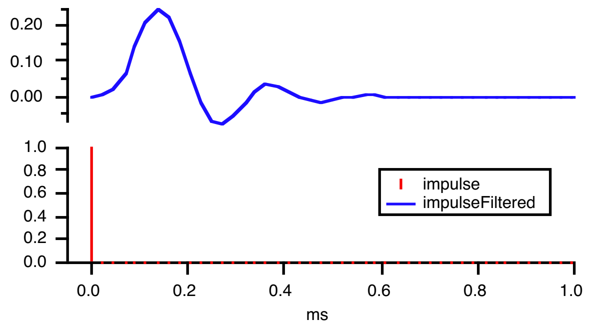
Then choose Pole and Zero Filter Design from the Analysis menu.

Examples

```
// Make test sound from three sine waves
Variable/G fs= 44100                          // Sampling frequency
Variable/G seconds= 0.5                       // Duration
Variable/G n= 2*round(seconds*fs/2)
Make/O/W/N=(n) sound                          // 16-bit integer sound wave
SetScale/p x, 0, 1/fs, "s", sound
Variable/G f1= 200, f2= 1000, f3= 7000
Variable/G a1=100, a2=3000,a3=1500
sound= a1*sin(2*pi*f1*x)
sound += a2*sin(2*pi*f2*x)
sound += a3*sin(2*pi*f3*x)+gnoise(10)            // Add a noise floor

// Compute the sound's spectrum in dB
FFT/MAG/WINF=Hanning/DEST=soundMag sound
soundMag= 20*log(soundMag)
SetScale d, 0, 0, "dB", soundMag

// Apply a 5 kHz, 6th order low-pass filter to the sound wave
Duplicate/O sound, soundFiltered
FilterIIR/LO=(5000/fs)/ORD=6 soundFiltered       // Second order by default

// Compute the filtered sound's spectrum in dB
FFT/MAG/WINF=Hanning/DEST=soundFilteredMag soundFiltered
soundFilteredMag= 20*log(soundFilteredMag)
SetScale d, 0, 0, "dB", soundFilteredMag

// Compute the filter's frequency and phase by filtering an impulse
Make/O/D/N=2048 impulse= p==0                    // Impulse at t==0
SetScale/P x, 0, 1/fs, "s", impulse
Duplicate/O impulse, impulseFiltered
FilterIIR/LO=(5000/fs)/ORD=6 impulseFiltered
FFT/MAG/DEST=impulseMag impulseFiltered
impulseMag= 20*log(impulseMag)
SetScale d, 0, 0, "dB", impulseMag
FFT/OUT=5/DEST=impulsePhase impulseFiltered
impulsePhase *= 180/pi                           // Convert to degrees
SetScale d, 0, 0, "deg", impulsePhase
Unwrap 360, impulsePhase                         // Continuous phase

// Graph the frequency responses
Display/R/T impulseMag as "IIR Lowpass Example"
AppendToGraph/L=phase/T impulsePhase
AppendToGraph soundMag, soundFilteredMag
ModifyGraph axisEnab(left)={0,0.6}
ModifyGraph axisEnab(right)={0.65,1}
ModifyGraph axisEnab(phase)={0.65,1}
ModifyGraph freePos=0, lblPos=60, rgb(soundFilteredMag)=(0,0,65535)
ModifyGraph rgb(impulseMag)=(0,0,0), rgb(impulsePhase)=(0,65535,0)
ModifyGraph axRGB(phase)=(3,52428,1), tlblRGB(phase)=(3,52428,1)
Legend
```



```
// Graph the unfiltered and filtered impulse time responses
Display/L=leftImpulse impulse as "IIR Filtered Impulse"
AppendToGraph/L=leftFiltered impulseFiltered
ModifyGraph axisEnab(leftImpulse)={0,0.45}, axisEnab(leftFiltered)={0.55,1}
ModifyGraph freePos=0, margin(left)=50
ModifyGraph mode(impulse)=1, rgb(impulseFiltered)=(0,0,65535)
SetAxis bottom -0.00005,0.001
Legend
```



```
// Listen to the sounds
PlaySound sound                    // This has a very high frequency tone
PlaySound soundFiltered            // This doesn't
```

References

Embree, P.M., and B. Kimble, *C Language Algorithms for Signal Processing*, 456 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Lynn, P.A., and W. Fuerst, *Introductory Digital Signal Processing with Computer Applications*, 479 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1998.

Oppenheim, A.V., and R.W. Schafer, *Digital Signal Processing*, 585 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1975.

Terrell, T.J., *Introduction to Digital Filters*, 2nd ed., 261 pp., John Wiley & Sons, New York, 1988.

See Also

Smoothing on page III-261; the **FFT** and **FilterFIR** operations.

FindContour

`FindContour [`*`flags`*`] `*`matrixWave, level`*

The FindContour operation creates an XY pair of waves representing the locus of the solution to *matrixWave=level* .

The FindContour operation was added in Igor Pro 7.00.

Flags

/DSTX=*destX*	Saves the output X data in the specified destination wave. The destination wave is created or overwritten if it already exists.
/DSTY=*destY*	Saves the output Y data in the specified destination wave. The destination wave is created or overwritten if it already exists.

Details

FindContour uses a contour-following algorithm to generate a pair of waves describing the locus of the solution to *matrixWave*=*level*.

If you omit /DSTX the output X data is written to W_XContour in the current data folder.

If you omit /DSTY the output Y data is written to W_YContour in the current data folder.

The output waves are written as double-precision floating point. They use NaNs to separate different contiguous solution points.

Example

```
Make/N=(100,200) dataWave = 1e4*gauss(x,50,10,y,100,20)
FindContour dataWave,4     // Find solution to dataWave=4
NewImage dataWave
AppendToGraph/T W_YContour vs W_XContour
```

See Also

AppendMatrixContour, **ContourZ**

FindDimLabel

FindDimLabel(*waveName*, *dimNumber*, *labelString*)

Returns the index value corresponding to the label for the given dimension. Returns -1 if the label is for the entire dimension. Returns -2 if the label is not found.

Use *dimNumber* =0 for rows, 1 for columns, 2 for layers, or 3 for chunks.

See Also

GetDimLabel, **SetDimLabel**

FindDuplicates

FindDuplicates [*flags*] *srcWave*

The FindDuplicates operation identifies duplicate values in a wave and optionally creates various output waves. *srcWave* can be either numeric or text.

When *srcWave* is numeric, the /DN, /INDX, /RN and /SN flags create output waves as described below. If you omit all of these flags then FindDuplicates does nothing.

When *srcWave* is text, the /DT, /INDX, /RT and /ST flags create output waves as described below. If you omit all of these flags then FindDuplicates does nothing.

The FindDuplicates operation was added in Igor Pro 7.

Flags

INDX=*indexWave*	Creates a numeric output wave containing the index of each encountered duplicate. The index is the point number in *srcWave* where a duplicate value was encountered. This flag applies to both numeric and text inputs.
/Z	Do not report any errors.

Flags for Numeric Source Wave

/DN=*dupsWave*	Creates a numeric output wave that contains the duplicates.
/RN=*dupsRemovedWave*	

Creates a numeric output wave that contains the source data with all duplicates removed.

/SN=*replacement* Creates a numeric output wave with all duplicates replaced with *replacement*. *replacement* can be any numeric value including NaN or INF.

The output wave is W_ReplacedDuplicates in the current data folder unless you specify a different output wave using the /SNDS flag.

/SNDS=*dupsReplacedWave*

Specifies the output wave generated by /SN. If you omit /SNDS then the output wave created by /SN is W_ReplacedDuplicates in the current data folder. /SNDS without /SN has no effect.

/TOL=*tolerance* Specifies the tolerance value for single-precision and double-precision numeric source waves.

Two values are considered duplicates if

```
abs(value1-value2) <= tolerance
```

By default *tolerance* is zero.

Flags for Text Source Wave

/DT=*dupsWave* Creates a text output wave that contains the duplicates.

/RT=*dupsRemovedWave*

Creates a text output wave that contains the source data with all duplicates removed.

/ST=*replacementStr* Creates a text output wave with all duplicates replaced with *replacementStr*. *replacementStr* can be any text value including "".

The output wave is T_ReplacedDuplicates in the current data folder unless you specify a different output wave using the /STDS flag.

/STDS=*dupsReplacedWave*

Specifies the output wave generated by /ST. If you omit /STDS then the output wave created by /ST is T_ReplacedDuplicates in the current data folder. /STDS without /ST has no effect.

Details

FindDuplicates scans *srcWave* and identifies duplicate values. The first instance of any value is not considered a duplicate. Duplicates are either identical, as is the case with integer or text waves, or values that are within a specified tolerance in the case of single-precision or double-precision numeric waves.

Text comparison is case-sensitive.

The operation creates wave references for the waves specified by the various flags above. See **Automatic Creation of WAVE References** on page IV-66 for details.

See Also

FindLevels, **FindValue**, **Sort**

FindLevel

`FindLevel [*flags*] *waveName, level*`

The FindLevel operation searches the named wave to find the X value at which the specified Y *level* is crossed.

Flags

/B=*box*	Sets box size for sliding average. If /B=*box* is omitted or *box* equals 1, no averaging is done. If you specify an even box size then the next higher (odd) integer is used. If you use a box size greater than 1, FindLevel will be unable to find a level crossing that occurs in the first or last *(box*-1)/2 points of the wave since these points don't have enough neighbors for computing the derived average wave values.
/EDGE=*e*	Specifies searches for either increasing or decreasing level crossing. *e*=1: Searches only for crossing where Y values are increasing as *level* is crossed from wave start towards wave end. *e*=2: Searches only for crossing where the Y values are decreasing as *level* is crossed from wave start towards wave end. *e*=0: Same as no /EDGE flag (searches for either increasing and decreasing level crossing).
/P	Computes the X crossing location in terms of point number. If /P is omitted, the level crossing location is computed in terms of X values.
/Q	Don't print results in history and don't report error if *level* is not found.
/R=(*startX*,*endX*)	Specifies an X range of the wave to search. You may exchange *startX* and *endX* to reverse the search direction.
/R=[*startP*,*endP*]	Specifies a point range of the wave to search. You may exchange *startP* and *endP* to reverse the search direction. If you specify the range as /R=[*startP*] then the end of the range is taken as the end of the wave. If /R is omitted, the entire wave is searched.
/T=*dx*	Search for two level crossings. *dx* must be less than *minWidthX*, so you must also specify /M if you use /T. (FindLevel limits *dx* so that second search start isn't beyond where the first search for next edge will be.)
/T=*dx*	Performs a second search after finding the initial level crossing. The second search starts *dx* units beyond the initial level crossing and looks back in the direction of the initial crossing. If FindLevel finds a second level crossing, it sets V_LevelX to the average of the initial and second crossings. Otherwise, it sets V_LevelX to the initial crossing.

Details

FindLevel scans through the wave comparing *level* to values derived from the Y values of the wave. Each derived value is a sliding average of the Y values.

FindLevel searches for two derived wave values that straddle *level*. If it finds these values it computes the X value at which *level* is located by linearly interpolating between the straddling Y values.

Note: FindLevel does not locate values exactly equal to *level*; it locates transitions through *level*. See **BinarySearch** for one method of locating exact values.

FindLevel reports its results by setting these variables:

`V_flag`	0: *level* was found. 1: *level* was not found.
`V_LevelX`	Interpolated X value at which *level* was found, or the corresponding point number if /P is specified.
`V_rising`	0: Y values at the crossing are decreasing from wave start towards wave end. 1: Y values at the crossing are increasing.

If you omit the /Q flag then FindLevel also reports its results by printing them in the history area.

If *level* is not found, and if you omit the /Q flag, FindLevel generates an error which puts up an error alert and halts execution of any command line or macro that is in progress.

V_LevelX is returned in terms of the X scaling of the named wave unless you use the /P flag, in which case it is in terms of point number.

The FindLevel operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

See Also

The **EdgeStats**, **FindLevels**, **FindValue**, and **PulseStats** operations and the **BinarySearch** and **BinarySearchInterp** functions.

FindLevels

`FindLevels [`*`flags`*`] `*`waveName, level`*

The FindLevels operation searches the named wave to find one or more X values at which the specified Y *level* is **crossed**.

To find where the wave is equal to a given value, use **FindValue** instead.

Flags

/B=*box*	Sets box size for sliding average. See the **FindLevel** operation.
/D=*destWaveName*	Specifies wave into which FindLevels is to store the level crossing values. If /D and /DEST are omitted, FindLevels creates a wave named W_FindLevels to store the level crossing values in.
/DEST=*destWaveName*	Same as /D. Both /D and /DEST create a real wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-66 for details.
/EDGE=*e*	Specifies searches for either increasing or decreasing level crossing. *e*=1: Searches only for crossings where the Y values are increasing as level is crossed from wave start towards wave end. *e*=2: Searches only for crossings where the Y values are decreasing as level is crossed from wave start towards wave end. *e*=0: Same as no /EDGE flag (searches for both increasing and decreasing level crossings).
/M=*minWidthX*	Sets the minimum X distance between level crossings. This determines where FindLevels searches for the next crossing after it has found a level crossing. The search starts *minWidthX* X units beyond the crossing. The default value for *minWidthX* is 0.
/N=*maxLevels*	Sets a maximum number of crossings that FindLevels is to find. The default value for *maxLevels* is the number of points in the specified range of *waveName*.
/P	Compute crossings in terms of points. See the **FindLevel** operation.
/Q	Doesn't print to history and doesn't abort if no levels are found.
/R=(*startX*,*endX*)	Specifies X range. See the **FindLevel** operation.
/R=[*startP*,*endP*]	Specifies point range. See the **FindLevel** operation.
/T=*dx*	Search for two level crossings. *dx* must be less than *minWidthX*, so you must also specify /M if you use /T. (FindLevels limits *dx* so that second search start isn't beyond where the first search for next edge will be.) See **FindLevel** for more about /T.

Details

The algorithm for finding a level crossing is the same one used by the **FindLevel** operation.

If FindLevels finds *maxLevels* crossings or can not find another level crossing, it stops searching.

FindLevels sets the following variables:

`V_flag`	0: *maxLevels* level crossings were found. 1: At least one but less than *maxLevels* level crossings were found. 2: No level crossings were found.
`V_LevelsFound`	Number of level crossings found.

Examples

```
Make/O/D/N=0 destWave
FindLevels/Q/D=destWave data, 5
if (V_LevelsFound)
   Print destWave[0]   // First crossing X location

FindLevels/Q/D=destWave data, 5
if (V_LevelsFound)
   Print destWave[0]   // First crossing X location
```

See Also

The **FindLevel** operation for details about the level crossing detection algorithm and the /B, /P, /Q, /R, and /T flag values.

The FindLevels operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

FindListItem

`FindListItem(`*`itemStr, listStr`*`[, `*`listSepStr`*`[, `*`start`*`[, `*`matchCase`*`]]])`

The FindListItem function returns a numeric offset into *listStr* where *itemStr* begins. *listStr* should contain items separated by the *listSepStr* character, such as "abc;def;".

Use FindListItem to locate the start of an item in a string containing a "wave0;wave1;" style list such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

Use WhichListItem to determine the index of an item in the list.

If *itemStr* is not found, if *listStr* is `""`, or if *start* is not within the range of 0 to strlen(*listStr*)-1, then -1 is returned.

listSepStr, *startIndex*, and *matchCase* are optional; their defaults are ";", 0, and 1 respectively.

Details

ItemStr may have any length.

listStr is searched for the first instance of the item string bound by a *listSepStr* on the left and a *listSepStr* on the right. The returned number is the character index where the first character of *itemStr* was found in *listSepStr*.

The search starts from the character position in *listStr* specified by *start*. A value of 0 starts with the first character in *listStr*, which is the default if *start* is not specified.

listString is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *listSepStr* are always case-sensitive. The comparison of *itemStr* to the contents of *listStr* is usually case-sensitive. Setting the optional *matchCase* parameter to 0 makes the comparison case insensitive.

In Igor6, only the first byte of *listSepStr* was used. In Igor7 and later, all bytes are used.

If *startIndex* is specified, then *listSepStr* must also be specified. If *matchCase* is specified, *startIndex* and *listSepStr* must be specified.

Examples

```
Print FindListItem("w1", "w0;w1;w2,")                  // prints 3
Print FindListItem("v2", "v1,v2,v3,", ",")             // prints 3
Print FindListItem("v2", "v0,v2,v2,", ",", 4)          // prints 6
Print FindListItem("C", "a;c;C;")                      // prints 4
Print FindListItem("C", "a;c;C;", ";", 0, 0)           // prints 2
```

See Also

The **AddListItem**, **strsearch**, **StringFromList**, **RemoveListItem**, **RemoveFromList**, **ItemsInList**, **WhichListItem**, **WaveList**, **TraceNameList**, **StringList**, **VariableList**, and **FunctionList** functions.

FindPeak

`FindPeak [`*`flags`*`]` ***`waveName`***

The FindPeak operation searches for a minimum or maximum by analyzing the smoothed first and second derivatives of the named wave. Information about the peak position, amplitude, and width are returned in the output variables.

Flags

Some of the flags have the same meaning as for the FindLevel operation.

/B=*box*	Sets box size for sliding average.
/I	Modify the search criteria to accommodate impulses (peaks of one sample) by requiring only one value to exceed *minLevel*. The default criteria requires that two successive values exceed *minLevel* for a peak to be found (or two successive values be less than the /M level when searching for negative peaks). Impulses can also be found by omitting *minLevel*, in which case /I is superfluous.
/M=*minLevel*	Defines minimum level of a peak. /N changes this to maximum level (see **Details**).
/N	Searches for a negative peak (minimum) rather then a positive peak (maximum).
/P	Location output variables (see **Details**) are reported in terms of (floating point) point numbers. If /P is omitted, they are reported as X values.
/Q	Doesn't print to history and doesn't abort if no peak is found.
/R=(*startX*,*endX*)	Specifies X range and direction for search.
/R=[*startP*,*endP*]	Specifies point range and direction for search.

Details

FindPeak sets the following variables:

`V_flag`	Set only when using the /Q flag. 0: Peak was found. Any nonzero value means the peak was not found.
`V_LeadingEdgeLoc`	Interpolated location of the peak edge closest to *startX* or *startP*. If you use the /P flag, V_LeadingEdgeLoc is a point number rather than to an X value. If the edge was not found, this value is NaN.
`V_PeakLoc`	Interpolated X value at which the peak was found. If you use the /P flag, FindPeak sets V_PeakLoc to a point number rather than to an X value. Set to NaN if peak wasn't found.
`V_PeakVal`	The *approximate* Y value of the found peak. If the peak was not found, this value is NaN (Not a Number).
`V_PeakWidth`	Interpolated peak width. If you use the /P flag, V_PeakWidth is expressed in point numbers rather than as an X value. V_PeakWidth is never negative. If either peak edge was not found, this value is NaN.
`V_TrailingEdgeLoc`	Interpolated location of the peak edge closest to *endX* or *endP*. If you use the /P flag, V_TrailingEdgeLoc is a point number rather than to an X value. If the edge was not found, this value is NaN.

FindPeak computes the sliding average of the input wave using the BoxSmooth algorithm with the *box* parameter. The peak center is found where the derivative of this smoothed result crosses zero. The peak edges are found where the second derivative of the smoothed result crosses zero. Linear interpolation of the derivatives is used to more precisely locate the center and edges. The peak value is simply the greater of the two unsmoothed values surrounding the peak center (if /N, then the lesser value).

FindPeak is not a high-accuracy measurement routine; it is intended as a simple peak-finder. Use the **PulseStats** operation for more precise statistics.

Without /M, a peak is found where the derivative crosses zero, regardless of the peak height.

If you use the /M=*minLevel* flag, FindPeak ignores peaks that are lower than *minLevel* (i.e., the Y value of a found peak will exceed *minLevel*) in the box-smoothed input wave. If /N is also specified (search for minimum), FindPeak ignores peaks whose amplitude is greater than *minLevel* (i.e., the Y value of a found peak will be *less* than *minLevel*).

Without /I, a peak must have two successive values that exceed *minLevel*. Use /I when you are searching for peaks that may have only one value exceeding *minLevel*.

The search for the peak begins at *startX* (or the first point of the wave if /R is not specified), and ends at *endX* (or the last point of the wave if no /R). Searching backwards is permitted, and exchanges the values of V_LeadingEdgeLoc and V_TrailingEdgeLoc.

A simple automatic peak-finder is implemented in the procedure file:

```
#include <Peak AutoFind>
```

one of the `#include <Multi-peak fitting 1.3>` procedures that provides support for Gaussian, Lorentzian, and Voigt fitting functions. See the “Multi-peak fit” example experiment for details (:Examples:Curve Fitting: folder).

The FindPeak operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

See Also

The **PulseStats** operation, the **FindLevel** operation for details about the /B, /P, /Q, and /R flag values.

FindPointsInPoly

`FindPointsInPoly` ***`xWaveName, yWaveName, xPolyWaveName, yPolyWaveName`***

The FindPointsInPoly operation determines if points fall within a certain polygon. It can be used to write procedures that operate on a subset of data identified graphically in a graph.

Details

FindPointsInPoly determines which points in *yWaveName* vs *xWaveName* fall within the polygon defined by *yPolyWaveName* vs *xPolyWaveName*.

xWaveName must have the same number of points as *yWaveName* and *xPolyWaveName* must have the same number of points as *yPolyWaveName*.

FindPointsInPoly creates an output wave named W_inPoly with the same number of points as *xWaveName*. FindPointsInPoly indicates whether the point *yWaveName*[p] vs *xWaveName*[p] falls within the polygon by setting `W_inPoly[p]=1` if it is within the polygon, or `W_inPoly[p]=0` if it is not.

FindPointsInPoly uses integer arithmetic with a precision of about 1 part in 1000. This should be good enough for visually determined (hand-drawn) polygons but might not be sufficient for mathematically generated polygons.

The FindPointsInPoly operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

See Also

The **GraphWaveDraw** operation.

FindRoots

`FindRoots` `[`***`flags`***`]` ***`funcspec, pWave`*** `[,` ***`funcspec, pwave`*** `[, …]]`

`FindRoots /P=`***`PolyCoefsWave`***

The FindRoots operation determines roots or zeros of a specified nonlinear function or system of functions. The function or system of functions must be defined in the form of Igor user procedures.

Using the second form of the command, FindRoots finds all the complex roots of a polynomial with real coefficients. The polynomial coefficients are specified by *PolyCoefsWave*.

Flags for roots of nonlinear functions

/B [= *doBracket*]	Specifies bracketing for roots of a single nonlinear function only.
	doBracket=0: Skips an initial check of the root bracketing values and the possible search for bracketing values. This means that you must provide good bracketing values via the /L and /H flags. See /L and /H flags for details on bracketing of roots. /B alone is the same as /B=0.
	doBracket=1: Uses default root bracketing.
/F=*trustRegion*	Sets the expansion factor of the trust region for the search algorithm when finding roots of systems of functions. Smaller numbers will result in a more stable search, although for some functions larger values will allow the search to zero in on a root more rapidly. Default is 1.0; useful values are usually between 0.1 and 100.
/I=*maxIters*	Sets the maximum number of iterations in searching for a root to *maxIters*. Default is 100.
/L=*lowBracket* /H=*highBracket*	/L and /H are used only when finding roots of a single nonlinear function. *lowBracket* and *highBracket* are X values that bracket a zero crossing of the function. A root is found between the bracketing values.
	If *lowBracket* and *highBracket* are on the same side of zero, it will try to find a minimum or maximum between *lowBracket* and *highBracket*. If it is found, and it is on the other side of zero, Igor will find two roots.
	If *lowBracket* and *highBracket* are on the same side of zero, but no suitable extreme point is found between, it will search outward from these values looking for a zero crossing. If it is found, Igor determines one root.
	If *lowBracket* and *highBracket* are equal, it adds 1.0 to *highBracket* before looking for a zero crossing.
	The default values for *lowBracket* and *highBracket* are zero. Thus, not using either *lowBracket* or *highBracket* is the same as /L=0/H=1.
/Q	Suppresses printout of results in the history area. Ordinarily, the results of root searches are printed in the history.
/T=*tol*	Sets the acceptable accuracy to *tol*. That is, the reported root should be within ±*tol* of the real root.
/X=*xWave* /X={*x1*, *x2*, …}	Sets the starting point for searching for a root of a system of functions. There must be as many X values as functions. The starting point can be specified with a wave having as many points as there are functions, or you can write out a list of X values in braces. If you are finding roots of a single function, use /L and /H instead.
	If you specify a wave, this wave is also used to receive the result of the root search.
/Z=*yValue*	Finds other solutions, that is, places where f(x) = *yValue*. FindRoots usually finds zeroes — places where f(x) = 0.

Flag for roots of polynomials

/P=*PolyCoefsWave*	Specifies a wave containing real polynomial coefficients. With this flag, it finds polynomial roots and does not expect to find user function names on the command line.
	The /P flag causes all other flags to be ignored.
	Use of this flag is not permitted in a thread-safe function.

Parameters

func specifies the name of a user-defined function.

pwave gives the name of a parameter wave that will be passed to your function as the first parameter. It is not modified. It is intended for your private use to pass adjustable constants to your function.

These parameters occur in pairs. For a one-dimensional problem, use a single *func*, *pwave* pair. An N-dimensional problem requires N pairs unless you use the combined function form (see **Combined Format for Systems of Functions**).

Function Format for 1D Nonlinear Functions

Finding roots of a nonlinear function or system of functions requires that you realize the function in the form of an Igor user function of a certain form. In the FindRoots command you then specify the functions with one or more function names paired with parameter wave names. See **Finding Function Roots** on page III-290 for detailed examples.

The functions must have a particular form. If you are finding the roots of a single 1D function, it should look like this:

```
Function myFunc(w,x)
    Wave w
    Variable x

    return f(x)          // an expression …
End
```

Replace "`f(x)`" with an appropriate expression. The FindRoots command might then look like this:

```
FindRoots /L=0 /H=1 myFunc, cw       // cw is a parameter wave for myFunc
```

Function Format for Systems of Multivariate Functions

If you need to find the roots of a system of multidimensional functions, you can use either of two forms. In one form, you provide N functions with N independent variables. You must have a function for each independent variable. For instance, to find the roots of two 2D functions, the functions must have this form:

```
Function myFunc1(w, x1, x2)
    Wave w
    Variable x1, x2

    return f1(x1, x2)       // an expression …
End

Function myFunc2(w, x1, x2)
    Wave w
    Variable x1, x2

    return f2(x1, x2)       // an expression …
End
```

In this case, the FindRoots command might look like this (where `cw1` and `cw2` are parameter waves that must be made before executing FindRoots):

```
FindRoots /X={0,1} myFunc1, cw1, myFunc2, cw2
```

You can also use a wave to pass in the X values. Make sure you have the right number of points in the X wave — it must have N points for a system of N functions.

```
Function myFunc1(w, xW)
    Wave w, xW

    return f1(xW[0], xW[1])        // an expression …
End

Function myFunc2(w, xW)
    Wave w, xW

    return f2(xW[0], xW[1])        // an expression …
End
```

Combined Format for Systems of Functions

For large systems of equations it may get tedious to write a separate function for each equation, and the FindRoots command line will get very long. Instead, you can write it all in one function that returns N Y values through a Y wave. The X values are passed to the function through a wave with N elements. The parameter wave for such a function must have N columns, one column for each equation. The parameters for equation N are stored in column N-1. FindRoots will complain if any of these waves has other than N rows.

Here is an template for such a function:

```
Function myCombinedFunc(w, xW, yW)
    Wave w, xW, yW

    yW[0] = f1(w[0][...], xW[0], xW[1],..., xW[N-1])
    yW[1] = f2(w[1][...], xW[0], xW[1],..., xW[N-1])
```

```
	…
	yW[N-1] = fN(w[N-1][...], xW[0], xW[1],..., xW[N-1])
End
```

When you use this form, you only have one function and parameter wave specification in the FindRoots command:

```
Make/N=(nrows, nequations) paramWave
fill in paramWave with values
Make/N=(number of equations) guessWave
guessWave = {x0, x1, …, xN}
FindRoots /X=guessWave myCombinedFunc, paramWave
```

FindRoots has no idea how many actual equations you have in the function. If it doesn't match the number of rows in your waves, your results will not be what you expect!

Coefficients for Polynomials

To find the roots of a polynomial, you first create a wave with the correct number of points. For a polynomial of degree N, create a wave with N+1 points. For instance, to find roots of a cubic equation you need a four-point wave.

The first point (row zero) of the wave contains the constant coefficient, the second point contains the coefficient for X, the third for X^2, etc.

There is no hard limit on the maximum degree, but note that there are significant numerical problems associated with computations involving high-degree polynomials. Round-off error most likely limits reasonably accurate results to polynomials with degree limited to 20 to 30.

Ultimately, if you are willing to accept very limited accuracy, the numerical problems will result in a failure to converge. In limited testing, we found no failures to converge with polynomials up to at least degree 100. At degree 150, we found occasional failures. At degree 200 the failures were frequent, and at degree 500 we found no successes.

Note that you really can't evaluate a polynomial with such high degree, and we have no idea if the computed roots for a degree-100 polynomial have any practical relationship to the actual roots.

While FindRoots is a thread-safe operation, finding polynomial roots is not. Using FindRoots/P=polyWave in a ThreadSafe function results in a compile error.

Results for Nonlinear Functions and Systems of Functions

The FindRoots operation reports success or failure via the V_flag variable. A nonzero value of V_flag indicates the reason for failure:

`V_flag`=0:	Successful search for a root. Otherwise, the value indicates what went wrong:
`V_flag`=1:	User abort.
`V_flag`=3:	Exceeded maximum allowed iterations
`V_flag`=4:	/T=*tol* was too small. Reported by the root finder for systems of nonlinear functions.
`V_flag`=5:	The search algorithm wasn't making sufficient progress. It may mean that /T=*tol* was set to too low a value, or that the search algorithm has gotten trapped at a false root. Try restarting from a different starting point.
`V_flag`=6:	Unable to bracket a root. Reported when finding roots of single nonlinear functions.
`V_flag`=7:	Fewer roots than expected. Reported by the polynomial root finder. This may indicate that roots were successfully found, but some are doubled. Happens only rarely.
`V_flag`=8:	Decreased degree. Reported by the polynomial root finder. This indicates that one or more of the highest-order coefficients was zero, and a lower degree polynomial was solved.
`V_flag`=9:	Convergence failure or other numerical problem. Reported by the polynomial root finder. This indicates that a numerical problem was detected during the computation. The results are not valid.

The results of finding roots of a single 1D function are put into several variables:

`V_numRoots`	The number of roots found. Either 1 or 2.
`V_Root`	The root.
`V_YatRoot`	The Y value of the function at the root. *Always* check this; some discontinuous functions may give an indication of success, but the Y value at the found root isn't even close to zero.
`V_Root2`	Second root if FindRoots found two roots.
`V_YatRoot2`	The Y value at the second root.

Results for roots of a system of nonlinear functions are reported in waves:

`W_Root`	X values of the root of a system of nonlinear functions. If you used /X=*xWave*, the root is reported in your wave instead.
`W_YatRoot`	The Y values of the functions at the root of a system of nonlinear functions. Only one root is found during a single call to FindRoots.

Roots of a polynomial are reported in a wave:

`W_polyRoots`	A complex wave containing the roots of a polynomial. The number of roots should be equal to the degree of the polynomial, unless a root is doubled.

See Also

Finding Function Roots on page III-290.

The FindRoots operation uses the Jenkins-Traub algorithm for finding roots of polynomials:

Jenkins, M.A., Algorithm 493, Zeros of a Real Polynomial, *ACM Transactions on Mathematical Software*, *1*, 178-189, 1975. Used by permission of ACM (1998).

FindSequence

`FindSequence [`*`flags`*`] `*`srcWave`*

The FindSequence operation finds the location of the specified sequence starting the search from the specified start point. The result of the search stored in V_value is the index of the entry in the wave where the first value is found or -1 if the sequence was not found.

Flags

/FNAN	Specifies searching for a NaN value when srcWave is floating point. This flag was added in Igor Pro 7.00.
/I=*wave*	Specifies an integer sequence wave for integer search.
/M=*val*	If there are repeating entries in the match sequence, *val* is a tolerance value that specifies the maximum difference between the number of repeats. So, for example, if the match sequence is aaabbccc and the *srcWave* contains a sequence aabbcc then the sequence will not be considered a match if *val*=0 but will be considered a match if *val*=1.
/S=*start*	Sets starting point of the search. If /S is not specified, start is 0.
/T=*tolerance*	Defines the tolerance (value ± *tolerance* will be accepted) when comparing floating point numbers.
/U=*uValue*	Specifies the match sequence wave in case of unsigned long range.
/V=*rValue*	Specifies the match sequence wave in the case of single/double precision numbers.
/Z	No error reporting.

Details

If the match sequence is specified via the /V flag, it is considered to be a floating point wave (i.e., single or double precision) in which case it is compared to data in the wave using a tolerance value. If the tolerance is not specified by the /T flag, the default value 1.0^{-7}.

If the match sequence is specified via the /I flag, the sequence is assumed to be an integer wave (this includes both signed and unsigned char, signed and unsigned short as well as long). In this case *srcWave* must also be of integer type and the operation searches for the sequence based on exact equality between the match sequence and entries in the wave as signed long integers.

If the match sequence is unsigned long wave use the /U flag to specify the value for an integer comparison.

You can also use this operation on waves of two or more dimensions. In this case you can calculate the rows, columns, etc. For example, in the case of a 2D wave:

```
col=floor(V_value/rowsInWave)
row=V_value-col*rowsInWave
```

See Also

The **FindValue** operation.

FindValue

`FindValue [`*`flags`*`] `*`srcWave`*

`FindValue [`*`flags`*`] `*`txtWave`*

This operation finds the location of the specified value starting the search from the specified start point. The result of the search stored in V_value is the index of the entry in the wave where the value is found or -1 if not found.

Flags

/I=*ivalue*	Specifies an integer value for integer search.
/S=*start*	Sets start of search in the wave. If /S is not specified, start is set to 0.
/T=*tolerance*	Use this flag when comparing floating point numbers to define a non-negative tolerance such that the specified value ± *tolerance* will be accepted.
/TEXT=*templateString*	Specifies a template string that will be searched for in *txtWave*.
/TXOP=*txOptions*	Specifies the search options using a combination of binary values. 1: Case sensitive 2: Whole word 4: Whole wave element
/U=*uValue*	Specifies the match value in case of unsigned long range.
/V=*rValue*	Specifies the match value in the case of single/double precision numbers. For most purposes you should also use /T to specify the tolerance.
/Z	No error reporting.

Details

If the match value is specified via the /V flag, it is considered to be a floating point value in which case it is compared to data in the wave using a tolerance value. If the tolerance is not specified by the /T flag, the value 10^{-7} is used.

If the match value is specified via the /I flag, the value is assumed to be an integer. In this case *srcWave* must be of integer type and the operation searches for the value based on exact equality between the match value and entries in the wave as signed long integers.

If the match value is unsigned long use the /U flag to specify the value for an integer comparison.

You can also use this operation on waves of two or more dimensions. In this case you can calculate the rows, columns, etc. For example, in the case of a 2D wave:

```
col=floor(V_value/rowsInWave)
row=V_value-col*rowsInWave
```

When searching for text in a text wave the operation creates the variable V_value as above but it also creates the variable V_startPos to specify the position of *templateString* from the start of the particular wave element.

Example

```
Make jack = sin(x/8)       // Single-precision floating point
Display jack

// This prints -1 because 0.5 +/- 1.0E-7 does not occur in wave jack
FindValue /V=.5 jack; Print V_value

// This prints 21 because 0.5 +/- 0.01 does occur in wave jack
FindValue /V=.5 /T=.01 jack; Print V_value

// The value of jack(21), to 6 decimal digits of precision, is 0.493920
Print jack(21)
```

See Also

FindSequence, **FindLevel**, **FindLevels**, **FindDuplicates**

FitFunc

`FitFunc`

Marks a user function as a user-defined curve fit function. By default, only functions marked with this keyword are displayed in the Function menu in the Curve Fit dialog.

If you wish other functions to be displayed in the Function menu, you can select the checkbox labelled "Show old-style functions (missing FitFunc keyword)".

See Also

User-Defined Fitting Functions on page III-219.

floor

`floor(`*`num`*`)`

The floor function returns the closest integer less than or equal to *num*.

See Also

The **round**, **ceil**, and **trunc** functions.

FontList

`FontList(`*`separatorStr`*` [, `*`options`*`])`

The FontList function returns a list of the installed fonts, separated by the characters in *separatorStr*.

Parameters

A maximum of 10 bytes from *separatorStr* are appended to each font name as the output string is generated. *separatorStr* is usually ";".

Use *options* to limit the returned font list according to font type. It is restricted to returning only scalable fonts (TrueType, PostScript, or OpenType), which you can do with *options* = 1.

To get a list of nonscalable fonts (bitmap or raster), use:

```
String bitmapFontList = RemoveFromList(FontList(";",1), FontList(";"))
```

(Most Mac OS X fonts are scalable, so bitmapFontList may be empty.)

Examples

```
Function SetFont(fontName)
    String fontName
    Prompt fontName,"font name:",popup,FontList(";")+"default;"
    DoPrompt "Pick a Font", fontName

    Print fontName

    Variable type= WinType("")                // target window type
    String windowName= WinName(0,127)
    if((type==1) || (type==3) || (type==7))   // graph, panel, layout
```

```
      Print "Setting drawing font for "+windowName
      Execute "SetDrawEnv fname=\""+fontName+"\""
   else
      if( type == 5 )                          // notebook
         Print "Setting font for selection in "+windowName
         Notebook $windowName font=fontName
      endif
   endif
End
```

See Also

The **FontSizeStringWidth**, **FontSizeHeight**, and **WinType** functions, and the **Execute**, **SetDrawEnv**, and **Notebook** Operations.

FontSizeHeight

FontSizeHeight(*fontNameStr*, *fontSize*, *fontstyle* [,*appearanceStr*])

The FontSizeHeight function returns the line height in pixels of any string when rendered with the named font and the given font style and size.

Parameters

fontNameStr is the name of the font, such as `"Helvetica"`.

fontSize is the size (height) of the font in pixels.

fontStyle is text style (bold, italic, etc.). Use 0 for plain text.

Details

The returned height is the sum of the font's ascent and descent heights. Variations in *fontStyle* and typeface design cause the actual font height to be different than *fontSize* would indicate. (Typically a font "height" refers to only the ascent height, so the total height will be slightly larger to accommodate letters that descend below the baseline, such as g, p, q, and y).

FontSize is in pixels. To obtain the height of a font specified in points, use the **ScreenResolution** function and the conversion factor of 72 points per inch (see Examples).

If the named font is not installed, FontSizeHeight returns NaN.

FontSizeHeight understands "default" to mean the current experiment's default font.

fontStyle is a binary coded integer with each bit controlling one aspect of the text style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

To set bit 0 and bit 2 (bold, underline), use $2^0+2^2 = 1+4 = 5$ for *fontStyle*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The optional *appearanceStr* parameter has no effect on Windows.

On Macintosh, the *appearanceStr* parameter is used for determining the height of a string drawn by a control. Set *appearanceStr* to "native" if you are measuring the height of a string drawn by a "native GUI" control or to "os9" if not.

Set *appearanceStr* to "default" to use the appearance set by the user in the Miscellaneous Settings dialog. "os9" is the default value.

Usually you will want to set *appearanceStr* to the S_Value output of **DefaultGUIControls**/W=winName when determining the height of a string drawn by a control.

Examples

```
Variable pixels= 12 * ScreenResolution/72         // convert 12 points to pixels
Variable pixelHeight= FontSizeHeight("Helvetica",pixels,0)
Print "Height in points= ", pixelHeight * 72/ScreenResolution
```

```
Function FontIsInstalled(fontName)
    String fontName
    if( numtype(FontSizeHeight(fontName,10,0)) == 2 )
        return 0            // NaN returned, font not installed
    else
        return 1
    endif
End
```

See Also

The **FontList**, **FontSizeStringWidth**, **numtype**, **ScreenResolution**, and **DefaultGUIControls** functions.

FontSizeStringWidth

FontSizeStringWidth(*fontNameStr*, *fontSize*, *fontstyle*, *theStr* [,*appearanceStr*])

The FontSizeStringWidth function returns the width of *theStr* in pixels, when rendered with the named font and the given font style and size.

Parameters

fontNameStr is the name of the font, such as `"Helvetica"`.

fontSize is the size (height) of the font in pixels.

fontStyle is text style (bold, italic, etc.). Use 0 for plain text.

theStr is the string whose width is being measured.

The optional *appearanceStr* parameter has no effect on Windows.

On Macintosh, the *appearanceStr* parameter is used for determining the width of a string drawn by a control. Set *appearanceStr* to "native" if you are measuring the width of a string drawn by a "native GUI" control or to "os9" if not.

Set *appearanceStr* to "default" to use the appearance set by the user in the Miscellaneous Settings dialog. "os9" is the default value.

Usually you will want to set *appearanceStr* to the S_Value output of **DefaultGUIControls**/W=winName when determining the width of a string drawn by a control.

Details

If the named font is not installed, FontSizeStringWidth returns NaN.

FontSizeStringWidth understands “default” to mean the current experiment’s default font.

FontSize is in pixels. To obtain the width of a font specified in points, use the **ScreenResolution** function and the conversion factor of 72 points per inch (see Examples).

fontStyle is a binary coded integer with each bit controlling one aspect of the text style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

To set bit 0 and bit 2 (bold, underline), use $2^0+2^2 = 1+4 = 5$ for *fontStyle*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Examples

Example 1

```
Variable fsPix= 10 * ScreenResolution/72          // 10 point text in pixels
String text= "How long is this text?"
Variable WidthPix= FontSizeStringWidth("Helvetica",fsPix,0,text)
Print "width in inches= ", WidthPix / ScreenResolution
```

Example 2

```
Variable fsPix= 13 * ScreenResolution/72          // 13 point text in pixels
String text= "text for a control"
```

```
DefaultGUIControls/W=Panel0                          // Sets S_Value
Variable WidthPix= FontSizeStringWidth("Helvetica",fsPix,0,text,S_Value)
Print "width in points= ", WidthPix / ScreenResolution * 72
```

See Also

The **FontList**, **FontSizeHeight**, **ScreenResolution** and **DefaultGUIControls** functions.

for-endfor

```
for(<initialize>;<exit test>;<update>)
   <loop body>
endfor
```

A for-endfor loop executes *loop body* code until *exit test* evaluates as FALSE, zero, or until a break statement is executed within the body code. When the loop starts, the *initialize* expressions are evaluated once. For each iteration, the *exit test* is evaluated at the beginning, and the *update* expressions are evaluated at the end.

See Also

For Loop on page IV-43 and **break** for more usage details.

FPClustering

```
FPClustering [flags] srcWave
```

The FPClustering operation performs cluster analysis using the farthest-point clustering algorithm. The input for the operation *srcWave* defines M points in N-dimensional space. Outputs are the waves W_FPCenterIndex and W_FPClusterIndex.

Flags

/CAC	Computes all the clusters specified by /MAXC.
/CM	Computes the center of mass for each cluster. The results are stored in the wave M_clustersCM in the current data folder. Each row corresponds to a single cluster with columns providing the respective dimensional components.
/INCD	Computes the inter-cluster distances. The result is stored in the current data folder in the wave M_InterClusterDistance, a 2D wave in which the [*i*][*j*] element contains the distance between cluster *i* and cluster *j*.
/MAXC=*nClusters*	Terminates the calculation when the number of clusters reaches the specified value. Note that this termination condition is sufficient but not necessary, i.e., the operation can terminate earlier if the farthest distance of an element from a hub is less than the average distance.
/MAXR=*maxRad*	Terminates the calculation when the maximum distance is less than or equal to *maxRad*.
/NOR	Normalizes the data on a column by column basis. The normalization makes each columns of the input span the range [0,1] so that even when *srcWave* contains columns that may be different by several orders of magnitude, the algorithm is not biased by a larger implied cartesian distance.
/Q	Don't print information to the history area.
/SHUB=*sHub*	Specifies the row which is used as a starting hub number. By default the operation uses the first row in *srcWave*.
/Z	No error reporting.

Details

The input for FPClustering is a 2D wave *srcWave* which consists of M rows by N columns where each row represents a point in N-dimensional space. *srcWave* can contain only finite real numbers and must be of type SP or DP. The operation computes the clustering and produces the wave W_FPCenterIndex which contains the centers or "hubs" of the clusters. The hubs are specified by the zero-based row number in *srcWave* which contains the cluster center. In addition, the operation creates the wave W_FPClusterIndex where each entry maps the corresponding input point to a cluster index. By default, the operation continues to add clusters as long as the largest possible distance is greater than the average intercluster distance. You can also stop the processing when the operation has formed a specified number of clusters (see /MAXC).

The variable V_max contains the maximum distance between any element and its cluster hub.

It is possible that in some circumstances you can get slightly different clustering depending on your starting point. The default starting hub is row zero of *srcWave* but you can use the /SHUB flag to specify a different starting point.

FPClustering computes the Cartesian distance between points. As a result, if the scale of any dimension is significantly larger than other dimensions it might bias the clustering towards that dimension. To avoid this situation you can use the /NOR flag which normalizes each column to the range [0,1] and hence equalizes the weight of each dimension in the clustering process.

See Also

The **KMeans** operation.

References

Gonzalez, T., Clustering to minimize the maximum intercluster distance, *Theoretical Computer Science*, *38*, 293-306, 1985.

fprintf

`fprintf` ***`refNum, formatStr`*** `[,` ***`parameter`***`]…`

The fprintf operation prints formatted output to a text file.

Parameters

refNum is a file reference number from the **Open** operation used to open the file.

formatStr is the format string, as used by the **printf** operation.

parameter varies depending on *formatStr*.

Details

If *refNum* is 1, fprintf will print to the history area instead of to a file, as if you used printf instead of fprintf. This useful for debugging purposes.

A zero value of *refNum* is used in conjunction with Program-to-Program Communication (PPC), Apple events (*Macintosh*) or DDE (*Windows*). Data that would normally be written to a file is appended to the PPC, Apple event or DDE result packet.

The fprintf operation supports numeric (real only) and string fields from structures. All other field types will cause a compile error.

See Also

The **printf** operation for complete format and parameter descriptions. The **Open** operation and **Creating Formatted Text** on page IV-244.

FReadLine

`FReadLine` `[/N/T]` ***`refNum, stringVarName`***

The FReadLine operation reads bytes from a file into the named string variable. The read starts at the current file position and continues until a terminator character is read, the end of the file is reached, or the maximum number of bytes is read.

Parameters

refNum is a file reference number from the **Open** operation used to create the file.

Flags

/N=*n*	Specifies the maximum number of bytes to read.
/T=*termcharStr*	Specifies the terminator character.
	`/T=(num2char(13))` specifies carriage return (CR, ASCII code 13).
	`/T=(num2char(10))` specifies linefeed (LF, ASCII code 10).
	`/T=";"` specifies the terminator as a semicolon.
	`/T=""` specifies the terminator as null (ASCII code 0).
	See **Details** for default behavior regarding the terminator.

Details

If /N is omitted, there is no maximum number of bytes to read. When reading lines of text from a normal text file, you will not need to use /N. It may be of use in specialized cases, such as reading text embedded in a binary file.

If /T is omitted, FReadLine will terminate on any of the following: CR, LF, CRLF, LFCR. (Most Macintosh files use CR. Most Windows files use CRLF. Most UNIX files use LF. LFCR is an invalid terminator but some buggy programs generate files that use it.) FReadLine reads whichever of these appears in the file, terminates the read, and returns just a CR in the output string. This default behavior transparently handles files that use CR, LF, CRLF, or LFCR as the terminator and will be suitable for most cases.

If you use the /T flag, then FReadLine will terminate on the specified character only and will return the specified character in the output string.

Once you have read all of the bytes in the file, FReadLine will return zero bytes via *stringVarName*. The example below illustrates testing for this.

In Igor Pro 7.00 or later, if the file data begins with the UTF-8 byte order mark (BOM) then FReadLine skips those bytes. In UTF-8, the BOM is the byte sequence 0xEF, 0xBB, 0xBF. If you want to check for the presence of the BOM, use **FBinRead** instead of FReadLine.

Example

```
Function PrintAllLinesInFile()
   Variable refNum
   Open/R refNum as ""           // Display dialog
   if (refNum == 0)
      return -1                  // User canceled
   endif

   Variable lineNumber, len
   String buffer
   lineNumber = 0
   do
      FReadLine refNum, buffer
      len = strlen(buffer)
      if (len == 0)
         break                   // No more lines to be read
      endif
      Printf "Line number %d: %s", lineNumber, buffer
      if (CmpStr(buffer[len-1],"\r") != 0)        // Last line has no CR ?
         Printf "\r"
      endif
   lineNumber += 1
   while (1)

   Close refNum
   return 0
End
```

See Also

The **Open** and **FBinRead** operations.

fresnelCos

`fresnelCos(`*`x`*`)`

The fresnelCos function returns the Fresnel cosine function *C*(*x*).

$$C(x) = \int_0^x \cos\left(\frac{\pi}{2}t^2\right)dt.$$

See Also

The **fresnelSin** and **fresnelCS** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

fresnelCS

`fresnelCS(`*`x`*`)`

The fresnelCS function returns both the Fresnel cosine in the real part of the result and the Fresnel sine in the imaginary part of the result.

See Also

The **fresnelSin** and **fresnelCos** functions.

fresnelSin

`fresnelSin(`*`x`*`)`

The fresnelSin function returns the Fresnel sine function $S(x)$.

$$S(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right)dt.$$

See Also

The **fresnelCos** and **fresnelCS** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

FSetPos

`FSetPos `*`refNum, filePos`*

The FSetPos operation attempts to set the current file position to the given position.

Parameters

refNum is a file reference number obtained from the **Open** operation when the file was opened.

filePos is the desired position of the file in bytes from the start of the file.

Details

FSetPos generates an error if *filePos* is greater than the number of bytes in the file. You can ascertain this limit with the **FStatus** operation.

When a file that is open for writing is closed, any bytes past the end of the current file position are deleted by the operating system. Therefore, if you use FSetPos, make sure to set the current file position properly before closing the file.

FSetPos supports files of any length.

See Also

Open, **FGetPos**, **FStatus**

FStatus

`FStatus` ***`refNum`***

The FStatus operation provides file status information for a file.

Parameters

refNum is a file reference number obtained from the **Open** operation.

Details

FStatus supports files of any length.

FStatus sets the following variables:

`V_flag`	Nonzero (true) if *refNum* is valid, in which case FStatus sets the other variables as well.
`V_filePos`	Current file position for the file in bytes from the start. In Igor7 or later, if you only want to know the current file position, use **FGetPos** instead of FStatus, which is slower.
`V_logEOF`	Total number of bytes in the file.
`S_fileName`	Name of the file.
`S_path`	Path from the volume to the folder containing the file. For example, "hd:Folder1:Folder2:". This is suitable for use as an input to the **NewPath** operation. Note that on the Windows operating system Igor uses a colon between folders instead of the Windows-standard backslash to avoid confusion with Igor's use of backslash to start an escape sequence (see **Escape Sequences in Strings** on page IV-13).
`S_info`	Keyword-packed information string.

The keyword-packed information string for S_info consists of a sequence of sections with the following form: *keyword*:*value*; You can pick a value out of a keyword-packed string using the **NumberByKey** and **StringByKey** functions.

Here are the keywords for S_info:

Keyword	Type	Meaning
PATH	string	Name of the symbolic path in which the file is located. This will be empty if there is no such symbolic path.
WRITEABLE	number	1 if file can be written to, 0 if not.

See Also

Open, **FGetPos**, **FSetPos**

FTPCreateDirectory

`FTPCreateDirectory` `[`***`flags`***`]` ***`urlStr`***

The FTPCreateDirectory operation creates a directory on an FTP server on the Internet.

For background information on Igor's FTP capabilities and other important details, see **File Transfer Protocol (FTP)** on page IV-257.

FTPCreateDirectory sets V_flag to zero if the operation succeeds or to a non-zero error code if it fails.

If the directory specified by *urlStr* already exists on the server, the server contents are not touched and V_flag is set to -1. This is not treated as an error.

Parameters

urlStr specifies the directory to create. It consists of a naming scheme (always "ftp://"), a computer name (e.g., "ftp.wavemetrics.com" or "38.170.234.2"), and a path (e.g., "/test/newDirectory"). For example:

```
"ftp://ftp.wavemetrics.com/test/newDirectory"
```

urlStr must always end with a directory name, and must not end with a slash.

To indicate that *urlStr* contains an absolute path, insert an extra '/' character between the computer name and the path. For example:

```
   ftp://ftp.wavemetrics.com//pub/test/newDirectory
```

If you do not specify that the path in *urlStr* is absolute, it is interpreted as relative to the FTP user's base directory. Since pub is the base directory for an anonymous user at wavemetrics.com, these URLs reference the same directory for an anonymous user:

```
   ftp://ftp.wavemetrics.com//pub/test/newDirectory  // Absolute path
   ftp://ftp.wavemetrics.com/test/newDirectory       // Relative to base directory
```

Special characters, such as punctuation, that are used in *urlStr* may be incorrectly interpreted by the operation. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. See **Percent Encoding** on page IV-253 for additional information.

Flags

/N=*portNumber*	Specifies the server's TCP/IP port number to use (default is 21). In almost all cases, the default will be correct so you won't need to use the /N flag.
/U=*userNameStr*	Specifies the user name to be used when logging in to the FTP server. If /U is omitted or if *userNameStr* is "", the login is done as an anonymous user. Use /U if you have an account on the FTP server.
/V=*diagnosticMode*	Determines what kind of diagnostic messages FTPCreateDirectory will display in the history area. *diagnosticMode* is a bitwise parameter, with the bits defined as follows: Bit 0: Show basic diagnostics. Currently this just displays the URL in the history. Bit 1: Show errors. This displays additional information when errors occur. Bit 2: Show status. This displays commands sent to the server and the server's response. The default value for *diagnosticMode* is 3 (show basic and error diagnostics). If you are having difficulties, you can try using 7 to show the commands sent to the server and the server's response. See **FTP Troubleshooting** on page IV-260 for other troubleshooting tips.
/W=*passwordStr*	Specifies the password to be used when logging in to the FTP server. Use /W if you have an account on the FTP server. If /W is omitted, the login is done using a default password that will work with most anonymous FTP servers. See **Safe Handling of Passwords** on page IV-254 for information on handling sensitive passwords.
/Z	Errors are not fatal. Will not abort procedure execution if an error occurs. Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded, -1 if the specified directory already exists, or another nonzero value if an error occurred.

Examples

```
// Create a directory.
String url = "ftp://ftp.wavemetrics.com/pub/test/newDirectory"
FTPCreateDirectory url
```

See Also

File Transfer Protocol (FTP) on page IV-257.

FTPDelete, **FTPDownload**, **FTPUpload**, **URLEncode**

FTPDelete

`FTPDelete [`*`flags`*`]` ***`urlStr`***

The FTPDelete operation deletes a file or a directory from an FTP server on the Internet.

Warning: If you delete a directory on an FTP server, all contents of that directory and any subdirectories are also deleted.

For background information on Igor's FTP capabilities and other important details, see **File Transfer Protocol (FTP)** on page IV-257.

FTPDelete sets V_flag to zero if the operation succeeds and to nonzero if it fails. This, in conjunction with the /Z flag, can be used to allow procedures to continue to execute if an FTP error occurs.

Parameters

urlStr specifies the file or directory to delete. It consists of a naming scheme (always "ftp://"), a computer name (e.g., "ftp.wavemetrics.com" or "38.170.234.2"), and a path (e.g., "/test/TestFile1.txt"). For example: "ftp://ftp.wavemetrics.com/test/TestFile1.txt"

urlStr must always end with a file name if you are deleting a file or with a directory name if you are deleting a directory. In the case of a directory, *urlStr* must not end with a slash.

To indicate that *urlStr* contains an absolute path, insert an extra '/' character between the computer name and the path. For example:

```
ftp://ftp.wavemetrics.com//pub/test
```

If you do not specify that the path in *urlStr* is absolute, it is interpreted as relative to the FTP user's base directory. Since pub is the base directory for an anonymous user at wavemetrics.com, these URLs reference the same directory for an anonymous user:

```
ftp://ftp.wavemetrics.com//pub/test
ftp://ftp.wavemetrics.com/test
```

Special characters such as punctuation that are used in *urlStr* may be incorrectly interpreted by the operation. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. See **Percent Encoding** on page IV-253 for additional information

Flags

/D	Deletes a complete directory and all its contents. Omit /D if you are deleting a file.
/N=*portNumber*	Specifies the server's TCP/IP port number to use (default is 21). In almost all cases, the default will be correct so you won't need to use the /N flag.
/U=*userNameStr*	Specifies the user name to be used when logging in to the FTP server. If /U is omitted or if userNameStr is "", the login is done as an anonymous user. Use /U if you have an account on the FTP server.
/V=*diagnosticMode*	Determines what kind of diagnostic messages FTPDelete will display in the history area. *diagnosticMode* is a bitwise parameter, with the bits defined as follows: Bit 0: Show basic diagnostics. Currently this just displays the URL in the history. Bit 1: Show errors. This displays additional information when errors occur. Bit 2: Show status. This displays commands sent to the server and the server's response. The default value for *diagnosticMode* is 3 (show basic and error diagnostics). If you are having difficulties, you can try using 7 to show the commands sent to the server and the server's response. See **FTP Troubleshooting** on page IV-260 for other troubleshooting tips.

/W=*passwordStr*	Specifies the password to be used when logging in to the FTP server. Use /W if you have an account on the FTP server. If /W is omitted, the login is done using a default password that will work with most anonymous FTP servers. See **Safe Handling of Passwords** on page IV-254 for information on handling sensitive passwords.
/Z	Errors are not fatal. Will not abort procedure execution if an error occurs. Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded, or a nonzero value if an error occurred.

Examples

```
// Delete a file.
String url = "ftp://ftp.wavemetrics.com/test/TestFile1.txt"
FTPDelete url

// Delete a directory.
String url = "ftp://ftp.wavemetrics.com/test/TestDir1"
FTPDelete/D url
```

See Also

File Transfer Protocol (FTP) on page IV-257.

FTPCreateDirectory, **FTPDownload**, **FTPUpload**, **URLEncode**

FTPDownload

`FTPDownload` `[`***`flags`***`]` ***`urlStr, localPathStr`***

The FTPDownload operation downloads a file or a directory from an FTP server on the Internet.

Warning: When you download a file or directory using the path and name of a file or directory that already exists on your local hard disk, all previous contents of the local file or directory are obliterated.

For background information on Igor’s FTP capabilities and other important details, see **File Transfer Protocol (FTP)** on page IV-257.

FTPDownload sets a variable named V_flag to zero if the operation succeeds and to nonzero if it fails. This, in conjunction with the /Z flag, can be used to allow procedures to continue to execute if a FTP error occurs.

If the operation succeeds, FTPDownload sets a string named S_Filename to the full file path of the downloaded file or, if the /D flag was used, the full path to the base directory that was downloaded. This is useful in conjunction with the /I flag.

If the operation fails, S_Filename is set to "".

Parameters

urlStr specifies the file or directory to download. It consists of a naming scheme (always `"ftp://"`), a computer name (e.g., `"ftp.wavemetrics.com"` or `"38.170.234.2"`), and a path (e.g., `"/Test/TestFile1.txt"`). For example: `"ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"`.

urlStr must always end with a file name if you are downloading a file or with a directory name if you are downloading a directory. In the case of a directory, *urlStr* must not end with a slash.

To indicate that *urlStr* contains an absolute path, insert an extra '/' character between the computer name and the path. For example:

```
ftp://ftp.wavemetrics.com//pub/test
```

If you do not specify that the path in *urlStr* is an absolute path, it is interpreted as a path relative to the FTP user's base directory. Since pub is the base directory for an anonymous user, this URL references the same directory:

```
ftp://ftp.wavemetrics.com/test
```

Special characters such as punctuation that are used in *urlStr* may be incorrectly interpreted by the operation. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. See **Percent Encoding** on page IV-253 for additional information.

localPathStr and *pathName* specify the name to use for the file or directory that will be created on your hard disk. If you use a full or partial path for *localPathStr*, see **Path Separators** on page III-401 for details on forming the path.

localPathStr must always end with a file name if you are downloading a file or with a directory name if you are downloading a directory. In the case of a directory, *localPathStr* must not end with a colon or backslash.

FTPDownload displays a dialog through which you can identify the local file or directory in the following cases:

1. You have used the /I (interactive) flag.
2. You did not completely specify the location of the local file or directory via *pathName* and *localPathStr*.
3. There is an error in *localPathStr*. This can be either a syntactical error or a reference to a nonexistent file or directory.
4. The specified local file or directory exists and you have not used the /O (overwrite) flag.

See **Examples** for examples of constructing a URL and local path.

Flags

/D	Downloads a complete directory. Omit it if you are downloading a file.
/I	Interactive mode which will prompt you to specify the name and location of the file or directory to be created on the local hard disk.
/M=*messageStr*	Specifies the prompt message used by the dialog in which you specify the name and location of the file or directory to be created. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/N=*portNumber*	Specifies the server's TCP/IP port number to use (default is 21). In almost all cases, this will be correct so you won't need to use the /N flag.
/O[=*mode*]	Controls whether a local file or directory whose name is in conflict with the file or directory being downloaded is overwritten without prompting the user.

mode=0:	Prompts the user to allow the overwrite. This is the default behavior if /O is omitted.
mode=1:	Overwrites without prompting the user. If the /D flag is also used, all contents of the destination directory are deleted if it already exists. /O=1 is the same as /O.
mode=2:	Merges files and subdirectories downloaded with the contents of the destination directory. Unlike /O=1, the contents of the destination directory are not deleted, however files and directories downloaded from the server will overwrite existing files and directories of the same name. When downloading a file this mode is accepted but has the same effect as /O=1.

/P=*pathName*	Contributes to the specification of the file or directory to be created on your hard disk. *pathName* is the name of an existing symbolic path. See **Examples**.
/S=*showProgress*	Determines if a progress dialog is displayed.

0:	No progress dialog.
1:	Show a progress dialog (default).

/T=*transferType*	Controls the FTP transfer type.

0:	Image (binary) transfer (default).
1:	ASCII transfer.

See **FTP Transfer Types** on page IV-260 for more discussion.

/U=*userNameStr*	Specifies the user name to be used when logging in to the FTP server. If this flag is omitted or if *userNameStr* is `""`, you will be logged in as an anonymous user. Use this flag if you have an account on the FTP server.
/V=*diagnosticMode*	Determines what kind of diagnostic messages FTPDownload will display in the history area. *diagnosticMode* is a bitwise parameter, with the bits defined as follows: Bit 0: Show basic diagnostics. Currently this just displays the URL in the history. Bit 1: Show errors. This displays additional information when errors occur. Bit 2: Show status. This displays commands sent to the server and the server's response. The default value for *diagnosticMode* is 3 (show basic and error diagnostics). If you are having difficulties, you can try using 7 to show the commands sent to the server and the server's response. See **FTP Troubleshooting** on page IV-260 for other troubleshooting tips.
/W=*passwordStr*	Specifies the password to be used when logging in to the FTP server. Use this flag if you have an account on the FTP server. If this flag is omitted, "nopassword" will be used for the login password. This will work with most anonymous FTP servers. Some anonymous FTP servers request that you use your email address as a password. You can do this by including the /W="<your email address>" flag. If /W is omitted, the login is done using a default password that will work with most anonymous FTP servers. See **Safe Handling of Passwords** on page IV-254 for information on handling sensitive passwords.
/Z	Errors are not fatal. Will not abort procedure execution if an error occurs. Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded, -1 if the user canceled in an interactive dialog, or another nonzero value if an error occurred.

Examples

Download a file using a full local path:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String localPath = "hd:Test Folder:TestFile1.txt"
FTPDownload url, localPath
```

Download a file using a local symbolic path and file name:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String pathName = "Igor"      // Igor is the name of a symbolic path.
String fileName = "TestFile1.txt"
FTPDownload/P=$pathName url, fileName
```

Download a directory using a full local path:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestDir1"
String localPath = "hd:Test Folder:TestDir1"
FTPDownload/D url, localPath
```

See Also

File Transfer Protocol (FTP) on page IV-257.

FTPCreateDirectory, **FTPDelete**, **FTPUpload**, **URLEncode**, **FetchURL**.

FTPUpload

FTPUpload [*flags*] *urlStr*, *localPathStr*

The FTPUpload operation uploads a file or a directory to an FTP server on the Internet.

Warning: When you upload a file or directory to an FTP server, all previous contents of the server file or directory are obliterated.

For background information on Igor’s FTP capabilities and other important details, see **File Transfer Protocol (FTP)** on page IV-257.

FTPUpload sets a variable named V_flag to zero if the operation succeeds and to nonzero if it fails. This, in conjunction with the /Z flag, can be used to allow procedures to continue to execute if a FTP error occurs.

If the operation succeeds, FTPUpload sets a string named S_Filename to the full file path of the uploaded file or, if the /D flag was used, to the full path to the base directory that was uploaded. This is useful in conjunction with the /I flag.

If the operation fails, S_Filename is set to "".

Parameters

urlStr specifies the file or directory to create. It consists of a naming scheme (always `"ftp://"`), a computer name (e.g., `"ftp.wavemetrics.com"` or `"38.170.234.2"`), and a path (e.g., `"/Test/TestFile1.txt"`). For example: `"ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"`.

urlStr must always end with a file name if you are uploading a file or with a directory name if you are uploading a directory, in which case *urlStr* must not end with a slash.

To indicate that *urlStr* contains an absolute path, insert an extra '/' character between the computer name and the path. For example:

```
ftp://ftp.wavemetrics.com//pub/test
```

If you do not specify that the path in *urlStr* is an absolute path, it is interpreted as a path relative to the FTP user's base directory. Since pub is the base directory for an anonymous user, this URL references the same directory:

```
ftp://ftp.wavemetrics.com/test
```

Special characters such as punctuation that are used in *urlStr* may be incorrectly interpreted by the operation. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. If you get unexpected results and *urlStr* contains such characters, you can try percent-encoding the special characters. See **Percent Encoding** on page IV-253 for additional information.

localPathStr and *pathName* specify the name and location on your hard disk of the local file to be uploaded. If you use a full or partial path for *localPathStr*, see **Path Separators** on page III-401 for details on forming the path.

localPathStr must always end with a file name if you are uploading a file or with a directory name if you are uploading a directory. In the case of a directory, *localPathStr* must not end with a colon or backslash.

FTPUpload displays a dialog that you can use to identify the file or directory to be uploaded in the following cases:

1. You used the /I (interactive) flag.
2. You did not completely specify the location of the file or folder to be uploaded via *pathName* and *localPathStr*.
3. There is an error in *localPathStr*. This can be either a syntactical error or a reference to a nonexistent directory.

See **Examples** for examples of constructing a URL and local path.

Flags

/D	Uploads a complete directory. Omit it if you are uploading a file.
/I	Interactive mode which displays a dialog for choosing the local file or directory to be uploaded.

/M=*messageStr*	Specifies the prompt message used by the dialog in which you choose the local file or directory to be uploaded. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/N=*portNumber*	Specifies the server's TCP/IP port number to use (default is 21). In almost all cases, this will be correct so you won't need to use the /N flag.
/O[=*mode*]	Overwrite. FTPUpload *always* overwrites the specified server file or directory, whether /O is used or not. If /O=2 is *not* used, all files and subdirectories in the destination directory on the server are first deleted and then the local files and directories are uploaded to the server. If /O=2 *is* used, the existing contents the contents of the local source directory are merged into the remote directory instead of completely overwriting it.
/P=*pathName*	Contributes to the specification of the file or directory to be uploaded. *pathName* is the name of an existing symbolic path. See **Examples**.
/S=*showProgress*	Determines if a progress dialog is displayed. 0: No progress dialog. 1: Show a progress dialog (default).
/T=*transferType*	Controls the FTP transfer type. 0: Image (binary) transfer (default). 1: ASCII transfer. See **FTP Transfer Types** on page IV-260 for more discussion.
/U=*userNameStr*	Specifies the user name to be used when logging in to the FTP server. If this flag is omitted or if *userNameStr* is `""`, you will be logged in as an anonymous user. Use this flag if you have an account on the FTP server.
/V=*diagnosticMode*	Determines what kind of diagnostic messages FTPUpload will display in the history area. *diagnosticMode* is a bitwise parameter, with the bits defined as follows: Bit 0: Show basic diagnostics. Currently this just displays the URL in the history. Bit 1: Show errors. This displays additional information when errors occur. Bit 2: Show status. This displays commands sent to the server and the server's response. The default value for *diagnosticMode* is 3 (show basic and error diagnostics). If you are having difficulties, you can try using 7 to show the commands sent to the server and the server's response. See **FTP Troubleshooting** on page IV-260 for other troubleshooting tips.
/W=*passwordStr*	Specifies the password used when logging in to the FTP server. Use this flag if you have an account on the FTP server. If this flag is omitted, "nopassword" will be used for the login password. This will work with most anonymous FTP servers. Some anonymous FTP servers request that you use your email address as a password. You can do this by including the /W="<your email address>" flag. If /W is omitted, the login is done using a default password that will work with most anonymous FTP servers. See **Safe Handling of Passwords** on page IV-254 for information on handling sensitive passwords.

/Z	Errors are not fatal. Will not abort procedure execution if an error occurs. Your procedure can inspect the V_flag variable to see if the transfer succeeded. V_flag will be zero if it succeeded, -1 if the user canceled in an interactive dialog, or another nonzero value if an error occurred.

Examples

Upload a file using a full local path:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String localPath = "hd:Test Folder:TestFile1.txt"
FTPUpload url, localPath
```

Upload a file using a local symbolic path and file name:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestFile1.txt"
String pathName = "Igor"      // Igor is the name of a symbolic path.
String fileName = "TestFile1.txt"
FTPUpload/P=$pathName url, fileName
```

Upload a directory using a full local path:

```
String url = "ftp://ftp.wavemetrics.com/pub/test/TestDir1"
String localPath = "hd:Test Folder:TestDir1"
FTPUpload/D url, localPath
```

See Also

File Transfer Protocol (FTP) on page IV-257.

FTPCreateDirectory, **FTPDelete**, **FTPDownload**, **URLEncode**.

FuncFit

`FuncFit [`*`flags`*`]` ***`fitFuncName, cwaveName, waveName`*** **`[`*`flag parameters`*`]`**

`FuncFit [`*`flags`*`]` **{*`fitFuncSpec`*}, *`waveName`*** **`[`*`flag parameters`*`]`**

The FuncFit operation performs a curve fit to a user defined function, or to a sum of fit functions using the second form (see **Fitting Sums of Fit Functions** on page V-236). Fitting can be done using any method that can be selected using the /ODR flag (see **CurveFit** for details).

FuncFit operation parameters are grouped in the following categories: flags, parameters (*fitFuncName*, *cwaveName*, *waveName* or {*fitFuncSpec* }, *waveName*), and flag parameters. The sections below correspond to these categories. Note that flags must precede the *fitFuncName* or `fitFuncSpec` and flag parameters must follow *waveName*.

Flags

See **CurveFit** for all available flags.

Parameters

fitFuncName	The user-defined function to fit to, which can be a function taking multiple independent variables (see also **FuncFitMD**). Multivariate fitting with FuncFit *requires* /X=*xwaveSpec*.
cwaveName	Wave containing the fitting coefficients.
waveName	The wave containing the dependent variable data to be fit to the specified function. For functions of just one independent variable, the dependent variable data is often referred to as "Y data". You can fit to a subrange of the wave by supplying (*startX*,*endX*) or [*startP*,*endP*] after the wave name. See **Wave Subrange Details** below for more information on subranges of waves in curve fitting.
fitFuncSpec	List of fit functions and coefficient waves, with some optional information. Using this format fits a model consisting of the sum of the listed fit functions. Intended for fitting multiple peaks, but probably useful for other applications as well. See **Fitting Sums of Fit Functions** on page V-236.

Flag Parameters

These flag parameters must follow *waveName*.

/E=*ewaveName*	A wave containing the epsilon values for each parameter. Must be the same length as the coefficient wave.
/STRC=*structureInstance*	Used only with **Structure Fit Functions** on page III-229. When using a structure fit function, you must specify an instance of the structure to FuncFit. This will be an instance that has been initialized by a user-defined function that you write in order to invoke FuncFit.
/X=*xwaveSpec*	An optional wave containing X values for each of the input data values. If the fitting function has more than one independent variable, *xwaveSpec* is required and must be either a 2D wave with a column for each independent variable, or a list of waves, one for each independent variable. A list must be in braces: /X={*xwave0*, *xwave1*,…}. There must be exactly one column or wave for each independent variable in the fitting function.
/NWOK	Allowed in user-defined functions only. When present, certain waves may be set to null wave references. Passing a null wave reference to FuncFit is normally treated as an error. By using /NWOK, you are telling FuncFit that a null wave reference is not an error but rather signifies that the corresponding flag should be ignored. This makes it easier to write function code that calls FuncFit with optional waves. The waves affected are the X wave or waves (/X), weight wave (/W), epsilon wave (/E) and mask wave (/M). The destination wave (/D=wave) and residual wave (/R=wave) are also affected, but the situation is more complicated because of the dual use of /D and /R to mean "do autodestination" and "do autoresidual". See /AR and /AD. If you don't need the choice, it is better not to include this flag, as it disables useful error messages when a mistake or run-time situation causes a wave to be missing unexpectedly. **Note**: To work properly this flag must be the last one in the command.

Other parameters are used as for the **CurveFit** operation, with some exceptions for multivariate fits.

Details for Multivariate Fits

The dependent variable data wave, *waveName*, must be a 1D wave even for multivariate fits. For fits to data in a multidimensional wave, see **FuncFitMD**.

For multivariate fits, the auto-residual (/R with no wave specified) is calculated and appended to the top graph if the dependent variable data wave is graphed in the top graph as a simple 1D trace. Auto residuals are calculated but not displayed if the data are displayed as a contour plot.

The autodest wave (/D with no wave specified) for multivariate fits has the same number of points as the data wave, with a model value calculated at the X values contained in the wave or waves specified with /X=*xwaveSpec*.

Confidence bands are not supported for multivariate fits.

Wave Subrange Details

Almost any wave you specify to FuncFit can be a subrange of a wave. The syntax for wave subranges is the same as for the Display command (see **Subrange Display Syntax** on page II-250 for details). See **Wave Subrange Details** on page V-112 for a discussion of the use of subranges in curve fitting.

The backwards compatibility rules for CurveFit apply to FuncFit as well.

In addition to the waves discussed in the CurveFit documentation, it is possible to use subranges when specifying the coefficient wave and the epsilon wave. Since the coefficient wave and epsilon wave must have the same number of points, it might make sense to make them two columns from a single multicolumn wave. For instance, here is an example in which the first column is used as the coefficient wave, the second is used as the epsilon wave, and the third is used to save a copy of the initial guesses for future reference:

```
Make/D/N=(5, 3) myCoefs
myCoefs[][0] = {1,2,3,4,5}          // hypothetical initial guess
myCoefs[][1] = 1e-6                 // reasonable epsilon values
```

```
myCoefs[][2] = myCoefs[p][0]            // save copy of initial guess
FuncFit myFitFunc, myCoefs[][0] myData /E=myCoefs[][1] …
```

You might have a fit function that uses a subset of the coefficients that are used by another. It might be useful to use a single wave for both. Here is an example in which a function that takes four coefficients is used to fit a subset of the coefficients, and then that solution is used as the initial guess for a function that takes six coefficients:

```
Make/D/N=6 Coefs6={1,2,3,4,5,6}
FuncFit Fit4Coefs, Coefs6[0,3] fitfunc4Coefs …
FuncFit Fit6Coefs, Coefs6 ...
```

Naturally, the two fit functions must be worked out carefully to allow this.

Fitting Sums of Fit Functions

If Igor encounters a left brace at the beginning of the fit function name, it expects a list of fit functions to be summed during the fit. This is useful for, for instance, fitting several peaks in a data set to a sum of peak functions.

The fit function specification includes at least the name of the fitting function and an associated coefficient wave. A sum of fit functions requires multiple coefficient waves, one for each fit function. Any coefficient wave-related options must be specified in the fit function specification via keyword-value pairs.

The syntax of the sum-of-fit-functions specification is as follows:

`{{`*`func1`*`, `*`coef1`*`, `*`keyword=value`*`},{`*`func2`*`, `*`coef2`*`, `*`keyword=value`*`}, …}`

or

`{`*`string=fitSpecStr`*`}`

Within outer braces, each fit function specification is enclosed within inner braces. You can use one or more fit function specifications, with no intrinsic limit on the number of fit functions.

The second format is available to overcome limitations on the length of a command line in Igor. This format is just like the first, but everything inside the outer braces is contained in a string expression (which may be just a single string variable).

You can use any fit function that can be used for ordinary fitting, including the built-in functions that are available using the CurveFit operation. If you should write a user-defined fitting function with the same name as a built-in fit function, the user-defined function will be used (this is strongly discouraged).

Every function specification must include an appropriate coefficient wave, pre-loaded with initial guesses.

The comma between each function specification is optional.

The keyword-value pairs are optional, and are used to communicate further options on a function-by-function basis. Available keywords are:

HOLD=*holdstr*	Indicates that a fit coefficient should be held fixed during fitting. *holdstr* works just like the hold string specified via the /H flag for normal fitting, but applies only to the coefficient wave associated with the fit function it appears with. If you include HOLD in a string expression (the `{string=fitSpecStr}` syntax), you must escape the quotation marks around the hold string. If you use the command-line syntax `{{func1,coef1,HOLD=`*`holdStr`*`}, ...}`, *holdStr* may be a reference to a global variable acquired using SVAR, or it may be a quoted literal string. If you use `{string=`*`fitSpecStr`*`}`, *fitSpecStr* is parsed at run-time outside the context of any running function. Consequently, you cannot use a general string expression. You can use either `HOLD="quotedLiteralString"` or `HOLD=root:globalString`.
CONST={*constants*}	Sets the values of constants in the fitting function. So far, only two built-in functions take constants: exp_XOffset and dblexp_XOffset. They each take just one constant (the X offset), so you will have a “list” of one number inside the braces.
EPSW=*epsilonWave*	Specifies a wave holding epsilon values. Use only with a user-defined fitting function to set the differencing interval used to calculate numerical estimates of derivatives of the fitting function.

STRC=*structureInstance*	Specifies an instance of the structure to FuncFit when using a structure fit function. *structureInstance* is an instance that was initialized by a user-defined function that invokes FuncFit. This keyword (and structure fitting functions) can be used only when calling FuncFit from within a user-defined function. See **Structure Fit Functions** on page III-229 for more details.

For more details, and for examples of sums of fit functions in use, **Fitting Sums of Fit Functions** on page III-214.

See Also

The **CurveFit** operation for parameter details. See also **FuncFitMD** for user-defined multivariate fits to data in a multidimensional wave.

The best way to create a user-defined fitting function is using the Curve Fitting dialog. See **Using the Curve Fitting Dialog** on page III-155, especially the section **Fitting to a User-Defined Function** on page III-163.

For details on the form of a user-defined function, see **User-Defined Fitting Functions** on page III-219.

FuncFitMD

`FuncFitMD [`*`flags`*`] `*`fitFuncName, cwaveName, waveName`*` [`*`flag parameters`*`]`

The FuncFitMD operation performs a curve fit to the specified multivariate user defined *fitFuncSpec*. FuncFitMD handles gridded data sets in multidimensional waves. Most parameters and flags are the same as for the **CurveFit** and **FuncFit** operations; differences are noted below.

cwaveName is a 1D wave containing the fitting coefficients, and *functionName* is the user-defined fitting function, which has 2 to 4 independent variables.

FuncFitMD operation parameters are grouped in the following categories: flags, parameters (*fitFuncName*, *cwaveName*, *waveName*), and flag parameters. The sections below correspond to these categories. Note that flags must precede the *fitFuncName* and flag parameters must follow *waveName*.

Flags

/L=*dimSize*	Sets the dimension size of the wave created by the auto-trace feature, that is, /D without destination wave. The wave fit_*waveName* will be a multidimensional wave of the same dimensionality as *waveName* that has *dimSize* elements in each dimension. That is, if you are fitting to a matrix wave, fit_*waveName* will be a square matrix that has dimensions *dimSize* X*dimSize*. **Beware**: *dimSize* =100 requires 100 million points for a 4-dimensional wave!

Parameters

fitFuncName	User-defined function to fit to, which must be a function taking 2 to 4 independent variables.
cwaveName	1D wave containing the fitting coefficients.
waveName	The wave containing the dependent variable data to be fit to the specified function. For functions of just one independent variable, the dependent variable data is often referred to as "Y data". You can fit to a subrange of the wave by supplying (*startX*,*endX*) or [*startP*,*endP*] for each dimension after the wave name. See **Wave Subrange Details** below for more information on subranges of waves in curve fitting.

Flag Parameters

These flag parameters must follow *waveName*.

/E=*ewaveName*	A wave containing the epsilon values for each parameter. Must be the same length as the coefficient wave.
/T=*twaveName*	Like /X except for the T independent variable. This is a 1D wave having as many elements as *waveName* has chunks.
/X=*xwaveName*	The X independent variable values for the data to fit come from *xwaveName* instead of from the X scaling of *waveName*. This is a 1D wave having as many elements as *waveName* has rows.

/Y=*ywaveName*	Like /X except for the Y independent variable. This is a 1D wave having as many elements as *waveName* has columns.
/Z=*ywaveName*	Like /X except for the Z independent variable. This is a 1D wave having as many elements as *waveName* has layers.
/NWOK	Allowed in user-defined functions only. When present, certain waves may be set to null wave references. Passing a null wave reference to FuncFitMD is normally treated as an error. By using /NWOK, you are telling FuncFitMD that a null wave reference is not an error but rather signifies that the corresponding flag should be ignored. This makes it easier to write function code that calls FuncFitMD with optional waves. The waves affected are the X wave or waves (/X), the Y spacing wave (/Y), the Z spacking wave (/Z) the T spacing wave (/T), weight wave (/W), epsilon wave (/E) and mask wave (/M). The destination wave (/D=wave) and residual wave (/R=wave) are also affected, but the situation is more complicated because of the dual use of /D and /R to mean "do autodestination" and "do autoresidual". See /AR and /AD. If you don't need the choice, it is better not to include this flag, as it disables useful error messages when a mistake or run-time situation causes a wave to be missing unexpectedly. **Note**: To work properly this flag must be the last one in the command.

Details

Auto-residual (/R with no wave specified) and auto-trace (/D with no wave specified) for functions having two independent variables are plotted in a separate graph window if *waveName* is plotted as a contour or image in the top graph. An attempt is made to plot the model values and residuals in the same way as the input data.

By default the auto-trace and auto-residual waves are 50x50 or 25x25x25 or 15x15x15x15. Use /L=*dimSize* for other sizes. Make your own wave and use /D=*waveName* or /R=*waveName* if you want a wave that isn't square. In this case, the wave dimensions must be the same as the dependent data wave.

Confidence bands are not available for multivariate fits.

Wave Subrange Details

Almost any wave you specify to FuncFitMD can be a subrange of a wave. The syntax for wave subranges is the same as for the Display command; see **Subrange Display Syntax** on page II-250 for details. Note that the dependent variable data (*waveName*) must be a multidimensional wave; this requires an extension of the subrange syntax to allow a multidimensional subrange. See **Wave Subrange Details** on page V-235 for a discussion of the use of subranges in curve fitting.

The backwards compatibility rules for **CurveFit** apply to FuncFitMD as well.

A subrange could be used to pick a plane from a 3D wave for fitting using a fit function taking two independent variables:

```
Make/N=(100,100,3) DepData
FuncFitMD fitfunc2D, myCoefs, DepData[][][0] …
```

See Also

The **CurveFit** operation for parameter details.

The best way to create a user-defined fitting function is using the Curve Fitting dialog. See **Using the Curve Fitting Dialog** on page III-155, especially the section **Fitting to a User-Defined Function** on page III-163.

For details on the form of a user-defined function, see **User-Defined Fitting Functions** on page III-219.

FUNCREF

`FUNCREF` ***`protoFunc func`*** `[=` ***`funcSpec`***`]`

Within a user function, FUNCREF is a reference that creates a local reference to a function or a variable containing a function reference.

When passing a function as an input parameter to a user function, the syntax is:

`FUNCREF` *protoFunc func*

In this FUNCREF reference, *protoFunc* is a function that specifies the format of the function that can be passed by the FUNCREF, and *func* is a function reference used as an input parameter.

When you declare a function reference variable within a user function, the syntax is:

`FUNCREF` *`protoFunc`* *`func = funcSpec`*

Here, the local FUNCREF variable, *`func`*, is assigned a *funcSpec*, which can be a literal function name, a $ string expression that evaluates at runtime, or another FUNCREF variable.

See Also

Function References on page IV-98 for an example and further usage details.

FuncRefInfo

`FuncRefInfo(`*`funcRef`*`)`

The FuncRefInfo function returns information about a **FUNCREF**.

Parameters

funcRef is a function reference variable declared by a FUNCREF statement in a user-defined function.

Details

FuncRefInfo returns a semicolon-separated keyword/value string containing the following information:

Keyword	Information
NAME	The name of the reference function or "" if the FUNCREF variable has not been assigned to point to a function.
ISPROTO	0 if the FUNCREF variable has been assigned to point to a function. 1 if it has not been assigned and therefore still points to the prototype function.
ISXFUNC	0 if it points to a user-defined function. 1 if the FUNCREF points to an external function.

See Also

Function References on page IV-98 and **FUNCREF** on page V-238.

Function

`Function [[/C /D /S /DF /WAVE]` ***`functionName`*** **`([`*`parameters`*`])`**

The Function keyword introduces a user-defined function in a procedure window.

The optional flags specify the return value type, if any, for the function.

Flags

/C	Returns a complex number.
/D	Returns a double-precision number. Obsolete, accepted for backward compatibility.
/S	Returns a string.
/DF	Returns a data folder reference. See **Data Folder Reference Function Results** on page IV-75.
/WAVE	Returns a wave reference. See **Wave Reference Function Results** on page IV-70.

Details

If you omit all flags, the result is a scalar double-precision number.

The /D flag is not needed because all numeric return values are double-precision.

See Also

Chapter IV-3, **User-Defined Functions** and **Function Syntax** on page IV-31 for further information.

FunctionInfo

`FunctionInfo(`*`functionNameStr`*` [, `*`procedureWinTitleStr`*`])`

The FunctionInfo function returns a keyword-value pair list of information about the user-defined or external function name in *functionNameStr*.

Parameters

functionNameStr a string expression containing the name or multipart name of a user-defined or external function. *functionNameStr* is usually just the name of a function.

To return information about a static function, supply both the module name and the function name in MyModule#MyFunction format (see **Regular Modules** on page IV-222), or specify the function name and *procedureWinTitleStr* (see below).

To return information about a function in a different independent module, supply the independent module name in addition to any module name and function name (a double or triple name):

Name	What It Refers To
MyIndependentModule#MyFunction	Refers to a non-static function in an independent module.
MyIndependentModule#MyModule#MyFunction	Refers to a static function in a procedure file with `#pragma moduleName=MyModule` in an independent module.

(See **Independent Modules** on page IV-224 for details on independent modules.)

The optional *procedureWinTitleStr* can be the title of a procedure window (such as "Procedure" or "File Name Utilities.ipf") in which to search for the named user-defined function. The information about the named function in the specified procedure window is returned.

The *procedureWinTitleStr* parameter makes it possible to select one of several static functions with identical names among different procedure windows, even if they do not contain a `#pragma moduleName=myModule` statement.

If you execute this command:

```
SetIgorOption IndependentModuleDev=1
```

then *procedureWinTitleStr* can also be a title followed by an independent module name in brackets to return information about the named function in the procedure window of the given title that belongs to named independent module.

procedureWinTitleStr can also be just an independent module name in brackets to return information about the named nonstatic function in any procedure window that belongs to named independent module.

Details

The returned string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

User-Defined and External Functions

Keyword	Information Following Keyword
`NAME`	The name of the function. Same as contents of *functionNameStr* in most cases. Just the function name if you use the `module#function` format.
`TYPE`	Value is "UserDefined" or "XFunc".
`THREADSAFE`	Either "yes" or "no". See **ThreadSafe Functions** on page IV-97.
`RETURNTYPE`	Number giving the return type of the function. See the table **Return Type and Parameter Type Codes** on page V-241.
`N_PARAMS`	Number of parameters for this function.
`PARAM_`*`n`*`_TYPE`	Number encoding the type of each parameter. There will be N of these keywords, one for each parameter. The part shown as *n* will be a number from 0 to N.

See **Examples** for a method for decoding these keywords.

User-Defined Functions Only

Keyword	Information Following Keyword	
PROCWIN	Title of procedure window containing the function definition.	
MODULE	Module containing function definition (see **Regular Modules** on page IV-222).	
INDEPENDENTMODULE	Independent module containing function definition (see **Independent Modules** on page IV-224).	
SPECIAL	The value part has one of three values:	
	no:	Not a “special” function.
	static:	Is a static function. Use the module#function format to get info about static functions.
	override:	Function is an override function. See **Function Overrides** on page IV-98.
SUBTYPE	The function subtype, for instance **FitFunc**. See **Procedure Subtypes** on page V-13 for others.	
PROCLINE	Line number within the procedure window of the function definition.	
VISIBLE	Either "yes" or "no". Set to "no" in the unlikely event that the function is defined in an invisible file.	
N_OPT_PARAMS	Number of optional parameters. Usually zero.	

External Functions Only

Keyword	Information Following Keyword
XOP	Name of the XOP module containing the function.

Return Type and Parameter Type Codes

Type	Code	Code in Hex
Complex	1	0x1
Single Precision	2	0x2
Variable	4	0x4
Double Precision	4	0x4
Byte	8	0x8
16-bit Integer	16	0x10
32-bit Integer	32	0x20
Single Precision	2	0x2
/WAVE	128	0x80
Data folder reference	256	0x100
Structure	512	0x200
Function reference	1024	0x400
Pass by reference parameter	4096	0x1000

Return Type and Parameter Type Codes

Type	Code	Code in Hex
String	8192	0x2000
Wave	16384	0x4000
/Z	-32768	0xFFFF8000

Igor functions can return a numeric value, a string value, a wave reference, or a data folder reference.

A returned numeric value is always double precision and may be complex. The return type for a normal numeric function is 4, for a complex function (Function/C) is 5 (4 for number +1 for complex).

A string function (Function/S) is 8192 (string). A Function/WAVE has a return type of 16384.

The return and parameter codes may be combined to indicate combinations of attributes. For instance, the code for a variable is 4 and the code for complex is 1. Consequently, the code for a complex variable parameter is 5. The code for a complex variable parameter passed by reference is (4+1+4096) = 4101.

Variables are always double-precision, hence the code of 4.

Waves may have a variety of codes. Numeric waves will combine with one of the number type codes such as 2 or 16. This does not reflect the numeric type of any actual wave, but rather any flag you may have used in the Wave reference. Thus, if the beginning of your function looks like

```
Function myFunc(w)
	Wave w
```

the code for the parameter w will be 16386 (16384 + 2) indicating a single-precision wave. You can use a numeric type flag with the Wave reference:

```
Function myFunc(w)
	Wave/I w
```

In this case, the code will be 16416 (16384 + 32).

Such codes are not very useful, as it is very rare to use a numeric type flag because the numeric type will be resolved correctly at runtime regardless of the flag.

A text wave has no numeric type, so its code is exactly 16384 (or -16384 if /Z is also specified.) Thus, the numeric type part of the code for a numeric wave serves to distinguish a numeric wave from a text wave. And a Wave/WAVE (a wave that contains references to other waves) has a code of 16512 (16384 + 128), unless /Z is also specified, which subtracts 32768, resulting in a code of -16256.

Examples

This function formats function information nicely and prints it in the history area in an organized fashion. You can copy it into the Procedure window to try it out. It uses the function `InterpretType()` below to print a human-readable version of the parameter and return types. To try `PrintFuncInfo()`, you will need to copy the code for `InterpretType()` as well.

```
Function PrintFuncInfo(functionName)
	String functionName

	String infostr = FunctionInfo(functionName)
	if (strlen(infostr) == 0)
		print "The function \""+functionName+"\" does not exist."
		return -1
	endif

	print "Name: ", StringByKey("NAME", infostr)

	String typeStr = StringByKey("TYPE", infostr)
	print "Function type: ", typeStr
	Variable IsUserDefined = CmpStr(typeStr, "UserDefined")==0

	// It's not really necessary to use an IF statement here;
	// it simply prevents lines with blank information being
	// printed for an XFUNC.

	if (IsUserDefined)
		print "Module: ", StringByKey("MODULE", infostr)
```

```
        print "Procedure window: ", StringByKey("PROCWIN", infostr)
        print "Subtype: ", StringByKey("SUBTYPE", infostr)
        print "Special? ", StringByKey("SPECIAL", infostr)

        // Note use of NumberByKey to get a numeric key value
        print "Line number: ", NumberByKey("PROCLINE", infostr)
    endif

    // See function InterpretType() below for example of
    // interpreting type information.

    Variable returnType = NumberByKey("RETURNTYPE", infostr)

    String returnTypeStr = InterpretType(returnType, 1)
    printf "Return type: %d (0x%X) %s\r", returnType, returnType, returnTypeStr

    Variable nparams = NumberByKey("N_PARAMS", infostr)
    print "Number of Parameters: ", nparams

    Variable nOptParams = 0
    if (IsUserDefined)
        nOptParams = NumberByKey("N_OPT_PARAMS", infostr)
        print "Optional Parameters: ", nOptParams
    endif

    Variable i
    for (i = 0; i < nparams; i += 1)
        // Note how the PARAM_n_TYPE keyword string is constructed here:
        String paramKeyStr = "PARAM_"+num2istr(i)+"_TYPE"
        Variable ptype = NumberByKey(paramKeyStr, infostr)

        String ptypeStr = InterpretType(ptype,0)
        String format = "Parameter %d; type as number: %g (0x%X); type as string: %s"
        String output
        sprintf output, format, i, ptype, ptype, pTypeStr
        print output
    endfor

    return 0
End
```

Function that creates a human-readable string with information about parameter and return types. Note that various attributes of the type info is tested using the bitwise AND operator (&) to test for individual bits. The constants are expressed as hexadecimal values (prefixed with “0x”) to make them more readable (at least to a programmer). Otherwise, 0x4000 would be 16384; at least, 0x4000 is clearly a single-bit constant.

```
Function/S InterpretType(type, isReturnType)
    Variable type
    Variable isReturnType      // 0: type is parameter type; 1: type is return type.

    String typeStr = ""

    // limit type to unsigned 16-bit values (remove sign extensions caused by 0x8000)
    type = type & 0xFFFF

    // isNumeric is flag to tell whether to print out "complex" and "real";
    // we don't want that information on strings, text waves or wave of wave references.
    Variable isNumeric = 1

    if (type & 0x4000)         // test for WAVE bit set
        typeStr += "Wave"

        if( !isReturnType )
            if (type & 0x80)    // test for WAVE/WAVE bit set
                typeStr += "/WAVE"
                // don't print "real" or "complex" for wave waves
                isNumeric = 0
            endif
            if (type & 0x8000)  // test for WAVE/Z bit set
                typeStr += "/Z"
            endif
        endif
        typeStr += " "
```

```
      if( (type == 0x4000) || (type == (0x4000 | 0x8000)) )    // WAVE/T or WAVE/Z/T
         if( !isReturnType )
            // For parameter types, if no numeric bits are set, it is a text wave.
            // A numeric wave has some other bits set causing the value
            // to be different from 0x4000 or 0xC000.
            typeStr += "text "
         endif
         // Function/WAVE doesn't (cannot) specify whether the returned wave
         // is text or numeric.
         // Don't print "real" or "complex" for text or unknown wave types.
         isNumeric = 0
      endif
   elseif (type & 0x2000)     // test for STRING bit set
      typeStr += "String "
      isNumeric = 0
   elseif (type & 4)          // test for VARIABLE bit
      typeStr += "Variable "
   elseif (type & 0x100)      // test for DFREF bit
      typeStr += "Data folder reference "
      isNumeric = 0
   elseif (type & 0x200)      // test for STRUCTURE bit
      typeStr += "Struct "
      isNumeric = 0
   elseif (type & 0x400)      // test for FUNCREF bit
      typeStr += "FuncRef "
      isNumeric = 0
   endif

   // print "real" or "complex" for numeric objects only
   if (isNumeric)
      if (type & 1)           // test for COMPLEX bit
         typeStr += "cmplx "
      else
         typeStr += "real "
      endif
   endif

   if( !isReturnType && (type & 0x1000) )  // test for PASS BY REFERENCE bit
      typeStr += "reference "
   endif

   return typeStr
End
```

See Also

The **StringByKey** and **NumberByKey** functions.

StringByKey, **NumberByKey**, and **FunctionList** functions.

Regular Modules on page IV-222 and **Independent Modules** on page IV-224.

FunctionList

`FunctionList(`*`matchStr`*`, `*`separatorStr`*`, `*`optionsStr`*`)`

The FunctionList function returns a string containing a list of built-in or user-defined function names satisfying certain criteria. This is useful for making a string to list functions in a pop-up menu control. Note that if the procedures need to be compiled, then FunctionList will not list user-defined functions.

Parameters

Only functions having names that match *matchStr* string are listed. Use `"*"` to match all names. See **WaveList** for examples.

The first character of *separatorStr* is appended to each function name as the output string is generated. *separatorStr* is usually `";"` for list processing (See **Processing Lists of Waves** on page IV-187 for details on list processing).

Use *optionsStr* to further qualify the list of functions. *optionsStr* is a string containing keyword-value pairs separated by commas. Available options are:

KIND:*nk* Controls the kinds of functions returned.

nk=1:	List built-in functions.
nk=2:	List normal and override user-defined functions.
nk=4:	List external functions (defined by an XOP).
nk=8:	List only curve fitting functions; must be summed with 1, 2, 4, or 16. For example, use 10 to list user-defined fitting functions.
nk=16:	Include static user-defined functions; requires WIN: option, must be summed with 1, 2, or 8. To list only static functions, subtract the non-static functions using RemoveFromList.

SUBTYPE:*typeName*

Lists functions that have the type *typeName*. That is, you could use ButtonControl as *typeName* to list only functions that are action procedures for buttons.

VALTYPE:*nv* Restricts list to functions whose return type is a certain kind.

nv=1:	Real-valued functions.
nv=2:	Complex-valued functions.
nv=4:	String functions.
nv=8:	WAVE functions
nv=16:	DFREF functions.

Use a sum of these values to include more than one type. The return type is not restricted if this option is omitted.

NPARAMS:*np* Restricts the list to functions having exactly *np* parameters. Omitting this option lists functions having any number of parameters.

NINDVARS:*ni* Restricts the list to fitting functions for exactly *ni* independent variables. NINDVARS is ignored if you have not elected to list curve fitting functions using the KIND option. Functions for any number of independent variables are listed if the NINDVARS option is omitted.

WIN:*windowTitle* Lists functions that are defined in the procedure window with the given title. “Procedure” is the title of the built-in procedure window.

Note: Because the *optionsStr* keyword-value pairs are comma separated and procedure window names can have commas in them, the WIN:keyword must be the last one specified.

WIN:windowTitle [*independentModuleName*]

Lists functions that are defined in the named procedure window that belongs to the independent module *independentModuleName*. See **Independent Modules** on page IV-224 for details. Requires `SetIgorOption IndependentModuleDev=1`, otherwise no functions are listed.

Requires *independentModuleName*=ProcGlobal or SetIgorOption independentModuleDev=1, otherwise no functions are listed.

Note: The syntax is literal and strict: the window title must be followed by one space and a left bracket, followed directly by the independent module name and a closing right bracket.

WIN:[*independentModuleName*]

Lists functions that are defined in any procedure file that belongs to the named independent module.

Requires *independentModuleName*=ProcGlobal or SetIgorOption independentModuleDev=1, otherwise no functions are listed.

Note: The syntax is literal and strict: 'WIN:' must be followed by a left bracket, followed directly by the independent module name and a closing right bracket, like this:

```
FunctionList(...,"WIN:[myIndependentModuleName]")
```

Examples

To list user-defined fitting functions for two independent variables:

```
Print FunctionList("*",";","KIND:10,NINDVARS:2")
```

To list button-control functions that start with the letter *b* (note that button-control functions are user-defined):

```
Print FunctionList("b*",";","KIND:2,SUBTYPE:ButtonControl")
```

See Also

Independent Modules on page IV-224.

For details on procedure subtypes, see **Procedure Subtypes** on page IV-193, as well as **Button**, **CheckBox**, **SetVariable**, and **PopupMenu**.

The **DisplayProcedure** operation and the **MacroList**, **OperationList**, **StringFromList**, and **WinList** functions.

FunctionPath

`FunctionPath(`*`functionNameStr`*`)`

The FunctionPath function returns a path to the file containing the named function. This is useful in certain specialized cases, such as if a function needs access to a lookup table of a large number of values.

The most likely use for this is to find the path to the file containing the currently running function. This is done by passing `""` for *functionNameStr*, as illustrated in the example below.

The returned path uses Macintosh syntax regardless of the current platform. See **Path Separators** on page III-401 for details.

If the procedure file is a normal standalone procedure file, the returned path will be a full path to the file such as `"hd:Igor Pro 7 Folder:WaveMetrics Procedures:Waves:Wave Lists.ipf"`.

If the function resides in the built-in procedure window the returned path will be `":Procedure"`. If the function resides in a packed procedure file, the returned path will be `":<packed procedure window title>"`.

If FunctionPath is called when procedures are in an uncompiled state, it returns ":".

Parameters

If *functionNameStr* is `""`, FunctionPath returns the path to the currently executing function or `""` if no function is executing.

Otherwise FunctionPath returns the path to the named function or `""` if no function by that name exists.

Examples

This example loads a lookup table into memory. The lookup table is stored as a wave in an Igor binary file.

```
Function LoadMyLookupTable()
    String path

    path = FunctionPath("")    // Path to file containing this function.
    if (CmpStr(path[0],":") == 0)
        // This is the built-in procedure window or a packed procedure
        // file, not a standalone file. Or procedures are not compiled.
        return -1
    endif

    // Create path to the lookup table file.
    path = ParseFilePath(1, path, ":", 0, 0) + "MyTable.ibw"
    DFREF dfSave = GetDataFolderDFR()
    // A previously-created place to store my private data.
    SetDataFolder root:Packages:MyData
    // Load the lookup table.
    LoadWave/O path
    SetDataFolder dfSave
    return 0
End
```

See Also

The **FunctionList** function.

GalleryGlobal

`GalleryGlobal#`*`pictureName`*

The GalleryGlobal keyword is used in an independent module to reference a picture in the global picture gallery which you can view by choosing Misc→Pictures.

See Also

See **Independent Modules and Pictures** on page IV-230.

gamma

`gamma(`*`num`*`)`

The gamma function returns the value of the gamma function of *num*. If *num* is complex, it returns a complex result. Note that the return value for *num* close to negative integers is NaN, not ±Inf.

See Also

The **gammln** function.

gammaEuler

`gammaEuler`

The gammaEuler function returns the Euler-Mascheroni constant 0.5772156649015328606065.

The gammaEuler function was added in Igor Pro 7.00.

gammaInc

`gammaInc(`*`a, x`*` [, `*`upperTail`*`])`

The gammaInc function returns the value of the incomplete gamma function, defined by the integral

$$\Gamma(a,x) = \int_x^{\infty} e^{-t} t^{a-1}\, dt.$$

If *upperTail* is zero, the limits of integration are 0 to x. If *upperTail* is absent, it defaults to 1, and the limits of integration are x to infinity, as shown. Note that `gammaInc(a, x) = gamma(a) - gammaInc(a, x, 0)`.

Defined for x > 0, a ≥ 0 (*upperTail* = zero or absent) or a > 0 (*upperTail* = 0).

See Also

The **gamma**, **gammp**, and **gammq** functions.

gammaNoise

`gammaNoise(`*`a`*` [, `*`b`*`])`

The gammaNoise function returns a pseudo-random value from the gamma distribution

$$f(x)=\frac{x^{a-1}\exp\left(-\frac{x}{b}\right)}{b^{a}\Gamma(a)}, \qquad x>0,\ a>0,\ b>0,$$

whose mean is *ab* and variance is ab^2. For backward compatibility you can omit the parameter *b* in which case its value is set to 1. When $a \to 1$ gammaNoise reduces to **expnoise**.

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable “random” numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

References

Marsaglia, G., and W. W. Tsang, *ACM*, 26, 363-372, 2000.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-344.

Chapter III-12, **Statistics** for a function and operation overview.

gammln

gammln(*num* [, *accuracy*])

The gammln function returns the natural log of the gamma function of *num*, where *num* > 0. If *num* is complex, it returns a complex result. Optionally, *accuracy* can be used to specify the desired fractional accuracy. If *num* is complex, it returns a complex result. In this case, *accuracy* is ignored.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of $(f_{actual} - f_{returned})/f_{actual}$ be less than 10^{-7}.

For backward compatibility, if you don’t include *accuracy*, gammln uses older code that achieves an accuracy of about $2x10^{-10}$.

With *accuracy*, newer code is used that is both faster and more accurate. The output has fractional accuracy better than $1x10^{-15}$ except for values near zero, where the absolute accuracy $(f_{actual} - f_{returned})$ is better than $2x10^{-16}$.

The speed of calculation depends only weakly on accuracy. Higher accuracy is significantly slower than lower accuracy only for *num* between 6 and about 10.

See Also

The **gamma** function.

gammp

gammp(*a*, *x* [, *accuracy*])

The gammp function returns the regularized incomplete gamma function P(*a*,*x*), where *a* > 0, *x* ≥ 0. Optionally, *accuracy* can be used to specify the desired fractional accuracy. Same as `gammaInc(a, x, 0)/gamma(a)`.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of $(f_{actual} - f_{returned})/f_{actual}$ be less than 10^{-7}.

For backward compatibility, if you don’t include *accuracy*, gammp uses older code that is slower for an equivalent accuracy, and cannot achieve as high accuracy.

The ability of gammp to return a value having full fractional accuracy is limited by double-precision calculations. This means that it will mostly have fractional accuracy better than about 10^{-15}, but this is not guaranteed, especially for extreme values of *a* and *x*.

See Also

The **gammaInc** and **gammq** functions.

gammq

gammq(*a*, *x* [, *accuracy*])

The gammq function returns the regularized incomplete gamma function 1-P(*a*,*x*), where $a > 0$, $x \geq 0$. Optionally, *accuracy* can be used to specify the desired fractional accuracy. Same as `gammaInc(a, x)/gamma(a)`.

Details

The *accuracy* parameter specifies the fractional accuracy that you desire. That is, if you set *accuracy* to 10^{-7}, that means that you wish that the absolute value of (f_{actual} - $f_{returned}$)/f_{actual} be less than 10^{-7}.

For backward compatibility, if you don't include *accuracy*, gammq uses older code that is slower for an equivalent accuracy, and cannot achieve as high accuracy.

The ability of gammq to return a value having full fractional accuracy is limited by double-precision calculations. This means that it will mostly have fractional accuracy better than about 10^{-15}, but this is not guaranteed, especially for extreme values of *a* and *x*.

See Also

The **gammaInc** and **gammp** functions.

Gauss

Gauss(x,*xc*,*wx* [,y,*yc*,*wy* [,z,*zc*,*wz* [,t,*tc*,*wt*]]])

The Gauss function returns a normalized Gaussian for the specified dimension.

$$Gauss(\mathbf{r},\mathbf{c},\mathbf{w}) = \prod_{i=1}^{n} \frac{1}{w_i\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{r_i - c_i}{w_i}\right)^2\right],$$

where *n* is the number of dimensions.

Parameters

xc, *yc*, *zc*, and *tc* are the centers of the Gaussian in the X, Y, Z, and T directions, respectively.

wx, *wy*, *wz*, and *wt* are the widths of the Gaussian in the X, Y, Z, and T directions, respectively.

Note that w_i here is the standard deviation of the Gaussian. This is different from the width parameter in the gauss curve fitting function, which is sqrt(2) times the standard deviation.

Note also that the Gauss function lacks the cross-correlation parameter that is included in the Gauss2D curve fitting function.

Examples

```
Make/n=100 eee=gauss(x,50,10)
Print area(eee,-inf,inf)
  0.999999
```

```
Make/n=(100,100) ddd=gauss(x,50,10,y,50,15)
Print area(ddd,-inf,inf)
  0.999137
```

Gauss1D

Gauss1D(*w*, *x*)

The Gauss1D function returns the value of a Gaussian peak defined by the coefficients in the wave *w*. The equation is the same as the Gauss curve fit:

$$w[0] + w[1]\exp\left[-\left(\frac{x - w[2]}{w[3]}\right)^2\right].$$

Examples

Do a fit to a Gaussian peak in a portion of a wave, then extend the model trace to the rest of the X range:

```
Make/O/N=100 junkg                      // fake data wave
Setscale/I x -1,1,junkg
Display junkg
junkg = 1+2.5*exp(-((x-.5)/.3)^2)+gnoise(.1)
Duplicate/O junkg, junkgfit
junkgfit = NaN
AppendToGraph junkgfit
CurveFit gauss junkg[50,99] /D=junkgfit
// now extend the model trace
junkgfit = Gauss1D(w_coef, x)
```

See Also

The **CurveFit** operation.

Gauss2D

Gauss2D(*w, x, y*)

The Gauss2D function returns the value of a two-dimensional Gaussian peak defined by the coefficients in the wave *w*. The equation is the same as the Gauss2D curve fit:

$$w[0]+w[1]\exp\left\{\frac{-1}{2\left(1-w[6]^2\right)}\left[\left(\frac{x-w[2]}{w[3]}\right)^2+\left(\frac{y-w[4]}{w[5]}\right)^2-\left(\frac{2w[6](x-w[2])(y-w[4])}{w[3]w[5]}\right)\right]\right\}.$$

Examples

Do a fit to a Gaussian peak in a portion of a wave, then extend the model trace to the rest of the X range (watch out for the very long wave assignment to junkg2D):

```
Make/O/N=(100,100) junkg2D                      // fake data wave
Setscale/I x -1,1,junkg2D
Setscale/I y -1,1,junkg2D
Display; AppendImage junkg2D
//Caution! Next command wrapped to fit page:
junkg2D = -1 + 2.5*exp((-1/(2*(1-.4^2)))*(((x-.1)/.2)^2+((y+.2)/.35)^2+2*.4*
          ((x-.1)/.2)*((y+.2)/.35)))
junkg2D += gnoise(.01)
Duplicate/O junkg2D, junkg2Dfit
junkg2Dfit = NaN
AppendMatrixContour junkg2Dfit
CurveFit gauss2D junkg2D[20,80][10,70] /D=junkg2Dfit[20,80][10,70]
// now extend the model trace
junkg2Dfit = Gauss2D(w_coef, x, y)
```

See Also

The **CurveFit** operation.

GBLoadWave

GBLoadWave [*flags*] [*fileNameStr*]

The GBLoadWave operation loads data from a binary file into waves.

For more complex applications such as loading structured data into Igor structures see the **FBinRead** operation.

Prior to Igor7, GBLoadWave was implemented as an XOP. It is now a built-in operation.

Parameters

If *fileNameStr* is omitted or is "", or if the /I flag is used, GBLoadWave presents an Open File dialog from which you can choose the file to load.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/A
Automatically assigns arbitrary wave names using "wave" as the base name. Skips names already in use.

/A=*baseName*
Same as /A but it automatically assigns wave names of the form *baseName*0, *baseName*1.

/D=*d*
New programming should use the /T flag instead of the /D, /L and /F flags.

d=0:	Creates single-precision waves.
d=1:	Creates double-precision waves.

/D by itself is equivalent to /D=1.

/F=f
New programming should use the /T flag instead of the /D, /L and /F flags.

f specifies the data format of the file:

f=1:	Signed integer (8, 16, 32 bits allowed)
f=2:	Creates double-precision waves
f=3:	Floating point (default, 32, 64 bits allowed)

/FILT=*fileFilterStr*
Provides control over the file filter menu in the Open File dialog. This flag was added in Igor Pro 7.00.

The construction of the *fileFilterStr* parameter is the same as for the /F=*fileFilterStr* flag of the Open operation. See **Open File Dialog File Filters** on page IV-137 for details.

/I [={*macFilterStr*, *winFilterStr*}]
Specifies interactive mode which displays the Open File dialog.

In Igor7, the *macFilterStr* and *winFilterStr* parameters are ignored. Use the /FILT flag instead.

/J=*j*
Specifies how input floating point data is interpreted.

j=0:	IEEE floating point (default)
j=1:	VAX floating point

/L=*length*
New programming should use the /T flag instead of the /D, /L and /F flags.

length specifies the data length of the data in the file in bits (default = 32). Allowable data lengths are 8, 16, 32, 64.

/N
Same as /A except that, instead of choosing names that are not in use, it overwrites existing waves.

/N=*baseName*
Same as /N except that it automatically assigns wave names of the form *baseName0*, *baseName1*.

/O=*o*
Controls overwriting of waves in case of a name conflict.

o=0:	Use unique wave names.
o=1:	Overwrite existing waves.

/O by itself is equivalent to /O=1.

/P=*pathName*
Specifies the folder to look in for *fileNameStr*. *pathName* is the name of an existing symbolic path.

/Q=*q*
Controls messages written to the history area of the command window.

q=0:	Write messages.
q=1:	Suppress messages.

/Q by itself is equivalent to /Q=1.

/S=*s*
s is the number of bytes at the start of the file to skip. It defaults to 0.

/T={*fType*,*wType*} Specifies the data type of the file (*fType*) and the data type of the wave or waves to be created (*wType*). The allowed codes for both *fType* and *wType* are:

2:	Single-precision floating point
4:	Double-precision floating point
8:	8-bit signed integer
16:	16-bit signed integer
32:	32-bit signed integer
64:	64-bit signed integer (Igor7 or later)
72:	8-bit unsigned integer (8+64)
80:	16-bit unsigned integer (16+64)
96:	32-bit unsigned integer (32+64)
192:	64-bit unsigned integer (128+64) (Igor7 or later)

/U=*u* Specifies the number of points of data per array in the file.

The default is 0 which means “auto”. In this case GBLoadWave calculate the number of data pointers per array based on the number of bytes in the file, the number of bytes to be skipped at the start of the file (/S flag), and the number of arrays in the file (/W flag).

/V=*v* Specifies interleaving of data in the file.

v=0:	Data in file is not interleaved (default)
v=1:	Data in file is interleaved

/V by itself is equivalent to /V=1.

/W=*w* Specifies the number of arrays in the file. The default is 1.

If you omit /W but specify the number of points per data array in the file via /U then GBLoadWave calculates the number of waves to be loaded based on the number of bytes in the file, the number of bytes to be skipped at the start of the file (/S flag), and the specified number of points per data array in the file (/U flag). Therefore, if you specify /U and want to load just one wave you must also specify /W=1.

/Y={*offset*, *mult*} Data loaded into waves is scaled using offset and mult:

```
output data = (input data + offset) * multiplier
```

This is useful to convert integer data into scaled, real numbers.

Details

The /N flag instructs Igor to automatically name new waves "wave" (or *baseName* if /N=*baseName* is used) plus a nimber. The nimber starts from zero and increments by one for each wave loaded from the file. If the resulting name conflicts with an existing wave, the existing wave is overwritten.

The /A flag is like /N except that Igor skips names already in use.

The /T flag allows you to specify a data type for both the input (data in the file) and the output (data in the waves). You should use the /T flag instead of the /D, /L and /F flags. These flags are obsolete but are still supported.

GBLoadWave Open File Dialog

If you include the /I flag, or if the /P=*pathName* and *fileNameStr* parameters do not fully specify the file to be loaded, GBLoadWave displays the Open File dialog.

The /FILT=*fileFilterStr* flag provides control over the file filter menu in the Open File dialog. This flag was added in Igor Pro 7.00. The construction of the *fileFilterStr* parameter is the same as for the /F=*fileFilterStr* flag of the Open operation. See **Open File Dialog File Filters** on page IV-137 for details.

In Igor7, the *macFilterStr* and *winFilterStr* parameters of the /I flag are ignored. Use the /FILT flag instead.

Output Variables

GBLoadWave sets the following output variables:

`V_flag`	Number of waves loaded or -1 if an error occurs during the file load.
`S_fileName`	Name of the file being loaded.
`S_path`	File system path to the folder containing the file.
`S_waveNames`	Semicolon-separated list of the names of loaded waves.

S_path uses Macintosh path syntax (e.g., "`hd:FolderA:FolderB:`"), even on Windows. It includes a trailing colon.

When GBLoadWave presents an Open File dialog and the user cancels, V_flag is set to 0 and S_fileName is set to `""`.

Example

```
// Load 128 point single precision version 2 Igor binary file
GBLoadWave/S=126/U=128 "fileName"

// Load 8 256 point arrays of 16 bit signed integers into single-precision waves
// after skipping 128 byte header
GBLoadWave/S=128/T={16,2}/W=8/U=256 "fileName"

// Load n 100 point arrays of double-precision floating point numbers
// into double-precision Igor waves with names like temp0, temp1, etc,
// overwriting existing waves. n is determined by the number of bytes
// in the file.
GBLoadWave/O/N=temp/T={4,4}/U=100 "fileName"

// Load a file containing a 1024 byte header followed by a 512 row
// by 384 column array of unsigned bytes into an unsigned byte matrix
// wave and display it as an image
GBLoadWave/S=1024/T={8+64,8+64}/N=temp "fileName"
Rename temp0, image
Redimension/N=(512,384) image
if (<file uses row-major order>)
      MatrixTranspose image
   endif
Display; AppendImage image
```

"Row-major order" relates to how a 2D array is stored in memory. In row-major order, all data for a given row is stored contiguously in memory. In column-major order, all data for a given column is stored contiguously in memory. Igor uses column-major order but row-major is more common.

See Also

Loading General Binary Files on page II-146.

FBinRead operation for more complex applications such as loading structured data into Igor structures.

gcd

`gcd(`*`A, B`*`)`

The gcd function calculates the greatest common divisor of *A* and *B*, which are both assumed to be integers.

Examples

Compute least common multiple (LCM) of two integers:

```
Function LCM(a,b)
   Variable a, b

   return((a*b)/gcd(a,b))
End
```

GetAxis

GetAxis [**/W=*winName* /Q**] ***axisName***

The GetAxis operation determines the axis range and sets the variables V_min and V_max to the minimum and maximum values of the named axis.

Parameters

axisName is usually `"left"`, `"right"`, `"top"` or `"bottom"`, though it may also be the name of a free axis such as `"VertCrossing"`.

Flags

/Q	Prevents values of V_flag, V_min, and V_max from being printed in the history area. The results are still stored in the variables.
/W=*winName*	Retrieves axis info from the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

GetAxis sets V_min according to the bottom of vertical axes or left of horizontal axes and V_max according to the top of vertical axes or right of horizontal axes. It also sets the variable V_flag to 0 if the specified axis is actually used in the graph, or to 1 if it is not.

See Also

The **AxisInfo** function.

GetBrowserLine

GetBrowserLine(*fullPathStr* [, *mode*])

The GetBrowserLine function returns the zero-based line number of the data folder referenced by *fullPathStr*.

Documentation for the GetBrowserLine function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "GetBrowserLine"
```

GetBrowserSelection

GetBrowserSelection(*index* [, *mode*])

The GetBrowserSelection returns a string containing the full path, quoted if necessary, to a selected Data Browser item.

Documentation for the GetBrowserSelection function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "GetBrowserSelection"
```

GetCamera

GetCamera [***flags***] [***keywords***]

The GetCamera operation provides information about a camera window.

Documentation for the GetCamera operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "GetCamera"
```

GetDataFolder

GetDataFolder(*mode* [, *dfr*])

The GetDataFolder function returns a string containing the name of or full path to the current data folder or, if *dfr* is present, the specified data folder.

GetDataFolderDFR is preferred.

Parameters

If *mode*=0, it returns just the name of the data folder.

If *mode*=1, GetDataFolder returns a string containing the full path to the data folder.

dfr, if present, specifies the data folder of interest.

Details

GetDataFolder can be used to save and restore the current data folder in a procedure. However GetDataFolderDFR is preferred for that purpose.

Examples

```
String savedDataFolder = GetDataFolder(1)          // Save
SetDataFolder root:
Variable/G gGlobalRootVar
SetDataFolder savedDataFolder                      // and restore
```

See Also

Chapter II-8, **Data Folders**.

The **SetDataFolder** operation and **GetDataFolderDFR** function.

GetDataFolderDFR

GetDataFolderDFR()

The GetDataFolderDFR function returns the data folder reference for the current data folder.

Details

GetDataFolderDFR can be used to save and restore the current data folder in a procedure. It is like GetDataFolder but returns a data folder reference rather than a string.

Example

```
DFREF saveDFR = GetDataFolderDFR()                 // Save
SetDataFolder root:
Variable/G gGlobalRootVar
SetDataFolder saveDFR                              // and restore
```

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-72.

The **SetDataFolder** operation.

GetDefaultFont

GetDefaultFont(*winName*)

The GetDefaultFont function returns a string containing the name of the default font for the named window or subwindow.

Parameters

If *winName* is null (that is, `""`) returns the default font for the experiment.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

Only graph windows and the experiment as a whole have default fonts. If *winName* is the name of a window other than a graph (e.g., a layout), or if *winName* is not the name of any window, GetDefaultFont returns the experiment default font.

In user-defined functions, font names are usually evaluated at compile time. To use the output of GetDefaultFont in a user-defined function, you will usually need to build a command as a string expression and execute it with the **Execute** operation.

Examples

```
String fontName = GetDefaultFont("Graph0")
String command= "SetDrawEnv fname=\"" + fontName + "\", save"
Execute command
```

See Also

The **GetDefaultFontSize**, **GetDefaultFontStyle**, **FontSizeHeight**, and **FontSizeStringWidth** functions.

GetDefaultFontSize

`GetDefaultFontSize(`*`graphNameStr, axisNameStr`*`)`

The GetDefaultFontSize function returns the default font size of the graph or of the graph's axis (in points) in the specified window or subwindow.

Details

If *graphNameStr* is `""` the top graph is examined.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

If *axisNameStr* is `""`, the font size of the default font for the graph is returned.

If named axis exists, the default font size for the named axis in the graph is returned.

If named axis does not exist, NaN is returned.

See Also

The **GetDefaultFont**, **GetDefaultFontStyle**, **FontSizeHeight**, and **FontSizeStringWidth** functions.

GetDefaultFontStyle

`GetDefaultFontStyle(`*`graphNameStr, axisNameStr`*`)`

The GetDefaultFontStyle function returns the default font style of the graph or of the graph's axis in the specified window or subwindow.

Details

If *graphNameStr* is `""` the top graph is examined.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

If *axisNameStr* is `""`, the font style of the default font for the graph is returned.

If named axis exists, the default font style for the named axis in the graph is returned.

If named axis does not exist, NaN is returned.

The function result is a bitwise value with each bit identifying one aspect of the font style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

See Also

The **GetDefaultFont**, **GetDefaultFontSize**, **FontSizeHeight**, and **FontSizeStringWidth** functions.

GetDimLabel

GetDimLabel(*waveName*, *dimNumber*, *dimIndex*)

The GetDimLabel function returns a string containing the label for the given dimension or dimension element.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers and 3 for chunks.

If *dimIndex* is -1, it returns the label for the entire dimension. If *dimIndex* is ≥ 0, it returns the dimension label for that element of the dimension.

See Also

SetDimLabel, **FindDimLabel**

Dimension Labels on page II-85 for further usage details and examples.

GetEnvironmentVariable

GetEnvironmentVariable(*varName*)

The GetEnvironmentVariable function returns a string containing the current value of the specified environment variable for the currently running Igor process. If the variable does not exist, an empty string ("") is returned.

The GetEnvironmentVariable function was added in Igor Pro 7.00.

Parameters

varName	The name of an environment variable which may or may not exist. It must not be an empty string and may not contain an equals sign (=). As a special case, if a single equals sign ("=") is passed for *varName*, a carriage return (\r) separated list of all current key=value environment variable pairs is returned.

Details

The environment of Igor's process is composed of a set of key=value pairs that are known as environment variables.

Any child process created by calling ExecuteScriptText inherits the environment variables of Igor's process.

On Windows, environment variable names are case-insensitive. On other platforms, they are case-sensitive.

GetEnvironmentVariable returns an empty string if *varName* does not exist or if it does exist but its value is empty. If you need to know whether or not the environment variable itself actually exists, you can use the following function:

```
Function EnvironmentVariableExists(varName)
    String varName
    String varList = GetEnvironmentVariable("=")

    // Replace \r with \n because GrepString treats \n only as line separator, not \r.
    varList = ReplaceString("\r", varList, "\n")
    String regExp = "(?m)^" + varName + "="
    return GrepString(varList, regExp)
End
```

Examples

```
String currentUser = GetEnvironmentVariable("USER")
String varList = GetEnvironmentVariable("=")
```

See Also

SetEnvironmentVariable, **UnsetEnvironmentVariable**

GetErrMessage

GetErrMessage(*errorCode* [, *substitutionOption*])

GetErrMessage returns a string containing an explanation of the error associated with *errorCode*.

Details

errorCode is a value sometimes returned in V_Flag, as per the documentation of the an Igor function or operation; the **Execute** operation is an example.

For a few error codes, the corresponding error message is designed to be combined with "substituted" information available only immediately after the error occurs. An example is the "parameter out of range" error which produces an error message such as "expected number between *x* and *y*". To get the correct error message, you must call GetErrMessage immediately after calling the function or operation that generated the error and you must pass the appropriate value for *substitutionOption* as explained below.

Substitution

Igor maintains two environments which store the substitution information: one for macros created using the Macro, Proc and Window keywords and another for user-defined functions created with the Function keyword. The optional *substitutionOption* parameter gives you the ability to choose between those environments or to not substitute at all. Set *substitutionOption* to one of:

substitutionOption	**GetErrMessage Action**
0	Substitution values are filled in with "_Not Available_". This is the default when *substitutionOption* is not specified.
1	Substitution values are blank.
2	Substitution is performed based on the presumption that the error was received while executing a macro or a command using Igor's command line. This includes a command executed via the Execute operation even from a user-defined function because such commands are executed as if entered in the command line.
3	Substitution is performed based on the presumption that the error was received while executing a user-defined function.

For most purposes you should pass 3 for *substitutionOption* when the error was generated in a user-defined function other than through the Execute operation and pass 2 otherwise.

Examples

```
// Macro, Execute or command line
Execute/Q/Z "Duplicate/O nonexistentWave, dup"
Print GetErrMessage(V_Flag,2)
```

Prints:

```
   expected wave name
```

```
// Function example
Function Test()
   Make/O/N=(2,2) data= 0
   FilterIIR/COEF=data/LO=999/Z data    // purposely wrong /LO value
   Print GetErrMessage(V_Flag)
   Print GetErrMessage(V_Flag,1)
   Print GetErrMessage(V_Flag,2)
   Print GetErrMessage(V_Flag,3)
End
```

Executing `Test()` prints:

```
  expected _Not Available_ between _Not Available_ and _Not Available_
  expected  between  and
  expected  between  and
  expected /LO frequency between 0 and 0.5
```

See Also

The **GetRTErrMessage** and **GetRTError** functions.

GetFileFolderInfo

`GetFileFolderInfo [`*`flags`*`][`*`fileOrFolderNameStr`*`]`

The GetFileFolderInfo operation returns information about a file or folder.

Parameters

fileOrFolderNameStr specifies the file (or folder) for which information is returned. It is optional if /P=*pathName* and /D are specified, in which case information about the directory associated with *pathName* is returned.

If you use a full or partial path for *fileOrFolderNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Folder paths should not end with single Path Separators. See the **MoveFolder Details** section.

If Igor can not determine the location of the file from *fileOrFolderNameStr* and *pathName*, it displays a dialog allowing you to specify the file to be examined. Use /D to select a folder.

Flags

/D	Uses the Select Folder dialog rather than Open File dialog when *pathName* and *fileOrFolderNameStr* do not specify an existing file or folder.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/Q	No information printed to the history area.
/Z[=z]	Prevents procedure execution from aborting if GetFileFolderInfo tries to get information about a file or folder that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort.

/Z=0:	Same as no /Z.
/Z=1:	Used for getting information for a file or folder only if it exists. /Z alone has the same effect as /Z=1.
/Z=2:	Used for getting information for a file or folder if it exists and displaying a dialog if it does not exist.

Variables

GetFileFolderInfo returns information in the following variables:

`V_flag`	0: File or folder was found. -1: User cancelled the Open File dialog. Other: An error occurred, such as the specified file or folder does not exist.
`S_path`	File system path of the selected file.
`V_isFile`	1: *fileOrFolderNameStr* is a file.
`V_isFolder`	1: *fileOrFolderNameStr* is a folder.
`V_isInvisible`	1: File is invisible (*Macintosh*) or Hidden (*Windows*).
`V_isReadOnly`	Set if the file is locked (*Macintosh*) or is read-only (*Windows*). On Macintosh, V_isReadOnly is either 0 (unlocked) or 1 (locked). To set this manually, display the Finder Info window for the file and then check or uncheck the “Locked” checkbox. On Windows, V_isReadOnly is either 0 (unlocked) or 1 (locked). To set this manually, display the Properties window for the file and then check or uncheck the “Read-only” checkbox. On both Macintosh and Windows, V_isReadOnly tells you only about the property set in the Finder or Windows desktop. It does not tell you if you have write permission for the file or for the folder containing the file. If your goal is to determine if you can write to the file, the only way to do that is to try to write to it and catch any resulting error.

`V_creationDate`	Number of seconds since midnight on January 1, 1904 when the file or folder was first created. Use **Secs2Date** to get a text format date.
`V_modificationDate`	Number of seconds since midnight on January 1, 1904 when the file or folder was last modified. Use **Secs2Date** to get a text format date.
`V_isAliasShortcut`	
	1: File is an alias (*Macintosh*) or a shortcut (*Windows*) and S_aliasPath is also set.

If *fileOrFolderNameStr* refers to a file (not a folder), GetFileFolderInfo returns additional information in the following variables:

`S_aliasPath`	Full path to the file or folder that is the source for an alias (*Macintosh*) or a shortcut (*Windows*). When the source is a folder, S_aliasPath ends with a ":" character.
`V_isStationery`	1: The stationery bit is set (*Macintosh*) or (*Windows*) the file type is one of the stationery file types (.pxt, .uxt, .ift).
`S_fileType`	Four-character file type code, such as 'TEXT' or 'IGsU' (packed experiment). On Windows, these codes are fabricated by translating from the equivalent file name extensions, such as .txt and .pxp.
`S_creator`	Four-character creator code, such as 'IGR0' (Igor Pro creator code). On Windows, S_creator is set to 'IGR0' if the file name extensions is one of those registered to Igor Pro, such as .pxp or .bwav (but not .txt). For other registered extensions, S_creator is set to the full file path of the registered application. Otherwise it is set to `""`.
`V_logEOF`	Number of bytes in the file data fork. For other forks, use **Open**/F and **FStatus**.
`V_version`	Version number of the file. On Macintosh, this is the value in the vers(1) resource. On Windows, a file version such as 3.10.2.1 is returned as 4.021: use S_fileVersion to avoid the problem of the second digit overflowing into the first digit. "0": File version can't be determined, or the file can't be examined because it is already open.
`S_fileVersion`	The file version as a string. On Macintosh, this is just a string representation of V_Version. On Windows, a file version such as 3.10.2.1 is returned as "3.10.2.1". "0": (*Macintosh*) file version can't be determined. "0.0.0.0": (*Windows*) file version can't be determined.

Details

You can change some of the file information by using **SetFileFolderInfo**.

Examples

Print the modification date of a file:

```
GetFileFolderInfo/Z "Macintosh HD:folder:afile.txt"
if( V_Flag == 0 && V_isFile )                   // file exists
   Print Secs2Date(V_modificationDate,0), Secs2Time(V_modificationDate,0)
endif
```

Determine if a folder exists (easier than creating a path with NewPath and then using PathInfo):

```
GetFileFolderInfo/Z "Macintosh HD:folder:subfolder"
if( V_Flag && V_isFolder )
   Print "Folder Exists!"
endif
```

Find the source for a shortcut or alias:

```
GetFileFolderInfo/Z "Macintosh HD:fileThatIsAlias"
if( V_Flag && V_isAliasShortcut )
```

```
    Print S_aliasPath
endif
```

See Also

The **SetFileFolderInfo**, **PathInfo**, and **FStatus** operations. The **IndexedFile**, **Secs2Date**, and **ParseFilePath** functions.

GetFormula

`GetFormula(`*`objName`*`)`

The GetFormula function returns a string containing the named object's dependency formula. The named object must be a wave, numeric variable or string variable.

Details

Normally an object will have an empty dependency formula and GetFormula will return an empty string (`""`). If you assign a expression to an object using the := operator or the SetFormula operation, the text on the right side of the := or the parameter to SetFormula is the object's dependency formula and this is what GetFormula will return.

Examples

```
Variable/G dependsOnIt
Make/O wave0 := dependsOnIt*2     //wave0 changes when dependsOnItdoes
Print GetFormula(wave0)
```

Prints the following in the history area:

```
  dependsOnIt*2
```

See Also

See **Dependency Formulas** on page IV-214, and the **SetFormula** operation.

GetGizmo

`GetGizmo [`*`flags`*`]` ***`keyword`*** **`[=`*`value`*`]`**

The GetGizmo operation provides information about a Gizmo display window.

Documentation for the GetGizmo operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "GetGizmo"
```

GetIndependentModuleName

`GetIndependentModuleName()`

The GetIndependentModuleName function returns the name of the currently running Independent Module. If no independent module is running, it returns “ProcGlobal”.

See Also

Independent Modules on page IV-224.

IndependentModuleList.

GetIndexedObjName

`GetIndexedObjName(`*`sourceFolderStr`*`, `*`objectType`*`, `*`index`*`)`

The GetIndexedObjName function returns a string containing the name of the indexth object of the specified type in the data folder specified by the string expression.

GetIndexedObjNameDFR is preferred.

Parameters

sourceFolderStr can be either `":"` or `""` to specify the current data folder. You can also use a full or partial data folder path. *index* starts from zero. If no such object exists a zero length string (`""`) is returned. *objectType* is one of the following values:

Examples

```
Function PrintAllWaveNames()
    String objName
```

objectType	**What You Get**
1	Waves
2	Numeric variables
3	String variables
4	Data folders

```
	Variable index = 0
	do
		objName = GetIndexedObjName(":", 1, index)
		if (strlen(objName) == 0)
			break
		endif
		Print objName
		index += 1
	while(1)
End
```

See Also

The **CountObjects** function, and Chapter II-8, **Data Folders**.

GetIndexedObjNameDFR

GetIndexedObjNameDFR(*dfr*, *objectType*, *index*)

The GetIndexedObjNameDFR function returns a string containing the name of the indexth object of the specified type in the data folder referenced by *dfr*.

GetIndexedObjNameDFR is the same as GetIndexedObjName except the first parameter, *dfr*, is a data folder reference instead of a string containing a path.

Parameters

index starts from zero. If no such object exists a zero length string ("") is returned.

objectType is one of the following values:

objectType	**What You Get**
1	Waves
2	Numeric variables
3	String variables
4	Data folders

Examples

```
Function PrintAllWaveNames()
	String objName
	Variable index = 0
	DFREF dfr = GetDataFolderDFR()	// Reference to current data folder
	do
		objName = GetIndexedObjNameDFR(dfr, 1, index)
		if (strlen(objName) == 0)
			break
		endif
		Print objName
		index += 1
	while(1)
End
```

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-72.

The **CountObjectsDFR** function.

GetKeyState

`GetKeyState(`*`flags`*`)`

The GetKeyState function returns a bitwise numeric value that indicates the state of certain keyboard keys.

To detect keyboard events directed toward a window that you have created, such as a panel window, use the window hook function instead of GetKeyState. See **Window Hook Functions** on page IV-276 for details.

GetKeyState is normally called from a procedure that is invoked directly through a user-defined button or user-defined menu item. The procedure tests the state of one or more modifier keys and adjusts its behavior accordingly.

Another use for GetKeyState is to determine if Escape is pressed. This can be used to detect that the user wants to stop a procedure.

GetKeyState tests the keyboard at the time it is called. It does not tell you if keys were pressed between calls to the function. Consequently, when a procedure uses the escape key to break a loop, the user must press Escape until the running procedure gets around to calling the function.

Parameters

flags is a bitwise parameter interpreted as follows:

Bit 0:	If set, GetKeyState reports keys keys even if Igor is not the active application. If cleared, GetKeyState reports keys only if Igor is the active application.

All other bits are reserved and must be zero.

Details

When set, the return value is interpreted bitwise as follows:

Bit 0:	Command (*Macintosh*) or Ctrl (*Windows*) pressed.
Bit 1:	Option (*Macintosh*) or Alt (*Windows*) pressed.
Bit 2:	Shift pressed.
Bit 3:	Caps Lock pressed.
Bit 4:	Control pressed (*Macintosh only*).
Bit 5:	Escape pressed.
Bit 6:	Left arrow key pressed. Supported in Igor Pro 7.00 or later.
Bit 7:	Right arrow key pressed. Supported in Igor Pro 7.00 or later.
Bit 8:	Up arrow key pressed. Supported in Igor Pro 7.00 or later.
Bit 9:	Down arrow key pressed. Supported in Igor Pro 7.00 or later.

To test if a particular key is pressed, do a bitwise AND of the return value with the mask value 1, 2, 4, 8, 16, 32, 64, 128, 256 and 512 for bits 0 through 9 respectively. To test if a particular key and only that key is pressed compare the return value with the mask value.

On Macintosh, it is currently possible to define a keyboard shortcut for a user-defined menu item and then, in the procedure invoked by the keyboard shortcut, to use GetKeyState to test for modifier keys. This is not possible on Windows because the keyboard shortcut will not be activated if a modifier key not specified in the keyboard shortcut is pressed. It is also possible that this ability on Macintosh will be compromised by future operating system changes. On both operating systems, however, you can test for a modifier key when the user chooses a user-defined menu item or clicks a user-defined button using the mouse.

Examples

```
Function ShiftKeyExample()
   Variable keys = GetKeyState(0)

   if (keys == 0)
      Print "No modifier keys are pressed."
   endif
```

```
    if (keys & 4)
        if (keys == 4)
            Print "The Shift key and only the Shift key is pressed."
        else
            Print "The Shift key is pressed."
        endif
    endif
End

Function EscapeKeyExample()
    Variable keys

    do
        keys = GetKeyState(0)
        if ((keys & 32) != 0)          // User is pressing escape?
            break
        endif
    while(1)
End
```

See Also

Keyboard Shortcuts on page IV-127. **Setting Bit Parameters** on page IV-12 for details about bit settings.

GetLastUserMenuInfo

`GetLastUserMenuInfo`

The GetLastUserMenuInfo operation sets variables in the local scope to indicate the value of the last selected user-defined menu item.

Details

GetLastUserMenuInfo creates and sets these special variables:

`V_flag` The kind of menu that was selected:

V_flag	Menu Kind
0	Normal text menu item, including **Optional Menu Items** (see page IV-122) and **Multiple Menu Items** (see page IV-122).
3	"*FONT*"
6	"*LINESTYLEPOP*"
7	"*PATTERNPOP*"
8	"*MARKERPOP*"
9	"*CHARACTER*"
10	"*COLORPOP*"
13	"*COLORTABLEPOP*"

See **Specialized Menu Item Definitions** on page IV-123 for details about these special user-defined menus.

`V_value` Which menu item was selected. The value also depends on the kind of menu the item was selected from:

V_flag	V_value meaning
0	Text menu item number (the first menu item is number 1).
3	Font menu item number (use S_Value, instead).
6	Line style number (0 is solid line)
7	Pattern number (1 is the first selection, a SW-NE light diagonal).
8	Marker number (1 is the first selection, the X marker).
9	Character as an integer, = char2num(S_Value). Use S_Value instead.
10	Color menu item (use V_Red, V_Green, V_Blue, and V_Alpha instead).
13	Color table list menu item (use S_Value instead).

`S_value` The menu item text, depending on the kind of menu it was selected from:

V_flag	S_value meaning
0	Text menu item text.
3	Font name or “default”.
6	Name of the line style menu or submenu.
7	Name of the pattern menu or submenu.
8	Name of the marker menu or submenu.
9	Character as string.
10	Name of the color menu or submenu.
13	Color table name.

In the case of **Specialized Menu Item Definitions** (see page IV-123), S_value will be the title of the menu or submenu, etc.

`V_Red`, `V_Green`, `V_Blue`, `V_Alpha`

If a user-defined color menu ("*COLORPOP*" menu item) was chosen then these values hold the red, green, and blue values of the selected color. The values range from 0 to 65535.

Will be 0 if the last user-defined menu selection was not a color menu selection.

`S_graphName`, `S_traceName`

These are set only when any user-defined menu is chosen from a graph’s trace contextual menu. (Menu “TracePopup” or Menu “AllTracesPopup” definitions).

Initially `""` until a user-defined menu selection was made from one of these contextual menus, these are not reset for each user-defined menu selection.

`S_graphName` is the full host-child specification for the graph. If the graph is embedded into a host window, `S_graphName` might be something like “Panel0#G0”. See **Subwindow Syntax** on page III-87.

`S_traceName` is name of the trace that was selected by the trace contextual menu, or `""` if the AllTracesPopup menu was chosen. See **Subwindow Syntax** on page III-87.

Examples

A Multiple Menu Items menu definition:

```
Menu "Wave List", dynamic
    "Menu Item 1", <some command>
    "Menu Item 2", <some command>
    WaveList("*",";",""), DoSomethingWithWave()
End
```

If the user selects one of the (many) menu items created by the "Wave List" menu item definition, the DoSomethingWithWave user function can call GetLastUserMenuInfo to determine which wave was selected:

```
Function DoSomethingWithWave()
    GetLastUserMenuInfo
    WAVE/Z selectedWave = $S_value
    …use selectedWave for something…
End
```

A trivial user-defined color menu definition:

```
Menu "Color"
    "*COLORPOP*", DoSomethingWithColor()
End

Function DoSomethingWithColor()
    GetLastUserMenuInfo
    ... do something with V_Red, V_Green, V_Blue, V_Alpha ...
End
```

See **Specialized Menu Item Definitions** on page IV-123 for another color menu example.

A Trace contextual menu Items menu definition:

```
Menu "TracePopup", dynamic        // menu when a trace is right-clicked
    "-"                    // separator divides this from built-in menu items
    ExportTraceName(), ExportSelectedTrace()
End

Function/S ExportTraceName()
    GetLastUserMenuInfo    // only S_graphName, S_traceName are set.
    Return "Export "+S_traceName
End

Function ExportSelectedTrace()
    GetLastUserMenuInfo
    …do something with S_graphName, S_traceName…
End
```

See Also

Chapter IV-5, **User-Defined Menus** and especially the sections **Optional Menu Items** on page IV-122, **Multiple Menu Items** on page IV-122, and **Specialized Menu Item Definitions** on page IV-123.

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

GetMarquee

`GetMarquee` `[`**`/K/W=`*`winName`*`/Z`**`]` `[`***`axisName`***` [, `***`axisName`***`]]`

The GetMarquee operation provides a way for you to use the marquee as an input mechanism in graphs and page layout windows. It puts information about the marquee into variables.

Parameters

If you specify *axisName* (allowed only for graphs) the coordinates are in axis units. If you specify an axis that does not exist, Igor generates an error.

If you specify only one axis then Igor sets only the variables appropriate to that axis. For example, if you execute "`GetMarquee left`" then Igor sets the V_bottom and V_top variables but does *not* set V_left and V_right.

Flags

/K	Kills the marquee. Usually you will want to kill the marquee when you call GetMarquee, so you should use the /K flag. This is modeled after what happens when you create a marquee in a graph and then choose Expand from the Marquee menu. There may be some situations in which you want the marquee to persist. Igor also automatically kills the marquee anytime the window containing the marquee is deactivated, including when a dialog is summoned.

/W=*winName*	Specifies the named window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/Z	No runtime error generated if the target window isn't a graph or layout, but V_flag will be zero. /Z does not prevent other kinds of problems from generating a runtime error.

Details

GetMarquee is intended to be used in procedures invoked through user menu items added to the graph Marquee menu and the layout Marquee menu.

GetMarquee sets the following variables and strings:

`V_flag`	0: There was no marquee when GetMarquee was invoked. 1: There was a marquee when GetMarquee was invoked.
`V_left`	Marquee left coordinate.
`V_right`	Marquee right coordinate.
`V_top`	Marquee top coordinate.
`V_bottom`	Marquee bottom coordinate.
`S_marqueeWin`	Name of window containing the marquee, or `""` if no marquee. If subwindow, subwindow syntax will be used.

When called from the command line, GetMarquee sets global variables and strings in the current data folder. When called from a procedure, it sets local variables and strings.

In addition, creating, adjusting, or removing a marquee may set additional marquee global variables (see the **Marquee Globals** section, below).

The target window must be a layout or a graph. Use /Z to avoid generating a runtime-error (V_flag will be 0 if the target window was not a layout or graph).

If the target is a layout then Igor sets the variables in units of points relative to the top/left corner of the paper.

If the target is a graph then Igor sets V_left and V_right based on the specified horizontal axis. If no horizontal axis was specified, V_left and V_right are set relative to the left edge of the base window in points.

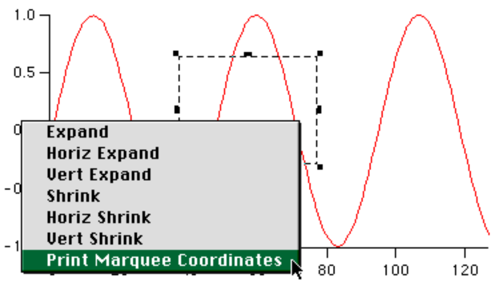
If the target is a graph then Igor sets V_bottom and V_top based on the specified vertical axis. If no vertical axis was specified, V_top and V_bottom are set relative to the top edge of the base window in points.

If there is no marquee when you invoke GetMarquee then Igor sets V_left, V_top, V_right, V_bottom based on the last time the marquee was active.

GetMarquee Example

```
Menu "GraphMarquee"
   "Print Marquee Coordinates", PrintMarqueeCoords()
End

Function PrintMarqueeCoords()
   GetMarquee left, bottom
   if (V_flag == 0)
      Print "There is no marquee"
   else
      printf "marquee left in bottom axis terms: %g\r", V_left
      printf "marquee right in bottom axis terms: %g\r", V_right
      printf "marquee top in left axis terms: %g\r", V_top
      printf "marquee bottom in left axis terms: %g\r", V_bottom
   endif
End
```

You can run this procedure by putting it into the procedure window, making a marquee in a graph, clicking in the marquee and choosing Print Marquee Coordinates:



The procedure calls GetMarquee to set the local marquee variables and then prints their values in the history area:

```
PrintMarqueeCoords()
  marquee left in bottom axis terms: 32.1149
  marquee right in bottom axis terms: 64.7165
  marquee top in left axis terms: 0.724075
  marquee bottom in left axis terms: -0.131061
```

Marquee Globals

You can cause Igor to update global marquee variables whenever the user adjusts the marquee (without the need for you to invoke GetMarquee) by creating a global variable named V_marquee in the root data folder:

```
Variable/G root:V_marquee = 1     //Creates V_marquee and sets bit 0 only
```

When the user adjusts the marquee Igor checks to see if root:V_marquee exists and which bits are set, and updates (and creates if necessary) these globals:

`Variable/G root:V_left`	Marquee left coordinate.
`Variable/G root:V_right`	Marquee right coordinate.
`Variable/G root:V_top`	Marquee top coordinate.
`Variable/G root:V_bottom`	Marquee bottom coordinate.
`String/G root:S_marqueeWin`	Name of window that contains marquee, or `""` if no marquee. Set only if root:V_Marquee has bit 15 (0x8000) set.

Unlike the local variables, for graphs these global variables are never in points. Root:V_left and V_right will be axis coordinates based on the first bottom axis created for the graph (if none, then for the first top axis). The axis creation order is the same as is returned by **AxisList**. Similarly, root:V_top and root:V_bottom will be axis coordinates based on the first left axis or the first right axis.

Igor examines the global root:V_marquee for bitwise flags to decide which globals to update, and when:

root:V_marquee Bit Meaning	Bit Number	Bit Value
Update global variables for graph marquees	0	1
Update global variables for layout marquees	2	4
Update S_marqueeWin when updating global variables	15	0x8000

Marquee Globals Example

By creating the global variable root:V_marquee this way:

```
Variable/G root:V_marquee = 1 + 4 + 0x8000
```

whenever the user creates, adjusts, or removes a marquee in any graph or layout Igor will create and update the global root:V_left, etc. coordinate variables and set the global string root:S_marqueeWin to the name of the window which has the marquee in it. When the marquee is removed, root:S_marqueeWin will be set to `""`.

This mechanism does neat things by making a **ValDisplay** or **SetVariable** control depend on any of the globals. See the Marquee Demo experiment in the Examples:Feature Demos folder for an example.

You can also cause a function to run whenever the user creates, adjusts, or removes a marquee by setting up a dependency formula using **SetFormula** to bind one of the marquee globals to one of the function’s input arguments:

```
Variable/G root:dependencyTarget

SetFormula root:dependencyTarget, "MyMarqueeFunction(root:S_marqueeWin)"

Function MyMarqueeFunction(marqueeWindow)
   String marqueeWindow          // this will be root:S_marqueeWin

   if( strlen(marqueeWindow) )
      NVAR V_left= root:V_left, V_right= root:V_right
      NVAR V_top= root:V_top, V_bottom= root:V_bottom
      Printf marqueeWindow + " has a marquee at: "
      Printf "%d, %d, %d, %d\r", V_left, V_right, V_top, V_bottom
   else
      Print "The marquee has disappeared."
   endif

   return 0                      // return value doesn't really matter
End
```

See Also

The **SetMarquee** and **SetFormula** operations. **Setting Bit Parameters** on page IV-12 for information about bit settings.

GetMouse

`GetMouse [/W=`*`winName`*`]`

The GetMouse operation returns information about the position of the input mouse, and the state of the mouse buttons.

GetMouse is useful in situations such as background tasks where the mouse position and state aren't available as they are in control procedures and window hook functions.

Flags

/W=*winName*	Returns the mouse position relative to the named window or subwindow. When identifying a subwindow with winName, see **Subwindow Syntax** on page III-87.
/W=kwTopWin	Returns the mouse position relative to the currently frontmost non-floating window.
/W=kwCmdHist	Returns the mouse position relative to the command window.
/W=Procedure	Returns the mouse position relative to the main Procedure window.

Details

GetMouse returns the mouse position in local coordinates relative to the specified window unless /W is omitted in which case the returned coordinates are global.

On Windows, global coordinates are actually relative to the frame window. See **GetWindow** wsizeDC kwFrameInner.

Information is returned via the following string and numeric variables:

V_left	Horizontal mouse position, in pixels.
V_top	Vertical mouse position, in pixels.
V_flag	Mouse button state. V_flag is a bitwise value with each bit reporting the mouse button states: Bit 0: 1 if the primary mouse button (usually the left) is down, 0 if it is up. Bit 1: 1 if the secondary mouse button (usually the right) is down, 0 if it is up. On Macintosh, the secondary mouse button can be invoked by pressing the control key while clicking the primary (often the only) mouse button, but GetMouse does not report this with bit 1 set. Use **GetKeyState**'s bit 4 to test if the control key is pressed. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

S_name	Name of the window or subwindow which the position is relative to, or "" if not a nameable window or if /W was omitted. Most useful with /W=kwTopWin. The result can be kwCmdHist or Procedure, or the name of a target window.

See Also

GetWindow, **GetKeyState**, **SetWindow**, **WMWinHookStruct**, **WMButtonAction**

Background Tasks on page IV-298, **Subwindow Syntax** on page III-87

GetRTError

`GetRTError(`*`flag`*`)`

The GetRTError function returns information about the error state of Igor's user-defined function runtime execution environment.

If *flag* is 0, GetRTError returns an error code if an error has occurred or 0 if no error has occurred.

If *flag* is 1, GetRTError returns an error code if an error has occurred or 0 if no error has occurred and it clears the error state of Igor’s runtime execution environment. Use this if you want to detect and handle runtime errors yourself.

If *flag* is 2, GetRTError returns the state of Igor's internal abort flag but does not clear it.

For *flag*=0 and *flag*=1, you can call **GetErrMessage** to obtain the error message associated with the returned error code, if any.

In Igor Pro 7.00 or later, using GetRTError(1) on the same line as a command that causes an error overrides the debugger “debug on error” setting and prevents the debugger from activating for that error.

Example

```
// This illustrates how to detect and handle a runtime error
// rather than allowing it to cause Igor to abort execution.
Function Demo()
    <Call an Igor operation or a user-defined function>
    Variable err = GetRTError(0)
    if (err != 0)
        String message = GetErrMessage(err)
        Printf "Error in Demo: %s\r", message
        err = GetRTError(1)                     // Clear error state
        Print "Continuing execution"
    endif
    <Call an Igor operation or a user-defined function>
End
```

See also

The **GetErrMessage** and **GetRTErrMessage** functions.

GetRTErrMessage

`GetRTErrMessage()`

In a user function, GetRTErrMessage returns a string containing the name of the operation that caused the error, a semicolon, and an error message explaining the cause of the error. This is the same information that appears in the alert dialog displayed. If no error has occurred, the string will be of zero length. GetRTErrMessage must be called before the error is cleared by calling GetRTError with a nonzero argument.

See also

The **GetRTError** and **GetErrMessage** functions.

GetRTLocation

`GetRTLocation(`*`sleepMS`*`)`

GetRTLocation is used for profiling Igor procedures.

You will typically not call GetRTLocation directly but instead will use it through FunctionProfiling.ipf which you can access using this include statement:

```
#include <FunctionProfiling>
```

GetRTLocation is called from an Igor preemptive thread to monitor the main thread. It returns a code that identifies the current location in the procedure files corresponding to the procedure line that is executing in the main thread.

Parameters

sleepMs is the number of milliseconds to sleep the preemptive thread after fetching a value.

Details

The result from GetRTLocation is passed to **GetRTLocInfo** to determine the location in the procedures. This samples the main thread only and the location becomes meaningless after any procedure editing.

See Also

GetRTLocInfo

GetRTLocInfo

`GetRTLocInfo(`*`code`*`)`

GetRTLocInfo is used for profiling Igor procedures.

You will typically not call GetRTLocInfo directly but instead will use it through FunctionProfiling.ipf which you can access using this include statement:

```
#include <FunctionProfiling>
```

GetRTLocation is called from an Igor preemptive thread to monitor the main thread. It returns a key/value string containing information about the procedure location associated with code or "" if the location could not be found.

Parameters

code is the result from a very recent call to **GetRTLocation**.

Details

The format of the result string is:

```
"PROCNAME:name;LINE:line;FUNCNAME:name;"
```

As of Igor Pro 7.03, if the code is in an independent module other than ProcGlobal then this appears at the beginning of the result string:

```
IMNAME:inName;
```

The line number is padded with zeros to facilitate sorting.

See Also

GetRTLocation

GetRTStackInfo

`GetRTStackInfo(`*`selector`*`)`

The GetRTStackInfo function returns information about “runtime stack” (the chain of macros and functions that are executing).

Details

If *selector* is 0, GetRTStackInfo returns a semicolon-separated list of the macros and procedures that are executing. This list is the same you would see in the debugger’s stack list.

The currently executing macro or function is the last item in the list, the macro or function that started execution is the first item in the list.

If *selector* is 1, it returns the name of the currently executing function or macro.

If *selector* is 2, it returns the name of the calling function or macro.

If *selector* is 3, GetRTStackInfo returns a semicolon-separated list of routine names, procedure file names and line numbers. This is intended for advanced debugging by advanced programmers only.

For example, if RoutineA in procedure file ProcA.ipf calls RoutineB in procedure file ProcB.ipf, and RoutineB calls GetRTStackInfo(3), it will return:

```
RoutineA,ProcA.ipf,7;RoutineB,ProcB.ipf,12;
```

The numbers 7 and 12 would be the actual numbers of the lines that were executing in each routine. Line numbers are zero-based.

GetRTStackInfo does not work correctly with string macros executed via the Execute operation.

In future versions of Igor, *selector* may request other kinds of information.

Examples

```
Function Called()
   Print "Called by " + GetRTStackInfo(2) + "()"
   Print "Routines in calling chain: " + GetRTStackInfo(0)
End

Function Calling()
   Called()
End

Macro StartItUp()
   Calling()
End

// Executing StartItUp() prints:
  Called by Calling()
  Routines in calling chain: StartItUp;Calling;Called;
```

See Also

StringFromList, **ItemsInList**, and **GetRTError** functions. **The Stack and Variables Lists** on page IV-202.

GetScrapText

`GetScrapText()`

The GetScrapText function returns a string containing any plain text on the Clipboard (aka “scrap”). This is the text that would be pasted into a text document if you used Paste in the Edit menu.

See Also

The **PutScrapText** and **LoadPICT** operations.

GetSelection

`GetSelection` ***`winType, winName, bitflags`***

The GetSelection operation returns information about the current selection in the specified window.

Parameters

winType is one of the following keywords:

`graph, panel, table, layout, notebook, procedure`

winName is the name of a window of the specified type.

When identifying a subwindow with winName, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

If *winType* is procedure then *winName* is actually a procedure window title inside a $"" wrapper, such as:

```
GetSelection procedure $"DemoLoader.ipf", 3
```

bitflags is a bitwise parameter that is used in different ways for different window types, as described in **Details**. You should use 0 for undefined bits. **Setting Bit Parameters** on page IV-12 for further details about bit settings.

Details

For all window types, GetSelection sets V_flag:

`V_flag`	0: No selection when GetSelection was invoked.
	1: There was a selection when GetSelection was invoked.

Here is a description of what GetSelection does for each window type:

Examples

In a new experiment, make a table named “Table0” with some columns, and select some combination of rows and columns:

winType	*bitFlags*	**Action**
`graph`		Does nothing.
`panel`		Does nothing.
`table`	1	Sets V_startRow, V_startCol, V_endRow, and V_endCol based on the selected cells in the table. The top/left cell, not including the Point column, is (0, 0).
	2	Sets S_selection to a semicolon-separated list of column names.
	4	Sets S_dataFolder to a semicolon-separated list of data folders, one for each column.
`layout`	2	Sets S_selection to a semicolon separated list of selected objects in the layout layer (not any drawing layers). S_selection will be `""` if no objects are selected.
`notebook`	1	Sets V_startParagraph, V_startPos, V_endParagraph, and V_endPos based on the selected text in the notebook.
	2	Sets S_selection to the selected text.
`procedure`	1	Sets V_startParagraph, V_startPos, V_endParagraph, V_endPos based on the selected text in the procedure window.
	2	Sets S_selection to the selected text.

```
Make wave0 = p
Make wave1 = p + 1
Edit wave0, wave1
ModifyTable selection = (3,0,8,1,3,0)
```

Now execute these commands in a procedure or in the command line:

```
GetSelection table, Table0, 3
Print V_flag, V_startRow, V_startCol, V_endRow, V_endCol
Print S_selection
```

This will print the following in the history area:

```
  1  3  0  8  1
  wave0.d;wave1.d;
```

GetUserData

`GetUserData(`*`winName, objID, userdataName`*`)`

The GetUserData function returns a string containing the user data for a window or subwindow. The return string will be empty if no user data exists.

Parameters

winName may specify a window or subwindow name. Use `""` for the top window.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

objID is a string specifying the name of a control or graph trace. Use `""` for a window or subwindow.

userdataName is the name of the user data or `""` for the default unnamed user data.

See Also

The **ControlInfo**, **GetWindow**, and **SetWindow** operations.

GetWavesDataFolder

`GetWavesDataFolder(`*`waveName, kind`*`)`

The GetWavesDataFolder function returns a string containing the name of a data folder containing the wave named by *waveName*. Variations on the theme are selected by *kind*.

The most common use for this is in a procedure, when you want to create a wave or a global variable in the data folder containing a wave passed as a parameter.

GetWavesDataFolderDFR is preferred.

Details

kind	**GetWavesDataFolder Returns**
0	Only the name of the data folder containing *waveName*.
1	Full path of data folder containing *waveName*, without wave name.
2	Full path of data folder containing *waveName*, including possibly quoted wave name.
3	Partial path from current data folder to the data folder containing *waveName*.
4	Partial path including possibly quoted wave name.

Kinds 2 and 4 are especially useful in creating command strings to be passed to **Execute**.

Examples

```
Function DuplicateWaveInDataFolder(w)
   Wave w
   DFREF dfSav = GetDataFolderDFR()
   SetDataFolder GetWavesDataFolder(w,1)
   Duplicate/O w, $(NameOfWave(w) + "_2")
   SetDataFolder dfSav
End
```

See Also

Chapter II-8, **Data Folders**.

GetWavesDataFolderDFR

GetWavesDataFolderDFR(*waveName*)

The GetWavesDataFolderDFR function returns a data folder reference for the data folder containing the specified wave.

GetWavesDataFolderDFR is the same as GetWavesDataFolder except that it returns a data folder reference instead of a string containing a path.

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-72.

GetWindow

GetWindow [/Z] *winName*, *keyword*

The GetWindow operation provides information about the named window or subwindow. Information is returned in variables, strings, and waves.

Parameters

winName can be the name of graph, table, panel, page layout, notebook, or any subwindow. It can also be the title of a procedure window or one of these four special keywords:

kwTopWin	Specifies the topmost graph, table, panel, page layout, or notebook window.
kwCmdHist	Specifies the command history area.
kwFrameOuter	Specifies the “frame” or “application” window that Igor Pro has only under Windows. This is the window that contains Igor’s menus and status bar.
kwFrameInner	Specifies the inside of the same “frame” window under Windows. This is the window that all other Igor windows are inside.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Only one of the following keywords may follow *winName*. The keyword chosen determines the information stored in the output variables:

active	Sets V_Value to 1 if the window is active or to 0 otherwise. Active usually means the window is the frontmost window.
activeSW	Stores the window “path” of currently active subwindow in S_Value. See **Subwindow Syntax** on page III-87 for details on the window hierarchy.
backRGB	Sets V_Red, V_Green, V_Blue, and V_Alpha to the background color of the window. The values range from 0 to 65535. The background color is set with **ModifyGraph (colors)** wbRGB, **ModifyLayout** bgRGB, and **Notebook** backRGB. Also returns the background color of procedure windows and the command/history windows. Other windows set these values to 65535 (opaque white). Added in Igor Pro 7.00.
exterior	Sets V_value to 1 if the window is an exterior panel window or to 0 otherwise. Useful for window hook functions that must work for both regular windows and exterior panel windows, since exterior panels use their own hook function.
bgRGB	Another name for backRGB. Added in Igor Pro 7.00.
cbRGB	Sets V_Red, V_Green, V_Blue, and V_Alpha to the control panel area background color of the window in graphs or panel windows, as set by **ModifyGraph (colors)** cbRGB and **ModifyPanel** cbRGB. Other windows set these values to 65535 (opaque white). Added in Igor Pro 7.00.
drawLayer	If the specified window is a graph, layout, or panel window then the window's current drawing layer is returned in S_value. S_value is set to "" for other windows. See **Drawing Layers** on page III-68. Added in Igor Pro 7.00.
expand	For graph windows, sets V_Value to the expand value set by **ModifyGraph (general)** expand=value, a value normally 0 or 1, where 1.5 means 150%. For notebook, procedure and command windows, sets V_Value to the magnification, normally 100. See the **Notebook** magnification=m documentation for details. For layout windows, sets V_Value to the **ModifyLayout** mag=m value, usually 0.5 (50%). Table windows set V_Value to 0, panels and other unsupported windows to NaN. Added in Igor Pro 7.00.
file	Works for notebook and procedure windows only. Returns via S_value a semicolon-separated list containing: - the file name - the Mac path to the folder containing the file with a colon at the end - the name of a symbolic path pointing to that folder, if any If the window was never saved to a standalone file then "" is returned in S_value. If the specified window is not a notebook or procedure window then "" is returned in S_value.

gbRGB	Sets V_Red, V_Green, V_Blue, and V_Alpha to the plot area background color of the window in graph windows, as set by **ModifyGraph (colors)** gbRGB. Other windows set these values to 65535 (opaque white). Use wbRGB to get the color of window background (the area outside of the axes). Added in Igor Pro 7.00.
gsize	Reads graph outer dimensions into V_left, V_right, V_top, and V_bottom in local coordinates. This includes axes but not the tool palette, control bar, or info panel. Dimensions are in points.
gsizeDC	Same as gsize but dimensions are in device coordinates (pixels).
hide	Sets V_Value bit 0 if the window or subwindow is hidden. Sets bit 1 if the host window is minimized. Sets bit 2 if the subwindow is hidden only because an ancestor window or subwindow is hidden. Added in Igor Pro 7.00. On Macintosh, if you execute `MoveWindow 0,0,0,0` to minimize a window to the dock, and then you immediately call `GetWindow hide`, bit 1 may not be correctly set because of the delay caused by the animation of the window sliding into the dock.
hook	Copies name of window hook function to S_value. See **Unnamed Window Hook Functions** on page IV-286.
hook(*hName*)	For the given named hook *hName*, copies name of window hook function to S_value. See **Named Window Hook Functions** on page IV-277.
logicalpapersize	Returns logical paper size of the page setup associated with the named window into V_left, V_right, V_top, and V_bottom. Dimensions are in points. If the Page Setup dialog uses 100% scaling, these are also the physical dimensions of the page. V_left and V_top are 0 and correspond to the left top corner of the physical page. On the Macintosh, using a Scale of 50% multiplies all of these dimensions by 2.
logicalprintablesize	Returns logical printable size of the page setup associated with the named window into V_left, V_right, V_top, and V_bottom. Dimensions are in points. If the Page Setup dialog uses 100% scaling, these are also the physical dimensions of the page minus the margins. V_left and V_top are the number of points from the left top corner of the physical page to the left top corner of the printable area of page. On the Macintosh, using a page setup scale of 50% multiplies all of these dimensions by 2.
magnification	Sets V_Value exactly the same way that expand does. Added in Igor Pro 7.00.
maximize	Sets V_Value to 1 if the window is maximized, 0 otherwise. On Macintosh, V_Value is always 0.
needUpdate	Sets V_Value to 1 if window or subwindow is marked as needing an update.
note	Copies window note to S_value.
psize	Reads graph plot area dimensions (where the traces are) into V_left, V_right, V_top, and V_bottom in local coordinates. Dimensions are in points.
psizeDC	Same as psize but dimensions are in device coordinates (pixels).

sizeLimit	Returns the size limits imposed on a window via `SetWindow sizeLimit` in the V_minWidth, V_minHeight, V_maxWidth and V_maxHeight. The values are scaled for screen resolution to the same units as GetWindow wsize, which is points. Very large limits are returned as INF. The sizeLimit keyword was added in Igor Pro 7.00. Also returns a sizeLimit status value in V_Value. 0 means no SetWindow sizeLimit command will appear in the window's recreation macro, usually because no `SetWindow sizeLimit` command was applied to the window. 1 means a `SetWindow sizeLimit` command will appear in the window's recreation macro. -1 means it won't appear because of conflicts with graph absolute sizing modes.
title	Gets the title (set by as *titleStr* with NewPanel, Display, etc., or by the Window Control dialog) and puts it into S_value. S_value is set to "" if *winName* specifies a subwindow. See also the wtitle keyword, below.
userdata	Returns the primary (unnamed) user data for a window in S_value. Use **GetUserData** to obtain any named user data.
wavelist	Creates a 3 column text wave called W_WaveList containing a list of waves used in the graph in *winName*. Each wave occupies one row in W_WaveList. This list includes all waves that can be in a graph, including the data waves for contour plots and images.
wbRGB	Another name for backRGB. Added in Igor Pro 7.00.
wsize	Reads window dimensions into V_left, V_right, V_top, and V_bottom in points from the top left of the screen. For subwindows, values are local coordinates in the host.
wsizeDC	Same as wsize but dimensions are in local device coordinates (pixels). The origin is the top left corner of the host window’s active rectangle.
wsizeOuter	Reads window dimensions into V_left, V_right, V_top, and V_bottom in points from the top left of the screen. Dimensions are for the entire window including any frame and title bar. For subwindows, values are for the host window.
wsizeOuterDC	Same as wsizeOuter but dimensions are in local device coordinates (pixels). The origin is the top left corner of the host window’s active rectangle, so V_top will be negative for a window with a title bar. V_left will be negative for windows with a frame; windows on Macintosh OS X have no frame, so V_left will be zero.
wsizeRM	Generally the same as wsize, but these are the coordinates that would actually be used by a recreation macro except that the coordinates are in points even if the window is a panel. Also, if the window is minimized or maximized, the coordinates represent the window’s restored location. On Windows, GetWindow kwFrameOuter wsizeRM returns the pixel coordinates of the MDI frame even when the frame is maximized. wsizeDC returns 2,2,2,2 in this case.
wtitle	Gets the actual window title displayed in the window's title bar, regardless of whether it was set by the user (see the title keyword above) or is the default title created by Igor, and puts it into S_value. S_value is set to "" if winName specifies a subwindow. If winName is kwFrameOuter or kwFrameInner, on Macintosh S_Value is set to the name of the Igor application. On Windows it is set to the full title of the application as seen on the frame's window, which can be altered using DoWindow/T kwFrame.

Flags

/Z	Suppresses error if, for instance, *winName* doesn't name an existing window. V_flag is set to zero if no error occurred or to a non-zero error code.

Details

winName can be the title of a procedure window. If the title contains spaces, use:

```
GetWindow $"Title With Spaces" wsize
```

However, if another window has a name which matches the given procedure window title, that window's properties are returned instead of the procedure window.

The wsize parameter is appropriate for all windows.

The gsize, psize, gbRGB, and wavelist parameters are appropriate only for graph windows.

The logicalpapersize, logicalprintablesize and expand/magnification parameters are appropriate for all printable windows except for control panels and Gizmo plots.

Local Coordinates

"Local coordinates" are relative to the top left of the graph area, regardless of where that is on the screen or within the graph window. All dimensions are reported in units of points (1/72 inch) regardless of screen resolution. On the Macintosh, this is the same as screen pixels.

Frame Window Coordinates

kwCmdHist, kwFrameInner, and kwFrameOuter may be used with only the wsize keyword.

On Windows computers, kwFrameInner and kwFrameOuter return coordinates into V_left, V_right, V_top, and V_bottom. On the Macintosh, they always return 0, because Igor has no frame on the Macintosh.

kwFrameOuter coordinates are the location of the outer edges of Igor's application window, expressed in screen (pixel) coordinates suitable for use with `MoveWindow/F` to restore, minimize, or maximize the Igor application window.

If Igor is currently minimized, kwFrameOuter returns 0 for all values. If maximized, it returns 2 for all values. Otherwise, the screen (pixel) coordinates of the frame are returned in V_left, V_right, V_top, and V_bottom. This is consistent with `MoveWindow/F`.

kwFrameInner coordinates, however, are the location of the inner edges of the application window, expressed in Igor window coordinates (points) suitable for positioning graphs and other windows with **MoveWindow**.

If Igor is currently minimized, kwFrameInner returns the inner frame coordinates Igor would have if Igor were "restored" with `MoveWindow/F 1,1,1,1`.

V_left and V_top will always both be 0, and V_Bottom and V_Right will be the maximum visible (or potentially visible) window (not screen) coordinates in points.

The Wavelist Keyword

The format of W_WaveList, created with the wavelist keyword, is as follows:

Column 1	Column 2	Column 3
Wave name	partial path to the wave	special ID number

The wave name in column 1 is simply the name of the wave with no path. It may be the same as other waves in the list, if there are waves from different data folders.

The partial path in column 2 includes the wave name and can be used with the $ operator to get access to the wave.

The special ID number in column 3 has the format ##<number>##. A version of the recreation macro for the graph can be generated that uses these ID numbers instead of wave names (see the **WinRecreation** function). This makes it relatively easy to find every occurrence of a particular wave using a function like **strsearch**.

Examples

```
// These commands draw a red foreground rectangle framing
// the printable area of a page layout window.
GetWindow Layout0 logicalpapersize
DoWindow/F Layout0
```

```
SetDrawLayer/K userFront
SetDrawEnv linefgc=(65535,0,0), fillpat=0         // Transparent fill
DrawRect V_left+1, V_top+1, V_right-1, V_bottom-1

// These commands demonstrate the difference between title and wtitle.
Make/O data=x
Display/N=MyGraph data
GetWindow MyGraph title;Print S_Value             // Prints nothing (S_Value = "")
GetWindow MyGraph wtitle;Print S_Value            // Prints "MyGraph:data"
DoWindow/T MyGraph, "My Title for My Graph"
GetWindow MyGraph title;Print S_Value             // Prints "My Title for My Graph"
GetWindow MyGraph wtitle;Print S_Value            // Prints "My Title for My Graph"
```

See Also

The **SetWindow**, **GetUserData**, **MoveWindow** and **DoWindow** operations.

The **IgorInfo** function.

GizmoInfo

`GizmoInfo(`*`nameStr`*`, `*`key`*`)`

The GizmoInfo function is used to determine if a particular name is valid and unique as a Gizmo item names.

Documentation for the GizmoInfo function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "GizmoInfo"
```

GizmoPlot

`GizmoPlot`

GizmoPlot is a procedure subtype keyword that identifies a macro as being a Gizmo recreation macro. It is automatically used when Igor creates a window recreation macro for a Gizmo plot. See **Procedure Subtypes** on page IV-193 and **Saving and Recreating Graphs** on page II-261 for details.

GizmoScale

`GizmoScale(`*`dataValue`*`, `*`dimNumber`*` [, `*`gizmoNameStr`*`] )`

The GizmoScale function returns a scaled *dataValue* for the specified dimension. The scaled values are used to position non-data drawing objects in the Gizmo window.

Documentation for the GizmoScale function is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "GizmoScale"
```

gnoise

`gnoise(`*`num`*` [, `*`RNG`*`])`

The gnoise function returns a random value from a Gaussian distribution such that the standard deviation of an infinite number of such values would be *num*.

The random number generator is initialized using the system clock when you start Igor, virtually guaranteeing that you will never get the same sequence twice. If you want repeatable “random” numbers, use **SetRandomSeed**.

The Gaussian distribution is achieved using a Box-Muller transformation of uniform random numbers.

The optional parameter *RNG* selects one of two different pseudo-random number generators used to create the uniformly-distributed random numbers used as the input to the Box-Muller transformation. If omitted, the default is 1. The *RNG*’s are:

RNG	**Description**
1	Linear Congruential generator by L’Ecuyer with added Bayes-Durham shuffle. The algorithm is described in *Numerical Recipes* (2nd edition) as the function `ran2()`. This RNG has nearly 2^{32} distinct values and the sequence of random numbers has a period in excess of 10^{18}.
2	Mersenne Twister by Matsumoto and Nishimura. It is claimed to have better distribution properties and period of 2^{19937}-1.

In a complex expression, the gnoise function returns a complex value, as if you had called:

```
cmplx(gnoise(num), gnoise(num))
```

Except for enoise, other noise functions do not have complex implementations.

See Also

The **SetRandomSeed** operation and the **enoise** function.

Noise Functions on page III-344.

References

Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

Details about the Mersene Twister are in:

Matsumoto, M., and T. Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Trans. on Modeling and Computer Simulation*, *8*, 3-30, 1998.

More information is available online at: <`http://en.wikipedia.org/wiki/Mersenne_twister`>

Graph

`Graph`

Graph is a procedure subtype keyword that identifies a macro as being a graph recreation macro. It is automatically used when Igor creates a window recreation macro for a graph. See **Procedure Subtypes** on page IV-193 and **Saving and Recreating Graphs** on page II-261 for details.

GraphMarquee

`GraphMarquee`

GraphMarquee is a procedure subtype keyword that puts the name of the procedure in the graph Marquee menu. See **Marquee Menu as Input Device** on page IV-151 for details.

GraphNormal

`GraphNormal` `[`**`/W=`*`winName`***`]`

The GraphNormal operation returns the target or named graph to the normal mode, exiting any drawing mode that it may be in.

You would usually enter normal mode by choosing ShowTools from the Graph menu and clicking the crosshair tool.

Flags

/W=*winName*	Reverts the named graph window. This must be the first flag specified when used in a Proc or Macro or on the command line.

See Also

The **GraphWaveDraw** and **GraphWaveEdit** operations.

GraphStyle

`GraphStyle`

GraphStyle is a procedure subtype keyword that puts the name of the procedure in the Style pop-up menu of the New Graph dialog and in the Graph Macros menu. See **Graph Style Macros** on page II-262 for details.

GraphWaveDraw

`GraphWaveDraw` `[`***`flags`***`] [`***`yWaveName, xWaveName`***`]`

The GraphWaveDraw operation initiates drawing a curve composed of *yWaveName* vs *xWaveName* in the target or named graph. The user draws the curve using the mouse, and the values are stored in a pair of waves as XY data.

Normally, you would initiate drawing by choosing ShowTools from the Graph menu and clicking in the appropriate tool rather than using GraphWaveDraw.

Parameters

yWaveName and *xWaveName* will contain the y and x values of the curve drawn by the user with the mouse.

If *yWaveName* and *xWaveName* do not already exist, they are created with two points which are initially set to NaN (Not a Number) and appended to the target.

If *yWaveName* and *xWaveName* already exist, an error is generated unless the /O (overwrite) flag is present. If /O is present, the waves are re-created — with two points which are initially set to NaN — and appended to the target if they are not already in it.

If *yWaveName* and *xWaveName* are omitted then waves called W_YPoly*n* and W_XPoly*n* are created with two points set to NaN and appended to the target (*n* is some digit, so Igor might create a wave named W_YPoly0, for example).

Flags

/F[=*f*]	Initiates freehand drawing. In normal drawing, you click where you want a data point. In freehand drawing, you click once and then draw with the mouse button held down. If present, *f* specifies the smoothing factor. Max value is 8 (which is really slow), min value is 0 (default). The drawing tools use a value of 3 which is the recommended value.
/L/R/B/T	Specifies which axes to use (Left, Right, Bottom, Top). Bottom and Left axes are used by default. Can specify free axes using /L=*axis name* type notation. See **AppendToGraph** for details. If necessary, the specified axes will be created. If an axis is created its range is set to -1 to 1.
/M	Specifies that the curve being edited must be monotonic in the X dimension. The user is not allowed drag points so that they cross horizontally.
/O	Overwrites *yWaveName* and *xWaveName* if they already exist.
/W=*winName*	Draws in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

Once drawing starts no other user actions are allowed.

In normal mode, drawing stops when you double-click or when you click the first point (in which case the last point is set equal to the first point). When drawing finishes, the edit mode is entered.

In freehand mode, drawing stops when the mouse is released or when 10000 points have been drawn.

If /O is used and the waves are already on the graph then the first instance on the graph will be used even if they use a different pair of axes than specified.

See Also

The **GraphNormal**, **GraphWaveEdit** and **DrawAction** operations.

GraphWaveEdit

`GraphWaveEdit [`*`flags`*`]` ***`traceName`***

The GraphWaveEdit operation initiates editing a wave trace in a graph. The wave trace must already be in the graph.

Normally, you would initiate editing by choosing ShowTools from the Graph menu and clicking in the appropriate tool rather than using GraphWaveEdit.

Parameters

traceName is a wave name, optionally followed by the `#` character and an instance number: “`myWave#1`” is the *second* instance of `myWave` appended to the graph (“`myWave`” is the first).

If *traceName* is omitted then you get to pick the wave trace to edit by clicking it.

Flags

/M	Specifies that the edited trace must be monotonic in the X dimension. You cannot drag points so that they cross horizontally.
/NI	Suppresses automatic new point insertion when clicking between points.
/T=*t*	Sets the trace mode. *t*=0: Lines and small square markers (default). *t*=1: User settings unchanged.
/W=*winName*	Edits traces in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details
The GraphWaveEdit operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

See Also
The **GraphNormal**, **GraphWaveDraw** and **DrawAction** operations.

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

Grep

```
Grep [flags][srcFileStr ][srcTextWaveName][ as [destFileOrFolderStr]
    [destTextWaveName]]
```

The Grep operation copies lines matching a search expression from a file on disk, the Clipboard, or rows from a text wave to a new or existing file, an existing text wave, the History area, the Clipboard, or to S_value as a string list.

Source Parameters
The optional *srcFileStr* can be

- The full path to the file to copy lines from (in which case /P is not needed).
- The partial path relative to the folder associated with *pathName.*
- The name of a file in the folder associated with *pathName.*
- "Clipboard" to read lines of text from the Clipboard (in which case /P is ignored).

If Igor can not determine the location of the source file from *srcFileStr* and *pathName*, it displays a dialog allowing you to specify the source file.

The optional *srcTextWaveName* is the name or path to a text wave.

Only one of *srcFileStr* or *srcTextWaveName* may be specified. If neither is specified then an Open File dialog is presented allowing you to specify a source file.

Destination Parameters
The optional *destFileOrFolderStr* can be

- The name of (or path to) an existing folder when /D is specified.
- The name of (or path to) a possibly existing file.
- "Clipboard", in which case the matching lines are copied to the Clipboard (and /P and /D are ignored). The text can be retrieved with the **GetScrapText** function.

If *destFileOrFolderStr* is a partial path, it is relative to the folder associated with *pathName*.

If /D is specified, the source file is created inside the folder using the source file name.

If Igor can not determine the location of the destination file from *pathName*, *srcFileStr*, and *destFileOrFolderStr*, it displays a Save File dialog allowing you to specify the destination file (and folder).

The optional *destTextWaveName* is the name or path to an existing text wave. It may be the same wave as *srcTextWaveName*.

Only one of *destFileOrFolderStr* or *destTextWaveName* may be specified.

If no destination file or text wave is specified then matching lines are printed in the history area, unless the /Q flag is specified, in which case the matching lines aren't printed or copied anywhere (though the output variables are still set).

Use /LIST to set S_value to a string list containing the matching lines or rows.

Use /INDX to create a wave W_index containing the zero-based row or line number where matches were found.

Parameter Details

If you use a full or partial path for either *srcFileStr* or *destFileOrFolderStr*, see **Path Separators** on page III-401 for details on forming the path.

Folder paths should not end with single path separators. See **MoveFolder**'s Details section.

Flags

/A	Appends matching lines to the destination file, creating it if necessary, appends text to the Clipboard if the *destFileOrFolderStr* is "Clipboard", or appends rows to the destination text wave. Has no effect on output to the History area.
/D	Interprets *destFileOrFolderStr* as the name of (or path to) an existing folder (or directory). Without /D, *destFileOrFolderStr* is the name of (or path to) a file. It is ignored if the destination is a text wave, the Clipboard, or the History area. If *destFileOrFolderStr* is not a full path to a folder, it is relative to the folder associated with *pathName*.
/DCOL={*colNum*}	Useful only when the destination is *destTextWaveName*. Copies matching lines of text from the source file, Clipboard, or *srcTextWaveName* to column *colNum* of *destTextWaveName*, with any terminator characters removed. The default when the source is a file or the Clipboard is `/DCOL={0}`, which copies matching lines into the first column of *destTextWaveName*. The default when the source is *srcTextWaveName* is to copy each column of a matched row to the corresponding column in *destTextWaveName*. /DCOL must be used with the /A flag, otherwise the destination wave will have only 1 column.
/DCOL={[*colNum*] [, *delimStr*], …}	Useful only when the source is *srcTextWaveName* and the destination is a file, the Clipboard, History area, or S_value. Copies multiple columns in any order from matching rows of *srcTextWaveName* to the destination file, Clipboard, History area, or S_value. Construct the line by appending the contents of the numbered column and the *delimStr* parameters in the order specified. The output line is terminated as described in **Line Termination**.
/E=*regExprStr*	Specifies the Perl-compatible regular expression string. A line must match the regular expression to be copied to the output file. See **Regular Expressions**. *Multiple /E flags may be specified*, in which case a line is copied only if it matches every regular expression.
/E={*regExprStr*, *reverse*}	

Specifies the Perl-compatible regular expression string, *regExprStr*, for which the sense of the match can be changed by *reverse*.

reverse=1:	Matching expressions are taken to **not** match, and vice versa. For example, use `/E={"CheckBox",1}` to list all lines that do not contain "CheckBox".
reverse=0:	Same as /E=*regExptrStr*.

/ENCG=*textEncoding*

Specifies the text encoding of named text file. This flag applies if the source is a file and is ignored if the source is the clipboard or a text wave.

See **Text Encoding Names and Codes** on page III-434 for a list of accepted values for *textEncoding*.

If you omit /ENCG, Grep uses the default text encoding as specified by the Misc→Text Encoding→Default Text Encoding menu.

Passing 0 for textEncoding acts as if /ENCG were omitted. Passing 255 (binary) for textEncoding is treated as an error because the source text must be internally converted to UTF-8 and there is no valid conversion from binary to UTF-8.

/GCOL=*grepCol* Greps the specified column of *srcTextWaveName*, which is a two-dimensional text wave. The default search is on the first column (`grepCol=0`). Use `grepCol=-1` to match against any column of *srcTextWaveName*.

Does not apply if the source is a file or Clipboard.

/I Requires interactive searching even if *srcFileStr* and *destFileOrFolderStr* are specified and if the source file exists. Same as /I=3.

/I=*i* Specifies the degree of file search interactivity with the user.

i=0:	Default; interactive only if *srcFileStr* is not specified or if the source file is missing. Same as if /I were not specified. **Note**: This is different behavior than other commands such as **CopyFile**.
i=1:	Interactive even if *srcFileStr* is specified and the source file exists.
i=2:	Interactive even if *destFileOrFolderStr* is specified.
i=3:	Interactive even if *srcFileStr* is specified, the source file exists, and *destFileOrFolderStr* is specified.

/INDX Creates in the current data folder an output wave W_Index containing the line numbers (or row numbers) where matching lines were found. If this is the only output you need, also use the /Q flag.

/LIST[=*listSepStr*] Creates an output string variable S_value containing a semicolon-separated list of the matching lines. If *listSepStr* is specified, then it is used to separate the list items. See **StringFromList** for details on string lists. If this is the only output you need, also use the /Q flag.

/M=*messageStr* Specifies the prompt message in any Open File dialog. If /S is not specified, then *messageStr* will be used for both the Open File and Save File dialogs. But see **Prompt Does Not Work on Macintosh** on page IV-137.

/O Overwrites any existing destination file.

/P=*pathName* Specifies the folder containing the source file or the folder into which the file is copied. *pathName* is the name of an existing symbolic path.

Both *srcFileStr* and *destFileOrFolderStr* must be either simple file or folder names, or paths relative to the folder specified by *pathName*.

/Q	Prevents printing results to an output file, text wave, History, or Clipboard. Use /Q to check for a match to *regExprStr* by testing the value of V_flag, V_value, S_value (/LIST), or W_Index (/INDX) without generating any other matching line output. **Note**: When using /Q neither *destFileOrFolderStr* nor *destTextWaveName* may be specified.
/S=*saveMessageStr*	Specifies the prompt message in any Save File dialog.
/T=*termCharStr*	Specifies the terminator character. `/T=(num2char(13))` Carriage return (CR, ASCII code 13). `/T=(num2char(10))` Linefeed (LF, ASCII code 10). `/T=";"` Semicolon. `/T=""` Null (ASCII code 0). See **Line Termination** for the default behavior of the terminator character.
/Z[=z]	Prevents aborting of procedure execution when attempting to open a nonexistent file for searching. Use /Z if you want to handle this case in your procedures rather than having execution abort. *z*=0: Same as no /Z at all. *z*=1: Open file only if it exists. `/Z` alone is the same as `/Z=1`. *z*=2: Open file if it exists and display a dialog if it does not exist.

Line Termination

Line termination applies mostly to source and destination files. (The Clipboard and history area delimit lines with CR, and text waves have no line terminators.)

If /T is omitted, Grep will break file lines on any of the following: CR, LF, CRLF, LFCR. (Most Macintosh files use CR. Most Windows files use CRLF. Most UNIX files use LF. LFCR is an invalid terminator but some buggy programs generate files that use it.)

Grep reads whichever of these terminator(s) appear in the source file and use them to write lines to any output file.

The terminator(s) are removed before the line is matched against the regular expression.

For lines that match *regExprStr*, terminator(s) in the input file are transferred to the output file unless the output is the Clipboard or history area, in which case the output terminator is always only CR (like **LoadWave**). This means you can transparently handle files that use CR, LF, CRLF, or LFCR as the terminator, and omitting /T will be suitable for most cases.

If you use the /T flag, then Grep will terminate line-from-file reads on the specified character only and will output the specified character into any output file.

"Lines" in One-dimensional Text Waves

Grep considers each row of *srcTextWaveName* or *destTextWaveName* to be a "line" of input or output.

When the destination is a file, the Clipboard, or the History area, Grep copies all of the text in a matching row of *srcTextWaveName* to the file and terminates the line. See **Line Termination** for the rules on line terminators.

When the destination is a *destTextWaveName*, Grep simply copies all the text in a matching row to a row in *destTextWaveName*, without adding or omitting any terminators.

"Lines" and Columns in Two-Dimensional Text Waves

Grep by default *matches* against only the first column (column 0) of each row of *srcTextWaveName*. You can use the `/GCOL=`*`grepCol`* flag to specify a different column to match against. Use `/GCOL=-1` to match against any column of *srcTextWaveName*.

When the source is a text wave and the destination is a file, the Clipboard, or the History area, Grep by default *copies* only the first column (column 0) to the destination.

Use the `/DCOL={`*`colNum1, delimStr1, colNum2, delimStr2,...colNumN`*`}` to print multiple columns (in any order) with delimiters after each column (the last column number need not be followed by a delimiter string). The output line is terminated with CR or *termcharStr* as described in **Line Termination**.

When both the source and destination are text waves and append (/A) is *not* specified, the destination text wave is redimensioned to have the same number of columns as the source text wave, and all columns of matching rows of *srcTextWaveName* are copied to *destTextWaveName*.

When both the source and destination are text waves and append /A is specified, then the number of columns in *destTextWaveName* is left unchanged, and each column of *srcTextWaveName* is copied to the corresponding column of *destTextWaveName*.

If the destination is a text wave and the source is a file or the Clipboard, each line (without the terminator) is copied to the first column of the destination text wave, or use `/DCOL={`*`destColNum`*`}` to put the text into a different column.

Output Variables

The Grep operation returns information in the following variables. When running in a user-defined function these are created as local variables. Otherwise they are created as global variables in the current data folder.

`V_flag`	0: Output successfully generated. -1: User cancelled either the Open File or Save File dialogs. Other: An error occurred, such as the specified file does not exist.
`V_value`	The number of input lines that matched the regular expression.
`V_startParagraph`	Zero-based line number into the file or Clipboard (or the row number of a source text wave) where the first regular expression was matched. Also see the **/INDX** flag.
`S_fileName`	Full path to the source file, the source text wave, or “Clipboard”. If an error occurred or if the user cancelled, it is an empty string.
`S_path`	Full path to the destination file or destination text wave. "Clipboard": If *destFileOrFolderStr* was the Clipboard. "History": If the output was printed to the history area of the window. `""`: If an error occurred, if the user cancelled, or if /Q was specified.
`S_value`	Contains matching lines as a string list only if /LIST is specified.

Regular Expressions

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the “subject”.

In the case of Grep, the “subject” is each line of the source file or Clipboard, or each row in the source text wave.

The regular expression syntax supported by Grep, **GrepList**, and **GrepString** is based on the “Perl-Compatible Regular Expression” (PCRE) library.

The syntax is similar to regular expressions supported by various UNIX and POSIX `egrep(1)` commands. See **Regular Expressions** on page IV-164 for more details.

As a trivial example, the pattern “Fred” as specified here:

```
Grep/P=myPath/E="Fred" "afile.txt" as "FredFile.txt"
```

matches lines that contain the string “Fred” anywhere on the line.

Character matching is *case-sensitive* by default, similar to strsearch. Prepend the Perl 5 modifier `(?i)` to match upper and lower-case versions of “Fred”:

```
// Copy lines that contain "Fred", "fred", "FRED", "fREd", etc
Grep/P=myPath/E="(?i)fred" "afile.txt" as "AnyFredFile.txt"
```

To copy lines that do *not* match the regular expression, set the /E flag’s reverse parameter:

```
// Copy lines that do NOT contain "Fred", "fred", "fREd", etc.
Grep/P=myPath/E={"(?i)fred",1} "afile.txt" as "NotFredFile.txt"
```

Note: Igor doesn’t use the opening and closing regular expression delimiters that UNIX grep or Perl use: they would have used `"/Fred/"` and `"/(?i)fred/"`.

Regular expressions in Igor support the expected metacharacters and character classes that make the whole grep paradigm so useful. For example:

```
// Copy lines that START with a space or tab character
Grep/P=myPath/E="^[ \\t]" "afile.txt" as "LeadingTabsFile.txt"
```

For a complete description of regular expressions, see **Regular Expressions** on page IV-164, especially for a description of the many uses of the regular expression backslash character (see **Backslash in Regular Expressions** on page IV-167).

Note: Because Igor Pro also has special uses for backslash (see **Escape Sequences in Strings** on page IV-13), you must double the number of backslashes you would normally use for a Perl or grep pattern. Each pair of backslashes identifies a single backslash for the Grep command.

For example, to copy lines that contain "`\z`", the Perl pattern would be `\\z`, but the equivalent Grep expression would be `/E="\\\\z"`.

See **Backslash in Regular Expressions** on page IV-167 for a more complete description of backslash behavior in Igor Pro.

Examples

```
// Copy lines in afile.txt containing "Fred" (case sensitive)
// to an output file named "AnyFredFile.txt" in the same directory.
Grep/P=myPath/E="Fred" "afile.txt" as "AnyFredFile.txt"

// Copy lines in afile.txt containing "Fred" and "Wilma" (case-insensitive)
// to a text wave (which must exist andis overwritten):
Make/O/N=0/T outputTextWave
Grep/P=myPath/E="(?i)fred"/E="(?i)wilma" "afile.txt" as outputTextWave

// Print lines in afile.txt containing "Fred" and "Wilma" (case-insensitive)
// to the history area
Make/O/N=0/T outputTextWave
Grep/P=myPath/E="(?i)fred"/E="(?i)wilma" "afile.txt"

// Test whether afile.txt contains the word "boondoggle", and if so,
// on which line the first occurence was found, WITHOUT creating any output.
//
// Note: the \\b sequences limit matches to a word boundary before and after
// "boondoggle", so "boondoggles" and "aboondoggle" won't match.
//
Grep/P=myPath/Q/E="(?i)\\bBoondoggle\\b" "afile.txt"
if( V_value )                     // at least one instance was found
   Print "First instance of \"boondoggle\" was found on line", V_startParagraph
endif

// Create in S_value a string list of the lines as \r - separated list items:
Grep/P=myPath/LIST="\r"/Q/E="(?i)\\bBoondoggle\\b" "afile.txt"
if( V_Value )                     // some were found
   Print S_value
endif

// Create in W_index a list of the 0-based line numbers where "boondoggle"
// or "boondoggles", etc was found in afile.txt.
Grep/P=myPath/INDX/Q/E="(?i)boondoggle" "afile.txt"
if( V_flag == 0 )   // grep succeeded, perhaps none were found; let's see where
   WAVE W_Index     // needed if in a function
   Edit W_Index     // show line numbers in a table.
endif

// (Create a string list and text wave for the following examples.)
String list= CTabList()           // "Grays;Rainbow;YellowHot;..."
Variable items= ItemsInList(list)
Make/O/T/N=(items) textWave= StringFromList(p,list)

// Copy rows of textWave that contain "Red" (case sensitive)
// to the Clipboard as carriage-return separated lines.
Grep/E="Red" textWave as "Clipboard"
```

```
// Copy lines of the Clipboard that do NOT contain "Blue"
// (case in-sensitve) back to the Clipboard, overwriting what was there:
Grep/E={"(?i)blue",1} "Clipboard" as "Clipboard"

// Format matching text wave row to the history area
Grep/E=("Red")/DCOL={"prefix text --- ", 0, " --- suffix text"} textWave
// Printed output:
   prefix text --- BlueRedGreen --- suffix text
   prefix text --- RedWhiteBlue --- suffix text
   prefix text --- BlueRedGreen256 --- suffix text
   prefix text --- RedWhiteBlue256 --- suffix text
   prefix text --- Red --- suffix text
   prefix text --- RedWhiteGreen --- suffix text
   prefix text --- BlueBlackRed --- suffix text


// Re-copy rows of textWave that contain "Red" (case sensitive)
// to the Clipboard as carriage-return separated lines.
Grep/E="Red" textWave as "Clipboard"
// Create a 2-column text wave whose column 1 (the second column)
// contains the matching text from the Clipboard
Make/O/N=(0,2)/T outputTextWave
// Grep with /A to preserve 2 columns of outputTextWave
Grep/A/E="Red"/GCOL=1/DCOL={1} "Clipboard" as outputTextWave
Edit outputTextWave

// Examples with two-dimensional source text waves
Make/O/T/N=(10, 3) sourceTW= StringFromList(p+10*q,list)
Edit sourceTW

// Copy rows of textWave that contain "Red" in column 2 to outputTextWave.
Make/O/N=0/T outputTextWave
Grep/E="Red"/GCOL=2 sourceTW as outputTextWave
Edit outputTextWave

// Format matching text wave columns to the history area.
// Match lines that contain "Red" in any column of sourceTW:
Grep/E=("Red")/GCOL=-1/DCOL={0,", ",1,", ",2} sourceTW
// Printed output:
   YellowHot, BlueRedGreen256, Magenta
   BlueHot, RedWhiteBlue256, Yellow
   BlueRedGreen, PlanetEarth256, Copper
   RedWhiteBlue, Terrain256, Gold
   Terrain, Rainbow16, RedWhiteGreen
   Grays256, Red, BlueBlackRed
```

References

The regular expression syntax supported by Grep, **GrepString**, and **GrepList** is based on the *PCRE — Perl-Compatible Regular Expression Library* by Philip Hazel, University of Cambridge, Cambridge, England. The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5.

Visit <`http://pcre.org/`> for more information about the PCRE library, and <`http://www.perldoc.com/`> for more about Perl regular expressions. The description of regular expressions above is taken from the PCRE documentation.

A good book on regular expressions is: Friedl, Jeffrey E. F., *Mastering Regular Expressions*, 2nd ed., 492 pp., O'Reilly Media, 2002.

A helpful web site is: http://www.regular-expressions.info

See Also

Regular Expressions on page IV-164 and **Symbolic Paths** on page II-21.

Demo, **CopyFile**, **PutScrapText**, **LoadWave** operations. The **GrepString**, **GrepList**, **StringMatch**, and **cmpstr** functions.

GrepList

`GrepList(`*`listStr`*`, `*`regExprStr`*` [,`*`reverse`*` [, `*`listSepStr`*`]])`

The GrepList function returns each list item in *listStr* that matches the regular expression *regExprStr*.

ListStr should contain items separated by the *listSepStr* character, such as in “abc;def;”.

regExprStr is a regular expression such as is used by the UNIX `grep(1)` command. It is much more powerful than the wildcard syntax used for **ListMatch**. See **Regular Expressions** on page IV-164 for *regExprStr* details.

reverse is optional. If missing, it is taken to be 0. If *reverse* is nonzero then the sense of the match is reversed. For example, if *regExprStr* is `"^abc"` and *reverse* is `1`, then all list items that do not start with “abc” are returned.

listSepStr is optional; the default is `";"`. In order to specify *listSepStr*, you must precede it with reverse.

Examples

To list ColorTables containing “Red”, “red”, or “RED” (etc.):

```
Print GrepList(CTabList(),"(?i)red")        // case-insensitive matching
```

To list window recreation commands starting with “\tCursor”:

```
Print GrepList(WinRecreation("Graph0", 0), "^\tCursor", 0 , "\r")
```

See Also

Regular Expressions on page IV-164.

ListMatch, **StringFromList**, and **WhichListItem** functions and the **Grep** operation.

GrepString

`GrepString(`*`string`*`, `*`regExprStr`*`)`

The GrepString function tests *string* for a match to the regular expression *regExprStr*. Returns 1 to indicate a match, or 0 for no match.

Details

regExprStr is a regular expression such as is used by the UNIX grep(1) command. It is much more powerful than the wildcard syntax used for **StringMatch**. See **Regular Expressions** on page IV-164 for *regExprStr* details.

Character matching is case-sensitive by default, similar to **strsearch**. Prepend the Perl 5 modifier "(?i)" to match upper and lower-case text

Examples

Test for truth that the string contains at least one digit:

```
if( GrepString(str,"[0-9]+") )
```

Test for truth that the string contains at least one “abc”, “Abc”, “ABC”, etc.:

```
if( GrepString(str,"(?i)abc") )             // case-insensitive test
```

See Also

Regular Expressions on page IV-164.

The **StringMatch**, **cmpstr**, **strsearch**, **ListMatch**, and **ReplaceString** functions and the **Demo** and **sscanf** operations.

GridStyle

`GridStyle`

GridStyle is a procedure subtype keyword that puts the name of the procedure in the Grid->Style Function submenu of the mover pop-up menu in the drawing tool palette. You can have Igor automatically create a grid style function for you by choosing Save Style Function from that submenu.

GroupBox

`GroupBox [/Z] `*`ctrlName`*` [`*`keyword`*` = `*`value`*` [, `*`keyword`*` = `*`value`*` …]]`

The GroupBox operation creates a box to surround and group related controls.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the GroupBox control to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **DefaultGUIControls Default Fonts and Sizes** for how enclosed controls are affected by native groupbox appearance.

See **Button** for more appearance details.

disable=*d*	Sets user editability of the control.
	d=0: Normal.
	d=1: Hide.
	d=2: Draw in gray state.
fColor=(*r*,*g*,*b*)	Sets color of the title text. *r*, *g*, and *b* are integers from 0 to 65535.
font="*fontName*"	Sets font used for the box title, e.g., `font="Helvetica"`.
frame=*f*	Sets frame mode. If 1 (default), the frame has a 3D look. If 0, then a simple gray line is used. Generally, you should not use frame=0 with a title if you want to be in accordance with human interface guidelines.
fsize=*s*	Sets font size for box title.
fstyle=fs	Sets the font style of the title text. *fs* is a bitwise parameter with each bit controlling one aspect of the font style for the tick mark labels as follows:
	Bit 0: Bold
	Bit 1: Italic
	Bit 2: Underline
	Bit 4: Strikethrough
	See **Setting Bit Parameters** on page IV-12 for details about bit settings.
labelBack=(*r*,*g*,*b*) or 0	Sets fill color for the interior. *r*, *g*, and *b* are integers from 0 to 65535. If not set, then interior is transparent. Note that if a fill color is used, draw objects can not be used because they will be covered up. Also, you will have to make sure the GroupBox is drawn before any interior controls.
pos={*left*,*top*}	Sets the postion of the box in pixels.
pos+={*dx*,*dy*}	Offsets the position of the box in pixels.
size={*width*,*height*}	Sets box size in pixels.

userdata(*UDName*)=*UDStr*

Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

title=*titleStr* Sets title to *titleStr*. Use `""` for no title.

win=*winName*	Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Flags

/Z	No error reporting.

Details

If no title is given and the width is less than 11 or height is specified as less than 6, then a vertical or horizontal separator line will be drawn rather than a box.

Note: Like TabControls, you need to click near the top of a GroupBox to select it.

See Also

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

GuideInfo

`GuideInfo(`*`winNameStr, guideNameStr`*`)`

The GuideInfo function returns a string containing a semicolon-separated list of information about the named guide line in the named host window or subwindow.

Parameters

winNameStr can be `""` to refer to the top host window.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

guideNameStr is the name of the guide line for which you want information.

Details

The returned string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

Keyword	**Information Following Keyword**
NAME	Name of the guide.
WIN	Name of the window or subwindow containing the guide.
TYPE	The value associated with this keyword is either *User* or *Builtin*. A *User* type denotes a guide created by the DefineGuide operation, equivalent to dragging a new guide from an existing one.
HORIZONTAL	Either 0 for a vertical guide, or 1 for a horizontal guide.
POSITION	The position of the guide in points. This is the actual position relative to the left or bottom edge of the window, not the relative position specified to DefineGuide.

The following keywords will be present only for user-defined guides:

Keyword	**Information Following Keyword**
GUIDE1	The guide is positioned relative to GUIDE1.
GUIDE2	In some cases, the guide is positioned at a fractional position between GUIDE1 and GUIDE2. If the guide does not use GUIDE2, the value will be `""`.
RELPOSITION	The position relative to GUIDE1 (and GUIDE2 if applicable). This is the same as the *val* parameter in DefineGuide. The returned value is in units of points if only GUIDE1 is used, or a fractional value if both GUIDE1 and GUIDE2 are used.

See Also

The **GuideNameList**,s **StringByKey** and **NumberByKey** functions; the **DefineGuide** operation.

GuideNameList

`GuideNameList(`*`winNameStr, optionsStr`*`)`

The GuideNameList function returns a string containing a semicolon-separated list of guide names from the named host window or subwindow.

Parameters

winNameStr can be "" to refer to the top host window.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

optionsStr is used to further qualify the list of guides. It is a string containing keyword-value pairs separated by commas. Use "" to list all guides. Available options are:

TYPE:type	type = BuiltIn: List only built-in guides.
	type = User: List only user-defined guides, those created by the DefineGuide operation or by manually dragging a new guide from an existing one.
HORIZONTAL:h	h = 0: List only non-horizontal (that is, vertical) guides.
	h = 1: List only horizontal guides.

Example

```
String list = GuideNameList("Graph0", "TYPE:Builtin,HORIZONTAL:1")
```

See Also

The **DefineGuide** operation and the **GuideInfo** function.

Hanning

`Hanning` ***`waveName`*** **`[,`** ***`waveName`*** **`]…`**

Note: The **WindowFunction** operation has replaced the Hanning operation.

The Hanning operation multiplies the named waves by a Hanning window (which is a raised cosine function).

You can use Hanning in preparation for performing an FFT on a wave if the wave is not an integral number of cycles long.

The Hanning operation is not multidimensional aware. See Chapter II-6, **Multidimensional Waves**, particularly **Analysis on Multidimensional Waves** on page II-86 for details.

See Also

The **WindowFunction** operation implements the Hanning window as well as other forms such as Hamming, Parzen, and Bartlet (triangle).

ImageWindow, **DPSS**

Hash

`Hash(`*`inputStr, method`*`)`

The Hash function returns a cryptographic hash of the data in *inputStr*.

Parameters

inputStr is string of length up to 2^31 bytes. *inputStr* can contain binary or text data.

method is a number indicating the hash algorithm to use:

1	SHA-256 (SHA-2)
2	MD4
3	MD5

4	SHA-1
5	SHA-224 (SHA-2)
6	SHA-384 (SHA-2)
7	SHA-512 (SHA-2)

Prior to Igor Pro 7.00, only method 1 was supported.

hcsr

`hcsr(`*`cursorName`*` [, `*`graphNameStr`*`])`

The hcsr function returns the horizontal coordinate of the named cursor (A through J) in the coordinate system of the top (or named) graph's X axis.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The X axis used is the one that controls the trace on which the cursor is placed.

Examples

```
Variable xAxisValueAtCursorA = hcsr(A)      // not hcsr("A")
String str="A"
Variable xA= hcsr($str,"Graph0")            // $str is a name, too
```

See Also

The **pcsr**, **qcsr**, **vcsr**, **xcsr**, and **zcsr** functions.

Programming With Cursors on page II-249.

hermite

`hermite(`*`n, x`*`)`

The hermite function returns the Hermite polynomial of order *n*:

$$H_n(x) = (-1)^n \exp\left(x^2\right)\frac{d^n}{dx^n}\exp\left(-x^2\right).$$

The first few polynomials are:

$$1$$

$$2x$$

$$4x^2 - 2$$

$$8x^3 - 12x$$

See Also

The **hermiteGauss** function.

hermiteGauss

hermiteGauss(*n, x*)

The hermiteGauss function returns the normalized Hermite polynomial of order *n*:

$$H_n(x) = \frac{1}{\sqrt{\sqrt{\pi}\,2^n n!}}(-1)^n \exp\left(x^2\right)\frac{d^n}{dx^n}\exp\left(-x^2\right).$$

Here the normalization was chosen such that

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x) H_m(x)\,dx = \delta_{mn},$$

where δ_{nm} is the Kronecker symbol.

You can verify the Hermite-Gauss normalization using the following functions:

```
Function TestNormalization(order)
   Variable order

   Variable/G theOrder = order
   // The integrand vanishes in double-precision outside [-30,30]
   Print/D Integrate1D(hermiteIntegrand,-30,30,2)
End

Function HermiteIntegrand(inX)
   Variable inX

   NVAR n = root:theOrder
   return HermiteGauss(n,inx)^2*exp(-inx*inx)
End
```

See Also

The **hermite** function.

hide

#pragma hide = *value*

The hide pragma allows you to make a procedure file invisible.

See Also

The **The hide Pragma** on page IV-50 and **#pragma**.

HideIgorMenus

HideIgorMenus [*MenuNameStr* [, *MenuNameStr*]…

The HideIgorMenus operation hides the named built-in menus or, if none are explicitly named, hides all built-in menus in the menu bar.

The effect of HideIgorMenus is lost when a new experiment is opened. The state of HideIgorMenus is saved with the experiment.

User-defined menus are not hidden by HideIgorMenus unless attached to built-in menus and the menu definition uses the hideable keyword.

Parameters

MenuNameStr	The name of an Igor menu, like “File”, “Data”, or “Graph”.

Details

The optional menu names are in English and not abbreviated. This ensures that code developed for a localized version of Igor will run on all versions.

The built-in menus that can be shown or hidden (the Help menu can be hidden only on Windows) are those that appear in the menu bar:

File	Edit	Data	Analysis	Macros	Windows	Graph
Layout	Notebook	Panel	Procedure	Table	Misc	Help

Hiding a built-in menu to which a user-defined menu is attached results in a built-in menu with only the user-defined items. For example, if this menu definition attaches items to the built-in Graph menu:

```
Menu "Graph"
   "Do My Graph Thing", ThingFunction()
End
```

Calling `HideIgorMenus "Graph"` will still leave a Graph menu showing (when a Graph is the top-most target window) with only the user-defined menu(s) in it: in this example the one “Do My Graph Thing” item.

Hiding the Macros menu hides menus created from Macro definitions like:

```
Macro MyMacro()
   Print "Hello, world."
End
```

but does not hide normal user-defined “Macros” definitions like:

```
Menu "Macros"
   "Macro 1", MyMacro(1)
End
```

You can set user-defined menus to hide and show along with built-in menus by adding the optional hideable keyword to the menu definition:

```
Menu "Graph", hideable
   "Do My Graph Thing", ThingFunction()
End
```

Then `HideIgorMenus "Graph"` will hide those items, too. If all user-defined Graph menu definitions use the hideable keyword, then no Graph menu will appear in the menu bar.

Some WaveMetrics procedures use the `hideable` keyword so that only customer-defined menus remain when HideIgorMenus is executed.

See Also

Chapter IV-5, **User-Defined Menus**.

The **ShowIgorMenus**, **DoIgorMenu**, and **SetIgorMenuMode** operations.

HideInfo

`HideInfo [/W=`*`winName`*`]`

The HideInfo operation removes the info panel from a graph if it was previously shown by the **ShowInfo** operation.

Flags

/W=*winName*	Hides the info panel in the named window.

See Also

The **ShowInfo** operation.

Programming With Cursors on page II-249.

HideProcedures

`HideProcedures`

The HideProcedures operation hides all procedure windows without closing or killing them.

See Also

The **DisplayProcedure** and **DoWindow** operations.

HideTools

`HideTools [/A/W=`*`winName`*`]`

The HideTools operation hides the tool palette in the top graph or control panel if it was previously shown by the **ShowTools** operation.

Flags

/A	Sizes the window automatically to make extra room for the tool palette. This preserves the proportion and size of the actual graph area.
/W=*winName*	Hides the tool palette in the named window. This must be the first flag specified when used in a Proc or Macro or on the command line.

See Also

The **ShowTools** operation.

HilbertTransform

`HilbertTransform [/Z][/O][/DEST=`*`destWave`*`]` ***`srcWave`***

The HilbertTransform operation computes the Hilbert transformation of *srcWave*, which is a real or complex (single or double precision) wave of 1-3 dimensions. The result of the HilbertTransform is stored in *destWave*, or in the wave W_Hilbert (1D) or M_Hilbert in the current data folder.

Flags

/DEST=*destWave*	Creates a real wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-66 for details.
/O	Overwrites *srcWave* with the transform.
/PAD={*dim1* [, *dim2*, *dim3*, *dim4*]}	Converts *srcWave* into a padded wave of dimensions *dim1*, *dim2*…. The padded wave contains the original data at the start of the dimension and adds zero entries to each dimension up to the specified dimension size. The *dim1*… values must be greater than or equal to the corresponding dimension size of *srcWave*. If you need to pad just the lowest dimension(s) you can omit the remaining dimensions; for example, /Pad=*dim1* will set *dim2* and above to match the dimensions in *srcWave*. This flag was added in Igor Pro 7.00.
/Z	No error reporting.

Details

The Hilbert transform of a function *f*(*x*) is defined by:

$$F(t)=\frac{1}{\pi t}\int_{-\infty}^{\infty}\frac{f(x)dx}{x-t}.$$

Theoretically, the integral is evaluated as a Cauchy principal value. Computationally one can write the Hilbert transform as the convolution:

$$F(t)=\frac{-1}{\pi t}*f(t),$$

which by the convolution theorem of Fourier transforms, may be evaluated as the product of the transform of *f*(*x*) with -*i**sgn(*x*) where:

$$\operatorname{sgn}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}.$$

Note that the Hilbert transform of a constant is zero. If you compute the Hilbert transform in more than one dimension and one of the dimensions does not vary (is a constant), the transform will be zero (or at least numerically close to zero).

There are various definitions for the extension of the Hilbert transform to more than one dimension. In two dimensions this operation computes the transform by multiplying the 2D Fourier transform of the input by the factor (-i)sgn(x)(-i)sgn(y) and then computing the inverse Fourier Transform. A similar procedure is used when the input is 3D.

Examples

Extract the instantaneous amplitude and frequency of a narrow-band signal.

```
Make/O/N=1000 w0,amp,phase
SetScale/I x 0,50,"", w0,amp,phase
w0 = exp(-x/10)*cos(2*pi*x)
HilbertTransform /DEST=w0h w0 // w0+i*w0h is the "analytic signal", i=cmplx(0,1)
amp = sqrt(w0^2 + w0h^2)   // extract the envelope
phase = atan2(-w0h,w0)     // extract the phase [SIGN CONVENTION?]
Unwrap 2*pi, phase         // eliminate the 2*pi phase jumps
Differentiate phase /D=freq   // would have less noise if fit to a line
                              // over interior points
freq /= 2*pi               // phase = 2*pi*freq*time
Display w0,amp   // original waveform and its envelope; note boundary effects
Display freq     // instantaneous frequency estimate, with boundary effects
```

See Also

The **FFT** operation.

References

Bracewell, R., *The Fourier Transform and Its Applications*, McGraw-Hill, 1965.

Histogram

Histogram [*flags*] *srcWaveName, destWaveName*

The Histogram operation generates a histogram of the data in *srcWaveName* and puts the result in *destWaveName* or in W_Histogram or in the wave specified by /DEST.

Parameters

srcWaveName specifies the wave containing the data to be histogrammed.

For historical reasons the meaning and use of *destWaveName* depend on the binning mode as specified by /B. See **Histogram Destination Wave** on page V-299 below for details.

Flags

/A	Accumulates the histogram result with the existing values in the destination wave instead of replacing the existing values with the result. Assumes /B=2 unless the /B flag is present. **Note**: The result will be incorrect if you also use /P.

/B=mode Controls binning:

mode=1: Semi-automatic mode that sets the bin range based on the range of the Y values in *srcWaveName*. The number of bins is determined by the number of points in the destination wave.

mode=2: Uses the bin range and number of bins determined by the X scaling and number of points in the destination wave.

mode=3: Uses Sturges' method to determine optimal number of bins and redimensions the destination wave as necessary. By this method

```
numBins=1+log2(N)
```

where N is the number of data points in *srcWaveName*. The bins will be distributed so that they include the minimum and maximum values.

mode=4: Uses a method due to Scott, which determines the optimal bin width as

$\texttt{binWidth=3.49*}\sigma\texttt{*N}^{-1/3}$

where N is the number of data points in *srcWaveName* and σ is the standard deviation of the distribution. The bins will be distributed so that they include the minimum and maximum values.

method=5: Uses the Freedman-Daiconis method where

$\texttt{binWidth=2*IQR*N}^{-1/3}$

where IQR is the interquartile distance (see **StatsQuantiles**) and the bins are evenly distributed between the minimum and maximum values.

/B={*binStart*,*binWidth*,*numBins*}

Sets the histogram bins from these parameters rather than from *destWaveName*. Changes the X scaling and length of the destination wave.

/C Sets the X scaling so that X values are in the centers of the bins, which is required when you do a curve fit to the histogram output. Ordinarily, wave scaling of the output wave is set with X values at the left bin edges.

/CUM Requests a cumulative histogram in which each bin is the sum of bins to the left. The last bin will contain the total number of input data points, or, with /P, 1.0.

/CUM cannot be used with a weighted histogram (/W flag).

When used with /A, the destination wave must be the result of a histogram created with /CUM.

Note that if you use a binning mode (/B flag) that sets a bin range that does not include the entire range of the input data, then the output will not count all of input points and the last bin will not contain the total number of input points. Input points whose values fall below the left edge of the first bin or above the right edge of the last bin will not be counted.

/DEST=*destWave* Saves the histogram output in a wave specified by *destWave*. The destination wave is created or overwritten if it already exists.

Creates a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-66 for details.

See **Histogram Destination Wave** on page V-299 below for further discussion.

The /DEST flag was added in Igor Pro 7.00.

/N Creates a wave (W_SqrtN) containing the square root of the number of counts in each bin. This is an appropriate wave to use as a weighting wave when doing a curve fit to the histogram results.

/NLIN=*binsWave* Computes a non-linear histogram using the bins specified in the wave *binsWave*. This option is not compatible with the flags /A, /B, /C, /CUM, /N, /P, /W.

The bins must be contiguous and non-overlapping so that *binsWave* contains monotonically increasing values with no NaNs and INFs. For example, if you want the 3 bins [1,10),[10,100),[100,1000), execute:

Make/O/N=4 *bins*={1,10,100,1000}

The upper end of each bin is open.

The /NLIN flag was added in Igor Pro 7.00.

/P Normalizes the histogram as a probability distribution function, and shifts wave scaling so that data correspond to the bin centers.

When using the results with **Integrate**, you must use /METH=0 or /METH=2 to select rectangular integration methods.

/R=(*startX*,*endX*) Specifies the range of X values of *srcWaveName* over which the histogram is to be computed.

/R=[*startP*,*endP*] Specifies the range of points of *srcWaveName* over which the histogram is to be computed.

/RMD=[*firstRow*,*lastRow*][*firstColumn*,*lastColumn*][*firstLayer*,*lastlayer*][*firstChunk*,*lastChunk*]

Designates a contiguous range of data in the source wave to which the operation is to be applied. This flag was added in Igor Pro 7.00.

You can include all higher dimensions by leaving off the corresponding brackets. For example:

```
/RMD=[firstRow,lastRow]
```

includes all available columns, layers and chunks.

You can use empty brackets to include all of a given dimension. For example:

```
/RMD=[][firstColumn,lastColumn]
```

means "all rows from column A to column B".

You can use a * to specify the end of any dimension. For example:

```
/RMD=[firstRow,*]
```

means "from firstRow through the last row".

/W=*weightWave* Creates a “weighted” histogram. In this case, instead of adding a single count to the appropriate bin, the corresponding value from *weightWave* is added to the bin. *weightWave* may be any number type, and it may be complex. If it is complex, then the destination wave will be complex.

/W cannot be used with a cumulative histogram (/CUM flag).

Histogram Destination Wave

For historical reasons there are multiple ways to specify the destination wave and the meaning and use of *destWaveName* depend on the binning mode as specified by /B. This section explains the details and then provides guidance and when to use which mode.

In binning modes 1 and 2 (/B=1 and /B=2, described above), the destination wave plays a role in determining the binning and *destWaveName* must be the name of an existing wave. If you omit /DEST then the output is written to *destWaveName*. If you provide /DEST then the output is written to the wave specified by /DEST.

In binning modes 3, 4 and 5 (/B=3, /B=4 and /B=5, described above), the destination wave plays no role in determining the binning. If you omit *destWaveName* and /DEST, Histogram stores its output in a wave named W_Histogram in the current data folder. If you omit /DEST and provide *destWaveName*, then *destWaveName* must name an existing wave to which the output is written. If you provide /DEST, you can omit *destWaveName*. If you provide both /DEST and *destWaveName* then *destWaveName* must name an existing wave but the operation ignores it.

Here is the recommended usage:

If you want to use specific binning that you have determined, use /B={*binStart*,*binWidth*,*numBins*}, use /DEST to specify the destination wave, and omit *destWaveName*.

If you want Igor to determine the binning, use /B=3, /B=4 or /B=5, use /DEST to specify the destination wave, and omit *destWaveName*.

For backward compatibility with Igor Pro 6, use /B=1, /B=2, /B=3, /B=4 or /B={*binStart*,*binWidth*,*numBins*}, create a destination wave, use it as *destWaveName* and omit /DEST.

Details

If you use /B={*binStart*, *binWidth*, *numBins*}, then the initial number of data points in the wave is immaterial since the Histogram operation changes the number of points.

Only one /B and only one /R flag is allowed.

If both /A and /B flags are missing, the bin range and number of bins is calculated as if /B=1 had been specified.

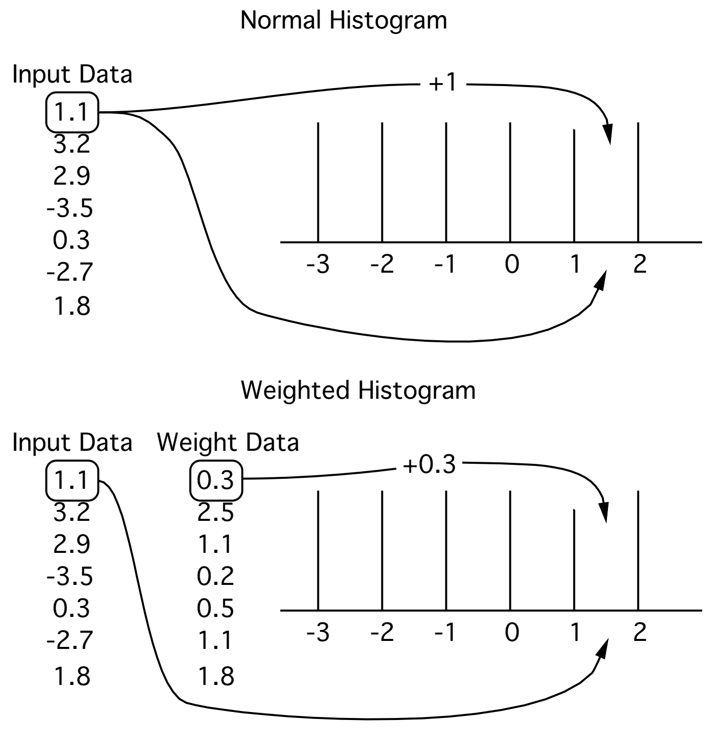
When accumulating multiple histograms in one output wave, typically you will want to use /B={*binStart*,*binWidth*,*numBins*} for the first histogram, and /A for successive histograms.

The Histogram operation works on single precision floating point destination waves. If necessary, Histogram redimensions the destination wave to be single precision floating point. However, Histogram/A requires that the destination wave already be single precision floating point.

For a weighted histogram, the destination wave will be double-precision.

If you specify the range as `/R=(start)`, then the end of the range is taken as the end of *srcWaveName*.

In an ordinary histogram, input data is examined one data point at a time. The operation determines which bin a data value falls into and a single count is added to that bin. A weighted histogram works similarly, except that it adds to the bin a value from another wave in which each row corresponds to the same row in the input wave.



The Histogram operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details. In fact, the Histogram operation can be usefully applied to multidimensional waves, such as those that represent images. The /R flag will not work as expected, however.

Examples

```
// Create histogram of two sets of data.
Make/N=1000 data1=gnoise(1), data2=gnoise(1)
Make/N=1 histResult
```

```
// Sets bins, does histogram.
Histogram/B={-5,1,10} data1, histResult
Display histResult; ModifyGraph mode=5

// Accumulates into existing bins.
Histogram/A data2, histResult
```

See Also

Histograms on page III-120, **ImageHistogram**, **JointHistogram**

References

Sturges, H.A., The choice of a class-interval, *J. Amer. Statist. Assoc.*, *21*, 65-66, 1926.

Scott, D., On optimal and data-based histograms, *Biometrika*, *66*, 605-610, 1979.

hyperG0F1

`hyperG0F1(`*`b, z`*`)`

The hyperG0F1 function the confluent hypergeometric function

$$ {}_0F_1(b,z) = \sum_{i=0}^{\infty} \frac{z^i}{\Gamma(b+i)i!}, $$

where $\Gamma(x)$ is the gamma function.

Note: The series evaluation may be computationally intensive. You can abort the computation by pressing the **User Abort Key Combinations**.

See Also

The **hyperG1F1**, **hyperG2F1**, and **hyperGPFQ** functions.

References

The PFQ algorithm was developed by Warren F. Perger, Atul Bhalla, and Mark Nardin.

hyperG1F1

`hyperG1F1(`*`a, b, z`*`)`

The hyperG1F1 function returns the confluent hypergeometric function

$$ {}_1F_1(a,b,z) = \sum_{n=0}^{\infty} \frac{(a)_n z^n}{(b)_n n!}, $$

where $(a)_n$ is the Pochhammer symbol

$$ (a)_n = a(a+1)\ldots(a+n-1). $$

Note: The series evaluation may be computationally intensive. You can abort the computation by pressing the **User Abort Key Combinations**.

See Also

The **hyperG0F1**, **hyperG2F1**, and **hyperGPFQ** functions.

References

The PFQ algorithm was developed by Warren F. Perger, Atul Bhalla, and Mark Nardin.

hyperG2F1

`hyperG2F1(`*`a, b, c, z`*`)`

The hyperG2F1 function returns the confluent hypergeometric function

$$ {}_2F_1(a,b,c,z) = \sum_{n=0}^{\infty} \frac{(a)_n (b)_n z^n}{(c)_n n!} $$

$$ {}_2F_1(a,b,c;z) = \sum_{n=0}^{\infty} \frac{(a)_n (b)_n z^n}{(c)_n n!}, $$

where $(a)_n$ is the Pochhammer symbol

$$ (a)_n = a(a+1)\ldots(a+n-1). $$

Note: The series evaluation may be computationally intensive. You can abort the computation by pressing the **User Abort Key Combinations**.

See Also

The **hyperG0F1**, **hyperG1F1**, and **hyperGPFQ** functions.

References

The PFQ algorithm was developed by Warren F. Perger, Atul Bhalla, and Mark Nardin.

hyperGNoise

`hyperGNoise(`*`m, n, k`*`)`

The hyperGNoise function returns a pseudo-random value from the hypergeometric distribution whose probability distribution function is

$$ f(x;m,n,k) = \frac{\binom{n}{x}\binom{m-n}{k-x}}{\binom{m}{k}} $$

where *m* is the total number of items, *n* is the number of marked items, and *k* is the number of items in a sample.

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable “random” numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

SetRandomSeed, **StatsHyperGCDF**, and **StatsHyperGPDF**.

Chapter III-12, **Statistics** for a function and operation overview.

Noise Functions on page III-344.

hyperGPFQ

`hyperGPFQ(`*`waveA, waveB, z`*`)`

The hyperGPFQ function returns the generalized hypergeometric function

$$ {}_pF_q\left(\{a_1,\ldots a_p\},\{b_1,\ldots b_q\};z\right) = \sum_{n=0}^{\infty} \frac{(a_1)_n (a_2)_n \cdots (a_p)_n z^n}{(b_1)_n (b_2)_n \cdots (b_q)_n n!}, $$

where $(a)_n$ is the Pochhammer symbol

$$(a)_n = a(a+1)\ldots(a+n-1).$$

Note: The series evaluation may be computationally intensive. You can abort the computation by pressing the **User Abort Key Combinations**.

See Also

hyperG0F1, **hyperG1F1**, **hyperG2F1**

CosIntegral, **ExpIntegralE1**, **SinIntegral**

References

The PFQ algorithm was developed by Warren F. Perger, Atul Bhalla, and Mark Nardin.

i

```
i
```

The i function returns the loop index of the inner most iterate loop in a macro. Not to be used in a function. iterate loops are archaic and should not be used.

ICA

`ICA [`*`flags`*`]` ***`srcWave`***

The ICA operation performs independent component analysis using the FastICA algorithm. Input data is in the form of a 2D wave where each column represents the equivalent of a single data acquisition channel. The results of the operation are stored in the waves M_ICAComponents, M_ICAUnMix and M_matrixW in the current data folder.

The ICA operation was added in Igor Pro 7.00.

Flags

/A=*alpha*	*alpha* is a constant used as a factor in the argument of the logCosh function. It is not used with the exp function. *alpha* is in the range [1,2] and its default value is 1. You will rarely need to change this value to affect the rate of convergence or the quality of the results.
/CF=num	Specifies the contrast function, also called the non-quadratic function, used by ICA. *num*=0: logCosh (default) *num*=1: exp
/COLS	Preconditions the input by subtracting the mean and then normalizing the input on a column-by-column basis. The algorithm appears to converge and produce better results when this flag is used.
/DFLT	Use the deflation/vector method where iterations solve for a single vector of the "unmixing" matrix at a time. By default the operation uses the matrix method which solves for the complete unmixing matrix at one time.
/PCA	Save the output of the "PCA" stage which is also the form of the data before the fastICA iterations. The data are saved in the wave M_PCA in the current data folder.
/Q	Quiet mode; do not print anything in the history.
/TOL=*tolerance*	The tolerance value is used to determine when iterations converge. In the deflation/vector method the tolerance measures the difference between the values of vectors in consecutive iterations. In the matrix method the tolerance measures the average deviation of all components. By default tolerance = 1e-5 for both methods.

/WINT=*w*	Provides an initial unmixing matrix W. If you do not provide this matrix the algorithm initializes using enoise. The wave *w* must be 2D having the same number type as *srcWave* and having dimensions nCols x nCols, where nCols is the number of columns of *srcWave*. Providing an initial matrix is useful if you have obtained one from a previous set of iterations which may have converged using inadequate tolerance.
/Z	No error reporting.

Details

srcWave is a 2D wave of nRows by nCols. It must be a single or double precision real-valued wave containing no NaNs or INFs. Each column of *srcWave* corresponds to a single data acquisition channel that is assumed to consist of a linear superposition of independent components. This can be expressed as a matrix product

X=A (S^t)

where **S** is an nRows by nCols matrix of independent components, **^t** denotes the transpose, **A** is an nCols by nCols mixing matrix and **X** is the "mixed" input. The ICA operation attempts to find the independent components of **S** from the transformation

S=W X

so that the mutual information between the resulting columns of **S** is minimized. Since mutual information is not affected by a multiplication of components by scalar constants, the resulting independent components can be specified up to a scalar factor.

The operation uses the FastICA algorithm to compute the independent components.

The algorithm has two available methods for computation. The default is to attempt to evaluate the full **W** matrix at once. The second method (/DFLT flag) also known as "deflation" computes one row of **W** at a time. The deflation method might have advantages in cases where there are fewer independent components than there are columns in the input.

Example

```
// Create the source
Make/O/N=(1000,3) ddd
ddd[][0]=sin(2*pi*x/13)
ddd[][1]=sin(2*pi*x/17)
ddd[][2]=sin(2*pi*x/23)

// Create mixing matrix
Make/O/N=(3,3) AA
AA[0][0]= {0.291,0.6557,-0.5439}
AA[0][1]= {0.5572,0.3,-0.2}
AA[0][2]= {-0.1,-0.7,0.4}

// Do the mixing
MatrixOp/O xx=ddd x AA

// Try the ICA
ICA/DFLT/COLS xx
Wave ICARes
Display M_ICAComponents[][0]
Display M_ICAComponents[][1]
Display M_ICAComponents[][2]
```

References

A. Hyvarinen and E. Oja (2000) Independent Component Analysis: Algorithms and Applications, Neural Networks, (13)411-430.

See Also

PCA

if-elseif-endif

```
if ( <expression1> )
   <TRUE part 1>
elseif ( <expression2> )
   <TRUE part 2>
[…]
[else
   <FALSE part>]
endif
```

In an if-elseif-endif conditional statement, when an expression first evaluates as TRUE (nonzero), then only code corresponding to the TRUE part of that expression is executed, and then the conditional statement is exited. If all expressions evaluate as FALSE (zero) then *FALSE part* is executed when present. After executing code in any TRUE part or the FALSE part, execution will next continue with any code following the if-elseif-endif statement.

See Also

If-Elseif-Endif on page IV-38 for more usage details.

if-endif

```
if ( <expression> )
   <TRUE part>
[else
   <FALSE part>]
endif
```

An if-endif conditional statement evaluates *expression*. If *expression* is TRUE (nonzero) then the code in *TRUE part* is executed, or if FALSE (zero) then the optional *FALSE part* is executed.

See Also

If-Else-Endif on page IV-37 for more usage details.s

IFFT

```
IFFT [flags] srcWave
```

The IFFT operation calculates the Inverse Discrete Fourier Transform of *srcWave* using a multidimensional fast prime factor decomposition algorithm. This operation is the inverse of the **FFT** operation.

Output Wave Name

For compatibility with earlier versions of Igor, if you use IFFT without /ROWS or /COLS, the operation overwrites *srcWave*.

If you use the /ROWS flag, IFFT uses the default output wave name M_RowFFT and if you use the /COLS flag, IFFT uses the default output wave name M_ColFFT.

We recommend that you use the /DEST flag to make the output wave explicit and to prevent overwriting *srcWave*.

Parameters

srcWave is a complex wave. The IFFT of *srcWave* is a either a real or complex wave, according to the length and flags.

Flags

/C	Forces the result of the IFFT to be complex. Normally, the IFFT produces a real result unless certain special conditions are detected as described in **Details**.
/COLS	Computes the 1D IFFT of 2D *srcWave* one column at a time, storing the results in the destination wave. You must specify a destination wave using the /DEST flag (no other flags are allowed). See the /ROWS flag and corresponding flags of the **FFT** operation.

/DEST=*destWave*	Specifies the output wave created by the IFFT operation.
	It is an error to specify the same wave as both *srcWave* and *destWave*.
	In a function, IFFT by default creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-66 for details.
/FREE	Creates *destWave* as a free wave.
	/FREE is allowed only in functions and only if *destWave*, as specified by /DEST, is a simple name or wave reference structure field.
	See **Free Waves** on page IV-84 for more discussion.
	The /FREE flag was added in Igor Pro 7.00.
/R	Forces real output when, due to a power of 2 number of points, IFFT would otherwise automatically produce a complex result.
/ROWS	Calculates the IFFT of only the first dimension of 2D *srcWave*. It computes the 1D FFT one row at a time. You must specify a destination wave using the /DEST flag (no other flags are allowed). See the /COLS flag and corresponding flags of the **FFT** operation.
/Z	Will not rotate *srcWave* when computing the IDFT of a complex wave whose length is an integral power of 2.
	This length indicates that the Inverse DFT result will also be a complex wave. When the result is complex, *and* the x scaling of *srcWave* is such that the first point is *not* x=0, it normally rotates *srcWave* by -N/2 points before performing the IFFT. This inverts the process of performing an FFT on a complex wave. However when /Z is specified, it does not perform this rotation.

Details

The data type of *srcWave* must be complex and must not be an integer type. You should be aware that an IFFT on a number of points that is prime can be slow.

By default, IFFT assumes you are performing an inverse transform on data that was originally real and therefore it produces a real result. However, for historical and compatibility reasons, IFFT detects the special conditions of a one-dimensional wave containing an integral power of 2 data points and automatically creates a complex result.

When the result is complex, the number of points (N) in the resulting wave will be of the same length. Otherwise the resulting wave will be real and of length (N-1)*2.

In either the complex or real case the X units of the output wave are changed to “s”. The X scaling also is changed appropriately, cancelling out the adjustments made by the **FFT** operation. When the data is multidimensional, the same considerations apply to the additional dimensions. The scaling description and IDFT equation below pretend that the IFFT is not performed in-place. After computing the IFFT values, the X scaling of *waveOut* is changed as if Igor had executed these commands:

```
Variable points                    // time-domain points, NtimeDomain
if( waveIn was complex wave )
   points= numpnts(waveIn)
else                               // waveIn was real wave
   points= (numpnts(waveIn) - 1) * 2
endif
Variable deltaT= 1 / (points*deltaX(waveIn))      // 1/(NtimeDomaindx)
SetScale/P waveOut 0,deltaT,"s"
```

The IDFT equation is:

$$waveOut[n] = \frac{1}{N}\sum_{k=0}^{N-1} waveIn[k]\exp\left(\frac{2\pi ikn}{N}\right), \qquad where \quad i=\sqrt{-1}.$$

See Also

The **FFT**, **DSPPeriodogram**, and **MatrixOp** operations.

IgorInfo

`IgorInfo(`*`selector`*`)`

The IgorInfo function returns information about the Igor application and the environment in which it is running.

Details

selector is a number from 0 to 6.

Always pass 0, 1, 2, 3, 4, 5, 6 or 7 as the input parameter. In future versions of Igor Pro, this parameter may request other kinds of information.

If *selector* is 0, IgorInfo returns a collection of assorted information. The result string contains five kinds of information. Each group is prefaced by a keyword and a colon, and terminated with a semicolon.

Keyword	Information Following Keyword For IgorInfo(0)
FREEMEM	The amount of free memory available to Igor.
PHYSMEM	The amount of total physical memory available to Igor. Added in Igor Pro 7.00.
USEDPHYSMEM	The amount of used physical memory available to Igor. Added in Igor Pro 7.00.
IGORKIND	The type of Igor application: "pro": Igor Pro 32-bit "pro demo": Igor Pro 32-bit in demo mode "pro64": Igor Pro 64-bit "pro64 demo": Igor Pro 64-bit in demo mode "pro64" and "pro64 demo" are returned by the 64-bit of Igor Pro 7.00 or later. The presence of "demo" indicates that Igor is running in demo mode, either because the user's fully-functional demo period has expired or because the user chose to run in demo mode using the License dialog.
IGORVERS	The version number of the Igor application. Also see IGORFILEVERSION returned by IgorInfo(3).
NSCREENS	Number of screens currently attached to the computer and used for the desktop.
SCREEN1	A description of the characteristics of screen 1. The format of the SCREEN1 description is: SCREEN1:DEPTH=*bitsPerPixel*,RECT=*left*,*top*,*right*,*bottom*; *left*, *top*, *right*, and *bottom* are all in pixels. If there are multiple screens, there will be additional SCREEN keywords, such as SCREEN2 and SCREEN3.

If *selector* is 1, IgorInfo returns the name of the current Igor experiment.

If *selector* is 2, IgorInfo returns the name of the current platform: "Macintosh" or "Windows".

If *selector* is 3, IgorInfo returns a collection of more detailed information about the operating system, localization information, and the actual file version of the Igor executable. The keywords are OS, OSVERSION, LOCALE, and IGORFILEVERSION.

Keyword	Information Following Keyword For IgorInfo(3)
IGORFILEVERSION	The actual version number of the Igor application file. On Macintosh, the version number is a floating point number with a possible suffix. Igor Pro 7.00, for example, returns "7.00". Igor Pro 7.02A returns "7.02A". On Windows, the version format is a period-separated list of four numbers. Igor Pro 7.02 returns "7.0.2.0". A revision to Igor Pro 7.02 would be indicated in the last digit, such as "7.0.2.12".

Keyword	Information Following Keyword For IgorInfo(3)
LOCALE	Country for which this version of Igor Pro is localized. “US” for most versions, “Japan” for the Japanese versions.
OS	On Mac OS X, this will be “Macintosh OS X”. On Windows, this might be “Windows XP (Build 1234)”. The actual build number and format of the text will vary with the operating system.
OSVERSION	Operating system number. On Macintosh, this is something like “10.9.5”. On Windows, this is something like “6.3.10586”.

If *selector* is 4, IgorInfo returns the name of the current processor architecture. Currently this is always “Intel”.

If *selector* is 5, IgorInfo returns, as a string, the serial number of the program if it is registered or "_none_" if it isn't registered. Use **str2num** to store the result in a numeric variable. str2num returns NaN if the program isn't registered.

If *selector* is 6, IgorInfo returns, as a string, the version of the Qt library under which Igor is running, for example “5.6.1”. This selector value was added in Igor Pro 7.00.

If *selector* is 7, IgorInfo returns, as a string, the name of the current user. This selector value was added in Igor Pro 7.00.

Examples

```
Print NumberByKey("NSCREENS", IgorInfo(0))     // Number of active displays
```

```
Function RunningWindows()                      // Returns 0 if Macintosh, 1 if Windows
   String platform = UpperStr(IgorInfo(2))
   Variable pos = strsearch(platform,"WINDOWS",0)
   return pos >= 0
End
```

IgorVersion

`#pragma IgorVersion =` ***`versNum`***

When a procedure file contains the directive, `#pragma IgorVersion=`*versNum*, an error will be generated if *versNum* is greater than the current Igor Pro version number. It prevents procedures that use new features added in later versions from running under older versions of Igor in which these features are missing. However, this version check is limited because it does not work with versions of Igor older than 4.0.

See Also

The **The IgorVersion Pragma** on page IV-50 and **#pragma**.

IgorVersion

The IgorVersion function returns version number of the Igor application as a floating point number. Igor Pro 7.00 returns 7.00, as does Igor Pro 7.00A.

Details

Because floating point numbers are not precise, exact comparisons to floating point values often behave in unexpected ways. For example:

```
Variable result = 6 + 0.1
if (result == 6.1)
   Print "result == 6.1"                // this is not printed!
else
   Print "difference = ", result - 6.1  // prints "difference =   -8.88178e-16"
endif
```

However, IgorVersion compensates for this so that the following will work as expected:

```
if (IgorVersion() == 6.1)
   Print "result == 6.1"// this is printed to the history area
endif
```

You can use IgorVersion in conditionally compile code expressions, which can be used to omit calls to new Igor features or to provide backwards compatibility code.

```
#if (IgorVersion() >= 7.00)
   [Code that compiles only on Igor Pro 7.00 or later]
#else
   [Code that compiles only on earlier versions of Igor]
#endif
```

If at all possible, it is better to require your users to use a later version of Igor rather than writing conditional code. Attempting this kind of backward-compatibility multiplies your testing requirements and the chances for bugs.

See Also

IgorInfo, **Conditional Compilation** on page IV-100, **The IgorVersion Pragma** on page IV-50

ilim

`ilim`

The ilim function returns the ending loop count for the inner most iterate loop Not to be used in a function. iterate loops are archaic and should not be used.

imag

`imag(`*`z`*`)`

The imag function returns the imaginary component of the complex number *z* as a real (not complex) number.

See Also

The **cmplx**, **conj**, **p2rect**, **r2polar**, and **real** functions.

ImageAnalyzeParticles

`ImageAnalyzeParticles` `[`***`flags`***`]` ***`keyword imageMatrix`***

The ImageAnalyzeParticles operation performs one of two particle analysis operations on a 2D or 3D source wave *imageMatrix*. The source image wave must be binary, i.e., an unsigned char format where the particles are designated by 0 and the background by 255 (the operation will produce erroneous results if your data uses the opposite designation). Note that all nonzero values in the source image will be considered part of the background. Grayscale images must be thresholded before invoking this operation (you may need to use the /I flag with the **ImageThreshold** operation).

Note: ImageAnalyzeParticles does not take into account wave scaling. All image metrics are in pixels and all pixels are assumed to be square.

Parameters

keyword is one of the following names:

mark	Creates a masking image for a single particle, which is specified by an internal (seed) pixel using the /L flag. The masking image is stored in the wave M_ParticleMarker, which is an unsigned char wave. All points in M_ParticleMarker are set to 64 (image operations on binary waves use the value 64 to designate the equivalent of NaN) except points in the particle which are set to the 0. This wave is designed to be used as an overlay on the original image (using the explicit=1 mode of ModifyImage). This keyword is superseded by the **ImageSeedFill** operation.
stats	Measures the particles in the image. See **ImageAnalyzeParticles Stats** on page V-312 for details.

Flags

/A=*minArea*	Specifies a minimum area as a threshold that must be exceeded for a particle to be counted (e.g., use *minArea*=0 to find single pixel particles). The minimum area is measured in pixels; its default value is *minArea*=5. When the source wave is 3D, *minArea* specifies the minimum number of voxels that constitute a particle. /A has no effect when used with the *mark* method.
/B	Erases a 1 pixel wide frame inset from the boundary. This insures that no particles will have boundary pixels (see /EBPC below) and all boundary waves will describe close contours.
/CIRC={*minCircularity*,*maxCircularity*}	Use this flag to filter the output so that only particles in the range of the specified circularity are counted.
/D=*dataWave*	Specify a wave from which the minimum, maximum, and total particle intensity are sampled when used with the stats keyword. dataWave must be of the same dimensions as the input binary image imageMatrix. It can be of any real numeric type. Results are returned in the waves W_IntMax, W_IntMin, and W_IntAvg.
/E	Calculates an ellipse that best fits each particle. The equivalent ellipse is calculated by first finding the moments of the particle (i.e., average x-value, average y-value, average x^2, average y^2, and average x*y), and then requiring that the area of the ellipse be equal to that of the particle. The resulting ellipses are saved in the wave M_Moments. When *imageMatrix* is a 2D wave, the results returned in M_Moments are the columns: the X-center of the ellipse, the Y-center of the ellipse, the major axis, the minor axis, and the angle (radians) that the major axis makes with the X-direction. When *imageMatrix* is a 3D wave, the results in M_Moments include the sum of the X, Y, and Z components as well as all second order permutations of their products. They are arranged in the order: sumX, sumY, sumZ, sumXX, sumYY, sumZZ, sumXY, sumXZ, and sumYZ.
/EBPC	Use this flag to exclude from counting any particle that has one or more pixels on any boundary of the image.
/F	Fills 2D particles having internal holes and adjusts their area measure for the removal of holes. Internal boundaries around the holes are also eliminated. When the boundary of the particle consists of thin elements that cannot be traversed as a single closed path which passes each boundary pixel only once, the particle will not be filled. Note that filling particles may increase execution time considerably and on some images it may require large amount of memory. It is likely that a more efficient approach would be to preprocess the binary image and remove holes using morphology operations. This flag is not supported when *imageMatrix* is a 3D wave.
/FILL	Use /FILL to fill holes inside particles. The reported values of area and perimeter are computed as if there are no holes. The filling algorithm could fail if, for example, there is a closed contour of zeros around the particles. If you specify both /F and /FILL the operation used /FILL only. Added in Igor Pro 7.00.
/L= (*row*,*col*)	Specifies a 2D particle location in connection with the mark method. (*row*, *col*) is a seed value corresponding to any pixel inside the particle. If the seed belongs to the particle boundary, the particle will not be filled. This flag is not supported when *imageMatrix* is a 3D wave.

/M=*markerVal*	Use this flag with the stats mode for 2D images. See **stats** keyword for a full description of the following waves: *markerVal*=0: No marker waves. *markerVal*=1: M_ParticlePerimeter. *markerVal*=2: M_ParticleArea. *markerVal*=3: M_Particle. This flag does not apply to 3D waves.
/MAXA=*maxArea*	Specifies an upper limit of the area of an acceptable particle when used with the stats keyword. The area is measured in pixels and the default value of *maxArea* is the number of pixels in the image. In 3D the maximum value applies to the number of voxels.
/NSW	Creates the marker wave (see /M flag) but not the particle statistics waves when used with the stats keyword. This should reduce execution time in images containing many particles.
/P=*plane*	Specifies the plane when operating on a single layer of a 3D wave.
/PADB	Use this flag with the stats keyword to pad the image with a 1 pixel wide background. This has the effect that particles touching the image boundary are now interior particles with closed perimeter (that extend one pixel beyond the original image frame). In addition, entries in the wave W_ObjPerimeter will be longer for all boundary particles which will also affect other derived parameters such as circularity. /PADB is different from /B in that it takes into account all pixels belonging to the particle that lie on the boundary of the image. The two flags are mutually exclusive. /PADB was added in Igor Pro 7.00.
/Q	Quiet flag, does not report the number of particles to the history area.
/R=*roiWave*	Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u) that has the same number of rows and columns as *imageMatrix*. The ROI itself is defined by the entries or pixels in the *roiWave* with value of 0. Pixels outside the ROI may have any nonzero value. The ROI does not have to be contiguous. When *imageMatrix* is a 3D wave, *roiWave* can be either a 2D wave (matching the number of rows and columns in *imageMatrix*) or it can be a 3D wave that must have the same number of rows, columns and layers as *imageMatrix*. When using a 2D *roiWave* with a 3D *imageMatrix* the ROI is understood to be defined by *roiWave* for each layer in the 3D wave. See **ImageGenerateROIMask** for more information on creating 2D ROI waves.
/U	Saves the wave M_ParticleMarker as an 8-bit unsigned instead of the default 16-bit when used with the mark keyword.
/W	Creates boundary waves W_BoundaryX, W_BoundaryY, and W_BoundaryIndex for a 2D *imageMatrix* wave. W_BoundaryX and W_BoundaryY contain the pixels along the particle boundaries. The boundary of each particle ends with a NaN entry in both waves. Each entry in W_BoundaryIndex is the index to the start of a new particle in W_BoundaryX and W_BoundaryY, so that you can quickly locate the boundary of each particle. When there are holes in particles, the entries in W_BoundaryX and W_BoundaryY start with the external boundary followed by all the internal boundaries for that particle. There are no index entries for internal boundaries. This flag is not supported when *imageMatrix* is a 3D wave.

Details

Particle analysis is accomplished by first converting the data from its original format into a binary representation where the particle is designated by zero and the background by any nonzero value. The algorithm searches for the first pixel or voxel that belongs to a particle and then grows the particle from that seed while keeping count of the area, perimeter and count of pixels or voxels in the particle. If you use additional flags, the algorithm must compute additional quantities for each pixel or voxel belonging to the particle.

If your goal is to mask only the particle, a more efficient approach is to use the **ImageSeedFill** operation, which similarly follows the particle but does not spend processing time on computing unrelated particle properties. ImageSeedFill also has the additional advantage of not requiring that the input wave be binary, which will save time on performing the initial threshold and, in fact, may produce much better results with the adaptive/fuzzy features that are not available in ImageAnalyzeParticles.

ImageAnalyzeParticles Stats

The ImageAnalyzeParticles stats keyword measures the particles in the image. Results of the measurements are reported for all particles whose area exceeds the *minArea* specified by the /A flag. The results of the measurements are:

V_NumParticles	Number of particles that exceed the *minArea* limit.
W_ImageObjArea	Area (in pixels) for each particle.
W_ImageObjPerimeter	Perimeter (in pixels) of each particle. The perimeter calculation involves estimates for 45-degree pixel edges resulting in noninteger values.
W_circularity	Ratio of the square of the perimeter to (4*π*objectArea). This value approaches 1 for a perfect circle.
W_rectangularity	Ratio of the area of the particle to the area of the inscribing (nonrotated) rectangle. This ratio is π/4 for a perfectly circular object and unity for a nonrotated rectangle.
W_SpotX and W_SpotY	Contain a single x, y point from each object. There is one entry per particle and the entries follow the same order as all other waves created by this operation. Each (x,y) point from these waves can used to define the position of a tag or annotation for a particle. Points can also be used as seed pixels for the associated *mark* method or for the ImageSeedFill operation.
W_xmin, W_xmax, W_ymin, W_ymax	Contain a single point for each particle defining an inscribing rectangular box with axes along the X and Y directions.

One of the following waves can be created depending on the /M specification. The waves are designed to be used as an overlay on the original image (using the explicit=1 mode of **ModifyImage**). **Note**: the additional time required to create these waves is negligible compared with the time it takes to generate the stats data.

M_ParticlePerimeter	Masking image of particle boundaries. It is an unsigned char wave that contains 0 values for the object boundaries and 64 for all other points.
M_ParticleArea	Masking image of the area occupied by the particles. It is an unsigned char wave containing 0 values for the object boundaries and 64 for all other points. It is also different from the input image in that particles smaller than the minimum size, specified by /A, are absent.
M_Particle	Image of both the area and the boundary of the particles. It is an unsigned char wave that contains the value 16 for object area, the value 18 for the object boundaries and the value 64 for all other points.
M_rawMoments	Contains five columns. The first column is the raw sum of the x values for each particle, and the second column contains the sum of the y values. To obtain the average or "center" of a particle divide these values by the corresponding area. The third column contains the sum of x^2, the fourth column the sum of y^2, and the fifth column the sum of x*y. The entries of this wave are used in calculating a fit to an ellipse (using the /E flag).

When *imageMatrix* is a 3D wave, the different results are packed into a single 2D wave M_3DParticleInfo, which consists of one row and 11 columns for each particle. Columns are arranged in the following order: minRow, maxRow, minCol, maxCol, minLayer, maxLayer, xSeed, ySeed, zSeed, volume, and area. Use `Edit M_3DParticleInfo.ld` to display the results in a table with dimension labels describing the different columns.

Examples

Convert a grayscale image (blobs) into a proper binary input:

```
ImageThreshold/M=4/Q/I blobs
```

Get the statistics on the thresholded image of blobs and create an image mask output wave for the perimeter of the particles:

```
ImageAnalyzeParticles/M=1 stats M_ImageThresh
```

Display an image of the blobs with a red overlay of the perimeter image:

```
NewImage/F blobs; AppendImage M_ParticlePerimeter
ModifyImage M_ParticlePerimeter explicit=1, eval={0,65000,0,0}
```

See Also

The **ImageThreshold**, **ImageGenerateROIMask**, **ImageSeedFill**, and **ModifyImage** operations. For more usage details see **Particle Analysis** on page III-328.

ImageBlend

ImageBlend [**/A=*alpha* /W=*alphaWave***] ***srcWaveA*, *srcWaveB*** [**, *destWave***]

The ImageBlend operation takes two RGB images (3D waves) in *srcWaveA* and *srcWaveB* and computes the alpha blending so that

destWave = *srcWaveA* * (1 - *alpha*) + *srcWaveB* * *alpha*

for each color component. If *destWave* is not specified or does not already exist, the result is saved in the current data folder in the wave M_alphaBlend.

The source and destination waves must be of the same data types and the same dimensions. The *alphaWave*, if used, must be a single precision (SP) float wave and it must have the same number of rows and columns as the source waves.

Flags

/A=*alpha*	Specifies a single alpha value for the whole image
/W=*alphaWave*	Single precision wave that specifies an alpha value for each pixel.

ImageBoundaryToMask

ImageBoundaryToMask width=*w*, height=*h*, xwave=*xwavename*, ywave=*ywavename* [**,**
scalingWave=*scalingWaveName*, [**seedX=*xVal*, seedY=*yVal***]]

The ImageBoundaryToMask operation scan-converts a pair of XY waves into an ROI mask wave.

Parameters

height = *h*	Specifies the mask height in pixels.
scalingWave = *scalingWaveName*	2D or 3D wave that provides scaling for the mask. If specified, the scaling of the first two dimensions of scalingWave are copied to M_ROIMask, and both the X and Y waves are assumed to describe pixels in the scaled domain.
seedX = *xVal*	Specifies seed pixel location. The operation fills the region defined by the seed and the boundary with the value 1. Background pixels are set to zero. Requires seedY.
seedY = *yVal*	Specifies seed pixel location. The operation fills the region defined by the seed and the boundary with the value 1. Background pixels are set to zero. Requires seedX.

width = *w*	Specifies the mask width in pixels.
xwave = *xwavename*	Name of X wave for mask region.
ywave = *ywavename*	Name of Y wave for mask region.

Details

ImageBoundaryToMask generates an unsigned char 2D wave named M_ROIMask, of dimensions specified by width and height. The wave consists of a background pixels that are set to 0 and pixels representing the mask that are set to 1.

The x and y waves can be of any type. However, if the waves describe disjoint regions there must be at least one NaN entry in each wave corresponding to the discontinuity, which requires that you use either single or double precision waves. The values stored in the waves must correspond to zero-based integer pixel values.

If the x and y waves include a vertex that lies outside the mask rectangle, the offending vertex is moved to the boundary before the associated line segment is scan converted.

If you want to obtain a true ROI mask in which closed regions are filled, you can specify the seedX and seedY keywords. The ROI mask is set with zero outside the boundary of the domain and 1 everywhere inside the domain.

Examples

```
Make/O/N=(100,200) src=gnoise(5)           // create a test image
SetScale/P x 500,1,"", src;DelayUpdate      // give it some funny scaling
SetScale/P y 600,1,"", src
Display; AppendImage src
Make/O/N=201 xxx,yyy                  // create boundary waves
xxx=550+25*sin(p*pi/100)              // representing a close ellipse
yyy=700+35*cos(p*pi/100)
AppendToGraph yyy vs xxx
```

Now create a mask from the ellipse and scale it so that it will be appropriate for src:

```
ImageBoundaryToMask ywave=yyy,xwave=xxx,width=100,height=200,scalingwave=src
```

To generate an ROI masked filled with 1 in a region defined by a seed value and the boundary curves:

```
ImageBoundaryToMask
   ywave=yyy,xwave=xxx,width=100,height=200,scalingwave=src,seedx=550,seedy=700
```

See Also

The **ImageAnalyzeParticles** and **ImageSeedFill** operations. For another example see **Converting Boundary to a Mask** on page III-331.

ImageEdgeDetection

`ImageEdgeDetection [`*`flags`*`]` ***`Method imageMatrix`***

The ImageEdgeDetection operation performs one of several standard image edge detection operations on the source wave *imageMatrix*.

Unless the /O flag is specified, the resulting image is saved in the wave M_ImageEdges.

The edge detection methods produce binary images on output; the background is set to 0 and the edges to 255. This is due, in most cases to a thresholding performed in the final stage.

Except for the case of marr and shen detectors, you can use the /M flag to specify a method for automatic thresholding; see the **ImageThreshold** /M flag.

Parameters

Method selects type of edge detection. *Method* is one of the following names:

canny	Canny edge detector uses smoothing before edge detection and thresholding. You can optionally specify the threshold using the /T flag and the smoothing factor using /S.
frei	Calculates the Frei-Chen edge operator (see Pratt p. 503) using only the row and column filters.
kirsch	Kirsch edge detector (see Pratt p. 509). Performs convolution with 8 masks calculating gradients.

marr	Marr-Hildreth edge detector. Performs two convolutions with Laplacian of Gaussian and then detects zero crossings. Use the /S flag to define the width of the convolution kernel.
prewitt	Calculates the Prewitt compass gradient filters. Returns the result for the largest filter response.
roberts	Calculates the square root of the magnitude squared of the convolution with the Robert's row and column edge detectors.
shen	Shen-Castan optimized edge detector. Supposed to be effective in the presence of noise. The flags that modify this operation are: /F for the threshold ratio (0.9 by default), /S for smoothness factor (0.9 by default), /W for window width (default is 10), /H for thinning factor which by default is 1.
sobel	Sobel edge detector using convolutions with row and column edge gradient masks (see Pratt p. 501).

Flags

/F=*fraction*
: Determines the threshold value for the shen algorithm by starting from the histogram of the image and choosing a threshold such that *fraction* specifies the portion of the image pixels whose values are below the threshold. Valid values are in the interval (0 < *fraction* < 1).

/H=*thinning*
: Thins edges when used with shen edge detector. By default the thinning value is 1. Higher values produce thinner edges.

/I
: Inverts the output, i.e., sets the edges to 255 and the background to 0.

/M=*threshMethod*
: See the **ImageThreshold** automatic methods for obtaining a threshold value. Methods 1, 2, 4 and 5 are supported in this operation. If you use *threshMethod* = -1, threshold is not applied.

  If you want to apply your own thresholding algorithm, use /M=6 to bypass the thresholding completely. The wave M_RawCanny contains the result regardless of any other flags you may have used.

/N
: Sets the background level to 64 (i.e., NaN)

/O
: Overwrites the source image with the output image.

/P=*layer*
: Applies the operation to the specified layer of a 3D wave.

  /P is incompatible with /O.

  /P was added in Igor Pro 7.00.

/R=*roiSpec*
: Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u). The ROI wave must have the same number of rows and columns as the image wave. The ROI itself is defined by entries/pixels whose values are 0. Pixels outside the ROI can be any nonzero value. The ROI does not have to be contiguous and can be any arbitrary shape. See **ImageGenerateROIMask** for more information on creating ROI waves.

  In general, the *roiSpec* has the form {*roiWaveName*, *roiFlag*}, where *roiFlag* can take the following values:

roiFlag=0:	Set pixels outside the ROI to 0.
roiFlag=1:	Set pixels outside the ROI as in original image.
roiFlag=2:	Set pixels outside the ROI to NaN (=64).

  By default *roiFlag* is set to 1 and it is then possible to use the /R flag using the abbreviated form /R=*roiWave*.

/S= *smoothVal*
: Specifies the standard deviation or the width of the smoothing filter. By default the operation uses 1. Larger values require longer computation time. In the shen operation the default value is 0.9 and the valid range is (0 < *smoothVal*< 1).

/T=*thresh*	Sets a manual threshold for any method above that uses a single threshold. This is faster than using /M.
/W=*width*	Specifies window width when used in the shen operation. By default width is set to 10 and it is clipped to 49.

See Also

The **ImageGenerateROIMask** operation for creating ROIs and the **ImageThreshold** operation.

Edge Detectors on page III-318 for a number of examples.

References

Pratt, William K., *Digital Image Processing*, John Wiley, New York, 1991.

ImageFileInfo

`ImageFileInfo [/P=pathName] fileNameStr`

As of Igor Pro 7, ImageFileInfo is no longer supported an always returns an error.

It is obsolete because it used QuickTime to obtain graphics file information and Apple is phasing out QuickTime.

ImageFilter

`ImageFilter [flags] Method dataMatrix`

The ImageFilter operation is identical to **MatrixFilter**, accepting the same parameters and flags, with the exception of the additional features described below.

Parameters

Method selects the filter type. *Method* is one of the following names:

avg3d	*nxnxn* average filter for 3D waves.
gauss3d	*nxnxn* gaussian filter for 3D waves.
hybridmedian	Implements ranking pixel values between two groups of pixels in a 5x5 neighborhood. The first group includes horizontal and vertical lines through the center, the second group includes diagonal lines through the center, and both groups include the center pixel itself. The resulting median value is the ranked median of both groups and the center pixel.
max3d	*nxnxn* maximum rank filter for 3D waves.
median3d	*nxnxn* median filter for 3D waves where *n* must be of the form 3^r (integer *r*), e.g., 3x3x3, 9x9x9 etc. The filter does not change the value of the voxel it is centered on if any of the filter voxels lies outside the domain of the data.
min3d	*nxnxn* minimum rank filter for 3D waves.
point3d	*nxnxn* point finding filter using normalized `(`n^3`-1)*`*`center-outer`* for 3D waves.

Flags

/N=*n*	Specifies the filter size. By default *n* =3. In most situations it will be useful to set *n* to an odd number in order to preserve the symmetry in the filters.
/O	Overwrites the source image with the output image. Used only with the hybridmedian filter, which does not automatically overwrite the source wave.

Details

You can operate on 3D waves using the 3D filters listed above. These filters are extensions of the 2D filters available under MatrixFilter. The avg3d, gauss3d, and point3d filters are implemented by a 3D convolution that uses an averaging compensation at the edges.

This operation does not support complex waves.

See Also

MatrixFilter for descriptions of the other available parameters and flags.

MatrixConvolve for information about convolving your own 3D kernels.

References

Russ, J., *Image Processing Handbook*, CRC Press, 1998.

ImageFocus

`ImageFocus [flags] stackWave`

The ImageFocus operation creates in focus image(s) from a stack of images that contain in and out of focus regions. It computes the variance in a small neighborhood around each pixel and then takes the pixel value from the plane in which the highest variance is found.

Flags

/ED=*edepth*	Sets the effective depth in planes. For example, an effective depth of one means that it computes the best focus for each plane using a stack of three planes, which includes the current plane and any one adjacent plane above and below it. Does not affect the default method (/METH=0).
/METH=*method*	Specifies the calculation method. *method*=0: Computes a single plane output for the stack (default). *method*=1: Computes the best image for each plane using /ED.
/Q	Quiet mode; no output to history area.
/Z	No error reporting.

See Also

Chapter III-11, **Image Processing** contains links to and descriptions of other image operations.

ImageFromXYZ

`ImageFromXYZ [flags] xyzWave, dataMatrix, countMatrix`

`ImageFromXYZ [flags] {xWave,yWave,zWave}, dataMatrix, countMatrix`

ImageFromXYZ converts XYZ data to matrix form. You might use it, for example, to convert a "sparse matrix" to an actual matrix for easier display and processing.

You provide the input data in the XYZ triplet *xyzWave* or in 1D waves *xwave*, *ywave*, and *zwave*.

dataMatrix and *countMatrix* receive output data but you must create them prior to calling ImageFromXYZ.

For each XY location in the input data, ImageFromXYZ adds the corresponding Z value to an element of *dataMatrix*. The element is determined based on the input XY location and the X and Y scaling of *dataMatrix*.

For each XY location in the input data, ImageFromXYZ increments the corresponding element of *countMatrix*. This permits you to obtain an average Z value if multiple input values fall into a given element of *dataMatrix*.

Parameters

xyzWave is a triplet wave containing the input XYZ data.

xWave, *yWave* and *zWave* are 1D input waves containing XYZ data.

You specify either *xyzWave* by itself or *xWave*, *yWave* and *zWave* in braces.

dataMatrix is a 2D wave to which the Z values are added. It must be either single-precision or double-precision floating point. The X and Y scaling of *dataMatrix* determines how input values are mapped to output matrix elements.

countMatrix is a 2D wave the elements of which store the number of Z values added to each corresponding element of *dataMatrix*. ImageFromXYZ sets it to 32-bit integer if it is not already so.

Flags

/AS	If /AS (autoscale) is specified, ImageFromXYZ clears both *dataMatrix* and *countMatrix* and sets the X and Y scaling of *dataMatrix* based on the range of X and Y input values.

Details

For each point in the XYZ input data, ImageFromXYZ adds the Z value to the appropriate element of *dataMatrix* and increments the corresponding element of *countMatrix*. Normally you will clear *dataMatrix* and *countMatrix* before calling it.

You can combine multiple XYZ datasets in one matrix by calling ImageFromXYZ multiple times with different input data and the same *dataMatrix* and *countMatrix*. In this case you would clear *dataMatrix* and *countMatrix* before the first call to ImageFromXYZ only.

What you do with the output is up to you but one technique is to divide *dataMatrix* by *countMatrix* to get the average and then use **MatrixFilter** NanZapMedian to eliminate any NaN values that result from zero divided by zero.

Example

```
Make /N=1000 /O wx=enoise(2), wy= enoise(2), wz= exp(-(wx^2+wy^2))
Make /O /N=(100,100) dataMat=0
SetScale x,-2,2,dataMat
SetScale y,-2,2,dataMat
Duplicate /O dataMat,countMat
ImageFromXYZ /AS {wx,wy,wz}, dataMat, countMat

// Execute these one at a time
NewImage dataMat
dataMat /= countMat                  // Replace cumulative z value with average
MatrixFilter NanZapMedian, dataMat   // Apply median filter, zapping NaNs
```

See Also

SetScale, **Image X and Y Coordinates** on page II-301.

ImageGenerateROIMask

`ImageGenerateROIMask [/W=`*`winName`*`/E=`*`e`*`/I=`*`i`*`]` ***`imageInstance`***

The ImageGenerateROIMask operation creates a Region Of Interest (ROI) mask for use with other ImageXXX commands. It assumes the top (or /W specified) graph contains an image and that the user has drawn shapes using Igor's drawing tools in a specific manner.

ImageGenerateROIMask creates an unsigned byte mask matrix with the same x and y dimensions and scaling as the specified image. The mask is initially filled with zeros. Then the drawing layer, progFront, in the graph is scanned for suitable fillable draw objects. The area inside each shape is filled with ones unless the fill mode for the shape is set to erase in which case the area is filled with zeros.

Flags

/E=*e*	Changes value used for the exterior from the default zero values to *e*.
/I=*i*	Changes value used for the interior from the default one values to *i*.
/W=*winName*	Looks for the named graph window or subwindow containing appropriate image masks drawn by the user. If /W is omitted, ImageGenerateROIMask uses the top graph window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

To generate an ROI wave for use with most image processing operations you need to set the values of interior pixels to zero and exterior pixels to one using /E=1/I=0.

Suitable objects are those that can be filled (rectangles, ovals, etc.) and which are plotted in axis coordinate mode specified using the same axes by which the specified image instance is displayed. Objects plotted in plot relative mode are also used, However, this is not recommended because it will give correct results only if the image exactly fills the plot rectangle. If you use axis coordinate mode then you can zoom in or out as desired and the resulting mask will still be correct.

Note that the shapes can have their fill mode set to none. This still results in a fill of ones. This is to allow the drawn ROI to be visible on the graph without obscuring the image. However cutouts (fills with erase mode) will obscure the image.

Note also that nonfill drawing objects are ignored. You can use this fact to create callouts and other annotations.

In a future version of Igor, we may create a new drawing layer in graphs dedicated to ROIs.

The mask generated is named M_ROIMask and is generated in the current data folder.

Variable V_flag is set to 1 if the top graph contained draw objects in the correct layer and 0 if not. If 0 then the M_ROIMask wave was not generated.

Examples

```
Make/O/N=(200,400) jack=x*y; NewImage jack; ShowTools
SetDrawLayer ProgFront
SetDrawEnv linefgc=(65535,65535,0),fillpat=0,xcoord=top,ycoord=left,save
DrawRect 63.5,79.5,140.5,191.5
DrawRRect 61.5,206.5,141.5,280.5
SetDrawEnv fillpat= -1
DrawOval 80.5,169.5,126.5,226.5
ImageGenerateROIMask jack
NewImage M_ROIMask
AutoPositionWindow/E
```

See Also

For another example see **Generating ROI Masks** on page III-331.

ImageGLCM

`ImageGLCM [`*`flags`*`]` ***`srcWave`***

The ImageGLCM operation calculates the gray-level co-occurrence matrix for an 8-bit grayscale image and optionally evaluates Haralick's texture parameters.

The ImageGLCM operation was added in Igor Pro 7.00.

Flags

/D=*distance*	Sets the offset in pixels for which the co-occurrence matrix is calculated. The default value is 1.
/DEST=*destGLCM*	Specifies the wave to hold the co-occurrence matrix. If you omit /DEST the operation stores the matrix in the wave M_GLCM in the current data folder.
/DETP=*destParamWave*	Specifies the wave to hold the computed texture parameters. If you omit /DETP the operation stores the texture parameters in the wave W_TextureParams in the current data folder. If the destination wave already exists it is overwritten. Note that you must specify the /HTFP flag to compute the texture parameters.
/E=*structureBits*	*structureBits* is a bitwise setting that lets you control the combination of co-occurrences that you want to compute. Consider a wave displayed in a table and a pixel at position x

$$\begin{array}{ccc} 0 & 3 & 5 \\ 1 & x & 6 \\ 2 & 4 & 7 \end{array}$$

The *structureBits* corresponding to co-occurrence between x and any direction is simply 2^direction. By default the operation computes all combinations. This is equivalent to *structureBits*=255.

Note that the *structureBits* only define directions. The combination of the distance (/D) and the *structureBits* define the full co-occurrence calculation.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/FREE	Creates output waves as free waves. /FREE is permitted in user-defined functions only, not from the command line or in macros. If you use /FREE then *destGLCM* and *destParamWave* must be simple names, not paths. See **Free Waves** on page IV-84 for details on free waves.
/HTFP	Computes Haralick's texture parameters. See the discussion in the Details section below for more information about the texture parameters.
/P=plane	If the image consists of more than one plane you can use this flag to determine which plane in srcWave is analysed. By default it is plane zero.
/Z	No error reporting.

Details

ImageGLCM computes the co-occurrence matrix for the image in *srcWave* and optionally evaluates Haralick's texture parameters. The operation supports 8-bit grayscale images and generates a 256x256 single-precision floating point co-occurrence matrix.

The elements of the matrix P[i][j] are defined as the normalized number of pixels that have a spatial relationship defined by the distance (/D) and the structure (/E) such that the first pixel has gray-level i and the second pixel has gray-level j. The matrix is normalized so that the sum of all its elements is 1.

If you specify the /HTFP flag the operation computes the 13 Haralick texture parameters and stores them sequentially in the destination wave (see /DETP). The wave is saved with dimension labels defining each element. The expressions for the texture parameters are:

$$f_1 = \sum_i \sum_j \left(p[i][j]\right)^2,$$

$$f_2 = \sum_{n=0}^{254} n^2 \left\{ \sum_{i=0}^{255} \sum_{\substack{j=0 \\ |i-j|=n}}^{255} p[i][j] \right\},$$

$$f_3 = \sum_i \sum_j \frac{(i-\mu_x)(j-\mu_y)p[i][j]}{\sigma_x \sigma_y},$$

$$f_4 = \sum_i \sum_j (i-\mu)^2 p[i][j],$$

$$f_5 = \sum_i \sum_j \frac{1}{1+(i-j)^2} p[i][j],$$

$$f_6 = \sum_i i p_{x+y}(i),$$

$$f_7 = \sum_i (i - f_6)^2 p_{x+y}(i),$$

$$f_8 = -\sum_i p_{x+y}(i) \log\left(p_{x+y}(i)\right),$$

$$f_9 = -\sum_i \sum_j p[i][j] \log\left(p[i][j]\right),$$

$$f_{10} = Variance\left(p_{x-y}\right),$$

$$f_{11} = \sum_i p_{x-y}(i) \log\left(p_{x-y}(i)\right),$$

$$f_{12} = \frac{f_9 - HXY1}{\max\left(HX, HY\right)},$$

$$f_{13} = \sqrt{1 - \exp\left(-2\left(HXY2 - f9\right)\right)}.$$

Here

$$p_x(i) = \sum_j p[i][j], \quad p_y(j) = \sum_i p[i][j],$$

$$\mu_x = \sum_i i p_x(i), \quad \mu_y = \sum_i i p_y(i), \quad \mu = (\mu_x + \mu_y)/2.$$

$$\sigma_x = \sqrt{\sum_i \left(1 - \mu_x\right)^2 p_x(i)}, \qquad \sigma_y = \sqrt{\sum_i \left(1 - \mu_y\right)^2 p_y(i)},$$

$$p_{x+y}(k) = \sum_i \sum_{\substack{j \\ i+j=k}} p[i][j],$$

$$p_{x-y}(k) = \sum_i \sum_{\substack{j \\ |i-j|=k}} p[i][j],$$

$$HXY1 = -\sum_i \sum_j p[i][j] \log\left(p_x(i) p_y(j)\right),$$

$$HXY2 = -\sum_i \sum_j p_x(i) p_y(j) \log\left(p_x(i) p_y(j)\right),$$

$$HX = -\sum_i p_x(i) \log\left(p_x(i)\right), \quad HY = -\sum_i p_y(i) \log\left(p_y(i)\right).$$

There are at least two versions of f7 used in the literature and in software. We know of at least three versions of f14 so ImageGLCM does not compute it.

References

R.M. Haralick, K. Shanmugam and Itshak Dinstein, "Textural Features for Image Classification", IEEE Transactions on Systems, Man, and Cybernetics, 1973.

ImageHistModification

`ImageHistModification [`*`flags`*`]` ***`imageMatrix`***

The ImageHistModification operation performs a modification of the image histogram and saves the results in the wave M_ImageHistEq. If /W is not specified, the operation is a simple histogram equalization of *imageMatrix*. If /W is specified, the operation attempts to produce an image with a histogram close to *waveName*. If /A is specified, the operation performs an adaptive histogram equalization. *imageMatrix* is a wave of any noncomplex numeric type. Adaptive histogram equalization applies only to 2D waves and the other parts apply to both 2D and 3D waves.

Flags

/A	Performs an adaptive histogram equalization by subdividing the image into a minimum of 4 rectangular domains and using interpolation to account for the boundaries between adjacent domains. When the /C flag is specified with contrast factor greater than 1, this operation amounts to contrast-limited adaptive histogram equalization. By default the operation divides the image into 8 horizontal and 8 vertical regions. See /H and /V.
/B=*bins*	Specifies the number of *bins* used with the /A flag. If not specified, this value defaults to 256.
/C=*cFactor*	Specifies a contrast factor (or clipping value) above which pixels are equally distributed over the whole range. *cFactor* must be greater than 1, in the limit as *cFactor* approaches 1 the operation is a regular adaptive histogram equalization. **Note**: this flag is used only with the /A flag.
/H=*hRegions*	Specifies the number of horizontal subdivisions to be used with the /A feature. Note, the number of image pixels in the horizontal direction must be an integer multiple of *hRegions*.
/I	Extends the standard histogram equalization by using 2^{16} bins instead of 2^8 when calculating histogram equalization. This feature does not apply to the adaptive histogram equalization (/A flag).

/O	Overwrites the source image. If this flag is not specified, the resulting image is saved in the wave M_ImageHistEq.
/R=*roiSpec*	Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u). The ROI wave must have the same number of rows and columns as *imageMatrix*. The ROI itself is defined by the entries whose values are 0. Regions outside the ROI can take any nonzero value. The ROI does not have to be contiguous and can take any arbitrary shape.

In general, the *roiSpec* has the form {*roiWaveName*, *roiFlag*}, where *roiFlag* can take the following values:

roiFlag=0:	Set pixels outside the ROI to 0.
roiFlag=1:	Set pixels outside the ROI as in original image (default).
roiFlag=2:	Set pixels outside the ROI to NaN (=64).

By default *roiFlag* is set to 1 and it is then possible to use the /R flag with the abbreviated form /R=*roiWave*. When *imageMatrix* is a 3D wave, *roiWave* can be either a 2D wave (matching the number of rows and columns in *imageMatrix*) or it can be a 3D wave which must have the same number of rows, columns, and layers as *imageMatrix*. When using a 2D *roiWave* with a 3D *imageMatrix*, the ROI is understood to be defined by *roiWave* for each layer in the 3D wave.

See **ImageGenerateROIMask** for more information on creating ROI waves.

/V=*vRegions*	Specifies the number of vertical subdivisions to be used with the /A flag. The number of image pixels in the horizontal direction must be an integer multiple of *vRegions*. If the image dimensions are not divisible by the number of regions that you want, you can pad the image using ImageTransform **padImage**.
/W=*waveName*	Specifies a 256-point wave that provides the desired histogram. The operation will attempt to produce an image having approximately the desired histogram values. This flag does not apply to the adaptive histogram equalization (/A flag)

See Also

The **ImageGenerateROIMask** and **ImageTransform** operations for creating ROIs. For examples see **Histograms** on page III-325 and **Adaptive Histogram Equalization** on page III-307.

ImageHistogram

`ImageHistogram [`*`flags`*`]` ***`imageMatrix`***

The ImageHistogram operation calculates the histogram of *imageMatrix*. The results are saved in the wave W_ImageHist. If *imageMatrix* is an RGB image stored as a 3D wave, the resulting histograms for each color plane are saved in W_ImageHistR, W_ImageHistG, W_ImageHistB.

imageMatrix must be a real-valued numeric wave.

Flags

/I	Calculates a histogram with 65536 bins evenly distributed between the minimum and maximum data values. The operation first finds the extrema and then calculates the bins and the resulting histogram. Data can be a 2D wave of any type including float or double.
/P=*plane*	Restricts the calculation of the histogram to a specific plane when *imageMatrix* is a non RGB 3D wave.

/R=*roiWave*	Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u) that has the same number of rows and columns as *imageMatrix*. The ROI itself is defined by the entries o pixels in the *roiWave* with value of 0. Pixels outside the ROI may have any nonzero value. The ROI does not have to be contiguous. When *imageMatrix* is a 3D wave, *roiWave* can be either a 2D wave (matching the number of rows and columns in *imageMatrix*) or it can be a 3D wave that must have the same number of rows, columns and layers as *imageMatrix*. When using a 2D *roiWave* with a 3D *imageMatrix* the ROI is understood to be defined by *roiWave* for each layer in the 3D wave. See **ImageGenerateROIMask** for more information on creating 2D ROI waves.
/S	Computes the histogram for a whole 3D wave possibly subject to 2D or 3D ROI masking. The /S and /P flags are mutually exclusive.

Details

The ImageHistogram operation works on images, but it handles both 2D and 3D waves of any data type. Unless you use one of the special features of this operation (e.g., ROI or /P or /I) you could alternatively use the **Histogram** operation, which computes the histogram for the full wave and includes additional options for controlling the number of bins.

If the data type of *imageMatrix* is single byte, the histogram will have 256 bins from 0 to 255. Otherwise, the 256 bins will be distributed between the minimum and maximum values encountered in the data. Use the /I flag to increase the number of bins to 65536, which may be useful for unsigned short (/W/U) data.

See Also

ImageHistModification, **ImageGenerateROIMask**, **JointHistogram**, **Histograms** on page III-325

ImageInfo

`ImageInfo(`*`graphNameStr`*`, `*`imageWaveNameStr`*`, `*`instanceNumber`*`)`

The ImageInfo function returns a string containing a semicolon-separated list of information about the specified image in the named graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

imageWaveNameStr contains either the name of a wave displayed as an image in the named graph, or an image instance name (wave name with “#n” appended to distinguish the nth image of the wave in the graph). You might get an image instance name from the **ImageNameList** function.

If *imageWaveNameStr* contains a wave name, *instanceNumber* identifies which instance you want information about. *instanceNumber* is usually 0 because there is normally only one instance of a wave displayed as an image in a graph. Set *instanceNumber* to 1 for information about the second image of the wave, etc. If *imageWaveNameStr* is `""`, then information is returned on the *instanceNumber*th image in the graph.

If *imageWaveNameStr* contains an instance name, and *instanceNumber* is zero, the instance is taken from *imageWaveNameStr*. If *instanceNumber* is greater than zero, the wave name is extracted from *imageWaveNameStr*, and information is returned concerning the *instanceNumber*th instance of the wave.

Details

The string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon for ease of use with **StringByKey**. The keywords are as follows:

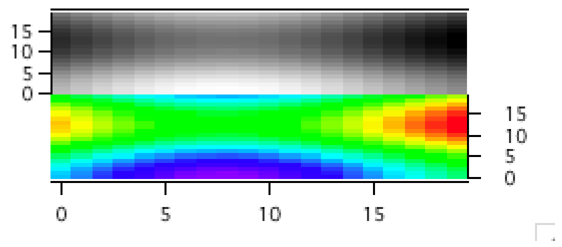
Keyword	Information Following Keyword
AXISFLAGS	Flags used to specify the axes. Usually blank because /L and /B (left and bottom axes) are the defaults.
COLORMODE	A number indicating how the image colors are derived: 1: Color table (see **Image Color Tables** on page II-305). 2: Scaled color index wave (see **Indexed Color Details** on page II-312). 3: Point-scaled color index (See **Example: Point-Scaled Color Index Wave** on page II-313). 4: Direct color (see **Direct Color Details** on page II-313). 5: Explicit Mode (See **ModifyImage** explicit keyword).
RECREATION	Semicolon-separated list of *keyword=modifyParameters* commands for the ModifyImage command.
XAXIS	X axis name.
XWAVE	X wave name if any, else blank.
XWAVEDF	The full path to the data folder containing the X wave or blank if there is no X wave.
YAXIS	Y axis name.
YWAVE	Y wave name if any, else blank.
YWAVEDF	The full path to the data folder containing the Y wave or blank if there is no Y wave.
ZWAVE	Name of wave containing Z data used to calculate the image plot.
ZWAVEDF	The full path to the data folder containing the Z data wave.

The format of the RECREATION information is designed so that you can extract a keyword command from the keyword and colon up to the "`;`", prepend "`ModifyImage `", replace the "`x`" with the name of a image plot ("`data#1`" for instance) and then **Execute** the resultant string as a command.

Example 1

This example gets the image information for the second image plot of the wave "jack" (which has an instance number of 1) and applies its **ModifyImage** settings to the first image plot.

```
#include <Graph Utility Procs>, version>=6.1   // For WMGetRECREATIONFromInfo

// Make two image plots of the same data on different left and right axes
Make/O/N=(20,20) jack=sin(x/5)+cos(y/4)
Display;AppendImage jack                        // bottom and left axes
AppendImage/R jack                              // bottom and right axes

// Put image plot jack#0 above jack#1
ModifyGraph axisEnab(left)={0.5,1},axisEnab(right)={0,0.5}

// Set jack#1 to use the Rainbow color table instead of the default Grays
ModifyImage jack#1 ctab={*,*,Rainbow,0}
```



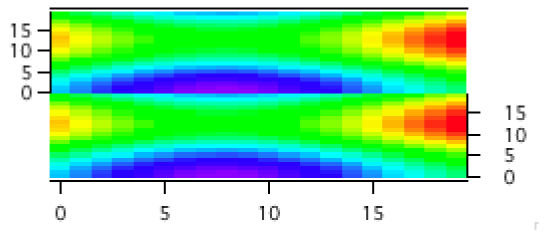
Now we peek at some of the image information for the second image plot of the wave "jack" (which has an instance number of 1) displayed in the top graph:

```
Print ImageInfo("","jack",1)[69,148]        // Just the interesting stuff
```

```
;ZWAVE:jack;ZWAVEDF:root:;COLORMODE:1;RECREATION:ctab= {*,*,Rainbow,0};plane= 0;
```

```
// Apply the color table, etc from jack#1 to jack:
String info= WMGetRECREATIONFromInfo(ImageInfo("","jack",1))
info= RemoveEnding(info)                    // Remove trailing semicolon
```

```
// Use comma instead of semicolon separators
String text = ReplaceString(";", info, ",")
Execute "ModifyImage jack " + text
```



Example 2

This example gets the full path to the wave containing the Z data from which the first image plot in the top graph was calculated.

```
String info= ImageInfo("","",0)      // 0 is index of first image plot
String pathToZ= StringByKey("ZWAVEDF",info)+StringByKey("ZWAVE",info)
Print pathToZ
   root:jack
```

See Also

The **ModifyImage**, **AppendImage**, **NewImage** and **Execute** operations.

Image Plots on page II-297.

Image Instance Names on page II-314.

ImageInterpolate

`ImageInterpolate [*flags*] *Method srcWave*`

The ImageInterpolate operation interpolates the source *srcWave* and stores the results in the wave M_InterpolatedImage in the current data folder unless you specify a different destination wave using the /DEST flag.

Parameters

Method selects type of interpolation. *Method* is one of the following names:

Affine2D	Performs an affine transformation on *srcWave* using parameters specified by the /APRM flag. The transformation applies to a general combination of rotation, scaling, and translation represented by a 3x3 matrix

$$M = \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & w \end{bmatrix}.$$

The upper 2x2 matrix is a composite of rotation and scaling, *tx* and *ty* are composite translations and *w* is usually 1. It computes the dimensions of the output wave and then uses the inverse transformation and bilinear interpolation to compute the value of each output pixel. When an output pixel does not map back into the source domain it is set to the user-specified background value. It supports 2D and 3D input waves. If *srcWave* is a 3D wave it applies on a layer by layer basis.

The output is stored in the wave M_Affine in the current data folder.

/ATOL Allows ImageInterpolate to use a tolerance value if the result of the interpolation at any point is NaN. The algorithm returns the first non-NaN value it finds by adding or subtracting 1/10000th of the size of the X or Y interpolation step. At worst this algorithm is 5 times slower than the default algorithm if the interpolation is performed in a region which is completely outside the convex source domain.

/ATOL was added in Igor Pro 7.00.

/CMSH Use this flag in Voronoi interpolation to create a triangle mesh surface representing the input data. After triangulating the X, Y locations (in a plane z=const), the mesh is generated from a sequence of XYZ vertices of all the resulting triangles. The output is stored in the wave M_ScatterMesh in the current data folder. It is in the form of a triplet wave where every consecutive 3 rows represent a disjoint triangle.

If the input contains a NaN or INF in any column, the corresponding row is excluded from the triangulation.

If you want to generate this mesh without generating the interpolated matrix you can omit the /S flag.

/CMSH was added in Igor Pro 7.00.

Bilinear Performs a bilinear interpolation subject to the specified flag. You can use either the /F or /S flag, but not both.

Kriging Uses Kriging to generate an interpolated matrix from a sparse data set. Kriging calculates interpolated values for a rectangular domain specified by the /S flag. The Kriging parameters are specified via the /K flag.

Kriging is computed globally for a single user-selected variogram model. If there are significant spatial variances within the domain occupied by the data, you should consider subdividing the domain along natural boundaries and use a single variogram model in each subdivision.

If there are N data points, the algorithm first computes the NxN matrix containing the distances between the data and then inverts an associated matrix of similar size to compute the result for the selected variogram model. Because inversion of an NxN matrix can be computationally expensive, you should consider restricting the calculation to regions that are similar to the range implied by the variogram. Such an approach can also be justified in the sense that the local interpolation should not be affected by a remote datum.

Note: Kriging does not support data containing NaNs or INFs. Wave scaling has no effect.

Pixelate Creates a lower resolution (pixelated) version of *srcWave* by averaging the pixels inside domain specified by /PXSZ flag. The results are saved in the wave M_PixelatedImage in the current data folder.

The computed wave has the same numeric type as *srcWave* so the averaging may be inaccurate in the case of integer waves.

When *srcWave* is a 3D wave you have the option of averaging only over data in each layer using /PXSZ={nx,ny} or averaging over data in a rectangular cube using /PXSZ={nx,ny,nz}.

When not averaging over layers, the number of rows and columns of the new image are obtained by integer division of the original number by the respective size of the averaging rectangle and adding one more pixel for any remainder.

When averaging over layers the resulting rows and columns are obtained by truncated integer division ignoring remainders (if any).

Resample Computes a new image based on the selected interpolation function and transformation parameters. Set the interpolation function with the /FUNC flag. Use the /TRNS flag to specify transformation parameters for grayscale images, or /TRNR, /TRNG, and /TRNB for the red, green, and blue components, respectively, of RGB images. M_InterpolatedImage contains the output image in the current data folder.

There are currently two transformation functions: the first magnifies an image and the second applies a radial polynomial sampling. The radial polynomial affects pixels based on their position relative to the image center. A linear polynomial reproduces the same image. Any nonlinear terms contribute to distortion (or correction thereof).

Spline Computes a 2D spline interpolation for 2D matrix data. The degree of the spline is specified by the /D flag.

Voronoi Generates an interpolated matrix from a sparse data set (*srcWave* must be a triplet wave) using Voronoi polygons. It calculates interpolated values for a rectangular domain as specified by the /S or /RESL flags.

It first computes the Delaunay triangulation of X, Y locations in the Z=0 plane (assuming that X, Y positions occupy a convex domain in the plane). It then uses the Voronoi dual to interpolate the Z values for X and Y pairs from the grid defined by the /S flag. The computed grid may exceed the bounds of the convex domain defined by the triangulation. Interpolated values for points outside the convex domain are set to NaN or the value specified by the /E flag.

Use the /I flag to iterate to finer triangulation by subdividing the original triangles into smaller domains. Each iteration increases computation time by approximately a factor of two, but improves the smoothness of the interpolation.

If you have multiple sets of data in which X,Y locations are unchanged, you can use the /STW flag to store one triangulation and then use the /PTW flag to apply the precomputed triangulation to a new interpolation. To use this option you should use the Voronoi keyword first with a triplet wave for *srcWave* and set xn = x0 and yn = y0. The operation creates the wave W_TriangulationData that you use in the next triangulation with a 1D wave as *srcWave*.

Voronoi interpolation is similar to what can be accomplished with the **ContourZ** function except that it does not require an existing contour plot, it computes the whole output matrix in one call, and it has the option of controlling the subdivision iterations.

See **Voronoi Interpolation Example** below.

XYWaves Performs bilinear interpolation on a matrix scaled using two X and Y 1D waves (specified by /W). The interpolation range is defined by /S. The data domain is defined between the centers of the first and last pixels (X in this example):

```
xmin=(xWave[0]+xWave[1])/2
xmax=(xWave[last]+xWave[last-1])/2
```

Values outside the domain of the data are set to NaN. The interpolation is contained in the M_InterpolatedImage wave, which is single precision floating point or double precision if *srcWave* is double precision.

Warp	Performs image warping interpolation using a two step algorithm with three optional interpolation methods. The operation warps the image based on the relative positions of source and destination grids. The warped image has the same size as the source image. The source and destination grids are each specified by a pair of 2D X and Y waves where the rows and columns correspond to the relative location of the source grid. The smallest supported dimensions of grid waves are 2x2. All grid waves must be double-precision floating point and must have the same number of points corresponding to pixel positions within the image. Grid waves must not contain NaNs or INFs. Wave scaling is ignored.

Flags

/APRM={*r11*,*r12*,*tx*,*r21*,*r22*,*ty*,*w*,*background*}

Sets elements of the affine transformation matrix and the background value.

/D=*splineDeg* Specifies the spline degree with the Spline method. The default spline degree is 2. Supported values are 2, 3, 4, and 5.

/DEST=*destWave* Specifies the wave to contain the output of the operation. If the specified wave already exists, it is overwritten.

Creates a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-66 for details.

/E=*outerValue* Assigns *outerValue* to all points outside the convex domain of the Delaunay triangulation. By default *outerValue* = NaN.

/F={*fx*,*fy*} Calculates a bilinear interpolation of all the source data. Here *fx* is the sampling factor for the X-direction and *fy* is the sampling factor in the Y-direction. The output number of points in a dimension is factor*(number of data intervals) +1. The number of data intervals is one less than the number of points in that dimension.

For example, if *srcWave* is a 2x2 matrix (you have a single data interval in each direction) and you use /F={2,2}, then the output wave is a 3x3 matrix (i.e., 2x2 intervals) which is a factor of 2 of the input. Sampling factors can be noninteger values.

/FDS Performs spline interpolation to a “local” cubic spline that depends only on the 2D neighboring data. The interpolation goes exactly through the original data and provides continuity of the function and its first derivative. The second derivative is set to vanish on the boundary. The implemented algorithm was provided by Francis Dalaudier. /FDS was added in Igor Pro 7.00.

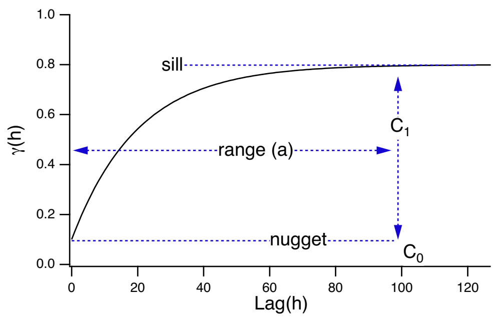
/FUNC=*funcName* Specifies the interpolation function. *funcName* can be:

nn	Nearest neighbor interpolation uses the value of the nearest neighbor without interpolation. This is the fastest function.
bilinear	Bilinear interpolation uses the immediately surrounding pixels and computes a linear interpolation in each dimension. This is the second fastest function.
cubic	Cubic polynomial (photoshop-like) uses a 4x4 neighborhood value to compute the sampled pixel value.
spline	Spline smoothed sampled value uses a 4x4 neighborhood around the pixel.
sinc	Slowest function using a 16x16 neighborhood.

/I=*iterations* Specifies the number of times the original triangulation is subdivided with the Voronoi interpolation method. By default the Voronoi interpolation computes the original triangulation without subdivision.

/K={*model*, *nugget*, *sill*, *range*}

Specifies the variogram parameters for kriging using standard notation, models are expressed in terms of the nugget value C_0, sill value C_0+C_1, and range a.



model	Selects the variogram model. Values and models are: 1: Spherical. $\gamma(h) = C_0 + C_1 \cdot (3h/2a - 0.5 \cdot h^3/a^3)$ 2: Exponential. $\gamma(h) = C_0 + C_1 \cdot (1 - \exp[-3 \cdot h/a])$ 3: Gaussian. $\gamma(h) = C_0 + C_1 \cdot (1 - \exp[-3 \cdot (h/a)^2])$
nugget	Specifies the lowest value in the variogram.
sill	Specifies the maximum (plateau) value in the variogram range the characteristic length of the different variogram models.

Wave scaling has no effect on kriging calculations.

/PFTL=*tol* In Voronoi interpolation, controls the rectangular neighborhood about each datum position XY where the algorithm returns the original Z-value. The tolerance value 0 <= *tol* < 1 is tested separately in X and Y (i.e., it is not a geometric distance) when both X and Y are normalized to the range [0,1]. By default *tol*=1e-15.

Added in Igor Pro 7.00.

/PTW=*tWave* Uses a previous triangulation wave with Voronoi interpolation. *tWave* will be the wave saved by the /STW flag. You can't use a triangulation wave that was computed and saved on a different computer platform.

See **PTW Flag Example** on page V-332.

/PXSZ={*nx*,*ny*} Specifies the size in pixels of the averaging rectangle used by the Pixelate operation. nx is the number of rows and ny is the number of columns that are averaged to yield a single output pixel. If srcWave is a 3D wave the resulting wave has the same number of layers as srcWave with pixelation computed on a layer-by-layer basis.

/PXSZ={*nx*,*ny*,*nz*} Specifies the size in pixels of the averaging rectangle used by the Pixelate operation. nx is the number of rows and ny is the number of columns and nz is the number of layers that are averaged to yield a single output pixel.

This form of the /PXSZ flag was added in Igor Pro 7.00.

/RESL={*nx*, *ny*} Specifies resampling the full input image to an output image having *nx* rows by *ny* columns.

/S={*x0*,*dx*,*xn*,*y0*,*dy*,*yn*}

Calculates a bilinear interpolation of a subset of the source data. Here *x0* is the starting point in the X-direction, *dx* is the sampling increment, *xn* is the end point in the X-direction and the corresponding values for the Y-direction. If you set *x0* equal to *xn* the operation will compute the triangulation but not the interpolation.

/SPRT Skips the XY perturbation step. The perturbation step is designed to break degeneracies that originate when the XY data are sampled on a rectangular grid. If your data are not sampled on a rectangular grid you can skip the perturbation and get better accuracy in reproducing the Z-values at the sampled locations. See also **ModifyContour** with the keyword Perturbation.

Added in Igor Pro 7.00.

/STW Saves the triangulation information in the wave W_TriangulationData in the current data folder. W_TriangulationData can only be used on the computer platform where it was created.

/SV Saves the Voronoi interpolation in the 2D wave M_VoronoiEdges, which contains sequential edges of the Voronoi polygons. Edges are separated from each other by a row of NaNs. The outer most polygons share one or more edges with a large triangle containing the convex domain.

/TRNS={*transformFunc*,*p1*,*p2*,*p3*,*p4*}

Determines the mapping between a pixel in the destination image and the source pixel. *transformFunc* can be:

scaleShift	Sets image scaling which could be anamorphic if the X and Y scaling are different.
radialPoly	Corrects both color as well as barrel and pincushion distortion. In `radialPoly` the mapping from a destination pixel to a source pixel is a polynomial in the pixel's radius relative to the center of the image.

A source pixel, *sr,* satisfies the equation:

$$sr = ar + br^2 + cr^3 + dr^4,$$

where *r* is the radius of a destination pixel having an origin at the center of the destination image.

The corresponding parameters are:

transformFunc	***p1***	***p2***	***p3***	***p4***
scaleShift	xOffset	xScale	yOffset	yScale
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

/U=*uniformScale* Calculates a bilinear interpolation of all the source data as with the /F flag but with two exceptions: A single uniform scale factor applies in both dimensions, and the scale factor applies to the number of points — not the intervals of the data.

/W={*xWave*, *yWave*} Provides the scaling waves for XYWaves interpolation. Both waves must be monotonic and must have one more point than the corresponding dimension in *srcWave*. The waves contain values corresponding to the edges of data points in *srcWave*, so that the X value at the first data point is equal to `(xWave[0]+xWave[1])/2`.

Flags for Warp

/dgrx=*wave* Sets the wave containing the destination grid X data.

/dgry=*wave* Sets the wave containing the destination grid Y data.

/sgrx=*wave* Sets the wave containing the source grid X data.

/sgry=*wave* Sets the wave containing the source grid Y data.

/WM=*im*	Sets the interpolation method for warping an image.	
	im=1:	Fast selection of original data values.
	im=2:	Linear interpolation.
	im=3:	Smoothing interpolation (slow)

Details

When computing Bilinear or Spline interpolation *srcWave* can be a 2D or a 3D wave. When *srcWave* is a 3D wave the interpolation is computed on a layer by layer basis and the result is stored in a corresponding 3D wave. When the interpolation method is Kriging or Voronoi, *srcWave* is a 2D triplet wave (3-column wave) where each row specifies the X, Y, Z values of a datum. *srcWave* can be of any real data type. Results are stored in the wave M_InterpolatedImage. If *srcWave* is double precision so is M_InterpolatedImage; otherwise M_InterpolatedImage is a single precision wave.

Voronoi Interpolation Example

```
Function DemoVoronoiInterpolation()
    Make/O/N=(100,3) sampleTriplet
    sampleTriplet[][0]=enoise(5)
    sampleTriplet[][1]=enoise(5)
    sampleTriplet[][2]=sqrt(sampleTriplet[p][0]^2+sampleTriplet[p][1]^2)

    // Interpolate the data to a rectangular grid of 50x50 pixels
    ImageInterpolate/RESL={50,50}/DEST=firstImage voronoi, sampleTriplet

    // Triangulate the XY locations and save the triangulation wave
    ImageInterpolate/STW voronoi, sampleTriplet

    // Use the previous triangulation on the Z column of the sample
    MatrixOp/O zData=col(sampleTriplet,2)
    Wave W_TriangulationData
    ImageInterpolate/PTW=W_TriangulationData/RESL={50,50}/DEST=secondImage voronoi, zData
End
```

PTW Flag Example

```
Function DemoPTW()
    // Create some random data
    Make/O/N=(100,3) eee = enoise(5)

    // Compute triangulation wave
    ImageInterpolate/RESL={1,1}/STW voronoi, eee
    Wave W_TriangulationData

    // Copy the Z-column to a 1D wave
    MatrixOp/O e3 = col(eee,2)

    // Use the previous triangulation with 1D wave
    ImageInterpolate/PTW=W_TriangulationData /RESL={100,100} voronoi, e3
    Wave M_InterpolatedImage
    Duplicate/O M_InterpolatedImage, oneD

    // Direct computation for comparison
    ImageInterpolate/RESL={100,100} voronoi, eee

    // Display
    NewImage M_InterpolatedImage
    NewImage oneD
End
```

See Also

The **interp**, **Interp3DPath**, **ImageRegistration**, and **Loess** operations. The **ContourZ** function. For examples see **Interpolation and Sampling** on page III-312.

References

Unser, M., A. Aldroubi, and M. Eden, B-Spline Signal Processing: Part I-Theory, *IEEE Transactions on Signal Processing*, *41*, 821-832, 1993.

Douglas B. Smythe, “A Two-Pass Mesh Warping Algorithm for Object Transformation and Image Interpolation” ILM Technical Memo #1030, Computer Graphics Department, Lucasfilm Ltd. 1990.

ImageLineProfile

`ImageLineProfile [`*`flags`*`] xWave=`*`xwave`*`, yWave=`*`ywave`*`, srcWave=`*`srcWave`*` [, width=`*`value`*`, widthWave=`*`wWave`*`]`

The ImageLineProfile operation provides sampling of a source image along an arbitrary path specified by the two waves: *xWave* and *yWave*. The arbitrary path is made of line segments between every two consecutive vertices of *xWave* and *yWave*. In each segment the profile is calculated at a number of points (profile points) equivalent to the sampling density of the original image (unless the /V flag is used). Both *xWave* and *yWave* should have the same scaling as *srcWave*. If *srcWave* does not have the same scaling in both dimensions you should remove the scaling to compute an accurate profile.

At each profile point, the profile value is calculated by averaging samples along the normal to the profile line segment. The number of samples in the average is determined by the keyword width. The operation actually averages the interpolated values at N equidistant points on the normal to profile line segment, with `N=2(width+0.5)`. Samples outside the domain of the source image do not contribute to the profile value.

The profile values are stored in the wave W_ImageLineProfile. The actual locations of the profile points are stored in the waves W_LineProfileX and W_LineProfileY. When the averaging width is greater than zero, the operation can also calculate at each profile point the standard deviation of the values sampled for that point (see /S flag). The results are then stored in the wave W_LineProfileStdv. When using this operation on 3D RGB images, the profile values are stored in the 3 column waves M_ImageLineProfile and M_LineProfileStdv respectively.

Parameters

srcWave=*srcWave*	Specifies the image for which the line profile is evaluated. The image may be a 2D wave of any type or a 3D wave or RGB data.
xWave=*xwave*	Specifies the wave containing the x coordinate of the line segments along the path.
yWave=*ywave*	Specifies the wave containing the y coordinate of the line segments along the path.
width=*value*	Specifies the width (diameter) in pixels (need not be an integer value) in a direction perpendicular to the path over which the data is interpolated and averaged for each path point. If you do not specify width or use width=0, only the interpolated value at the path point is used.
widthWave=*wWave*	Specifies the width of the profile (see definition above) on a segment by segment basis. *wWave* should be a 1D wave that has the same number of entries as xWave and yWave. If you provide a widthWave any value assigned with the width keyword is ignored. All values in the wave must be positive and finite.

Flags

/P=*plane*	Specifies which plane (layer) of a 3D wave is to be profiled. By default *plane* =-1 and the profiles are of either the single layer of a 2D wave or all three layers of a 3D RGB wave. Use *plane* =-2 if you want to profile all layers of a 3D wave.
/S	Calculates standard deviations for each profile point.
/SC	Saves W_LineProfileX and W_LineProfileY using the X and Y scaling of *srcWave*.
/V	Calculate profile points only at the vertices of xWave and yWave.

Examples

```
Make/N=(50, 50) sampleData
sampleData = sin((x-25) / 10) * cos((y-25) / 10)
NewImage sampleData
Make/n=2 xTrace={0,50} ,yTrace={20,20}
ImageLineProfile srcWave=sampleData, xWave=xTrace, yWave=yTrace
AppendtoGraph/T yTrace vs xTrace
Display W_ImageLineProfile
```

See Also

For additional examples see **ImageLineProfile Operation** on page III-325.

ImageLoad

`ImageLoad [`*`flags`*`] [`*`fileNameStr`*`]`

The ImageLoad operation loads an image file into an Igor wave. It can load PNG, JPEG, BMP, TIFF, and Sun Raster Files.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where pathName is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

If you want to force a dialog to select the file, omit the *fileNameStr* parameter or pass "" for it.

Flags

/AINF	Loads all of the image files in a disk folder into the current data folder. For example, if you have created an Igor symbolic path named ImagePath that points to a folder containing image files, you can execute: `ImageLoad/P=ImagePath/T=TIFF/AINF` When using /AINF, you must include /T to specify the type of image file to be loaded.
/BIGT=*mode*	When mode is 1, ImageLoad uses the LibTIFF library to load TIFF files. This library supports the traditional TIFF file format and the Big TIFF file format, which supports file sizes greater than 4 GB. When mode is 0, ImageLoad uses Igor's internal TIFF code to load image data. This internal code does not support Big TIFF and is limited to file sizes less than 2 GB. If you omit /BIGT, ImageLoad first attempts to load the file using LibTIFF. If an error occurs, it automatically attempts to load the file using Igor's internal TIFF code. The /SCNL, /STRP and /TILE flags require using LibTIFF. If you use any of these flags, /BIGT=1 is automatically in effect. The /RAT and /RTIO flags require using Igor's internal TIFF code. If you use these flags, /BIGT=0 is automatically in effect. /BIGT=1 supports 8, 16, 32, and 64 bits per color component. /BIGT=0 supports 8, 16, and 32 bits per color component.
/C=*count*	Specifies the number of images to load from a TIFF stack containing multiple images. The images are stored in individual waves if /LR3D is omitted or in a single 3D wave if /LR3D is present. By default, it loads only a single image (i.e., /C=1). Use /C=-1 to load all images. Images must be either 8 bits, 16 bits, or 32 bits/pixel for this option. To load a subset of the images in a TIFF stack, use /S to specify the starting image. If you specify a *count* that exceeds the number of images in the file, ImageLoad loads all images beginning with the first image or the image specified by /S.
/G	Displays the loaded image in a new image plot window.
/LR3D	Specifies that the images in a TIFF stack are to be loaded into a 3D wave rather than into multiple 2D waves. This option works with grayscale images only, not with full color (e.g., RGB). To load a subset of the images into the 3D wave, also use /S and /C.

/N=*baseName*

Stores the waves using *baseName* as the wave name. Only when *baseName* conflicts with an existing wave name will a numeric suffix be appended to the new wave names.

If you omit /N, ImageLoad uses the name of the file as the base name.

/O

Overwrites an existing wave with the same name.

If you omit /O and there is an existing wave with the same name, a numeric suffix is appended to the image name to create a unique name.

/P=*pathName*

Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.

/Q

Quiet mode. Suppresses printing a description of the loaded data to the history area.

/RAT

Read All Tags reads all of the tags in a TIFF file into one or more waves.

/RAT creates a data folder named "Tag*n*" with a numeric suffix, *n*, starting from zero for each loaded image. When reading multiple images from a stack TIFF file, /RAT creates a corresponding number of data folders.

Each data folder contains a text wave named T_Tags consisting of 5 columns. The first row contains the offset of the current Image File Directory (IFD) from the start of the file. The remaining rows describe the individual TIFF Tags as they appear in the IFD.

The first column contains the tag number, the second contains the tag description, the third contains the tag type, the fourth contains the tag length, and the fifth contains either the value of the tag or a statement identifying the name of the wave in which the data was stored. For example, a simple tag that contains a single value has the form:

Num	Desc	Type	Length	Value
256	IMAGEWIDTH	4	1	2560

A tag that contains more data, such as an array of values has the form:

Num	Desc	Type	Length	Value
273	STRIPOFFSETS	4	-120	tifTag273

Here the Length field is negative (-1*realLength) and the Value field contains the name of the wave tifTag273 which contains the array of strip offsets.

When the Value field consists of ASCII characters it is stored in the T_Tags wave itself. All other types are stored in a wave in the same Tag data folder.

Private tags are usually designated by negative tag numbers. If their data type is anything other than ASCII, they are saved in separate waves.

/RTIO

Reads tag information only from a TIFF file. /RTIO is similar to /RAT but it loads tag information only without loading any images.

If you are loading a stack of images you can use the /C and /S flags to obtain tags from a specific range of images.

/S=*start*

Specifies the first image to load from a TIFF stack containing multiple images.

start is zero-based and defaults to 0.

Use /C to specify the number of images to load.

/SCNL=*num*

Reads the specified scanline from a TIFF file using LibTiff.

Added in Igor Pro 7.00.

/STRP=*num*

Reads the specified strip from a TIFF file using LibTiff.

Added in Igor Pro 7.00.

/T=*type*	Identifies what kind of image file to load. *type* is one of the following image file formats:

type	Loads this Image Format
any	Any graphic file type
bmp	Windows bitmap file
jpeg	JPEG file
png	PNG file
rpng	Raw PNG file (see **Details**)
sunraster	Sun Raster file
tiff	TIFF file (see also **Loading TIFF Files**).

If you omit /T or specifiy /T=any, Igor makes a guess based on the file name extension. ImageLoad reports an error if it is unable to determine the image file type.

/T=any allows the user to choose any file, regardless of its file name extension, if ImageLoad displays an Open File dialog.

When loading TIFF, we recommend that you use /T=tiff. See **Loading TIFF Files** below for details.

/TILE=*num*	Reads the specified tile from a TIFF file using LibTiff. Added in Igor Pro 7.00.
/Z	No error reporting.

Details

The name of the wave created by ImageLoad is based on the file name or on *baseName* if you provide the /N=*baseName* flag. In either case, if and only if there is a name conflict, ImageLoad appends a number to create a unique wave name.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like “hd:Folder1:” or “C:\\Folder1\\”. See **Symbolic Paths** on page II-21 for details.

Output Variables

ImageLoad sets the following variables:

`V_flag`	Set to 1 if the image was successfully loaded or to 0 otherwise.
`S_fileName`	Set to the name of the file that was loaded.
`V_numImages`	Set to the number of images loaded. Applies to TIFF files only.
`S_path`	Set to the file system path to the folder containing the file.
`S_waveNames`	Set to a semicolon-separated list of the names of loaded waves.

S_path uses Macintosh path syntax (e.g., “hd:FolderA:FolderB:”), even on Windows. It includes a trailing colon.

Loading PNG Files

If you use /T=rpng (“raw PNG”) or if you omit /T and the file as a .png extension, ImageLoad interprets the PNG file as raw data.

We recommend that you use /T=rpng and use /T=png only if /T=rpng does not produce the desired results.

/T=rpng creates an 8-bit or 16-bit unsigned integer wave with 1 to 4 layers.

PNG images with physical units produce waves with X and Y units of meters.

If a PNG image has a color table, ImageLoad creates two waves: a main image wave with one layer and a color table wave of the same name but with an “_pal” suffix. If the name is too long it creates a wave named PNG_pal instead.

Loading TIFF Files

ImageLoad supports 1-bit, 8-bit, 16-bit, 24-bit, and 32-bit TIFF files as well as floating point TIFFs.

1-bit/pixel images are loaded into a unsigned byte waves.

8-bit/pixel images are loaded into a unsigned byte waves.

16-bit/pixel images are loaded into unsigned 16-bit waves.

24-bit/pixel images and 32-bit/pixel images loaded into 3D RGB and RGBA waves respectively.

Loading a TIFF File With a Color Table

If your TIFF file includes a color table, ImageLoad /T=tiff loads the data into a 2D wave and loads the color table into a separate color table wave which can be used when creating an image plot.

If you want to load the TIFF file into a 3D RGB wave, use /T=tiff to load it into a 2D wave plus a color table and then use **ImageTransform** cmap2RGB to create the 3D RGB wave.

Loading TIFF Stacks

A TIFF stack is a TIFF file that contains multiple images. When loading a stack, you can:

- Load all images
- Load a range of images specified by /S (starting image) and /C (image count)

You can also load the images into:

- Separate 2D waves by omitting the /LR3D flag
- A single 3D wave by using the /LR3D flag

When you use /LR3D, ImageLoad stores each image from the TIFF file in a layer of the 3D output wave. This option works with grayscale images only, not with full color (e.g., RGB).

EXIF Metadata

Some applications embed metadata (information about the image) in EXIF format. In both JPEG and TIFF files, the metadata is stored using TIFF tags. To read the metadata, use the /RAT flag, even if you are loading a JPEG file.

Examples

```
// Load all images from a TIFF stack into separate 2D waves
ImageLoad /C=-1 /T=TIFF

// Load a single image from a TIFF stack into a 2D wave
ImageLoad/S=10/C=1/T=TIFF     // Load image 10 (zero based)

// Load all images from a TIFF stack into a single 3D wave
ImageLoad/LR3D/S=0/C=-1/T=TIFF

// Read all tags without loading any images
ImageLoad/C=-1/T=TIFF/RTIO

// Get the number of images in a TIFF stack
NewDataFolder/O/S tmp
ImageLoad/C=-1/T=TIFF/RTIO
Print V_numImages
KillDataFolder :
```

See Also

Loading Image Files on page II-138.

The **ImageSave** operation for saving waves as image files.

ImageMorphology

`ImageMorphology [`*`flags`*`]` ***`Method imageMatrix`***

The ImageMorphology operation performs one of several standard image morphology operations on the source *imageMatrix*. Unless the /O flag is specified, the resulting image is saved in the wave M_ImageMorph. The operation applies only to waves of type unsigned byte. All ImageMorphology methods except for watershed use a structure element. The structure element may be one of the built-in elements (see /E flag) or a user specified element.

Erosion, Dilation, Opening, and Closing are the only methods supported for a 3D *imageMatrix*.

Parameters

Method is one of the following names:

BinaryErosion	Erodes the source binary image using a built-in or user specified structure element (see /E and /S flags).
BinaryDilation	Dilates the source binary image using a built-in or user specified structure element (see /E and /S flags).
Closing	Performs the closing operation (dilation followed by erosion). The same structure element is used in both erosion and dilation. Note that this operation is an idempotent, which means that there is no point of executing it more than once.
Dilation	Performs a dilation of the source grayscale image using either a built-in structure element or a user specified structure element. The operation supports only 8-bit gray images.
Erosion	Erodes the source grayscale image using either a built-in structure element or a user specified structure element. The operation supports only 8-bit gray images.
Opening	Performs an opening operation (erosion followed by dilation). The same structure element is used in both erosion and dilation. Note that this operation is an idempotent which means that there is no point of executing it more than once.
TopHat	Calculates the difference between the eroded image and dilated image using the same structure element.
Watershed	Calculates the watershed regions for grayscale or binary image. Use the /N flag to mark all nonwatershed lines as NaNs. The /L flag switches from using 4 neighboring pixels (default) to 8 neighboring pixels.

Flags

/E=*id* Uses a particular built in structure element. The following are the built-in structure element. The following are the built-in structure elements; make sure to use the appropriate id for the dimensionality of *imageMatrix*:

id	Element	Origin	Shape
1	2x2	(0,0)	square (default)
2	1x3	(1,1)	row (in 3x3 square)
3	3x1	(1,1)	column (in 3x3 square)
4	3x3	(1,1)	cross (in 3x3 square)
5	5x5	(2,2)	circle (in 5x5 square)
6	3x3	(1,1)	full 3x3 square
200	2x2x2	(1,1,1)	symmetric cube
202	2x2x2	(1,1,1)	2 voxel column in Y direction
203	2x2x2	(1,1,1)	2 voxel column in X direction
204	2x2x2	(1,1,1)	2 voxel column in Z direction
205	2x2x2	(1,1,1)	XY plane
206	2x2x2	(1,1,1)	YZ plane
207	2x2x2	(1,1,1)	XZ plane
300	3x3x3	(1,1,1)	symmetric cube
301	3x3x3	(1,1,1)	symmetric ball
302	3x3x3	(1,1,1)	3 voxel column in Y direction
303	3x3x3	(1,1,1)	3 voxel column in X direction
304	3x3x3	(1,1,1)	3 voxel column in Z direction
305	3x3x3	(1,1,1)	XY plane
306	3x3x3	(1,1,1)	YZ plane
307	3x3x3	(1,1,1)	XY plane
500	5x5x5	(2,2,2)	symmetric cube
501	5x5x5	(2,2,2)	symmetric ball
700	7x7x7	(3,3,3)	symmetric cube
701	7x7x7	(3,3,3)	symmetric ball

Note that this flag has no effect on watershed calculations.

/I= *iterations* Repeats the operation the specified number of *iterations*.

/L Uses 8-connected neighbors instead of 4.

/N Sets the background level to 64 (= NaN).

/O Overwrites the source wave with the output.

/R=*roiSpec*	Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u). The ROI wave must have the same number of rows and columns as the image wave. The ROI itself is defined by the entries/pixels whose values are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous and can take any arbitrary shape. See **ImageGenerateROIMask** for more information on creating ROI waves. In general, the *roiSpec* has the form {*roiWaveName*, *roiFlag*}, where *roiFlag* can take the following values: *roiFlag*=0: Set pixels outside the ROI to 0. *roiFlag*=1: Set pixels outside the ROI as in original image. *roiFlag*=2: Set pixels outside the ROI to NaN (=64). By default *roiFlag* is set to 1 and it is then possible to use the /R flag using the abbreviated form /R=*roiWave*.
/S= *seWave*	Specifies your own structure element. *seWave* must be of type unsigned byte with pixels that belong to the structure element set to 1 and background pixels set to 0. There are no limitations on the size of the structure element and you can use the /X and /Y flags to specify the origin of your structure element.
/W= *whiteVal*	Sets the white value in the binary image if it is different than 255. The black level is assumed to be zero.
/X= *xOrigin*	Specifies the X-origin of a user-defined structure element starting at 0. If you do not use this flag Igor sets the origin to the center of the specified structure element.
/Y= *yOrigin*	Specifies the Y-origin of a user defined structure element starting at 0. If you do not use this flag Igor sets the origin to the center of the specified structure element.
/Z= *zOrigin*	Specifies the Z-origin of the element for 3D structure elements. If you do not use this flag Igor sets the origin to the center of the specified structure element.

Examples

If you would like to apply a morphological operation to a wave whose data type is not an unsigned byte and you wish to retain the wave's dynamic range, you can use the following approach:

```
Function ScaledErosion(inWave)
    Wave inWave

    WaveStats/Q inWave
    Variable nor=255/(V_max-V_min)
    MatrixOp/O tmp=nor*(inWave-V_min)
    Redimension/B/U tmp
    ImageMorphology/E=5 Erosion tmp
    Wave M_ImageMorph
    MatrixOp/O inWave=(M_ImageMorph/nor)+V_min
    KillWaves/Z tmp,M_ImageMorph
End
```

See Also

The **ImageGenerateROIMask** operation for creating ROIs. For details and usage examples see **Morphological Operations** on page III-321 and **Particle Analysis** on page III-328.

ImageNameList

`ImageNameList(`*`graphNameStr, separatorStr`*`)`

The ImageNameList function returns a string containing a list of image names in the graph window or subwindow identified by *graphNameStr*.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

separatorStr should contain a single character such as "," or ";" to separate the names.

An image name is defined as the name of the 2D wave that defines the image with an optional #ddd suffix that distinguishes between two or more images that have the same wave name. Since the image name has to be parsed, it is quoted if necessary.

Examples

The following command lines create a very unlikely image display. If you did this, you would want to put each image on different axes, and arrange the axes such that they don't overlap. That would greatly complicate the example.

```
Make/O/N=(20,20) jack,'jack # 2';
Display;AppendImage jack
AppendImage/T/R jack
AppendImage 'jack # 2'
AppendImage/T/R 'jack # 2'
Print ImageNameList("",";")
```

prints jack;jack#1;'jack # 2';'jack # 2'#1;

See Also

Another command related to images and waves: **ImageNameToWaveRef**.

For commands referencing other waves in a graph: **TraceNameList**, **WaveRefIndexed**, **XWaveRefFromTrace**, **TraceNameToWaveRef**, **CsrWaveRef**, **CsrXWaveRef**, **ContourNameList**, and **ContourNameToWaveRef**.

ImageNameToWaveRef

`ImageNameToWaveRef(`*`graphNameStr, imageNameStr`*`)`

The ImageNameToWaveRef function returns a wave reference to the 2D wave corresponding to the given image name in the graph window or subwindow named by *graphNameStr*.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

The image is identified by the string in *imageNameStr*, which could be a string determined using **ImageNameList**. Note that the same image name can refer to different waves in different graphs.

See Also

The **ImageNameList** function.

For a discussion of wave references, see **Wave Reference Functions** on page IV-186.

ImageRegistration

`ImageRegistration [`*`flags`*`] [testMask=`*`testMaskWave`*`] [refMask=`*`refMaskWave`*`]`
`testWave=`*`imageWave1`*`, refWave=`*`imageWave2`*

The ImageRegistration operation adjusts the test image wave, testWave, to match the reference image wave, refWave, possibly subject to auxiliary mask waves. The registration may involve offset, rotation, scaling or skewing.

Image data may be in two or three dimensions.

ImageRegistration is designed to find accurate registration for relatively small variation on the order of a few degrees of rotation and a few pixels offset between the reference and test images.

All the input waves are expected to be single precision float (SP) so you may have to redimension your images before using ImageRegistration.

ImageRegistration does not tolerate NaNs or INFs; use the masks if you need to exclude pixels from the registration process.

Parameters

refMask=*refMaskWave*	Specifies an optional ROI wave used to mask refWave. Omit refMask to use all of the refWave. The wave must have the same dimensions as refWave. refMask is a single precision floating point wave with nonzero entries marking the “ON” state. Note that the operation modifies this wave and that you should not use the same wave for both the reference and the test masks.
refWave=*imageWave2*	Specifies the name of the reference image wave used to adjust testWave.
testMask=*testMaskWave*	Specifies an optional ROI wave used to mask testWave. Omit testMask to use all of the testWave. The wave must have the same dimensions as refWave. testMask is a single precision floating point wave with nonzero entries marking the “ON” state. Note that the operation modifies this wave and that you should not use the same wave for both the reference and the test masks.
testWave=*imageWave1*	Specifies the name of the image wave that will be adjusted to match refWave.

testMaskWave and *refMaskWave* are optional ROI waves. The waves must have the same dimensions as testWave and refWave respectively. They must be single precision floating point waves with nonzero entries marking the “ON” state. If you need to include the whole region described by testWave or the whole region described by refWave you can omit the respective mask wave

Flags

/ASTP=*val*	Sets the adaptation step for the Levenberg-Marquardt algorithm. Default value is 4.
/BVAL=*val*	Enables clipping and sets the background values to which masked out voxels of the test data will be set.
/CONV=*val*	Sets the convergence method. *val*=0: Gravity, use if the difference between the images is only in translation. This option is frequently useful as a first step when the test and reference data are too far apart for accurate registration. The result of this registration is then passed to a subsequent ImageRegistration with /CONV=1. *val*=1: Marquardt.
/CSNR=*val*	Determines if the operation calculates the signal to noise ratio (SNR) *val*=0: The SNR is not calculated. *val*=1: The SNR is calculated (default). Skipping the SNR calculation saves time and may be particularly useful when performing the registration on a stack of images.
/FLVL=*val*	Specifies the finest level on which the optimization is to be performed. If this is the same as /PRDL, then only the coarsest registration calculation is done. If /FLVL=1 (default), then the full multiresolution pyramid is processed. You can use this flag to terminate the computation at a specified coarseness level greater than 1.
/GRYM	Optimizes the gray level scaling factor. It is sometimes dangerous to let the program adjust for gray levels because in some situations it might result in a null image.
/GRYR	Renders output using the gray scaling parameter. This is more meaningful if the operation computes the optimal gray scaling (see /GRYM).

/GWDT={*sx*,*sy*,*sz*} Sets the three fields to the half-width of a Gaussian window that is used to smooth the data when computing the default masks. Defaults are {1,1,1}. See /REFM and /TSTM for more details.

/INTR=*val* Sets the interpolation method.

val=0:	Nearest neighbor. Used when registering the center of gravity of the test and reference images.
val=1:	Trilinear.
val=2:	Tricubic (default).

/ISOS Optimizes the isometric scaling. This option is inappropriate if voxels are not cubic.

/ISR Computes the multiresolution pyramid with isotropic size reduction.

If the flag is not specified, the size reduction is in the XY plane only.

/MING=*val* Sets the minimum gain at which the computations will stop. Default value is zero, but you can use a slightly larger value to stop the iterations earlier.

/MSKC=*val* Sets mask combination value. During computation the masks for the test data and the mask for the reference data are also transformed. This flag determines how the two masks are to be combined. The registration criteria are computed for the combination of the two masks.

val=0:	or.
val=1:	nor.
val=2:	and (default).
val=3:	nand.
val=4:	xor.
val=5:	nxor.

/PRDL=*depth* Specifies the depth of the multiresolution pyramid. The finest level is *depth*=1. Each level of the pyramid decreases the resolution by a factor of 2. By default, the pyramid *depth*=4, which corresponds to a resolution reduction by a factor of $2^{(depth\text{-}1)}$=8.

The algorithm starts by computing the first registration on large scale features in the image (deepest level of the pyramid). It then makes small corrections to the registration at each consecutive pyramid level.

For best results, the coarsest representation the data should be between 30 and 60 pixels on a side. For example, for an image that is *H* by *V* pixels, you should choose the depth such that $H/2^{(depth\text{-}1)} \approx 30$.

/PSTK When performing registration of a stack of images, use this flag to apply the registration parameters of the previous layer as the initial guess for the registration of each layer after the first in the 3D stack.

/Q Quiet mode; no messages printed in the history area.

/REFM=*val* Sets the reference mask.

val=0:	To leave blank and then every pixel is taken into account.
val=1:	Value will be set if a valid reference mask is provided.
val=2:	The test mask is computed (default).

When computing the reference mask it is assumed that brighter features are more important. This is done by using a low pass filter on the data (using the parameters in /GWDT) which is then converted into a binary mask. Note that you do not need to specify /REFM=1 if you are providing a reference mask wave. See also /TSTM.

/ROT={*rotX*,*rotY*,*rotZ*}

Determines if optimization will take into account rotation about the X, Y, or Z axes. A value of one allows optimization and zero prevents optimization from affecting the corresponding rotation parameter. Defaults are {0,0,1}, which are the appropriate values for rotating images.

/SKEW={*skewX*,*skewY*,*skewZ*}

Determines if optimization will take into account skewness about the X, Y, or Z axes. A value of one allows optimization and zero prevents optimization from affecting the corresponding skewness parameter. Defaults are {0,0,0}. Note that skewing and rotation or isometric scaling are mutually exclusive operations.

/STCK Use /STCK to perform the registration between a 2D reference image and each of the layers in a 3D image. The number of rows and columns of the refWave must match exactly the number of rows and columns in testWave. The transformation parameters are saved in the wave M_RegParams where each column contains the parameters for the corresponding layer in testWave.

/STRT=*val* Sets the first value of the adaptation parameter in the Levenberg-Marquardt algorithm.

The default value of this parameter is 1.

/TRNS={*transX*,*transY*,*transZ*}

Determines if optimization will take into account translation about the X, Y, or Z axes. A value of one allows optimization and zero prevents optimization from affecting the corresponding translation parameter. Defaults are {1,1,0}, which are appropriate for finding the X and Y translations of an image.

/TSTM=*val* Sets the test mask.

val=0:	To leave blank and then every pixel is taken into account.
val=1:	This value will be set if a valid reference mask is provided.
val=2:	The reference mask is computed. This is the default value.

The test image mask is computed in the same way as the reference image mask (see /REFM) using the same set of smoothing parameters. Note that you do not need to specify /TSTM=1 if you are providing a test mask wave.

/USER=*pWave* Provides a user transformation that will be applied to the input testWave in order to create the trasnsformed image. *pWave* must be a double precision wave which contains the same number of rows as W_RegParams.

Note: If you use a previously created W_RegParams make sure to change its name as it is overwritten by the operation.

If *pWave* has only one column and testWave contains multiple layers, then the same transformation applies to all layers. If *pWave* contains more than one column, then each layer of testWave is processed with corresponding column. If there are more layers than columns the first column is used in place of the missing columns.

/ZMAN Modifies the test and reference data by subtracting their mean values prior to optimization.

Details

ImageRegistration will register images that have sufficiently similar features. It will not work if key features are too different. For example, ImageRegistration can handle two images that are rotated relative to each other by a few degrees, but cannot register images if the relative rotation is as large as 45 degrees. The algorithm is capable of subpixel resolution but it does not handle large variations between the test image and the reference image. If the centers of the two images are too far from each other, you should first try ImageRegistration using /CON=0 to remove the translation offset before proceeding with a finer registration of details.

The algorithm is based on an iterative processing that proceeds from coarse to fine detail. Optimization uses a modified Levenberg-Marquardt algorithm and produces an affine transformation for the relative rotation and translation as well as for isometric scaling and contrast adjustment. The algorithm is most effective with square images where the center of rotation is not far from the center of the image.

When using gravity for convergence, skew parameters can't be evaluated (only translation is supported). Skew and isoscaling are mutually exclusive options. Mask waves are defined to have zero entries for pixels outside the region of interest and nonzero entries otherwise. If a mask is not provided, every pixel is used.

ImageRegistration creates the waves M_RegOut and M_RegMaskOut, which are both single precision waves. In addition, the operation creates the wave W_RegParams which stores 20 double precision registration parameters. M_RegOut contains the transformed (registered) test image and M_RegMaskOut contains the transformed mask (which is not affected by mask combination). ImageRegistration ignores wave scaling; images are compared and registered based on pixel values only.

The results printed in the history include:

dx, dy, dz	translation offsets measured in pixels.
a*ij*	Elements in the skewing transformation matrix.
phi	Rotation angle in degrees about the X-axis. Zero for 2D waves.
tht	Rotation angle in degrees about the Y-axis. Zero for 2D waves.
psi	Rotation angle in degrees about the Z-axis.
det	Absolute value of determinant of the skewing matrix (a*ij*).
err	Mean square error defined as $$\frac{1}{N}\sum\left(x_i - y_i\right)^2,$$ where x_i is the original pixel value, y_i the computed value, and N is the number of pixels.
snr	Signal to noise ratio in dB. It is given by: $$10\log\left(\frac{\sum x_i^2}{\sum\left(x_i - y_i\right)^2}\right).$$

These parameters are stored in the wave W_RegParams (or M_RegParams in the case of registering a stack). Angles are in radians. Dimension labels are used to describe the contents of each row of the output wave. Each column of the wave consists of the following rows (also indicated by dimension labels):

Point	**Contents**	**Point**	**Contents**	**Point**	**Contents**	**Point**	**Content**
0	dx	6	a21	12	gamma	17	origin_x
1	dy	7	a22	13	phi	18	origin_y
2	dz	8	a23	14	theta	19	origin_z
3	a11	9	a31	15	psi	21	MSE
4	a12	10	a32	16	lambda	21	SNR
5	a13	11	a33				

You can view the output waves with dimension labels by executing:

```
Edit W_RegParams.ld
```

See Also

The ImageInterpolate **Warp** operation.

References

The ImageRegistration operation is based on an algorithm described by:

Thévenaz, P., and M. Unser, A Pyramid Approach to Subpixel Registration Based on Intensity, *IEEE Transactions on Image Processing*, *7*, 27-41, 1998.

ImageRemoveBackground

`ImageRemoveBackground /R=`*`roiWave`*` [`*`flags`*`] `*`srcWave`*

The ImageRemoveBackground operation removes a general background level, described by a polynomial of a specified order, from the image in *srcWave*. The result of the operation are stored in the wave M_RemovedBackground.

Flags

/F	Computes only the background surface fit. Will only store the resulting fit in M_RemovedBackground. This will not subtract the fit from the image.
/O	Overwrites the original wave.
/P=*polynomial order*	Specifies the order of the polynomial fit to the background surface. If omitted, the order is assumed to be 1.
/R=*roiWave*	Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/B/U), which has the same number of rows and columns as the image wave. Set the pixels that define the background region to 1. The remaining pixels can be any value other than 1. We recommend using 64 which Igor image processing operations often interpret as "blank" in unsigned byte image waves. The ROI does not have to be contiguous. See **ImageGenerateROIMask** for more information on creating ROI waves.
/W	Specifies that polynomial coefficients are to be saved in the wave W_BackgroundCoeff.

Details

The identification of the background is done via the ROI wave. Set the pixels that define the background region to 1. The remaining pixels can be any value other than 1. We recommend using 64 which Igor image processing operations often interpret as "blank" in unsigned byte image waves.

The operation first performs a polynomial fit to the points designated by the ROI wave using the specified polynomial order. A polynomial of order N corresponds to the function:

$$F_N(x,y)=\sum_{m=0}^{N}\sum_{n=0}^{m}c_{nm}x^{m-n}y^{n}.$$

Using the polynomial fit, a surface corresponding to the polynomial is subtracted from the source wave and the result is saved in M_RemovedBackground, unless the /O flag is used, in which case the original wave is overwritten.

Use the /W flag if you want polynomial coefficients to be saved in the W_BackgroundCoeff wave. Coefficients are stored in the same order as the terms in the sums above.

If you do not specify the polynomial order using the /P flag, the default order is 1, which means that the operation subtracts a plane (fitted to the ROI data) from the source image.

Note, if the image is stored as a wave of unsigned byte, short, or long, you might consider converting it into single precision (using `Redimension/S`) before removing the background. To see why this is important, consider an image containing a region of pixels equal to zero and subtracting a background plane corresponding to a nonconstant value. After subtraction, at least some of the pixels in the zero region should become negative, but because they are stored as unsigned quantities, they appear incorrectly as large values.

Examples

See **Background Removal** on page III-332.

See Also

The **ImageGenerateROIMask** operation for creating ROIs.

ImageRestore

```
ImageRestore [flags] srcWave=wSrc, psfWave=wPSF [, relaxationGamma=h,
   startingImage=wRecon ]
```

The ImageRestore operation performs the Richardson-Lucy iterative image restoration.

Flags

/DEST=*destWave*	Specifies the desired output wave. If /DEST is omitted, the output from the operation is stored in the wave M_Reconstructed in the current data folder.
/ITER=*iterations*	Specifies the number of iterations. The default number of iterations is 100.
/Z	Do not report errors.

Parameters

psfWave=*wPSF*	Specifies a known point spread function. *wPSF* must be a 2D (square NxN) wave of the same numeric type as *wSRC*. N must be an odd number greater than 1.
relaxationGamma=*h*	Specifies positive power gamma of in the relaxation mapping (see Details).
startingImage=*wRecon*	Use this keyword to specify a starting image that could be for example the output from a previous call to this operation. *wRecon* must have the same dimensions as *wSRC* and the same numeric type. You must make sure that *wRecon* is not the user-specified or the default destination wave of the operation.
srcWave=*wSrc*	Specifies the degraded image which must be a 2D single-precision or double-precision real wave.

Details

ImageRestore performs the Richardson-Lucy iteration solution to the deconvolution of an image. The input consists of the degraded image and point spread function as well as the desired number of iterations.

The operation allows you to apply additional iterations by setting the starting image to the restored output wave from a previous call to ImageRestore using the startingImage keyword. If startingImage is omitted, the starting image is created by ImageRestore with each pixel set to the value 1.

In the case of stellar images it may be useful to apply a relaxation step that involves scaling the correction evaluated at each iteration by

$$factor(v) = \sin\left(\frac{\pi}{2}\frac{v - v_{\min}}{v_{\max} - v_{\min}}\right)^{\gamma},$$

where v is pixel value, vmax and vmin are the maximum and minimum level pixels in the image and gamma is the user-specified relaxationGamma.

References

W.H. Richardson, "Bayesian-Based Iterative Method of Image Restoration". *JOSA 62, 1*: 55-59, 1972.

L.B. Lucy, "An iterative technique for the rectification of observed distributions", *Astronomical Journal 79, 6*: 745-754, 1974.

ImageRotate

`ImageRotate [`*`flags`*`]` ***`imageMatrix`***

The ImageRotate operation rotates the image clockwise by *angle* (degrees) or counter-clockwise if /W is used.

The resulting image is saved in the wave M_RotatedImage unless the /O flag is specified. The size of the resulting image depends on the angle of rotation.

The portions of the image corresponding to points outside the domain of the original image are set to the default value 64 or the value specified by the /E flag.

You can apply ImageRotate to 2D and 3D waves of any data type.

Flags

/A=*angle*	Specifies the rotation angle measured in degrees in a clockwise direction. For rotations by exactly 90 degrees use /C or /W instead.
/C	Specifies clockwise rotation. Clockwise is the default direction so the rotation will be clockwise whether you use /C or not, so long as you do not use /W.
/E= *val*	Specifies the value for pixels that are outside the domain of the original image. By default pixels are set to 64. If you specify `/E=(NaN)` and your data is of type char, short, or long, the operation sets the external values to 64.
/F	Rotates image by 180 degrees.
/H	Flip the image horizontally.
/O	Overwrites the original image with the rotated image.
/Q	Quiet mode. Without this flag the operation writes warnings in the history area.
/RGBA=[*R*, *G*, *B* [, *A*])	Specifies the RGB or RGBA values of pixels that lie outside the domain occupied by the original image. This flag was added in Igor Pro 7.00.
/S	Uses source image wave scaling to preserve scaling and relative locations of objects in the image for rotation angles that are multiples of 90 degrees.
/V	Flip the image vertically.
/W	Specifies counter-clockwise rotation.
/Z	Ignore errors.

See Also

The **MatrixTranspose** operation.

ImageSave

`ImageSave [`*`flags`*`]` ***`waveName`*** **`[[as]`*`fileNameStr`*`]`**

The ImageSave operation saves the named wave as an image file.

Previously this operation used QuickTime to implement the saving of some file formats. As of Igor Pro 7.00, it no longer uses QuickTime. Consequently, some file formats re no longer supported and some flags have changed.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where pathName is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

If you want to force a dialog to select the file, omit the *fileNameStr* parameter or specify `""` for *fileNameStr* or use the /I flag. The "as" keyword before *fileNameStr* is optional.

In Igor Pro 7.00 or later, if you specify "Clipboard" for *fileNameStr* and the file type is JPEG or PNG, Igor writes the image to the clipboard.

In Igor Pro 7.00 or later, if the file type is PNG or JPEG you can write the image to the picture gallery using the magic path name _PictGallery_. For example

```
ImageSave/T=PNG/P=_PictGallery_ waveName
```

Flags

/ALPH=*mode*	Sets the value used for the ExtraSamples tag (338) in the TIFF file format. This flag was added in Igor Pro 7.00. The tag tells a TIFF file reader how to use the alpha values. Supported modes are 1 for premultiplied and 2 for unassociated alpha. Premultiplied means that the RGB components have already been multiplied by alpha. Unassociated means that the RGB components are raw. By default, alpha is assumed to be premultiplied (*mode*=1).
/D=*depth*	As of Igor7, this flag is deprecated and should not be used in new code. Specifies color depth in bits-per-pixel. Integer values of 1, 8, 16, 24, 32, and 40 are supported. A *depth* of 40 specifies 8-bit grayscale; a *depth* of 8 saves the file as 8-bits/pixel with a color lookup table. If /D is omitted, it acts like /D=8. Saving with a color table may cause some loss of fidelity. See also the discussion of saving TIFF files below.
/DS=*depth*	Saves TIFF data in *depth* bits/sample. Values for *depth* of 8, 16, 32 and 64 are supported. The default is 8. The total number of bits/pixel is: (bits/sample) * (samples/pixel). When using 32 or 64 bits/sample, *srcWave* is saved without normalization.
/F	As of Igor7, this flag is deprecated and should not be used in new code. Saves the wave as single precision float. The data is not normalized. Applies only to TIFF files.
/I	Interactive mode. Forces ImageSave to display a Save File dialog, even if the file and folder location are fully specified.
/IGOR	In Igor6 this flag told Igor to use internal code rather than QuickTime to write TIFF files. As of Igor7, Igor no longer uses QuickTime and always uses internal coe. This flag is still accepted but has no effect.
/O	Overwrites the specified file if it exists. If /O is omitted and the file exists, ImageSave displays a Save File dialog.
/P=*pathName*	Specifies the folder in which to save the image file. *pathName* is the name of an existing symbolic path.
/Q=*quality*	Specifies the quality of the compressed image. *quality* is a number between 0 and 1, with 1 being the highest quality. This is only applicable when saving in formats with lossy compression like JPEG.
/S	Saves as stack. Applies only to 3D or 4D source waves saved as TIFF files.

/T=*fileTypeStr*	Specifies the type of file to be saved.

fileTypeStr	Saved Image Format
`"jpeg"`	JPEG file
`"png"`	PNG file
`"rpng"`	raw PNG file (see **Saving as Raw PNG**)
`"tiff"`	TIFF file

If /T is omitted, the default type is `"tiff"`.

/U	Prevents normalization. This works when saving TIFF only. See **Saving as TIFF** below.
/WT=*tagWave*	Saves TIFF files with file tags as specified by *tagWave*, which is a text wave consisting of a row and 5 columns for each tag (see description of the /RAT flag under **ImageLoad**). It ignores any information in the second column but will write the tags sequentially (only in the first Image File Directory (IFD) if there is more than one image). If the fourth column contains a negative number, there must be a wave, whose name is given in the fifth column of *tagWave*, in the same data folder as *tagWave*. You must make sure that: (1) tag numbers are legal and do not conflict with any existing tags; (2) the data type and data size are consistent with the amount of data saved in external waves.
/Z	No error reporting.

Details

Image files are characterized by the number of color components per pixel and the bit-depth of each components. Igor displays images that consist of 1 components (gray-scale/false color), 3 components (RGB) or 4 components (RGBA) per pixel. You can display waves of all real numeric types as images, but wave data are assumed to be either in the range of [0,255] for 8-bits/components or [0,65535] for all other numeric types. For more information see **Creating an Image Plot** on page II-299.

When you save a numeric wave as image file your options depend on the number of components (layers) of your wave and its number type.

Saving as PNG

You can save an Igor wave as a PNG file with 24 bits per pixel or 32 bits per pixel. The following table describes how the wave's data type corresponds to the PNG pixel format.

Source Wave	PNG Pixel Format
1-layer any type	RGB 24 bits per pixel gray normalized [0,255]
3-layer unsigned byte	RGB 24 bits per pixel no scaling
3-layer unsigned short	RGB 24 bits per pixel data divided by 256
3-layer all other types	RGB 24 bits per pixel data normalized [0,255]
4-layer unsigned byte	RGBA 32 bits per pixel no scaling
4-layer unsigned short	RGBA 32 bits per pixel data divided by 256
4-layer all other types	RGBA 32 bits per pixel data normalized [0,255]

Gray means that the three components of each pixel have the identical value. Data normalization consists of finding the global (over all layers) minimum and maximum values of *srcWave* followed by scaling to the full range, e.g.,

```
minValue = WaveMin(srcWave)
maxValue = WaveMax(srcWave)
outValue[i][j][k]  = 255*(srcWave[i][j][k]-minValue)/(maxValue-minValue)
```

Saving as Raw PNG

The rpng image format requires a wave in 8- or 16-bit unsigned integer format with 1 to 4 layers. Use one layer for grayscale, 3 layers for rgb color, and the extra layer for an alpha channel. If X or Y scaling is not unity, they both must be valid and must be either inches per pixel or meters per pixel. If the units are not inches they are taken to be meters.

Saving as TIFF

ImageSave supports saving uncompressed TIFF images of 8, 16, 32 and 64 bits per color component with 1, 3 or 4 components per pixel.

Depending on the data type of your wave and the depth of the image file being created, ImageSave may save a "normalized" version of your data. The normalized version is scaled to fit in the range of the image file data type. For example, if you save a 16-bit Igor wave containing pixel values from -1000 to +1000 in an 8-bit grayscale TIFF file, ImageSave will map the wave values -1000 and +1000 to the file values 0 and 255 respectively. When saving an image file of 16-bits/component, Igor normalizes to 65535 as the full-scale value.

There is no normalization when you save in floating point (32 or 64 bits/component). Normalization is also not done when saving 8-bit wave data to an 8-bit image file. You can disable normalization with the /U flag.

Saving in floating point can lead to large image files (e.g., 64-bit/component RGBA has 256 bits/pixel) which are not supported by many applications.

Saving 3D or 4D Waves as a Stack of TIFF Images

If your Igor data is a 3D wave other than an RGB wave or a 4D wave, you can save it as a stack of grayscale images without a color map.

Use /S to indicate that you want to save a stack of images rather than a single image from the first layer of the wave.

Use /D=8 to save as 8 bits. Normalization is done except if the wave data is 8 bits.

Use /D=16 to save as 16 bits with normalization.

Use /F to save as single-precision floating point without normalization. Many programs can not read this format.

Use /U to prevent normalization.

Stacked images are normalized on a layer by layer basis. If you want to have uniform scaling and normalization you should convert your wave to the file data type before executing ImageSave.

See Also

The **ImageLoad** operation for loading image files into waves.

ImageSeedFill

ImageSeedFill [*flags*] [*keyword*], seedX=*xLoc*, seedY=*yLoc*, target=*setValue*, srcWave=*srcImage*

The ImageSeedFill operation takes a seed pixel and fills a contiguous region with the target value, storing the result in the wave M_SeedFill. The filled region is defined by all contiguous pixels that include the seed pixel and whose pixel values lie between the specified minimum and maximum values (inclusive). ImageSeedFill works on 2D and 3D waves.

Parameters

keyword is one of the following names:

adaptive=*factor*	Invokes the adaptive algorithm where a pixel or voxel is accepted if its value is between the specified minimum and maximum or its value satisfies: $\lvert val - avg \rvert < factor * stdv.$ Here *val* is the value of the pixel or voxel in question, *avg* is the average value of the pixels or voxels in the neighborhood and *stdv* is the standard deviation of these values. By choosing a small *factor* you can constrain the acceptable values to be very close to the neighborhood average. A large *factor* allows for more deviation assuming that the *stdv* is greater than zero. This requirement means that a connected pixel has to be between the specified minimum and maximum value **and** satisfy the adaptive relationship. In most situations it is best to set wide limits on the minimum and maximum values and allow the adaptive parameter to control the local connectivity.
fillNumber=*num*	Specifies the number, in the range 1 to 26, of voxels in each 3x3x3 cube that belong to the set. If fillNumber is exceeded, the operation fills the remaining members of the cube. If you do not specify this keyword, the operation does not fill the cube. Used only in the fuzzy algorithm.
fuzzyCenter=*fcVal*	Specifies the center value for the fuzzy probability with the fuzzy algorithm (see **Details**). The default value is 0.25. Its standard range is 0 to 0.5, although interesting results might be obtained outside this range.
fuzzyProb=*fpVal*	Specifies a probability threshold that must be met by a voxel to be accepted to the seeded set. The value must be in the range 0 to 1. The default value is 0.75.
fuzzyScale=*fsVal*	Determines if a voxel is to be considered in a second stage using fuzzy probability. *fsVal* must be nonzero in order to invoke the fuzzy algorithm. The scale is used in comparing the value of the voxel to the value of the seed voxel. The scale should normally be in the range 0.5 to 2.0.
fuzzyWidth=*fwVal*	Defines the width of the fuzzy probability distribution with the fuzzy algorithm (see **Details**). In most situations you should not need to specify this parameter. The default value is 1.
min=*minval*	Specifies the minimum value that is accepted in the seeded set. Not needed when using fuzzy algorithm.
max=*maxval*	Specifies the maximum value that is accepted to the seeded set. Not needed when using the fuzzy algorithm.
seedP=row	Specifies the integer row location of the seed pixel or voxel. This avoids roundoff issues when srcWave has wave scaling. You must provide either seedP or seedX with all algorithms. It is sometimes convenient to use this with cursors e.g., `seedP=pcsr(a)`.
seedQ=col	Specifies the integer column location of the seed pixel or voxel. This avoids roundoff difficulties when srcWave has wave scaling. You must provide either seedQ or seedY with all algorithms.
seedR=layer	Specifies the integer layer position of the seed voxel. When srcWave is a 3D wave you must use either seedR or seedZ.
seedX=*xLoc*	Specifies the pixel or voxel index. If srcWave has wave scaling, seedX must be expressed in terms of the scaled coordinate. This keyword or seedP is required with all algorithms.
seedY=*yLoc*	Specifies the pixel or voxel index. If srcWave has wave scaling, seedY must be expressed in terms of the scaled coordinate. This keyword or seedQ is required with all algorithms.

seedZ=*zLoc*	Specifies the voxel index. If srcWave has wave scaling, seedZ must be expressed in terms of the scaled coordinate. You must use this keyword or seedR whenever srcWave is 3D.
srcWave=*srcImage*	Specifies the source image wave.
target=*val*	Sets the value assigned to pixels or voxels that belonging to the seeded set.

Flags

/B=*bValue*	Specifies the value assigned to pixels or voxels that do not belong to the fuzzy set.
/C	Uses 8-connectivity where a pixel can be connected to any one of its neighbors and with which it shares as little as a single boundary point. The default setting is 4-connectivity where pixels can be connected if they are neighbors along a row or a column. This has no effect in 3D, where 26-connectivity is the only option.
/K=*killCount*	Terminates the fill operation after *killCount* elements have been accepted.
/O	Overwrites the source wave with the output (2D only).
/R=*roiWave*	Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u), that has the same number of rows and columns and layers as the image wave. The ROI itself is defined by the entries/pixels whose value are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous. See ImageGenerateROIMask for more information on creating ROI waves.

Details

In two dimensions, the operation takes a seed pixel, optional minimum and maximum pixel values and optional adaptive coefficient. It then fills a contiguous region (in a copy of the source image) with the target value. There are two algorithms for 2D seed fill. In direct seed fill (only min, max, seedX and seedY are specified) the filled region is defined by all contiguous pixels that include the seed pixel and whose pixel values lie between the specified minimum and maximum values (inclusive). In adaptive fill, there is an additional condition for the pixel or voxel to be selected, which requires that the pixel value must be within the standard deviation of the average in the 3x3 (2D) or 3x3x3 (3D) nearest neighbors. If you do not specify the minimum and maximum values then the operation selects only values identical to that of the seed pixel.

In 3D, there are three available algorithms. The direct seed fill algorithm uses the limits specified by the user to fill the seeded domain. In adaptive seed fill the algorithm requires the limits as well as the adaptive parameter. It fills the domain by accepting only voxels that lie within the adaptive factor times the standard deviation of the immediate voxel neighborhood. To invoke the third algorithm you must set fuzzyScale to a nonzero value. The fuzzy seed fill uses two steps to determine if a voxel should be in the filled domain. In the first step the value of the voxel is compared to the seed value using the fuzzy scale. If accepted, it passes to the second stage where a fuzzy probability is calculated based on the number of voxels in the 3x3x3 cell which passed the first step together with the user-specified probability center (fuzzyCenter) and width (fuzzyWidth). If the result is greater than fuzzyProb, the voxel is set to belong to the filled domain.

If the /O flag is not specified, the result is stored in the wave M_SeedFill.

If you specify a background value with the /B flag, the resulting image consists of the background value and the target value in the area corresponding to the seed fill. Although the wave is now bi-level, it retains the same number type as the source image.

ImageSeedFill returns a “bad seed pixel specification” if the seed pixel location derived from the various keywords above satisfies one or more of the following conditions:

- The computed integer pixel/voxel is outside the image.
- The value stored in the computed integer pixel/voxel location does not satisfy the min/max or fuzzy conditions. This is the most common condition when srcWave has wave scaling. To avoid this difficulty you should use the keywords seedP, seedQ, and seedR.

Examples

Using Cursor A position and value to supply parameter inputs for a 2D seedFill (**Warning**: command wrapped over two lines):

```
ImageSeedFill
```

```
   seedP=pcsr(a),seedQ=qcsr(a),min=zcsr(a),max=zcsr(a),target=0,srcwave=image0
```

Using the fuzzy algorithm for a 3D wave (**Warning**: command wrapped over two lines):

```
ImageSeedFill seedX=232,seedY=175,seedZ=42,target=1,fillnumber=10,fuzzycenter=.25,
   fuzzywidth=1,threshold=1,fuzzyprob=0.4,srcWave=ddd
```

See Also

For an additional example see **Seed Fill** on page III-330. To display the result of the operation for 3D waves it is useful to convert the 3D wave M_SeedFill into an array of quads. See ImageTransform **vol2surf**.

ImageSnake

`ImageSnake [`*`flags`*`]` ***`srcWave`***

The ImageSnake operation creates or modifies an active contour/snake in the grayscale image srcWave. The operation iterates to find the “lowest total energy” snake. The energy is defined by a range of optional flags, each corresponding to an individual term in the total energy expression. Iterations terminate by reaching the maximum number of iterations, when the snake does not move between iterations or when the user aborts the operation.

Flags

/ALPH=*alpha*	Sets the coefficient of the energy term arising from the “tightness” of the snake.
/BETA=*beta*	Sets the coefficient of the energy term corresponding to curvature of the snake. A high value for beta makes the snake more rounded.
/DELT=*delta*	Sets the coefficient of the repulsion energy. A high value of *delta* keeps nonconsecutive snake points far from each other.
/EPS=*num*	Sets the maximum number of vertices which are allowed to move in one iteration. If the number of vertices which move during an iteration is smaller than *num* then iterations terminate.
/EXEF=*eta*	Sets the coefficient of the optional external energy component. By default this value is set to zero and there is no external energy contribution to the snakes energy. Note, this component is referred to as “external” because it is completely up to the user to specify both its coefficient and the value associated with each pixel. It should not be confused with what is called external snake energy in the literature, which usually applies to energy proportional to the gradient image (see /GAMM and /GRDI).
/EXEN=*wave*	Specify a wave that contains energy values that will be added to the snakes energy calculation. The wave must have the same dimensions as *srcWave* and must be single precision float. Each pixel value corresponds to user defined energy which will be multiplied by the /EXEF coefficient and added to the sum which the snake minimizes. Note that when /EXEF is set to zero this component is ignored. An external energy wave may be useful, for example, if you want to attract the snake to the picture boundaries. In that case you can set:

```
Duplicate/O srcWave,extWave
Redimension/S extWave
Variable rows=DimSize(srcWave,0)-1
Variable cols=DimSize(srcWave,1)-1
extWave=(p==0 || q==0 || p==rows || q==cols) ? 0:1
```

/GAMM=*gamma*	Sets the coefficient of the energy term corresponding to the gradient. A high value of gamma makes the snake follow lines of high image gradient.
/GRDI=*gWave*	Specify the gradient image. This wave must have the same dimensions as *srcWave* and it must be single precision float. The wave corresponds to the quantity `abs(grad(gauss**`*`srcWave`*`))`, where grad is the gradient operator and ** denotes convolution of the source wave with a Gaussian kernel. It is best to run the operation the first time without specifying this wave. When the operation executes, it creates the wave M_GradImage which can then be used in subsequent executions of this operation. If you want to modify the wave to express some other form of energy that you want the operation to minimize, you should use the /EXEN and /EXEF flags.

/ITER=*iterations*	Sets the maximum number of iterations. Convergence can be achieved if the value specified by /EPS is met. You can also terminate the process earlier by pressing the **User Abort Key Combinations**.
/N=*snakePts*	Specify the number of vertices in the snake or the number of snake points. Note that if you are providing snake waves in /SX and /SY, you do not need to specify this flag. If you do not specify this flag the default value is 40.
/SIG=*sigma*	Sets the size of the Gaussian kernel that is used to convolve the input image when creating the gradient image. Note that you do not need to use this flag if you provide a gradient image. *sigma* is 3 by default. You can use larger odd integers for larger Gaussian kernels which would correspond to a stronger blur.
/STRT={*centerX*, *centerY*, *radius*}	Sets the starting snake to be a circle with the given center and radius. If you use this flag you should also provide the number of snake points using the /N flag.
/STEP=*pixels*	Sets the maximum radius of search. By default the radius of the search is 6 pixels and the search follows a clockwise pattern from radius of 1 pixel to maximum radius specified by this flag. Note: the search radius should be smaller than the dimension of a typical feature in the image. If the radius is larger the snake may encompass more than one object. Larger radius is also less efficient because many of the pixels in that range would result in a snake that crosses itself and hence get rejected in the process.
/SX=*xSnake*	Provide an X-wave for the starting snake. You must also provide an appropriate Y-wave using /SY.
/SY=*ySnake*	Provide a Y-Wave for the starting snake. Must work in combination with /SX.
/UPDM=*mode*	Sets the update mode using any combination of the following:

Value	Update
0	Once when the operation terminates.
1	Once at the end of every iteration.
2	Once after every snake vertex moves.
4	Once for every search position.

/Q	Quiet mode; don't print information in the history.
/Z	Don't report any errors.

Details

A snake is a two-dimensional, usually closed, path drawn over an image. The snake is described by a pair of XY waves consisting of N vertices (sometimes called "snake elements" or "snaksels"). In this implementation it is assumed that the snake is closed so that the last point in the snake is connected to the first. Snakes are used in image segmentation, when you want to automatically select a contiguous portion of the plane based on some criteria. Unlike the classic contours, snakes do not have to follow a constant level. Their structure (or path) is found by associating the concept of energy with every snake configuration and attempting to find the configuration for which the associated energy is a minimum. The search for a minimum energy configuration is usually time consuming and it strongly depends on the format of the energy function and the initial conditions (as defined by the starting snake). The operation computes the energy as a sum of the following 5 terms:

1. The coefficient alpha times a sum of absolute deviations from the average snake segment length. This term tends to distribute the vertices of the snake at even intervals.
2. The coefficient beta times a sum of energies associated with the curvature of the snake at each vertex.
3. The coefficient gamma times a sum of energies computed from the negative magnitude of the gradient of a Gaussian kernel convolved with the image. This term is usually referred to in the literature as the external energy and usually drives the snake to follow the direction of high image gradients.

4. The coefficient delta times a repulsion energy. Repulsion is computed as an inverse square law by adding contributions from all vertices except the two that are immediately connected to each vertex. This energy term is designed to make sure that the snake does not fold itself into “valleys”.
5. The coefficient eta times the sum of values corresponding to the positions of all snake vertices in the wave you provide in /EXEN.

The energy calculation skips all terms for which the coefficient is zero. In addition there is a built-in scan which adds a very high penalty for configurations in which the snake crosses itself.

ImageSkeleton3D

ImageSkeleton3D [/DEST=*destWave* /METH=*method* /Z] *srcWave*

The ImageSkeleton3D operation computes the skeleton of a 3D binary object in srcWave by "thinning". Thinning is a layer-by-layer erosion until only the "skeleton" of an object remains. (See reference below.) It is used in neuroscience to trace neurons.

The ImageSkeleton3D operation was added in Igor Pro 7.

Parameters

srcWave is a 3D unsigned-byte wave where object voxels are set to 1 and the background is set to 0.

Flags

/DEST=*destWave*	Specifies the wave to contain the output of the operation. If the specified wave already exists, it is overwritten. When used in a user-defined function, ImageSkeleton3D creates wave reference for *destWave* if it is a simple name. See **Automatic Creation of WAVE References** on page IV-66 for details. If you omit /DEST the output wave is M_Skeleton in the current data folder.
/METH=*m*	Sets the method used to compute the skeleton. *m*=1: Uses elements of an algorithm by Kalman Palagyi (default). This is currently the only supported method.
/Z	Do not report any errors.

Details

The output is stored in the wave M_Skeleton in the current data folder or in the wave specified by /DEST.

Skeleton voxels are set to the value 1 and background voxels are set to 0.

Example

```
// Create a cube with orthogonal square holes
Make/B/U/N=(30,30,30) ddd=0
ddd[2,27][2,27][2,27]=1
ddd[2,27][10,20][10,20]=0
ddd[10,20][2,27][10,20]=0
ddd[10,20][10,20][2,27]=0
ImageSkeleton3D ddd
```

See Also

Chapter III-11, **Image Processing**, **ImageMorphology**, **ImageSeedFill**

Reference

K. Palagyi, "A 3D fully parallel surface-thinning algorithm", *Theoretical Computer Science* **406** (2008) 119-135.

ImageStats

ImageStats [flags] ***imageWave***

The ImageStats operation calculates wave statistics for specified regions of interest in a real matrix wave. The operation applies to image pixels whose corresponding pixels in the ROI wave are set to zero. It does not print any results in the history area.

Flags

/BEAM Computes the average, minimum, and maximum pixel values in each layer of a 3D wave and 2D ROI. Output is to waves W_ISBeamAvg, W_ISBeamMax, and W_ISBeamMin in the current data folder. Use /RECT to improve efficiency for simple ROI domains. V_ variable results correspond to the last evaluated layer of the 3D wave. Do not use /G, /GS, or /P with this flag. Set /M=1 for maximum efficiency.

/BRXY={*xWave*, *yWave*}

Use this option with a 3D imageWave. It provides a more efficient method for computing average, minimum and maximum values when the set of points of interest is much smaller than the dimensions of an image.

Here *xWave* and *yWave* are 1D waves with the same number of points containing XY integer pixel locations specifying arbitrary pixels for which the statistics are calculated on a plane by plane basis as follows:

$$\text{W\_ISBeamAvg[k]}=\frac{1}{\text{n}}\sum_{i=1}^{n}\text{Im}\,age[xWave[i]][yWave[i]].$$

Pixel locations are zero-based; non-integer entries may produce unpredictable results.

The calculated statistics for each plane are stored in the current data folder in the waves W_ISBeamAvg, W_ISBeamMax and W_ISBeamMin.

Note: This flag is not compatible with any other flag except /BEAM.

/C=*chunk* When imageWave is a 4D wave, /C specifies the chunk for which statistics are calculated. By default *chunk* = 0.

Added in Igor Pro 7.00.

/G={*startP*, *endP*, *startQ*, *endQ*}

Specifies the corners of a rectangular ROI. When this flag is used an ROI wave is not required. This flag requires that $startP \leq endP$ and $startQ \leq endQ$. When the parameters extend beyond the image area, the command will not execute and V_flag will be set to -1. You should therefore verify that V_flag=0 before using the results of this operation.

/GS={*sMinRow*,*sMaxRow*,*sMinCol*,*sMaxCol*}

Specifies a rectangular region of interest in terms of the scaled image coordinates. Each one of the 4 values will be translated to an integer pixel using truncation.

This flag, /G, and an ROI specification are mutually exclusive.

/M=*val* Calculates the average and locates the minimum and the maximum in the ROI when /M=1. This will save you the computation time associated with the higher order statistical moments.

/P=*planeNumber* Restricts the calculation to a particular layer of a 3D wave. By default, *planeNumber*= -1 and only the first layer of the wave is processed.

/R=*roiWave* Specifies a region of interest (ROI) in the image. The ROI is defined by a wave of type unsigned byte (/b/u), which has the same number of rows and columns as the image wave. The ROI itself is defined by the entries/pixels whose value are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous. See **ImageGenerateROIMask** for more information on creating ROI waves.

/RECT={*minRow*, *maxRow*, *minCol*, *maxCol*}

Limits the range of the ROI to a rectangular pixel range with /BEAM.

Details

The image statistics are returned via the following variables:

`V_adev`	Average deviation of pixel values.
`V_avg`	Average of pixel values.
`V_kurt`	Kurtosis of pixel values.
`V_min`	Minimum pixel value.
`V_minColLoc`	Specifies the location of the column in which the minimum pixel value was found or the first eligible column if no single column was found.
`V_minRowLoc`	Specifies the location of the row in which the minimum pixel value was found or the first eligible row if no single minimum was found.
`V_max`	Maximum pixel value.
`V_maxColLoc`	Specifies the location of the column in which the maximum pixel value was found or the first eligible column if no single column was found.
`V_maxRowLoc`	Specifies the location of the row in which the maximum pixel value was found or the first eligible row if no single maximum was found.
`V_npnts`	Number of points in the ROI.
`V_rms`	Root mean squared of pixel values.
`V_sdev`	Standard deviation of pixel values.
`V_skew`	Skewness of pixel values.

Most of these statistical results are similarly defined as for the WaveStats operation. WaveStats will be more convenient to use when calculating statistics for an entire wave.

If *imageWave* is 4D it is often useful to use the reversible conversion

```
Redimension/N=(rows,cols,layers*chunks) ImageWave
```

which allows you to obtain the statistics for each layer and all chunks of the wave. To convert back to 4D, execute:

```
Redimension/N=(rows,cols,layers,chunks) ImageWave
```

See Also

The **ImageGenerateROIMask** and **WaveStats** operations. **ImageStats Operation** on page III-324.

ImageThreshold

`ImageThreshold` `[`***`flags`***`]` ***`imageMatrix`***

The ImageThreshold operation converts a grayscale *imageMatrix* into a binary image. The operation supports all data types. However, the source wave must be a 2D matrix. If *imageMatrix* contains NaNs, the pixels corresponding to NaN values are mapped into the value 64. The values for the On and Off pixels are 255 and 0 respectively. The resulting image is stored in the wave M_ImageThresh.

Flags

/C	Calculates the correlation coefficient between the original image and the image generated by the threshold operation. The correlation value is printed to the history area (unless the /Q flag is specified), it is also stored in the variable V_correlation.
/I	Inverts values written to the image, i.e., sets to zero all pixels above threshold.

/M= *method*	Specifies the thresholding method. The calculated value will be printed to the history area (unless /Q is specified) and stored in the variable V_threshold.

method=0:	Default. In thie case you must use the /T flag to specify a manually-selected threshold.
method=1:	Automatically calculate a threshold value using an iterative method.
method=2:	Image histogram is a simple bimodal distribution.
method=3:	Adaptive thresholding. Evaluates threshold based on the last 8 pixels in each row, using alternating rows. The output wave M_ImageThresh has the same numeric type as the input wave. In particular, when the input is signed byte, the on and off pixel values are 127 and 0 respectively. Note that this method is not supported when used as part of the operation **ImageEdgeDetection**.
method=4:	Fuzzy thresholding using entropy as the measure for "fuzziness".
method=5:	Fuzzy thresholding using a method that minimizes a "fuzziness" measure involving the mean gray level in the object and background.
method=6:	Determines an ideal threshold by histograming the data and representing the image as a set of clusters that is iteratively reduced until there are two clusters left. The threshold value is then set to the highest level of the lower cluster. This method is based on a paper by A.Z. Arifin and A. Asano (see reference below) but modified for handling images with relatively flat histograms. If the image histogram results in less than two clusters, it is impossible to determine a threshold using this method and the threshold value is set to NaN. Added in Igor Pro 7.00.
method=7:	Determines the ideal threshold value by maximizing the total variance between the "object" and the "background". See http://en.wikipedia.org/wiki/Otsu%27s_method.

/N	Sets the background level to 64 (i.e., NaN).
/O	Overwrites the original image with the calculated threshold image. If you do not specify the /O flag, the threshold image is written into the wave M_ImageThresh.
/P=*layer*	When *imageMatrix* is a 3D wave /P selects a specific layer for which to compute the threshold. *layer* is the zero-based layer index. If *layer* is -1, which is the default value, the threshold is computed for all layers of *imageMatrix*. The /P flag is not compatible with /O. The /P flag was added in Igor Pro 7.00.
/Q	Suppresses printing calculated correlation coefficients (/C) and calculated thresholds (/M) to the history area.
/R=*roiSpec*	Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/b/u). The ROI wave must have the same number of rows and columns as the image wave. The ROI itself is defined by the entries/pixels whose values are 0. Pixels outside the ROI can take any nonzero value. The ROI does not have to be contiguous and can take any arbitrary shape. See **ImageGenerateROIMask** for more information on creating ROI waves.

In general, the *roiSpec* has the form {*roiWaveName*, *roiFlag*}, where *roiFlag* can take the following values:

roiFlag=0:	Set pixels outside the ROI to 0.
roiFlag=1:	Set pixels outside the ROI as in original image.
roiFlag=2:	Set pixels outside the ROI to NaN (=64).

By default *roiFlag* is set to 1 and it is then possible to use the /R flag using the abbreviated form /R=*roiWave*.

/T=*thresh*	Sets the threshold value.
/W= *Twave*	Sets the threshold intervals. Each interval is specified by a pair of values in the wave *Twave*. The first element in each pair is the low value and the second element is the high value. Pixel values that lie outside all the specified intervals are set to 0.

References

The automatic thresholding method (/M=1) is described in: T. W. Ridler and S. Calvard, *IEEE Transactions on Systems, Man and Cybernetics*, SMC-8, 630-632, 1978.

The thresholding method used with /M=6 is described in: A.Z. Arifin and A Asano, "Image segmentation by histogram thresholding using hierarchical cluster analysis", *Pattern Recognition Letters* 27 (2006) 1515-1521.

See Also

For usage examples see **Threshold Examples** on page III-309. The **ImageGenerateROIMask** and **ImageEdgeDetection** operations.

ImageTransform

`ImageTransform [flags] Method imageMatrix`

The ImageTransform operation performs one of a number of transformations on *imageMatrix*. The result of using most keywords is a new wave stored in the current data folder. Most flags in this operation are exclusive to the keywords in which they are mentioned.

Parameters

Method selects type of transform. It is one of the following names:

averageImage	Computes an average image for a stack of images contained in 3D *imageMatrix*. The average image is stored in the wave M_AveImage and the standard deviation for each pixel is stored in the wave M_StdvImage in the current data folder. You can use this keyword together with the optional /R flag where a region of interest is defined by zero value points in a ROI wave. The operation sets to NaN the entries in M_AveImage and M_StdvImage that correspond to pixels outside the ROI. *imageMatrix* must have at least three layers.
averageImages	Computes an average image from a sequence of RGB images represented by a 4D wave. The average is computed separately for the R, G and B channels and the resulting RGB image is stored in the wave M_AverageRGBImage in the current data folder. The operation supports all real data types. Added in Igor Pro 7.00.

backProjection	Reconstructs the source image from a projection slice and stores the result in the wave M_BackProjection. The projection slice should either be a wave produced by the projectionSlice keyword of this operation or a wave that would be similarly scaled. The input must be a single or double precision real 2D wave. Row scaling must range symmetrically about zero. For example, if the reconstructed image is expected to have 256 rows then the row scaling of the input should be from -128 to 127. Similarly, the column scaling of the input should range between zero and π. An equivalent implementation as a user function is provided in the demo experiment. You can use this implementation as a starting point if you want to develop filtered back projection. See the **projectionSlice** keyword and the RadonTransformDemo experiment. For algorithm details see the chapter "Reconstruction of cross-sections and the projection-slice theorem of computerized tomography" in Born and Wolf, 1999.
ccsubdivision	Performs a Catmull-Clark recursive generation of B-spline surfaces. There are two valid inputs: triangular meshes or quad meshes. Quad meshes are assumed to be in the form of a 3D wave where the first plane contains the X-values, the second the Y-values and the third the Z-values. Triangle meshes are much more complicated to convert into face-edge-vertex arrays so they are less desirable. They are stored in a three column (triplet) wave where the first column corresponds to the X coordinate, the second to the Y coordinate and the third to the Z coordinate. Each triangle is described by three rows in the wave and common vertices have to be repeated so that each sequential three rows in the triplet wave correspond to a single triangle. You can also associate a scalar value with each vertex and it will be suitably interpolated as new vertices are computed and old ones are shifted. In this case the input source wave contains one more column in the case of a triplet wave or one more plane in the case of a quad wave. The scalar value is added everywhere as an additional dimension to the spatial part of any point calculation. You can specify the number of iterations using the /I flag. By default the operation executes a single iteration. The output is saved in a quad wave M_CCBSplines that consists of 4 columns. Each row corresponds to a Quad where the 3 planes contain the X, Y, and Z components. If you are using an optional scalar in the input, the scalar result is stored in the wave M_CCBScalar.
cmap2rgb	Converts an image and its associated colormap wave (specified using the /C flag) into an RGB image stored in a 3D wave M_RGBOut.
CMYK2RGB	Converts a CMYK image, stored as 4 layer unsigned byte wave, into a 3 layer, standard RGB image wave. The output wave is M_CMYK2RGB that is stored in the current data folder.
compress	Compresses the data in the *imageWave* using a nonlossy algorithm and stores it in the wave W_Compressed in the current data folder. The compressed wave includes all data associated with *imageWave* including its units and wavenote. Use the decompress keyword to recover the original wave. The operation supports all numeric data types. **NOTE**: The compression format for waves greater than 2GB in size was changed in version 6.30B02. If you compressed a wave greater than 2 GB in IGOR64 6.30B01, you will need to decompress it using the same version. You can not decompress it in 6.30B02 or later.
convert2gray	Converts an arbitrary 2D wave into an 8-bit normalized 2D wave. The default output wave name is M_Image2Gray.
decompress	Decompresses a compressed wave. It saves a copy of the decompressed wave under the name W_DeCompressed in the current data folder. **NOTE**: The compression format for waves greater than 2GB in size was changed in version 6.30B02. If you compressed a wave greater than 2 GB in IGOR64 6.30B01, you will need to decompress it using the same version. You can not decompress it in 6.30B02 or later.

distance	Computes the distance transform, also called "distance map", for the input image. Added in Igor Pro 7.00. The distance transform/map is an image where every pixel belonging to the "object" is replaced by the shortest distance of that pixel from the background/boundary. *imageMatrix* must be a 2D wave of type unsigned byte (Make/B/U) where pixels corresponding to the object are set to zero and pixels corresponding to the background are set to 255. Such an image can be obtained, for example, using **ImageThreshold** with the /I flag. The resulting distance map is stored in the wave M_DistanceMap in the current data folder. The operation supports three metrics (Manhattan, Euclidean, Euclidean + scaling) set via the /METR flag.
extractSurface	Extracts values corresponding to a plane that intersects a 3D volume wave (*imageMatrix*). You must specify the extraction parameters using the /X flag. The volume is defined by the wave scaling of the 3D wave. The result, obtained by trilinear interpolation, is stored in the wave M_ExtractedSurface and is of the type NT_FP64. Points in the plane that lie outside the volume are set to NaN.
fht	Performs a Fast Hartley Transform subject to the /T flag. The source wave must be a 2D real matrix with a power of 2 number of rows and columns. Default output is saved in the double-precision wave M_Hartley in the current data folder. If you use the /O flag the result overwrites *imageMatrix* without changing the numeric type. If *imageMatrix* is single-precision float the conversion is straightforward. All other numeric types are scaled. Single- and double-byte types are scaled to the full dynamic range. 32 bit integers are scaled to the range of the equivalent 16 bit types (i.e., unsigned int is scaled to unsigned short range etc.). It does not support wave scaling or NaN entries.
fillImage	Fills a 2D target image wave with data from a 1D image wave (specified using /D). Both waves must be the same data type, and the number of data points in the target wave must match the number of points in the data wave. There are four fill modes that are specified via the /M flag. The operation supports all noncomplex numeric data types.
findLakes	Originally intended to identify lakes in geographical data, this operation creates a mask for a 2D wave for all the contiguous points whose values are close to each other. You can determine the minimum number of pixels within a contiguous region using the /LARA=*minPixels* flag (default is 100). You can determine how close values must be in order to belong to a contiguous region using the /LTOL=*tolerance* flag (default is zero). You can also limit the search region using the /LRCT flag. Use the flag /LTAR=*target* to set the value of the masked regions. By default, the algorithm uses 4-connectivity when looking at adjacent pixels. You can set it to 8-connectivity using the /LCVT flag. The result of the operation is saved in the wave M_LakeFill. It has the same data type as the source wave and contains all the source values outside the masked pixels.
flipCols	Rearrange pixels by exchanging columns symmetrically about a center column or the center of the image (if the number of columns is even). The exchange is performed in place and can be reverted by repeating the operation. When working with 3D waves, use the /P flag to specify the plane that you want to operate on.
flipRows	Rearrange pixels in the image by exchanging rows symmetrically about the middle row or the middle of the image (if the image has an even number of rows). The exchange is performed in place and can be reverted by repeating the operation. When working with 3D waves, use the /P flag to specify the plane that you want to operate on.
flipPlanes	Rearrange data in a 3D wave by exchanging planes symmetrically about the middle plane. The operation is performed in place and can be reverted by repeating the operation.

fuzzyClassify	Segments grayscale and color images using fuzzy logic. Iteration stops when it reaches convergence defined by /TOL or the maximum number of iterations specified by /I. It is a good practice to specify the tolerance and the number of iterations. If the number of classes is small, the operation prints the class values in the history. Use /Q to eliminate printing and increase performance. Use /CLAS to specify the number of classes and optionally use /CLAM to modify the fuzzy probability values. Use /SEG to generate the segmentation image. The classes are stored in the wave W_FuzzyClasses in the current data folder and it will be overwritten if it already exists. When *imageMatrix* is a grayscale image, each class is a single wave entry. When *imageMatrix* is a RGB image, classes are stored consecutively in the wave W_FuzzyClasses. If you request more classes than are present in *imageMatrix*, you will likely find a degeneracy where the space of a data class is spanned by more than one class. It is a good idea to compute the Euclidean distance between every possible pair of classes and eliminate degeneracies when the distance falls below some threshold. Any real data type is allowed but values in the range [0,255] are optimal. You can segment 3D waves of more than 3 layers in which a class will be a vector of dimensionality equal to the number of layers in *imageMatrix*. For examples see Examples/Imaging/fuzzyClassifyDemo.pxp.
getBeam	Extracts a beam from a 3D wave. A “beam” is a 1D array in the Z-direction. If a row is a 1D array in the first dimension and a column is a 1D array in the second dimension then a beam is a 1D array in the third dimension. The number of points in a beam is equal to the number of layers in *imageWave*. Specify the beam with the /BEAM={*row*,*column*} flag. It stores the result in the wave W_Beam in the current data folder. W_Beam has the same numeric type as *imageWave*. Use **setBeam** to set beam values. (See also, **MatrixOp** beam.)
getChunk	Extracts the chunk specified by chunk index /CHIX from *imageMatrix* and stores it in the wave M_Chunk in the current data folder. For example, if *imageMatrix* has the dimensions (10,20,3,10), the resulting M_Chunk has the dimensions (10,20,3). See also **setChunk** and **insertChunk**.
getCol	Extracts a 1D wave, named W_ExtractedCol, from any type of 2D or 3D wave. You specify the column using the /G flag. For a 3D source wave, it will use the first plane unless you specify a plane using the /P flag. *imageMatrix* can be real or complex. (See also putCol keyword, **MatrixOp** col.)
getPlane	Creates a new wave, named M_ImagePlane, that contains data in the plane specified by the /P flag. The new wave is of the same data type as the source wave. You can specify the type of plane using the /PTYP flag. (See also setPlane keyword, **MatrixOp**.)
getRow	Extracts a 1D wave, named W_ExtractedRow, from any type of 2D or 3D wave. You specify the row using the /G flag. For a 3D source wave, it will use the first plane unless you specify a plane using the /P flag. *imageMatrix* can be real or complex. (See also setRow keyword, **MatrixOp** row.)

Hough

Performs the Hough transform of the input wave. The result is saved to a 2D wave M_Hough in which the columns correspond to angle and the rows correspond to the radial domain.

By default the output consists of 180 columns. Use the /F flag to modify the angular resolution.

If the input image has N rows and M columns then the number of rows in the M_Hough is set to 1+sqrt(N^2+M^2).

It is assumed that the input wave has square pixels of unit size and that is binary (/B/U) where the background value is 0.

See also **Hough Transform** on page III-317.

hsl2rgb

Transforms a 3-plane HSL wave into a 3-plane RGB wave. If the source wave for this operation is of any type other than byte or unsigned byte, the HSL values are expected to be between 0 and 65535. For all source wave types the resulting RGB wave is of type unsigned short. The result of the operation is the wave M_HSL2RGB (of type unsigned word), where the RGB values are in the range 0 to 65535.

hslSegment

Creates a binary image of the same dimensions as the source image, in which all pixels that belong to all three of the specified Hue, Saturation, and Lightness ranges are set to 255 and the others to zero. You can specify the HSL ranges using the /H, /S, and /L flags. Each flag takes as an argument either a pair of values or a wave containing pairs of values. You must specify the /H flag but you can omit the /S and /L flags in which case the default values (corresponding to full range 0 to 1) are used. *imageMatrix* is assumed to be an RGB image.

imageToTexture

Transforms a 2D or 3D image wave into a 1D wave of contiguous pixel components. The transformation is useful for creating an OpenGL texture (for Gizmo) or for saving a color image in a format requiring either RGB or RGBA sequences.

The /O flag does not apply to imageToTexture.

Use the /TEXT flag to specify the type of transformation. *imageMatrix* must be an unsigned byte wave. A 1D unsigned byte wave named W_Texture is created in the current data folder.

W_Texture's wave note is set to a semicolon-separated list of keyword -value pairs that can be parsed using **StringByKey** and **NumberByKey**:

Keyword	**Information Following Keyword**
WIDTHPIXELS	**DimSize**(imageMatrix,0) or truncated to nearest power of 2 if /TEXT value is odd
HEIGHTPIXELS	**DimSize**(imageMatrix,1) or truncated to nearest power of 2
LAYERS	**DimSize**(imageMatrix,2)
TEXTUREMODE	val parameter from /TEXT flag
SOURCEWAVE	**GetWavesDataFolder**(imageMatrix, 2)

insertChunk

Inserts a chunk (a 3D wave specified by the /D flag) into *imageMatrix* at chunk index specified by the /CHIX flag. The dimensions of the inserted chunk must match the first three dimensions of *imageMatrix*. The wave must also have the same numeric type. The 4th dimension of *imageMatrix* is incremented by 1 to accommodate the new data. See also **getChunk** and **setChunk**.

insertImage

Inserts the image specified by the flag /INSI into *imageMatrix* starting at the position specified by the flags /INSX and /INSY. If the *imageMatrix* is a 3D wave then it inserts the image in the layer specified by the /P flag. The inserted image and *imageMatrix* must be the same data type. The inserted data is clipped to the boundaries of *imageMatrix*.

insertXplane	Inserts a 2D wave as a new plane perpendicular to the X-axis in a 3D wave. The /P flag specifies the insertion point and the /INSW flag specifies the inserted wave. The 2D wave must be the same numeric data type as the 3D wave and its dimensions must be cols x layers of the 3D wave. For example, if the 3D wave has the dimensions (10x20x30) the 2D wave must be 20x30. If you do not use the /O flag, it stores the result in the wave M_InsertedWave in the current data folder.
insertYplane	Inserts a 2D wave as a new plane perpendicular to the Y-axis in a 3D wave. The /P flag specifies the insertion point and the /INSW flag specifies the inserted wave. The 2D wave must be the same numeric data type as the 3D wave and its dimensions must be rows x layers of the 3D wave. For example, if the 3D wave has the dimensions (10x20x30) the 2D wave must be 10x30. If you do not use the /O flag, it stores the result in the wave M_InsertedWave in the current data folder.
insertZplane	Inserts a 2D wave as a new plane perpendicular to the Z-axis in a 3D wave. The /P flag specifies the insertion point and the /INSW flag specifies the inserted wave. The 2D wave must be the same numeric data type as the 3D wave and its dimensions must be rows x cols of the 3D wave. For example, if the 3D wave has the dimensions (10x20x30) the 2D wave must be 10x20. If you do not use the /O flag, it stores the result in the wave M_InsertedWave in the current data folder. This keyword is included for completeness. You can accomplish the same task using InsertPoints.
Invert	Converts pixel values using the formula `newValue=255-oldValue`. Works on waves of any dimension, but only on waves of type unsigned byte. The result is stored in the wave M_Inverted unless specifying the /O flag.
indexWave	Creates a 1D wave W_IndexedValues in the current data folder containing values from *imageMatrix* that are pointed to by the index wave (see /IWAV). Each row in the index wave corresponds to a single value of *imageMatrix*. If any row does not point to a valid index (within the dimensions of *imageMatrix*), the corresponding value is set to zero and the operation returns an error. Indices are zero based integers; the operation does not support interpolation and ignores wave scaling.
matchPlanes	Finds pixels that match test conditions in all layers of a 3D wave. It creates a 2D unsigned byte output wave, M_matchPlanes, that is set to the values 0 and 255. A value of 255 indicates that the corresponding pixel has satisfied test conditions in all layers of the wave for which conditions were provided. Otherwise the pixel value is 0. Test conditions are entered as a 2D wave using the /D flag. The condition wave must be double precision and it must contain the same number of columns as the number of layers in the 3D source wave. A condition for layer j of the source wave is specified by two rows in column j of the condition wave. The first row entry, say *A*, and the second row entry, say *B*, imply a condition on pixels in layer *j* such that $A \le x < B$. You can have more than one condition for a given layer by adding pairs of rows to the condition wave. For example, if you add in consecutive rows the values *C* and *D*, this implies the test: $\left(A \le x \le B\right) \Vert \left(C \le x \le D\right).$ If you do not have any conditions for some layer, set its corresponding condition column to NaN. Similarly, if you have two conditions for the first layer and one condition for the second layer, pad the bottom of column 1 in the condition wave with NaNs. See Examples for use of this keyword to perform hue/saturation segmentation.
offsetImage	Shifts an image in the XY plane by dx, dy pixels (specified by the /IOFF flag). Pixels outside the shifted image will be set to the specified background value. The operation works on 2D waves or on 3D waves with the optional /P flag. When shifting a 3D wave with no specified plane, it creates a 3D wave with all planes offset by the same amount. The wave M_OffsetImage contains the result in the current data folder. The /O flag is not supported with offsetImage.

padImage	Resizes the source image. When enlarged, values from the last row and column fill in the new area. The /N flag specifies the new image size in terms of the rows and columns change. The /W flag specifies whether data should be wrapped when padding the image. Unless you use the /O flag, the result is stored in the wave M_PaddedImage in the current data folder.
projectionSlice	Computes a projection slice for a parallel fan of rays going through the image at various angles. For every ray in the fan the operation computes a line integral through the image (equivalent to the sum of the line profile along the ray). The operation computes the line integrals for multiple fans defined by the number and position of the rays as well as the angle that they make with the positive X-direction. Use the /PSL flag to specify the projection parameters. The projection slice itself is stored in a 2D wave M_ProjectionSlice where the rows correspond to the rays and the columns correspond to the selected range of angles. The operation does not support wave scaling. If the source wave is 3D the projection slice currently supports slices that are perpendicular to the z-axis and specified by their plane number. See the **backProjection** keyword and the RadonTransformDemo experiment. For algorithm details see the chapter “Reconstruction of cross-sections and the projection-slice theorem of computerized tomography” in Born and Wolf, 1999.
putCol	Sets a column of *imageMatrix* to the values in the wave specified by the /D flag. Use the /G flag to specify column number and the /P flag to specify the plane. Note that if there is a mismatch in the number of entries between the specified waves, the operation uses the smaller number. See also getCol keyword.
putRow	Sets a row of *imageMatrix* to the values in the wave specified by the /D flag. Use the /G flag to specify column number and the /P flag to specify the plane. Note that if there is a mismatch in the number of entries between the specified waves, the operation uses the smaller number. See also getRow keyword.

rgb2cmap	Computes a default color map of 256 colors to represent the input RGB image. The colors are computed by clustering the input pixels in RGB space. The resulting color map is stored in the wave M_ColorIndex in the current data folder. The operation also saves the wave M_IndexImage which contains an index into the colormap that can be used to display the image using the commands:
	
	`NewImage M_IndexImage`
	`ModifyImage M_IndexImage cindex= M_ColorIndex`
	
	To change the default number of colors use the /NCLR flag. When the number of colors are greater than 256, M_IndexImage will be a 16-bit unsigned integer or a 32 bit integer wave depending on the number. rgb2cmap supports input images in the form of 3D waves of type unsigned byte or single precision float. The floating point option may be used to input images in colorspaces that use signed numeric data.

rgb2gray	When the input *imageMatrix* is a 3D RGB wave, rgb2gray produces a 2D wave of type unsigned byte containing the grayscale representation of the input. By default, the operation stores the output in the wave M_RGB2Gray in the current data folder. The RGB values are converted into the luminance Y of the YIQ standard using: `Y = 0.299R + 0.587G + 0.114B` When the input imageMatrix is a 4D wave containing multiple (3 layer) RGB chunks, the conversion produces a 3D wave where each layer corresponds to the grayscale conversion of the corresponding chunk in the input wave. In this case the numeric type of the output is the same as that of the input but the conversion formula is the same. The /O flag is not supported when transforming a 4D RGB wave.
rgb2hsl	Converts an RGB image stored in a 3D wave into another 3D wave in which the three planes correspond to Hue, Saturation and Lightness in the HSL color model. Values of all components are normalized to the range 0 to 255 unless the /U flag is used or if the source wave is not 8-bit, in which case the range is 0 to 65535. The default output wave name is M_RGB2HSL.

rgb2i123	Performs a colorspace conversion of an RGB image into the following quantities:

$$\begin{aligned} I_1 &= \left|R-G\right|D \\ I_2 &= \left|R-B\right|D \quad \text{where } D = \frac{255}{\left|R-G\right|+\left|R-B\right|+\left|G-B\right|}. \\ I_3 &= \left|G-B\right|D, \end{aligned}$$

*I*1, *I*2, and *I*3 are stored in the wave M_I123 (using the same data type as the original RGB wave) in the current data folder. *I*1 is stored in the first layer, *I*2 in the second and *I*3 in the third. This color transformation is said to have useful applications in machine vision.

For more information see: Gevers and Smeulders (1999).

removeXplane — Removes one or more planes perpendicular to the X-axis from a 3D wave. The /P flag specifies the starting position. By default, it removes a single plane but you can remove more planes with the /NP flag. If you do not use the /O flag, it saves the result in the wave M_ReducedWave in the current data folder.

removeYplane — Removes one or more planes perpendicular to the Y-axis from a 3D wave. The /P flag specifies the starting position. By default, it removes a single plane but you can remove more planes with the /NP flag. If you do not use the /O flag, it saves the result in the wave M_ReducedWave in the current data folder.

removeZplane — Removes one or more planes perpendicular to the Z-axis from a 3D wave. The /P flag specifies the starting position. By default, it removes a single plane but you can remove more planes with the /NP flag. If you do not use the /O flag, it saves the result in the wave M_ReducedWave in the current data folder.

rgb2xyz — Converts a 3D RGB image wave into a 3D wave containing the XYZ color space equivalent. The conversion is based on the D65 white point and uses the following transformation:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}.$$

The XYZ values are stored in a wave named "M_RGB2XYZ" unless the /O flag is used, in which case the source image is overwritten and converted into single precision wave (NT_FP32).

roiTo1D — Copies all pixels in an ROI and saves them sequentially in a 1D wave. The ROI is specified by /R. The ROI wave must have the same dimensions as *imageMatrix*. If *imageMatrix* is a 3D wave, the ROI must have as many layers as *imageMatrix*. The wave W_roi_to_1d contains the output in the current data folder, has the same numeric type as *imageMatrix*, and contains the selected pixels in a column-major order.

rotateCols — Rotates rows in place. This operation is analogous to the **Rotate** operation except that it works on images and rotates an integer number of rows. The number of rows is specified by the /G flag.

When *imageMatrix* contains multiple layers you can use the /P flag to specify the layer of the wave that will undergo rotation. By default, if you do not specify the /P flag and if *imageMatrix* consists of three layers (RGB), then all three layers are rotated. Otherwise the operation rotates only the first layer of the wave.

rotateRows	Rotates columns in place. This operation is analogous to the **Rotate** operation except that it works on images and rotates an integer number of columns. The number of columns is specified by the /G flag. When *imageMatrix* contains multiple layers you can use the /P flag to specify the layer of the wave that will undergo rotation. By default, if you do not specify the /P flag and if *imageMatrix* consists of three layers (RGB), then all three layers are rotated. Otherwise the operation rotates only the first layer of the wave.
scalePlanes	Scales each plane of the 3D wave, *imageMatrix*, by a constant taken from the corresponding entry in the 1D wave specified by the /D flag. The result is stored in the wave M_ScaledPlanes unless the /O flag is specified, in which case scaling is done in place. When using /O, first redimension the wave to a different data type to make sure there are no artifacts due to type clipping. If *imageMatrix* is double precision, M_ScaledPlanes is double precision. Otherwise M_ScaledPlanes is single precision. This operation also supports the optional flag. Note that when you display M_ScaledPlanes, which has three planes that originated from scaling byte data, you will have to multiply the wave by 255 to see the image because the RGB format for single and double precision data requires values in the range 0 to 65535.
selectColor	Creates a mask for the image in which pixel values depend on the proximity of the color of the image to a given central color. The central color, the tolerance and a grayscale indicator must be specified using the /E flag. For example, `/E={174,187,75,10,1}` specifies an RGB of (174,187,75), a tolerance level of 10 and a requested grayscale output. RGB values must be provided in a range appropriate for the source image. If the source wave type is not byte or unsigned byte, then the range of the RGB components should be 0 to 65535. The color proximity is calculated in the nonuniform RGB space and the tolerance applies to the maximum component difference from the corresponding component of the central color. The tolerance, just like the central color, should be appropriate to the type of the source wave. The generated mask is stored in the wave M_SelectColor in the current data folder. If a wave by that name exists prior to the execution of this operation, it is overwritten. You can also use the /R flag with this operation to limit the color selection to pixels in the ROI wave whose value is zero.
setBeam	Sets the data of a particular beam in *imageMatrix*. A “beam” is a 1D array in the Z-direction. If a row is a 1D array in the first dimension and a column is a 1D array in the second dimension then a beam is a 1D array in the third dimension. Specify the beam with the /BEAM={*row*,*column*} flag and the 1D beam data wave with the /D flag. The beam data wave must have the same number of elements as the number of layers and same numeric type as *imageMatrix*. Use **getBeam** to extract the beam.
setChunk	Overwrites the data in the wave *imageMatrix* at chunk index specified by /CHIX with the data contained in a 3D wave specified by the /D flag. The assigned data must be contained in a wave that matches the first three dimensions of *imageMatrix* and must have the same number type. See also **getChunk** and **insertChunk**.

setPlane	Sets a plane (given by the /P flag) in the designated image with the data in a wave specified by the /D flag. It is designed as a complement of the **getPlane** keyword to provide an easier (faster) way to create multiplane images. Note that the operation supports setting a plane when the source data is smaller than the destination plane in which case the source data is placed in memory contiguously starting from the corner pixel of the destination plane. If the source data is larger than the destination plane it is clipped to the appropriate rows and columns. If you are setting all planes in the destination wave using algorithmically named source waves you could use the **stackImages** keyword instead. See also **getPlane** keyword.
shading	Calculates relative reflectance of a surface for a light source position defined by the /A flag. The operation estimates the slope of the surface and then computes a relative reflectance defined as the dot product of the direction of the light and the normal to the surface at the point of interest. Reflectivity is scaled using the expression: *outPixel* = *shadingA* * (*sunDirection* · *surfaceNormal*) + *shadingB* By default *shadingA*=1, *shadingB*=0. The result is stored in the wave M_ShadedImage, which has the same data type as the source wave. If the source wave is any integer type, and the value of *shadingA*=1 the operation sets that value to 255. The smallest supported wave size is 4x4 elements. Values along the boundary (1 pixel wide) are arbitrary because there are no derivatives calculable for those pixels, so these pixels are filled with duplicates of the inner rows and columns.
shrinkBox	Shrinks 3D imageMatrix to include only the minimum three dimension rectangular range that contains all the voxels whose values are different from an outer value. The outer value is specified with the /F flag. This feature is useful in situations where ImageSeedFill has set the voxels around an object of interest to some outer value and it is desired to extract the smallest box that contains interesting data. The output is stored in the wave M_shrunkBox. Added in Igor Pro 7.00.
shrinkRect	Shrinks *imageMatrix* to include only the minimum rectangle that contains all the pixels whose value is different from an outer value. The outer value is specified with the /F flag. This is useful in situations where **ImageSeedFill** has set the pixels around the object of interest to some outer value and it is desired to extract the smallest rectangle that contains interesting data. The output is stored in the wave M_Shrunk.
stackImages	Creates a 3D or 4D stack from individual image waves in the current data folder. The waves should be of the form *baseNameN*, where *N* is a numeric suffix specifying the sequence order. *imageMatrix* should be the name of the first wave that you want to add to the stack. You can use the /NP flag to specify the number of waves that you want to add to the stack. The result is a 3D or 4D wave named M_Stack, which overwrites any existing wave of that name in the current data folder. With /K, it kills all waves copied into the stack.
sumAllCols	Creates a wave W_sumCols in which every entry is the sum of the pixels on the corresponding image column. For a 3D wave, unless you specify a plane using the /P flag it will use the first plane by default.
sumAllRows	Creates a wave W_sumRows in which every entry is the sum of the pixels on the corresponding image row. For a 3D wave, unless you specify a plane using the /P flag it will use the first plane by default.

sumCol	Stores in the variable V_value the sum of the elements in the column specified by /G flag and optionally the /P flag.
sumPlane	Stores in the variable V_value the sum of the elements in the plane specified by the /P flag.
sumPlanes	Creates a 2D wave M_SumPlanes which contains the same number of rows and columns as the 3D source wave. Each entry in M_SumPlanes is the sum of the corresponding pixels in all the planes of the source wave. M_SumPlanes is a double precision wave if the source wave is double precision. Otherwise it is a single precision wave.
sumRow	Stores in the variable V_value the sum of the elements of a row specified by /G flag and optionally the /P flag.
swap	Swaps image data following a 2D FFT. The transform swaps diagonal quadrants of the image in one or more planes. This keyword does not support any flags. The swapping is done in place and it overwrites the source wave.
swap3D	Swaps data following a 3D FFT. The transform swaps diagonal quadrants of the data. This keyword does not support any flags. The swapping is done in place and the source wave is overwritten.
transpose4D	Converts a 4D wave into a new 4D wave where the data are reordered by dimensions specified by the /TM4D flag. The results are stored in the wave M_4DTranspose in the current data folder. There is no option to overwrite the input wave so /O has no effect with transpose4D. Added in Igor Pro 7.00.
transposeVol	Transposes a 3D wave. The transposed wave is stored in M_VolumeTranspose. The /O flag does not apply. The operation supports the following 5 transpose modes which are specified using the /G flag:

mode	**Equivalent Command**
1	`M_VolumeTranspose=`*`imageMatrix`*`[p][r][q]`
2	`M_VolumeTranspose=`*`imageMatrix`*`[r][p][q]`
3	`M_VolumeTranspose=`*`imageMatrix`*`[r][q][p]`
4	`M_VolumeTranspose=`*`imageMatrix`*`[q][r][p]`
5	`M_VolumeTranspose=`*`imageMatrix`*`[q][p][r]`

vol2surf	Creates a quad-wave output (appropriate for display in Gizmo) that wraps around 3D “particles”. A particle is defined as a region of nonzero value voxels in a 3D wave. The algorithm effectively computes a box at the resolution of the input wave which completely encloses the data. The output wave M_Boxy is a 2D single precision wave of 12 columns where each row corresponds to one disjoint quad and the columns provide the sequential X, Y, and Z coordinates of the quad vertices.
voronoi	Computes the voronoi tesselation of a convex domain defined by the X, Y positions of the input wave. *imageMatrix* must be a triplet wave where the first column contains the X-values, the second column contains the Y-values and the third column is an arbitrary (zero is recommended) constant. The result of the operation is stored in the two column wave M_VoronoiEdges which contains sequential edges of the Voronoi polygons. Edges are separated from each other by a row of NaNs. The outer most polygons share one or more edges with a large triangle which contains the convex domain.
xProjection	Computes the projection in the X-direction and stores the result in the wave M_xProjection. See **zProjection** for more information.

xyz2rgb	Converts a 3D single precision XYZ color-space data into RGB based on the D65 white point. The transformation used is:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

If you do not specify the /O flag, the results are saved in a single precision 3D wave (NT_FP32) “M_XYZ2RGB”.

Note that not all XYZ values map into positive RGB triplets (consider colors that reside outside the RGB triangle in the XYZ diagram). This operation gives you the following choices: by default, the output wave is a single precision wave that will include possible negative RGB values. If you specify the /U flag, for unsigned short output wave, the operation will set to zero all negative components and scale the remaining ones in the range 0 to 65535.

yProjection	Computes the projection in the Y-direction and stores the result in the wave M_yProjection. See **zProjection** for more information.
zDot	Computes the dot product of a beam in *srcWave* with a 1D zVector wave (specified with the /D flag). This will convert stacked images of spectral scans into RGB or XYZ depending on the scaling in zVector. The *srcWave* and zVector must be the same data type (float or double). The wave M_StackDot contains the result in the current data folder.
zProjection	Computes the projection in the Z-direction and stores the result in the wave M_zProjection. The source wave must be a 3D wave of arbitrary data type. The value of the projection depends on the method specified via the /METH flag.

Flags

/A={*azimuth*, *elevation* [, *shadingA*, *shadingB*]}

Specifies parameters for shade. Position of the light source is given by *azimuth* (measured in degrees counter-clockwise) and *elevation* (measured in degrees above the horizon).

The parameters *shadingA* and *shadingB* are optional. By default their values are 1 and 0, respectively.

/Beam={*row*,*column*}	Designates a beam in a 3D wave; both *row* and *column* are zero based.
/BPJ={*width*,*height*}	Specifies the **backProjection** parameters: *width* and *height* are the width and height of the reconstructed image and should be equal to the size of the original wave.
/C=*CMapWave*	Specifies the colormap wave for **cmap2rgb** keyword. The *CMapWave* is expected to be a 2D wave consisting of three columns corresponding to the RGB entries.
/CHIX=*chunkIndex*	Identifies the chunk index for getting, inserting or setting a chunk of data in a 4D wave. *chunkIndex* ranges from 0 to the number of chunks in *imageMatrix*.
/CLAM=*fuzzy*	Sets the value used to compute the fuzzy probability in **fuzzyClassify**. It must satisfy *fuzzy* > 1 (default is 2).
/CLAS=*num*	Sets the number of requested classes in **fuzzyClassify**. If you don’t know the number of expected classes and *num* is too high, fuzzyClassify will likely produce some degenerate classes.
/D=*waveName*	Specifies a data wave. Check the appropriate keyword documentation for more information about this wave.

/F=*value*	Increases the sampling in the angle domain when used with the Hough keyword. By default *value*=1 and the operation results in single degree increments in the interval 0 to 180, and if *value*=1.5 there will be 180*1.5 rows in the transform. Specifies the outer pixel value surrounding the region of interest when used with **shrinkRect** keyword.
/G=*colNumber*	Specifies either the row or column number used in connection with **getRow** or **getCol** keywords. This flag also specifies the transpose *mode* with the **transposeVol** keyword.
/H={*minHue*, *maxHue*} /H=*hueWave*	Specifies the range of hue values for selecting pixels. The hue values are specified in degrees in the range 0 to 360. Hue intervals that contain the zero point should be specified with the higher value first, e.g., `/H={330,10}`. Use *hueWave* when you have more than one pair of hue values that bracket the pixels that you want to select. See **HSL Segmentation** on page III-327 for an example.
/I=*iterations*	Sets the number of iterations in ccsubdivision and in **fuzzyClassify**.
/INSI=*imageWave*	Specifies the wave, *imageWave*, to be used with the insertImage keyword. *imageWave* is a 2D wave of the same numeric data type as *imageMatrix*.
/INSW=*wave*	Specifies the 2D wave to be inserted into a 3D wave using the keywords: **insertXplane**, **insertYplane**, or **insertZplane**.
/INSX=*xPos*	Specifies the pixel position at which the first row is inserted. Ignores wave scaling.
/INSY=*yPos*	Specifies the pixel position at which the first column is inserted. Ignores wave scaling.
/IOFF={*dx*,*dy*,*bgValue*}	Specifies the amount of positive or negative integer offset with *dx* and *dy* and the new background value, *bgValue*, with the offsetImage keyword.
/IWAV=*wave*	Specifies the wave which provides the indices when used with the keyword **indexWave**. The wave should have as many columns as the dimensions of *imageMatrix* (2, 3, or 4). For example, to specify indices for pixels in an image, the wave should have two columns. The first column corresponds to the row designation and the second to the column designation of the pixel. The wave can be of any number type (other than complex) and entries are assumed to be integer indices; there is no support for interpolation or for wave scaling.
/L={*minLight*, *maxLight*} /L=*lightnessWave*	Specifies the range of lightness for selecting pixels. The lightness values are in the range 0-1. If you do not use the /L flag than the default full range is used. Use *lightnessWave* when you have more than one pair of lightness values corresponding to the pixels that you want to select. For each pair, values should be arranged so that the smaller one is first and the larger is second. There is no restriction on the order of pairs in the wave except that they match the other waves used by the operation.
/LARA=*minPixels*	Specifies the minimum number of pixels required for an area to be masked by the **findLakes** keyword. If you do not specify this flag, the default value used is 100.
/LCVT	Use 8-connectivity instead of 4-connectivity.
/LRCT={*minX*,*minY*,*maxX*,*maxY*}	Sets the rectangular region of interest for the **findLakes** keyword. The operation will not affect the original data outside the specified rectangle. The X and Y values are the scaled values (i.e., using wave scaling).
/LTAR=*target*	Set the target value for the masked region in the **findLakes** keyword.

/LTOL=*tol*	Specifies the tolerance for the **findLakes** keyword. By default the tolerance is zero. The tolerance must be a positive number. The operation uses the tolerance by requiring neighboring pixels to have a value between that of the current pixel V and V+*tol*.
/M=*n*	Specifies the method by which a 2D target image is filled with data from a 1D wave using the **fillImage** keyword.

n =0:	Straight column fill, which you can also accomplish by a redimension operation.
n=1:	Straight row fill.
n=2:	Serpentine column fill. The points from the data wave are sequentially loaded onto the first column and continue from the last to the first point of the second column, and then sequentially through the third column, etc.
n=3:	Serpentine row fill.

/METH=*method* — Determines the values of the projected pixels for **xProjection**, **yProjection**, and **zProjection** keywords.

method=1:	Pixel (i,j) in M_zProjection is assigned the maximum value that (i,j,*) takes among all layers of *imageMatrix* (default).
method=2:	Pixel (i,j) in M_zProjection is assigned the average value that (i,j,*) takes among all layers of *imageMatrix*.
method=3:	Pixel (i,j) in M_zProjection is assigned the minimum value that (i,j,*) takes among all layers of *imageMatrix*.

/METR=*method* — Sets the metric used by the distance transform. Added in Igor Pro 7.00.

method=0:	Manhattan distance. Default.
method=1:	Euclidean distance where the distance is measured between centers of pixels (assumed square).
method=2:	Euclidean distance that also takes into account actual pixel size using the input's wave scaling.

method=0 executes faster than the other methods. *method*=2 can be 2x slower especially if different scaling is applied along the two axes.

/N={*rowsToAdd, colsToAdd*}

Creates an image that is larger or smaller by *rowsToAdd*, *colsToAdd*. The additional pixels are set by duplicating the values in the last row and the last column of the source image.

/NCLR=*M*	Specifies the maximum number of colors to find with the rgb2cmap keyword. *M* must be a positive number; the default value is 256 colors. The result of the operation is saved in the wave M_paddedImage.
/NP=*numPlanes*	Specifies the number of planes to remove from a 3D wave when using the removeXplane, removeYplane, or removeZplane keywords. Specifies the number of waves to be added to the stack with the stackImages keyword.
/O	Overwrites the input wave with the result except in the cases of **Hough** transform and **cmap2rgb**. Does not apply to the transposeVol parameter.
/P=*planeNum*	Specifies the plane on which you want to operate with the **rgb2gray** or **getPlane** keywords. Also used for **getRow** or **getCol** if the source wave is 3D.

/PSL={*xStart,dx,Nx,aStart,da,Na*}

Specifies projection slice parameters. *xStart* is the first offset of the parallel rays measured from the center of the image. *dx* is the directed offset to the next ray in the fan and *Nx* is the number of rays in the fan. *aStart* is the first angle for which the projection is calculated. The angle is measured between the positive X-direction and the direction of the ray. *da* is the offset to the next angle at which the fan of rays is rotated and *Na* is the total number of angles for which the projection is computed.

/PTYP=*num* Specifies the plane to use with the **getPlane** keyword.

num=0:	XY plane.
num=1:	XZ plane.
num=2:	YZ plane.

/Q Quiet flag. When used with the **Hough** transform, it suppresses report to the history of the angle corresponding to the maximum.

/R=*roiWave* Specifies a region of interest (ROI) defined by *roiWave*. For use with the keywords: **averageImage**, **scalePlanes** and **selectColor**.

/S={*minSat, maxSat*}

/S=*saturationWave* Specifies the range of saturation values for selecting pixels. The saturation values are in the range 0 to 1. If you do not use the /S flag, the default value is the full saturation range.

Use *saturationWave* when you have more than one pair of saturation values. If you use a saturation wave you must also use a lightness wave (see /L). *saturationWave* should consist of pairs of values where the first point is the lower saturation value and the second point is the higher saturation value. There is no restriction on the order of pairs within the wave.

/SEG Computes the segmentation image for **fuzzyClassify**. The image is stored in the 2D wave M_FuzzySegments. The value of each pixel is 255**classIndex*/*number of classes*. Here *classIndex* is the index of the class to which the pixel belongs with the highest probability.

/T=*flag* Use one or more of the following flags.

1: Swaps the data so that the DC is at the center of the image.

2: Calculates the power defined as: $P(f) = 0.5 \cdot (H(f)^2 + H(-f)^2)$.

/TEXT=*val* Specifies the type of texture to create with the **imageToTexture** keyword. *val* is a binary flag that can be a combination of the following values.

val	Texture
1	Truncates each dimension to the nearest power of 2, which is required for OpenGL textures.
2	Creates a 1D texture (all other textures are for 2D applications).
4	Creates a single channel texture suitable for alpha or luminance channels.
8	Creates a RGB texture from a 3 (or more) layer data.
16	Creates a RGBA texture. If *imageMatrix* does not have a 4th layer, alpha is set to 255.

/TOL=*tolerance* Sets the tolerance for iteration convergence with **fuzzyClassify**. Convergence is satisfied when the sum of the squared differences of all classes drops below *tolerance*, which must not be negative.

/TM4D=*mode*	Used with transpose4D to specify the format of the output wave. Here *mode* is a 4 digit integer that describes the mapping of the transformation. The digit 1 is used to represent the first dimension, 2 for the second, 4 for the third and 8 for the 4th dimension such that the original wave corresponds to mode=1248. All other modes are obtained by permuting one or more of the four digits and mode must consist of 4 distinct digits. Added in Igor Pro 7.00.
/U	Creates an HSL wave of type unsigned short that contains values between 0 and 65535 when used with **rgb2hsl**.
/W	Pads the image by wrapping the data. If you are adding more rows or more columns than are available in the source wave, the operation cycles through the source data as many times as necessary.
/X={*Nx*,*Ny*,*x1*,*y1*,*z1*,*x2*,*y2*,*z2*,*x3*,*y3*,*z3*}	*Nx* and *Ny* are the rows and columns of the wave M_ExtractedSurface. The remaining parameters specify three 3D points on the extracted plane. The three points must be chosen at the vertices of the plane and entered in clock-wise order without skipping a vertex.
/Z	Ignores errors.

Examples

If you want to insert a 2D (M x N) wave, plane0, into plane number 0 of an (M x N x 3) wave, rgbWave:

```
ImageTransform /P=0/D=plane0 setPlane rgbWave
```

If your source wave is 100 rows by 100 columns and you want to create a montage of this image use:

```
ImageTransform /W/N={200,200} padImage srcWaveName
```

An example of hue and saturation segmentation on an HSL wave.

```
Function hueSatSegment(hslW,lowH,highH,lowS,highS)
    Wave hslW
    Variable lowH,highH,lowS,highS

    Make/D/O/N=(2,3) conditionW
    conditionW={{lowH,highH},{lowS,highS},{NaN,NaN}}
    ImageTransform/D=conditionW matchPlanes hslW
    KillWaves/Z conditionW
End
```

An example of voronoi tesselation.

```
Make/O/N=(33,3) ddd=gnoise(4)
ImageTransform voronoi ddd
Display ddd[][1] vs ddd[][0]
ModifyGraph mode=3,marker=19,msize=1,rgb=(0,0,65535)
Appendtograph M_VoronoiEdges[][1] vs M_VoronoiEdges[][0]
SetAxis left -15,15
SetAxis bottom -5,10
```

See Also

Chapter III-11, **Image Processing**, for many examples. In particular see: **Color Transforms** on page III-305, **Handling Color** on page III-332, and **General Utilities: ImageTransform Operation** on page III-334. The **MatrixOp** operation.

References

Born, Max, and Emil Wolf, *Principles of Optics*, 7th ed., Cambridge University Press, 1999.

Details about the rgb2i123 transform:

Gevers, T., and A.W.M. Smeulders, Color Based Object Recognition, *Pattern Recognition, 32*, 453-464, 1999.

ImageUnwrapPhase

`ImageUnwrapPhase [`*`flags`*`][qualityWave=`*`qWave`*`,] srcwave=`*`waveName`*

The ImageUnwrapPhase operation unwraps the 2D phase in srcWave and stores the result in the wave M_UnwrappedPhase in the current data folder. srcWave must be a real valued wave of single or double precision. Phase is measured in cycles (units of 2π).

Parameters

qualityWave=*qWave*	Specifies a wave, *qWave,* containing numbers that rate the quality of the phase stored in the pixels. *qWave* is 2D wave of the same dimensions as srcWave that can be any real data type and values can have an arbitrary scale. If used with /M=1 the quality values determine the order of phase unwrapping subject to branch cuts, with higher quality unwrapped first. If used with /M=2 the unwrapping is guided by the quality values only. This wave must not contain any NaNs or INFs.
srcwave=*waveName*	Specifies a real-valued SP or DP wave that may contain NaNs or INFs but is otherwise assumed to contain the phase modulo 1.

Flags

/E — Eliminate dipoles. Only applies to Goldstein's method (/M=1). Dipoles are a pair of a positive and negative residues that are side by side. They are eliminated from the unwrapping process by replacing them with a branch cut. The variable V_numResidues contains the number of residues remaining after removal of the dipoles.

/L — Saves the lookup table(LUT) used in the analysis with /M=1. This information may be useful in analyzing your results. The LUT is saved as a 2D unsigned byte wave M_PhaseLUT in the current data folder. Each entry consists of 8-bit fields:

bit=0:	Positive residue.
bit=1:	Negative residue.
bit=2:	Branch cut.
bit=3:	Image boundary exclusion.

Other bits are reserved and subject to change. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/M=*method* — Determines the method for computing the unwrapped phase:

method =0:	Modified Itoh's algorithm, which assumes that there are no residues in the phase. The phase is unwrapped in a contiguous way subject only to the ROI or singularities in the data (e.g., NaNs or INFs). You will get wrong results for the unwrapped phase if you use this method and your data contains residues.
method=1:	Modified Goldstein's algorithm. Creates the variables V_numResidues and V_numRegions. Optional *qWave* can determine order of unwrapping around the branch cuts.
method=2:	Uses a quality map to decide the unwrapping path priority. The quality map is a 2D wave that has the same dimensions as the source wave but could have an arbitrary data type. The phase is unwrapped starting from the largest value in the quality map.

/MAX=*len* — Specifies the maximum length of a branch cut. Only applicable to Goldstein's method (/M=1). By default this is set to the largest of rows or columns.

/Q — Suppresses messages to the history.

/R=*roiWave*	Specifies a region of interest (ROI). The ROI is defined by a wave of type unsigned byte (/B/U) that has the same number of rows and columns as *waveName*. The ROI itself is defined by entries or pixels in the *roiWave* with value of 1. Pixels outside the ROI should be set to zero. The ROI does not have to be contiguous but it is best if you choose a convex ROI in order to make sure that any branch cuts computed by the algorithm lie completely inside the ROI domain.
/REST = *threshold*	Sets the threshold value for evaluating a residue. The residue is evaluated by the equivalent of a closed path integral. If the path integral value exceeds the threshold value, the top-left corner of the quad is taken to be a positive residue. If the path integral is less than -*threshold,* it is a negative residue.

Details

Phase unwrapping in two dimensions is difficult because the result of the operation needs to be such that any path integral over a closed contour will vanish. In many practical situations, certain points in the plane have the property that a path integral around them is not zero. These nonzero points are residues. We use the definition that when a counterclockwise path integral leads to a positive value the residue is called a positive residue.

ImageUnwrapPhase uses the modified Itoh's method by default. Phase is unwrapped with an offset equal to the first element that is allowed by the ROI starting at (0,0) and scanning by rows. If there are no residues or if you unwrap the phase using Itoh's algorithm, then the phase is unwrapped only subject to the optional ROI using a seed-fill type algorithm that unwraps by growing a region outward from the seed pixel. Each time that the region growing is terminated by boundaries (external or due to the ROI), the algorithm returns to the row scanning to find a new starting point.

If there are residues and you choose Goldstein's method, the residues are first mapped into a lookup table (LUT) and branch-cuts are determined between residues and boundaries. It is also possible to remove some residues (dipoles) using the /E flag. Phase is then unwrapped in regions bounded by branch cuts using a seed-fill type algorithm that does not cross branch cuts. With a quality wave, the algorithm follows the same seed-fill approach except that it gives priority to pixels with high quality level. The phase on the branch cuts themselves is subsequently calculated.

The output wave M_UnwrappedPhase has the same wave scaling and dimension units as srcWave. The unwrapped phase is units of cycles; you will have to multiply it by 2π if you need the results in radians.

The operation creates two variables:

V_numResidues	Number of residues encountered(if using /M=1).
V_numRegions	Number of independent phase regions. In Goldstein's method the regions are bounded by branch cuts, but in Itoh's method they depend on the content of the ROI wave.

Examples

To unwrap a complex wave wCmplx:

```
Make/O/N=(DimSize(wCmplx,0),DimSize(wCmplx,1) phaseWave
phaseWave=atan2(imag(wCmplx),real(wCmplx))/2*pi
ImageUnwrapPhase/N=1 srcWave=phaseWave
```

To find the locations of positive residues in the phase:

```
ImageUnwrapPhase/N=1/L srcWave=phaseWave
Duplicate/O M_PhaseLUT ee
ee=M_PhaseLUT&3 ? 1:0
```

To find the branch cuts:

```
Duplicate/O M_PhaseLUT bc
bc=M_PhaseLUT&4 ? 1:0
```

See Also

The **Unwrap** operation and the **mod** function.

References

The following reference is an excellent text containing in-depth theory and detailed explanation of many two-dimensional phase unwrapping algorithms:

Ghiglia, Dennis C., and Mark D. Pritt, *Two Dimensional Phase Unwrapping — Theory, Algorithms and Software*, Wiley, 1998.

ImageWindow

`ImageWindow [/I/O/P=*param*] *method srcWave*`

The ImageWindow operation multiplies the named waves by the specified windowing method.

ImageWindow is useful in preparing an image for FFT analysis by reducing FFT artifacts produced at the image boundaries.

Parameters

srcWave	Two-dimensional wave of any numerical type. See **WindowFunction** for windowing one-dimensional data.
method	Selects the type of windowing filter. See **ImageWindow Methods** on page V-379.

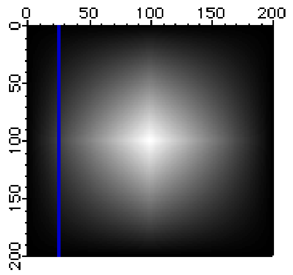
Flags

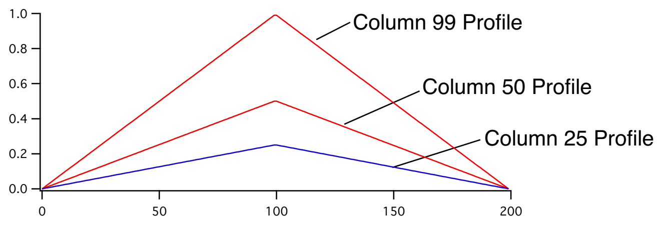
/I	Creates only the output wave containing the windowing filter values that are used to multiply each pixel in *srcWave*. It does not filter the source image.
/O	Overwrites the source image with the output image. If /O is not used then the operation creates the M_WindowedImage wave containing the filtered source image.
/P=*param*	Specifies the design parameter for the Kaiser window.

Details

The 1-dimensional window for each column is multiplied by the value of the corresponding row's window value. In other words, each point is multiplied by the both the row-oriented and column-oriented window value.

This means that all four edges of the image are decreased while the center remains at or near its original value. For example, applying the Bartlett window to an image whose values are all equal results in a trapezoidal pyramid of values:





The default output wave is created with the same data type as the source image. Therefore, if the source image is of type unsigned byte (/b/u) the result of using /I will be identically zero (except possibly for the middle-most pixel). If you keep in mind that you need to convert the source image to a wave type of single or double precision in order to perform the FFT, it is best if you convert your source image (e.g., `Redimension/S *srcImage*`) before using the ImageWindow operation.

The windowed output is in the M_WindowedImage wave unless the source is overwritten using the /O flag.

The necessary normalization value (equals to the average squared window factor) is stored in V_value.

ImageWindow Methods

This section describes the supported keywords for the *method* parameter. In all equations, *L* is the array width and *n* is the pixel number.

Hanning:

$$w(n) = \frac{1}{2}\left[1-\cos\left(\frac{2\pi n}{L-1}\right)\right] \quad 0 \le n \le L-1$$

Hamming:

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{L-1}\right) \quad 0 \le n \le L-1$$

Bartlet: Synonym for Bartlett.

Bartlett:

$$w(n) = \begin{cases} \dfrac{2n}{L-1} & 0 \le n \le \dfrac{L-1}{2} \\ 2-\dfrac{2n}{L-1} & \dfrac{L-1}{2} \le n \le L-1 \end{cases}$$

Blackman:

$$w(n) = 0.42 - 0.5\cos\left(\frac{2\pi n}{L-1}\right) + 0.08\cos\left(\frac{4\pi n}{L-1}\right)$$

$$0 \le n \le L-1$$

Kaiser:

$$\frac{I_0\left(\omega_a\sqrt{\left(\frac{L-1}{2}\right)^2-\left(n-\frac{L-1}{2}\right)^2}\right)}{I_0\left(\omega_a\left(\frac{L-1}{2}\right)\right)} \quad 0 \le n \le L-1$$

where $I_0\{\ldots\}$ is the zeroth-order Bessel function of the first kind and ω_a is the design parameter specified by /P=*param*.

KaiserBessel20: $\alpha = 2.0$

KaiserBessel25: $\alpha = 2.5$

KaiserBessel30: $\alpha = 3.0$

$$w(n) = \frac{I_0\left(\pi\alpha\sqrt{1-\left(\frac{n}{L/2}\right)^2}\right)}{I_0(\pi\alpha)} \quad 0 \le |n| \le \frac{L}{2}$$

$$I_0(x) = \sum_{k=0}^{\infty}\frac{\left(x^2/4\right)^k}{(k!)^2}.$$

Examples

To see what one of the windowing filters looks like:

```
Make/N=(80,80) wShape                   // Make a matrix
ImageWindow/I/O Blackman wShape         // Replace with windowing filter
Display;AppendImage wShape                      // Display windowing filter
Make/N=2 xTrace={0,79},yTrace={39,39}           // Prepare for 1D section
AppendToGraph yTrace vs xTrace
ImageLineProfile srcWave=wShape, xWave=xTrace, yWave=yTrace
Display W_ImageLineProfile              // Display 1D section of filter
```

See Also

The **WindowFunction** operation for information about 1D applications.

Spectral Windowing on page III-244. Chapter III-11, **Image Processing** contains links to and descriptions of other image operations.

See **FFT** operation for other 1D windowing functions for use with FFTs; **DSPPeriodogram** uses the same window functions. See **Correlations** on page III-315.

DPSS

References

For further windowing information, see page 243 of:

Pratt, William K., *Digital Image Processing*, John Wiley, New York, 1991.

IndependentModule

`#pragma IndependentModule = imName`

The IndependentModule pragma designates groups of one or more procedure files that are compiled and linked separately. Once compiled and linked, the code remains in place and is usable even though other procedures may fail to compile. This allows functioning control panels and menus to continue to work regardless of user programming errors.

See Also

Independent Modules on page IV-224, **The IndependentModule Pragma** on page IV-51 and #**pragma**.

IndependentModuleList

`IndependentModuleList(listSepStr)`

The IndependentModuleList function returns a string containing a list of independent module names separated by listSepStr.

Use **StringFromList** to access individual names.

Parameters

listSepStr contains the character, usually ";", to be used to to separate the names in the returned list.

Details

In Igor6, only the first byte of *listSepStr* was used. In Igor7 and later, all bytes are used.

ProcGlobal is not in the returned list, and the order of returned names is not defined.

See Also

Independent Modules on page IV-224.

GetIndependentModuleName, **StringFromList**, **FunctionList**.

IndexedDir

`IndexedDir(pathName, index, flags)`

The IndexedDir function returns a string containing the name of or the full path to the *index*th folder in the folder referenced by *pathName*.

Parameters

pathName is the name of an Igor symbolic path pointing to the parent directory.

index is the index number of the directory (within the parent directory) of interest starting from 0. If *index* is -1, IndexedDir will return the name of *all* of the folders in the parent, separated by semicolons.

flags is a bitwise parameter:

Bit 0:	Set if you want a full path. Cleared if you want just the directory name.

All other bits are reserved and should be cleared.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Details

You create the symbolic path identifying the parent directory using the **NewPath** operation or the New Path dialog (Misc menu).

Prior to Igor Pro 3.1, IndexedDir was an external function and took a string as the first parameter rather than a name. The *pathName* parameter can now be either a name or a string containing a name. Any of the following will work:

```
String str = "IGOR"
Print IndexedDir(IGOR, 0, 0)          // First parameter is a name.
Print IndexedDir($str, 0, 0)          // First parameter is a name.
Print IndexedDir("IGOR", 0, 0)        // First parameter is a string.
Print IndexedDir(str, 0, 0)           // First parameter is a string.
```

The acceptance of a string is for backward compatibility only. New code should be written using a name.

The returned path uses the native conventions of the OS under which Igor is running.

Examples

Example: Recursively Listing Directories and Files

Here is an example for heavy-duty Igor Pro programmers. It is an Igor Pro user-defined function that prints the paths of all of the files and folders in a given folder with or without recursion. You can rework this to do something with each file instead of just printing its path.

To try the function, copy and paste it into the Procedure window. Then execute the example shown in the comments.

```
//  PrintFoldersAndFiles(pathName, extension, recurse, level)
//  Shows how to recursively find all files in a folder and subfolders.
//  pathName is the name of an Igor symbolic path that you created
//  using NewPath or the Misc->New Path menu item.
//  extension is a file name extension like ".txt" or "????" for all files.
//  recurse is 1 to recurse or 0 to list just the top-level folder.
//  level is the recursion level - pass 0 when calling PrintFoldersAndFiles.
//  Example: PrintFoldersAndFiles("Igor", ".ihf", 1, 0)
Function PrintFoldersAndFiles(pathName, extension, recurse, level)
    String pathName      // Name of symbolic path in which to look for folders and files.
    String extension     // File name extension (e.g., ".txt") or "????" for all files.
    Variable recurse     // True to recurse (do it for subfolders too).
    Variable level       // Recursion level. Pass 0 for the top level.

    Variable folderIndex, fileIndex
    String prefix

    // Build a prefix (a number of tabs to indicate the folder level by indentation)
    prefix = ""
    folderIndex = 0
    do
        if (folderIndex >= level)
            break
        endif
        prefix += "\t"       // Indent one more tab
        folderIndex += 1
    while(1)

    // Print folder
    String path
    PathInfo $pathName       // Sets S_path
    path = S_path
    Printf "%s%s\r", prefix, path
    // Print files
    fileIndex = 0
    do
        String fileName
        fileName = IndexedFile($pathName, fileIndex, extension)
        if (strlen(fileName) == 0)
            break
        endif
        Printf "%s\t%s%s\r", prefix, path, fileName
        fileIndex += 1
    while(1)

    if (recurse)            // Do we want to go into subfolder?
        folderIndex = 0
        do
            path = IndexedDir($pathName, folderIndex, 1)
            if (strlen(path) == 0)
                break         // No more folders
            endif

            String subFolderPathName = "tempPrintFoldersPath_" + num2istr(level+1)

            // Now we get the path to the new parent folder
            String subFolderPath
            subFolderPath = path

            NewPath/Q/O $subFolderPathName, subFolderPath
            PrintFoldersAndFiles(subFolderPathName, extension, recurse, level+1)
            KillPath/Z $subFolderPathName

            folderIndex += 1
        while(1)
    endif
End
```

Example: Fast Scan of Directories

Calling IndexedDir for each directory is an O(N^2) problem because to get the nth directory the OS routines underlying IndexedDir need to iterate through directories 0..n-1. This becomes an issue only if you are dealing with hundreds or thousands of directories.

This function illustrates a technique for converting this to an O(N) problem by getting a complete list of paths from IndexedDir in one call and storing them in a text wave. This approach could also be used with IndexedFile.

```
Function ScanDirectories(pathName, printDirNames)
    String pathName            // Name of Igor symbolic path
    Variable printDirNames     // True if you want to print the directory names

    Variable t0 = StopMSTimer(-2)

    String dirList = IndexedDir($pathName, -1, 0)
    Variable numDirs = ItemsInList(dirList)

    // Store directory list in a free text wave.
    // The free wave is automatically killed when the function returns.
    Make/N=(numDirs)/T/FREE dirs = StringFromList(p, dirList)

    String dirName
    Variable i
    for(i=0; i<numDirs; i+=1)
        dirName = dirs[i]
        Print i, dirName
    endfor

    Variable t1 = StopMSTimer(-2)
    Variable elapsed = (t1 - t0) / 1E6
    Printf "Took %g seconds\r", elapsed
End
```

IndexedFile

`IndexedFile(`*`pathName, index, fileTypeOrExtStr`*` [, `*`creatorStr`*`])`

If *index* is greater than or equal to zero, IndexedFile returns a string containing the name of the *index*th file in the folder specified by *pathName* which matches the file type or extension specified by *fileTypeOrExtStr*.

If *index* is -1, IndexedFile returns a semicolon-separated list of all matching files.

IndexedFile returns an empty string (`""`) if there is no such file.

Parameters

pathName is the *name* of an Igor symbolic path. It is *not* a string.

index normally starts from zero. However, if *index* is -1, IndexedFile returns a string containing a semicolon-separated list of the names of all files in the folder associated with the specified symbolic path which match *fileTypeOrExtStr*.

fileTypeOrExtStr is either:

- A string starting with ".", such as ".txt", ".bwav", or ".c". Only files with a matching file name extension are indexed. Set *fileTypeOrExtStr* to "." to index file names that end with "." such as "myFileNameEndsWithThisDot."
- A string containing exactly four ASCII characters, such as "`TEXT`" or "`IGBW`". Only files of the specified Macintosh file type are indexed. However, if *fileTypeOrExtStr* is "????", files of any type are indexed.
- On Windows, Igor considers files with ".txt" extensions to be of type TEXT. It does similar mappings for other extensions. See **File Types and Extensions** on page III-404 for details.

creatorStr is an optional string argument containing four ASCII characters such as "IGR0". Only files with the specified Macintosh creator code are indexed. Set *creatorStr* to "`????`" to index all files (or omit the argument altogether). This argument is ignored on Windows systems.

Order of Files Returned By IndexedFile

The order in which files are returned by IndexedFile is determined by the operating system. Empirically, this is alphabetical order for both Macintosh and Windows. If the order matters to you, you should explicitly sort the file names.

For example, assume you have files named "File1.txt", "File2.txt" and "File10.txt". An alphabetic order gives you: "File1.txt;File10.txt;File2.txt;". Often what you really want is a combination of alphabetic and numeric sorting returning "File1.txt;File2.txt;File10.txt;". Here is a function that does that:

```
Function DemoIndexedFile()
	String pathName = "Igor"// Refers to "Igor Pro 7 Folder"

	// Get a semicolon-separated list of all files in the folder
	String list = IndexedFile($pathName, -1, ".txt")

	// Sort using combined alpha and numeric sort
	list = SortList(list, ";", 16)

	// Process the list
	Variable numItems = ItemsInList(list)
	Variable i
	for(i=0; i<numItems; i+=1)
		String fileName = StringFromList(i, list)
		Print i, fileName
	endfor
End
```

Treatment of Macintosh Dot-Underscore Files

IndexedFile ignores "dot-underscore" files unless *fileTypeOrExtStr* is "????".

A dot-underscore file is a file created by Macintosh when it writes to a non-HFS volume, for example, when it writes to a Windows volume via SMB file sharing. The dot-underscore file stores Macintosh HFS-specific data such as the file's type and creator codes, and the file's resource fork, if it has one.

For example, if a file named "wave0.ibw" is copied via SMB to a Windows volume, Mac OS X creates two files on the Windows volume: "wave0.ibw" and "._wave0.ibw". Mac OS X makes these two files appear as one to Macintosh applications. However, Windows does not do this. As a consequence, when a Windows program sees "._wave0.ibw", it expects it to be a valid .ibw file, but it is not. This causes problems.

By ignoring dot-underscore files, IndexedFile prevents this type of problem. However, if fileTypeOrExtStr is "????", IndexedFile will return dot-underscore files on Windows.

Examples

```
NewPath/O myPath "MyDisk:MyFolder:"
Print IndexedFile(myPath,-1,"TEXT")         // all text-type files
Print IndexedFile(myPath,0,"TEXT")          // only the first text file
Print IndexedFile(myPath,-1,".dat")         // *.dat
Print IndexedFile(myPath,-1,"TEXT","IGR0")  // all Igor text files
Print IndexedFile(myPath,-1,"????")         // all files, all creators
```

See **IndexedDir** for another example using IndexedFile and for a method for speeding up scanning of very large numbers of files.

See Also

The **TextFile** and **IndexedDir** functions.

IndexSort

`IndexSort [ /DIML ]` ***`indexWaveName, sortedWaveName`*** **`[,`** ***`sortedWaveName`*** **`]…`**

The IndexSort operation sorts the values in each *sortedWaveName* wave according to the Y values of *indexWaveName*.

Flags

/DIML	Moves the dimension labels with the values (keeps any row dimension label with the row's value).

Details

indexWaveName can not be complex. *indexWaveName* is presumed to have been the destination of a previous **MakeIndex** operation.

This has the effect of putting the *sortedWaveName* waves in the same order as the wave from which the index values in *indexWaveName* was made.

All of the *sortedWaveName* waves must be of equal length.

See Also

MakeIndex and IndexSort Operations on page III-127.

IndexToScale

`IndexToScale(wave, index, dim)`

The IndexToScale function returns the scaled coordinate value corresponding to wave element index in the specified dimension.

The IndexToScale function was added in Igor Pro 7.00.

Details

The function returns the expression:

DimOffset(*wave*,*dim*) + index*DimDelta(*wave*,*dim*)

index is an integer.

dim is 0 for rows, 1 for columns, 2 for layers or 3 for chunks.

The function returns NaN if *dim* is not a valid dimension or if *index* is greater than the number of elements in the specified dimension.

Examples

```
Make/N=(10,20,30,40) w4D
SetScale/P y 2,3,"", w4D
SetScale/P z 4,5,"", w4D
SetScale/P t 6,7,"", w4D
Print IndexToScale(w4D,1,0)
Print IndexToScale(w4D,1,1)
Print IndexToScale(w4D,1,2)
Print IndexToScale(w4D,1,3)
Print IndexToScale(w4D,1,4)
Print IndexToScale(w4D,-1,0)
Print IndexToScale(w4D,11,0)
```

See Also

ScaleToIndex, **pnt2x**, **DimDelta**, **DimOffset**

Waveform Model of Data on page II-57 for an explanation of wave scaling.

Inf

`Inf`

The Inf function returns the “infinity” value.

InsertPoints

`InsertPoints [/M=`*`dim`*`]` ***`beforePoint, numPoints, waveName`*** **`[,`** ***`waveName`*****`]`**…

The InsertPoints operation inserts *numPoints* points in front of point *beforePoint* in each *waveName*. The new points have the value zero.

Flags

/M=*dim*	Specifies the dimension into which elements are to be inserted. Values are 0 for rows, 1 for columns, 2 for layers, 3 for chunks. If /M is omitted, InsertPoints inserts in the rows dimension.

Details

Trying to insert points into any but the rows of a zero-point wave results in a zero-point wave. You must first make the number of rows nonzero before anything else has an effect.

See Also

Lists of Values on page II-72.

Int

`int` ***`localName`***

In a user-defined function or structure, declares a local 32-bit integer in IGOR32, a local 64-bit integer in IGOR64.

Int is available in Igor Pro 7 and later. See **Integer Expressions** on page IV-36 for details.

See Also

Int64, **UInt64**

Int64

`int64` ***`localName`***

Declares a local 64-bit integer in a user-defined function or structure.

Int64 is available in Igor Pro 7 and later. See **Integer Expressions** on page IV-36 for details.

See Also

Int, **UInt64**

Integrate

Integrate [***type flags***][***flags***] ***yWaveA*** [**/X** = ***xWaveA***][**/D** = ***destWaveA***]
  [, ***yWaveB*** [**/X** = ***xWaveB***][**/D** = ***destWaveB***][, …]]

The Integrate operation calculates the 1D numeric integral of a wave. X values may be supplied by the X-scaling of the source wave or by an optional X wave. Rectangular integration is used by default.

Integrate is multi-dimension-aware in the sense that it computes a 1D integration along the dimension specified by the /DIM flag or along the rows dimension if you omit /DIM.

Complex waves have their real and imaginary components integrated individually.

Flags

/DIM= *d*	Specifies the wave dimension along which to integrate when *yWave* is multi-dimensional.
	d=-1: Treats entire wave as 1D (default).
	d=0: Integrates along rows.
	d=1: Integrates along columns.
	d=2: Integrates along layers.
	d=3: Integrates along rows.
	For example, for a 2D wave, /DIM=0 integrates each row and /DIM=1 integrates each column.
/METH=*m*	Sets the integration method.
	m=0: Rectangular integration (default). Results at a point are stored at the same point (rather than at the next point as for /METH=2). This method keeps the dimension size the same.
	m=1: Trapezoidal integration.
	m=2: Rectangular integration. Results at a point are stored at the next point (rather than at the same point as for /METH=0). This method increases the dimension size by one to provide a place for the last bin.
/P	Forces point scaling.
/T	Trapezoidal integration. Same as /METH=1.

Type Flags *(used only in functions)*

Integrate also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when it needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-67 and **WAVE Reference Type Flags** on page IV-68 for a complete list of type flags and further details.

For example, when the input (and output) waves are complex, the output wave will be complex. To get the Igor compiler to create a complex output wave reference, use the /C type flag with /D=destwave:

```
Make/O/C cInput=cmplx(sin(p/8), cos(p/8))
Make/O/C/N=0 cOutput
Integrate/C cInput /D=cOutput
```

Wave Parameters

Note: ***All*** wave parameters must follow *yWave* in the command. All wave parameter flags and type flags must appear immediately after the operation name (`Integrate`).

/D=*destWave*	Specifies the name of the wave to hold the integrated data. It creates *destWave* if it does not already exist or overwrites it if it exists.
/X=*xWave*	Specifies the name of corresponding X wave. For rectangular integration, the number of points in the X wave must be one greater than the number of elements in the Y wave dimension being integrated.

Details

The computation equation for rectangular integration using /METH=0 is:

$$waveOut[p] = \sum_{i=0}^{p} waveIn[i] \cdot \Delta x.$$

The computation equation for rectangular integration using /METH=2 is:

$$waveOut[0] = 0$$

$$waveOut[p+1] = \sum_{i=0}^{p} (x_{i+1} - x_i) waveIn[i].$$

The inverse of this rectangular integration is the backwards difference.

Trapezoidal integration (/METH=1) is a more accurate method of computing the integral than rectangular integration. The computation equation is:

$$waveOut[0] = 0$$

$$waveOut[p] = waveOut[p-1] + \frac{\Delta x}{2}\big(waveIn[p-1] + waveIn[p]\big).$$

If the optional /D = *destWave* flag is omitted, then the wave is integrated in place overwriting the source wave.

When using an X wave, the X wave must be a 1D wave with data type matching the Y wave (excluding the complex type flag). Rectangular integration (/METH=0 or 2) requires an X wave having one more point than the number of elements in the dimension of the Y wave being integrated. X waves with number points plus one are allowed for rectangular integration with methods needing only the number of points. X waves are not used with integer source waves.

Although it is mathematically suspect, rectangular integration using /METH=0 would be correct if the X scaling of the output wave is offset by ΔX.

`Differentiate/METH=1/EP=1` is the inverse of `Integrate/METH=2`, but `Integrate/METH=2` is the inverse of `Differentiate/METH=1/EP=1` only if the original first data point is added to the output wave.

Integrate applied to an XY pair of waves does not check the ordering of the X values and doesn't care about it. However, it is usually the case that your X values should be monotonic. If your X values are not monotonic, you should be aware that the X values will be taken from your X wave in the order they are found, which will result in random X intervals for the X differences. It is usually best to sort the X and Y waves first (see **Sort**).

See Also

Differentiate, **Integrate2D**, **Integrate1D**, **area** , **areaXY**

Integrate1D

Integrate1D(*UserFunctionName*, *min_x*, *max_x* [, *options* [, *count*] [, *pWave*]]])

The Integrate1D function performs numerical integration of a user function between the specified limits (*min_x* and *max_x*).

Parameters

UserFunctionName must have this format:

```
Function UserFunctionName(inX)
   Variable inX
   ... do something
   return result
End
```

However, if you supply the optional *pWave* parameter then it must have this format:

```
Function UserFunctionName(pWave, inX)
   Wave pWave
   Variable inX
   ... do something
   return result
End
```

options is one of the following:

0:	Trapezoidal integration (default).
1:	Romberg integration.
2:	Gaussian Quadrature integration.

By default, *options* is 0 and the function performs trapezoidal integration. In this case Igor evaluates the integral iteratively. In each iteration the number of points where Igor evaluates the function increases by a factor of 2. The iterations terminate at convergence to tolerance or when the number of evaluations is 2^{23}.

The *count* parameter specifies the number of subintervals in which the integral is evaluated. If you specify 0 or a negative number for count, the function performs an adaptive Gaussian Quadrature integration in which Igor bisects the interval and performs a recursive refining of the integration only in parts of the interval where the integral does not converge to tolerance.

pWave is an optional parameter that, if present, is passed to your function as the first parameter. It is intended for your private use, to pass one or more values to your function, and is not modified by Igor. The *pWave* parameter was added in Igor Pro 7.00.

Details

You can integrate complex-valued functions using a function with the format:

```
Function/C complexUserFunction(inX)
   Variable inX
   Variable/C result
   //… do something
   return result
End
```

The syntax used to invoke the function is:

```
Variable/C cIntegralResult=Integrate1D(complexUserFunction,min_x,max_x…)
```

You can also use Integrate1D to perform a higher dimensional integrals. For example, consider the function: $F(x,y) = 2x + 3y + xy$.

In this case, the integral

$$h = \int dy \int f(x,y)\,dx$$

can be performed by establishing two user functions:

```
Function Do2dIntegration(xmin,xmax,ymin,ymax)
   Variable xmin,xmax,ymin,ymax

   Variable/G globalXmin=xmin
   Variable/G globalXmax=xmax
   Variable/G globalY
   return Integrate1d(userFunction2,ymin,ymax,1)  // Romberg integration
End

Function UserFunction1(inX)
   Variable inX

   NVAR globalY=globalY

   return (3*inX+2*globalY+inX*globalY)
End

Function UserFunction2(inY)
   Variable inY

   NVAR globalY=globalY
   globalY=inY
   NVAR globalXmin=globalXmin
   NVAR globalXmax=globalXmax

   // Romberg integration
   return Integrate1D(userFunction1,globalXmin,globalXmax,1)
End
```

This method can be extended to higher dimensions.

If the integration fails to converge or if the integrand diverges, Integrate1D returns NaN. When a function fails to converge it is a good idea to try another integration method or to use a user-defined number of intervals (as specified by the count parameter). Note that the trapezoidal method is prone to convergence problems when the absolute value of the integral is very small.

See Also

Integrate, **Integrate2D**, **SumSeries**

Integrate2D

`Integrate2D [`*`flags`*`] [`*`keyword = value`* `[,` *`keyword = value`* `…]]`

The Integrate2D operation calculates a two-dimensional numeric integral of a real-valued user-defined function or a wave. The result of the operation is stored in the variable V_value and the variable V_Flag is set to zero if there are no errors.

This operation was added in Igor Pro 7.00.

Flags

/OPTS=*op*	Sets the integration options. By default, both the x and the y integrations are performed using the adaptive trapezoidal method. *op* is a bitwise parameter that you set to select the x and y integration methods. Set one bit for x and one bit for y: Bit 0: Trapezoidal in Y (1) Bit 1: Romberg in Y (2) Bit 2: Gaussian Quadrature in Y (4) Bit 3: Trapezoidal in X (8) Bit 4: Romberg in X (16) Bit 5: Gaussian Quadrature in X (32) See **Setting Bit Parameters** on page IV-12 for details about bit settings. Using these constants you can specify, for example, Romberg integration in the Y direction and Gaussian Quadrature in the X direction using /OPTS=(2 | 32).
/Q	Suppress printing to the history area.
/Z=zFlag	Set zFlag to 1 to suppress error reporting.

Keywords

epsilon=*ep*	Specifies the convergence parameter. By default *ep*=1e-5. Smaller values lead to more accurate integration result but the tradeoff is longer computation time.
integrand=*uF*	Specifies the user function to be integrated. See **The Integrand Function** below for details.
innerLowerLimit=*y1*	Specifies the lower limit of the inner integral if this limit is fixed, i.e., if it is not a function of x. See the innerLowerFunc keyword if you need to specify a function for this limit.
innerUpperLimit=*y2*	Specifies the upper limit of the inner integral if this limit is fixed, i.e., if it is not a function of x. See the innerUpperFunc keyword if you need to specify a function for this limit.
innerLowerFunc=*y1Func*	Specifies a user-defined function for the lower limit of the inner integral. See **The Limit Functions** below.
innerUpperFunc=*y2Func*	Specifies a user-defined function for the upper limit of the inner integral. See **The Limit Functions** below.
outerLowerLimit=*x1*	Specifies the lower limit of the outer integral.
outerUpperLimit=*x2*	Specifies the upper limit of the outer integral.
paramWave=*pWave*	Specifies a wave to be passed to the integrand and limit user-defined functions as the pWave parameter. The wave may contain any number of values that you might need to evaluate the integrand or the integration limits. If you omit paramWave then the pWave parameter to the functions will be NULL.
srcWave=*mWave*	If you need to perform 2D integration of some data, you can specify the data directly instead of providing a user-defined function that returns interpolated data. mWave must be a 2D wave. Higher dimensional waves are accepted but only the first layer of the wave is used in the integration.

The Integrand Function

Integrate2D computes the general two-dimensional integral of a user-defined integrand function which you specify using the integrand keyword. The integrand function has this form:

```
Function integrandFunc(pWave,inX,inY)
   Wave/Z pWave
   Variable inX,inY

   ... do something
   return result
End
```

The function can have any name - integrandFunc is just an example. The function must take the parameters shown and must return a real numeric result. Returning a NaN terminates the integration.

pWave is a parameter wave that you specify using the paramWave keyword. The operation passes this wave on every call to the integrand function. If you omit paramWave when invoking Integrate2D then pWave will be NULL.

The Limit Functions

The limit functions provide lower and/or upper limits of integration for the inner integral if they are functions of x rather than fixed values. You specify a limit function using the innerLowerFunc and innerUpperFunc keywords. The form of the limit function is:

```
Function limitFunction(pWave,inX)
   Wave/Z pWave
   Variable inX

   ... do something
```

```
    return result
End
```

Details

The operation computes the general two-dimensional integral of the form

$$I = \int_{x1}^{x2} dx \int_{y1(x)}^{y2(x)} f(x,y)dy.$$

Here y1 and y2 are in general functions of x but could also be simple constants, and f(x,y) is real valued function. The integral is evaluated by considering the "outer" integral

$$I = \int_{x1}^{x2} G(x)dx,$$

where G(x) is the "inner" integral

$$G(x) = \int_{y1(x)}^{y2(x)} f(x,y)dy.$$

The operation allows you to specify different algorithms for integrating the inner and outer integrals. The simplest integration algorithm is the Trapezoidal method. You can typically improve on the accuracy of the calculation using Romberg integration and the performance of Gaussian quadrature depends significantly on the nature of the integrand.

Example 1: Integrating a 2D function over fixed limits

Suppose we wanted to check the normalization of the built-in two-dimensional Gauss function. The user-defined function would be:

```
Function myIntegrand1(pWave,inX,inY)
    Wave/Z pWave
    Variable inX,inY
    return Gauss(inX,50,10,inY,50,10)
End
```

To perform the integration, execute:

```
Integrate2D outerLowerLimit=0, outerUpperLimit=100, innerLowerLimit=0,
    innerUpperLimit=100, integrand=myIntegrand1
Print/D V_Value
```

Example 2: Integrating a 2D function using function limits

In this example we compute the volume of a sphere of radius 1. To do so we use symmetry and integrate the volume in one octant only. The limits of integration are [0,1] in the x direction and [0,sqrt(1-x^2)] in the y direction. In this case we need to define two user-defined functions: one for the integrand and one for the upper limit of the inner integral:

```
Function myIntegrand2(pWave,inX,inY)
    Wave/Z pWave
    Variable inx,iny
    Variable r2=inX^2+inY^2
    if(r2>=1)
        return 0
    else
        return sqrt(1-r2)
    endif
End

Function y2Func(pWave,inX)
    Wave pWave
    Variable inX
    return sqrt(1-inX^2)
End
```

To perform the integration, execute:

```
Integrate2D outerLowerLimit=0, outerUpperLimit=1, innerLowerLimit=0,
   innerUpperFunc=y2Func, integrand=myIntegrand2
Print/D 4*pi/3 -8*V_Value // Calculation error
```

Note that the integrand function tests that r2 does not exceed 1. This is because the sqrt function would return a NaN if r2>1 which can happen due to floating point rounding errors. Returning a NaN terminates the integration.

Example 3: Integrating a 2D wave using fixed limits

In this example we compute the volume between the surface defined by the wave my2DWave and the plane z=0 with integration limits [0,1] in the x-direction and [0,2] in the y-direction. For simplicity we set the wave's value to be a constant (pi).

```
Make/O/N=(5,9) my2DWave=pi
SetScale/P x 0,0.3,"", my2DWave
SetScale/P y 0,0.4,"", my2DWave
Integrate2D outerLowerLimit=0, outerupperLimit=1, innerlowerLimit=0, innerUpperLimit=2,
   srcWave=my2DWave
Print/D 2*pi-V_Value          // Calculation error
```

See Also

Integrate, **Differentiate**, **Integrate1D**, **area**, **areaXY**

IntegrateODE

`IntegrateODE [`*`flags`*`] `*`derivFunc, cwaveName, ywaveSpec`*

The IntegrateODE operation calculates a numerical solution to a set of coupled ordinary differential equations by integrating derivatives. The derivatives are user-specified via a user-defined function, *derivFunc*. The equations must be a set of first-order equations; a single second-order equation must be recast as two first-order equations, a third-order equation to three first order equations, etc. For more details on how to write the function, see **Solving Differential Equations** on page III-274.

IntegrateODE offers two ways to specify the values of the independent variable (commonly called X or t) at which output Y values are recorded. You can specify the X values or you can request a “free-run” mode.

The algorithms used by IntegrateODE calculate results at intervals that vary according to the characteristics of the ODE system and the required accuracy. You can set specific X values where you need output (see the /X flag below) and arrangements will be made to get values at those specific X values. In between those values, IntegrateODE will calculate whatever spacing is needed, but intermediate values will not be output to you.

If you specify free-run mode, IntegrateODE will simply output all steps taken regardless of the spacing of the X values that results.

Parameters

cwaveName	Name of wave containing constant coefficients to be passed to *derivFunc*.
derivFunc	Name of user function that calculates derivatives. For details on the form of the function, see **Solving Differential Equations** on page III-274.
ywaveSpec	Specifies a wave or waves to receive calculated results. The waves also contain initial conditions. The *ywaveSpec* can have either of two forms: *ywaveName*: *ywaveName* is a single, multicolumn wave having one column for each equation in your equation set (if you have just one equation, the wave will be a simple 1D wave). {*ywave0*, *ywave1*, …}: The *ywaveSpec* is a list of 1D waves, one wave for each equation. The ordering is important — it must correspond to the elements of the y wave and dydx wave passed to *derivFunc*. Unless you use the /R flag to alter the start point, the solution to the equations is calculated at each point in the waves, starting with row 1. You must store the initial conditions in row 0.

Flags

/CVOP={*solver*, *jacobian*, *extendedErrors* [, *maxStep*]}

Selects options that affect how the Adams-Moulton and BDF integration schemes operate. This flag applies only when using /M = 2 or /M = 3. These methods are based on the CVODE package, hence the flag letters “CV”.

The *solver* parameter selects a solver method for each step. The values of *solver* can be:

solver=0:	Select the default for the integration method. That is functional for /M=2 or Newton for /M=3.
solver=1:	Functional solver.
solver=2:	Newton solver.

The *jacobian* parameter selects the method used to approximate the jacobian matrix (matrix of df/dy_i where f is the derivative function).

jacobian=0:	Full jacobian matrix.
jacobian=1:	Diagonal approximation.

In both cases, the derivatives are approximated by finite differences.

In our experience, *jacobian* = 1 causes the integration to proceed by much smaller steps. It might decrease overall integration time by reducing the computation required to approximate the jacobian matrix.

If the *extendedErrors* parameter is nonzero, extra error information will be printed to the history area in case of an error during integration using /M=2 or /M=3. This extra information is mostly of the sort that will be meaningful only to WaveMetrics software engineers, but may occasionally help you to solve problems. It is printed out as a bug message (`BUG:...`) regardless of whether it is our bug or yours.

If *maxStep* is present and greater than zero, this option sets the maximum internal step size that the CVODE package is allowed to take. This is particularly useful with /M=3, as the BDF method is capable of taking extremely large steps if the derivatives don't change much. Use of this option may be necessary to make sure that the CVODE package doesn't step right over a brief excursion in, say, a forcing function. If you have something in your derivative function that may be step-like and brief, set *maxStep* to something smaller than the duration of the excursion.

If you want to set *maxStep* only, set the other three options to zero.

/E=*eps*

Adjusts the step size used in the calculations by comparing an estimate of the truncation error against a fraction of a scaled number. The fraction is *eps*. For instance, to achieve error less than one part in a thousand, set *eps* to 0.001. The number itself is set by a combination of the /F flag and possibly the wave specified with the /S flag. See **Solving Differential Equations** on page III-274 for details.

If you do not use the /E flag, *eps* is set to 10^{-6}.

For details, see **Error Monitoring** on page III-284.

/F=*errMethod* Adjusts the step size used in the calculations by comparing an estimate of the truncation error against a scaled number. *errMethod* is a bitwise parameter that specifies what to include in that number:

bit 0:	Add a constant from the error scaling wave set by the /S flag.
bit 1:	Add the current value of the results.
bit 2:	Add the current value of the derivatives.
bit 3:	Multiply by the current step size (/M=0 or /M=1 only).

Each bit that you set of bits 0, 1, or 2 adds a term to the number; setting bit 3 multiplies the sum by the current step size to achieve a global error limit. Note that bit 3 has no effect if you use the Adams or BDF integrators (/M=2 or /M=3). See **Setting Bit Parameters** on page IV-12 for further details about bit settings.

If you don't include the /F flag, a constant is used. Unless you use the /S flag, that constant is set to 1.0.

For details, see **Error Monitoring** on page III-284.

/M=*m* Specifies the method to use in calculating the solution.

m=0:	Fifth-order Runge-Kutta-Fehlberg (default).
m=1:	Bulirsch-Stoer method using Richardson extrapolation.
m=2:	Adams-Moulton method.
m=3:	BDF (Backwards Differentiation Formula, or Gear method). This method is the preferred method for stiff problems.

If you don't specify a method, the default is the Runge-Kutta method (*m*=0). Bulirsch-Stoer (*m*=1) should be faster than Runge-Kutta for problems with smooth solutions, but we find that this is often not the case. Simple experiments indicate that Adams-Moulton (*m*=2 may be fastest for nonstiff problems. BDF (*m*=3) is definitely the preferred one for stiff problems. Runge-Kutta is a robust method that may work on problems that fail with other methods.

/Q [= *quiet*] *quiet* = 1 or simply /Q sets quiet mode. In quiet mode, no messages are printed in the history, and errors do not cause macro abort. The variable V_flag returns an error code. See **Details** for the meanings of the V_flag error codes.

/R=(*startX*,*endX*) Specifies an X range of the waves in *ywaveSpec*.

/R=[*startP*,*endP*] Specifies a point range in *ywaveSpec*.

If you specify the range as /R=[*startP*] then the end of the range is taken as the end of the wave. If /R is omitted, the entire wave is evaluated. If you specify only the end point (/R = [,*endP*]) the start point is taken as point 0.

You must store initial conditions in *startP*. The first point is *startP*+1.

/S=*errScaleWaveName*

If you set bit 0 of *errMethod* using the /F flag, or if you don't include the /F flag, a constant is required for scaling the estimated error for each differential equation. By default, the constants are simply set to 1.0.

You provide custom values of the constants via the /S flag and a wave. Make a wave having one element for each derivative, set a reasonable scale factor for the corresponding equation, and set *errScaleWaveName* to the name of that wave.

If you don't use the /S flag, the constants are all set to 1.0.

/STOP = {*stopWave*, *mode*}

Requests that IntegrateODE stop when certain conditions are met on either the solution values (Y values) or the derivatives.

stopWave contains information that IntegrateODE uses to determine when to stop.

mode controls the logical operations applied to the elements of stopWave:

mode=0:	OR mode. If *stopWave* contains more than one condition, any one condition will stop the integration when it is satisfied.
mode=1:	AND mode. If *stopWave* contains more than one condition, all conditions must be satisfied to cause the integration to stop.

See Details, below, for more information.

/U=*u*

Update the display every *u* points. By default, it will update the display every 10 points. To disable updates, set *u* to a very large number.

/X=*xvaluespec*

Specifies the values of the independent variable (commonly called x or t) at which values are to be calculated (see parameter *ywaveSpec*).

You can provide a wave or *x0* and *deltaX*:

/X = *xWaveName*

Use this form to provide a list of values for the independent variable. They can have arbitrary spacing and may increase or decrease, but should be monotonic.

If you use the /XRUN flag to specify free-run mode, /X = *xWaveName* is required. In this case, the X wave becomes an output wave and any contents are overwritten. See the description of /XRUN for details.

xValues = {*x0, deltaX*}

If you use this form, *x0* is the initial value of the independent variable. This is the value at which the initial conditions apply. It will calculate the first result at *x0*+*deltaX*, and subsequent results with spacing of *deltaX*.

deltaX can be negative.

If you do not use the xValues keyword, it reads independent variable values from the X scaling of the results wave (see *ywaveSpec* parameter).

/XRUN={*dx0*, *Xmax*}

If *dx0* is nonzero, the output is generated in a free-running mode. That is, the output values are generated at whatever values if the independent variable (*x* or *t*) the integration method requires to achieve the requested accuracy. Thus, you will get solution points at variably-spaced X values.

The parameter *dx0* sets the step size for the first integration step. If this is smaller than necessary, the step size will increase rapidly. If it is too large for the requested accuracy, the integration method will decrease the step size as necessary.

If *dx0* is set to zero, free-run mode is not used; this is the same as if the XRUN flag is not used.

When using free-run mode, you must provide an X wave using /X = *xWaveName*. Set the first value of the wave (this is usually point zero, but may not be if you use the /R flag) to the initial value of X.

As the integration proceeds, the X value reached for each output point is written into the X wave. The integration stops when the latest step taken goes beyond *Xmax* or when the output waves are filled.

Details

The various waves you may use with the IntegrateODE operation must meet certain criteria. The wave to receive the results (*ywaveSpec*), and which contains the initial conditions, must have exactly one column for each equation in your system of equations, or you must supply a list of waves containing one wave for each equation. Because IntegrateODE can't determine how many equations there are from your function, it uses the number of columns or the number of waves in the list to determine the number of equations.

If you supply a list of waves for *ywaveSpec*, all the waves must have the same number of rows. If you supply a wave containing values of the independent variable or to receive X values in free-run mode (using /X=*waveName*) the wave must have the same number of rows as the *ywaveSpec* waves.

The wave you provide for error scaling via the /S flag must have one point for each equation. That is, one point for each *ywaveSpec* wave, or one point for each column of a multicolumn *ywaveSpec*.

By default, the display will update after each tenth point is calculated. If you display one of your *ywaveSpec* waves in a graph, you can watch the progress of the integration.

The display update may slow down the calculation considerably. Use the /U flag to change the interval between updates. To disable the updates entirely, set the update interval to a number larger than the length of the waves in *ywaveSpec*.

In free-run mode, it is impossible to predict how many output values you will get. IntegrateODE will stop when either your waves are filled, or when the X value exceeds *Xmax* set in the /XRUN flag. The best strategy is to make the waves quite large; unused rows in the waves will not be touched. To avoid having “funny” traces on a graph, you can prefill your waves with NaN. Make sure that you don’t set the initial condition row and initial X value row to Nan!

Stopping IntegrateODE

In some circumstances it is useful to be able to stop the integration early, before the full number of output values has been computed. You can do this two ways: using the /STOP flag to put conditions on the solution, or by returning 1 from your derivative function.

When using /STOP={*stopWave*, *mode*}, *stopWave* must have one column for each equation in your system or, equivalently, a number of columns equal to the order of your system. Each column represents a condition on either the solution value or the derivatives for a given equation in your system.

Row 0 of *stopWave* contains a flag telling what sort of condition to apply to the solution values. If the flag is zero, that value is ignored. If the flag is 1, the integration is stopped if the solution value exceeds the value you put in row 1. If the flag is -1, integration is stopped when the solution value is less than the value in row 1.

Rows 2 and 3 work just like rows 0 and 1, but the conditions are applied to the derivatives rather than to the solution values.

If *stopWave* has two rows, only the solution values are checked. If *stopWave* has four rows, you can specify conditions on both solution values and derivatives.

You can set more than one flag value non-zero. If you do that, then *mode* determines how the multiple conditions are applied. If *mode* is 0, then any one condition can stop integration when it is satisfied. If *mode* is 1, all conditions with a non-zero flag value must be satisfied at the same time. If row 0 and row 2 have nothing but zeroes, then *stopWave* is ignored.

For further discussion, see **Stopping IntegrateODE on a Condition** on page III-287.

Output Variables

The IntegrateODE operation sets a variety of variables to give you information about the integration. These variables are updated at the same time as the display so you can monitor an integration in progress. They are:

V_ODEStepCompleted	Point number of the last result calculated.
V_ODEStepSize	Size of the last step in the calculation.
V_ODETotalSteps	Total number of steps required to arrive at the current result. In free-run mode, this is the same as V_ODEStepCompleted.
V_ODEMinStep	Minimum step size used during the entire calculation.
V_ODEFunctionCalls	The total number of calls made to your derivative function.
V_Flag	Indicates why IntegrateODE stopped.

The values for V_Flag are:

0:	Finished normally.
1:	User aborted the integration.

2: Integration stopped because the step size became too small.
That is, dX was so small that X + dX = X.

3: IntegrateODE ran out of memory.

4: In /M=2 or /M=3, the integrator received illegal inputs. Please report this to WaveMetrics (see **Technical Support** on page II-4 for contact details).

5: In /M=2 or /M=3, the integrator stopped with a failure in the step solver. The method chosen may not be suitable to the problem.

6: Indicates a bug in IntegrateODE. Please report this to WaveMetrics (see **Technical Support** on page II-4 for contact details).

7: An error scaling factor was zero (see /S and /F flags).

8: IntegrateODE stopped because the conditions specified by the /STOP flag were met.

9: IntegrateODE stopped because the derivative function returned a value requesting the stop.

See Also

Solving Differential Equations on page III-274 gives the form of the derivative function, details on the error specifications and what they mean, along with several examples.

References

The Runge-Kutta (/M=0) and Bulirsh-Stoer (/M=1) methods are based on routines in Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992, and are used by permission.

The Adams-Moulton (/M=2) and BDF methods (/M=3) are based on the CVODE package developed at Lawrence Livermore National Laboratory:

Cohen, Scott D., and Alan C. Hindmarsh, *CVODE User Guide*, LLNL Report UCRL-MA-118618, September 1994.

The CVODE package was derived in part from the VODE package. The parts used in Igor are described in this paper:

Brown, P.N., G. D. Byrne, and A. C. Hindmarsh, VODE, a Variable-Coefficient ODE Solver, *SIAM J. Sci. Stat. Comput.*, *10*, 1038-1051, 1989.

interp

`interp(`*`x1, xwaveName, ywaveName`*`)`

The interp function returns a linearly interpolated value at the location x = *x1* of a curve whose X components come from the Y values of *xwaveName* and whose y components come from the Y values of *ywaveName*.

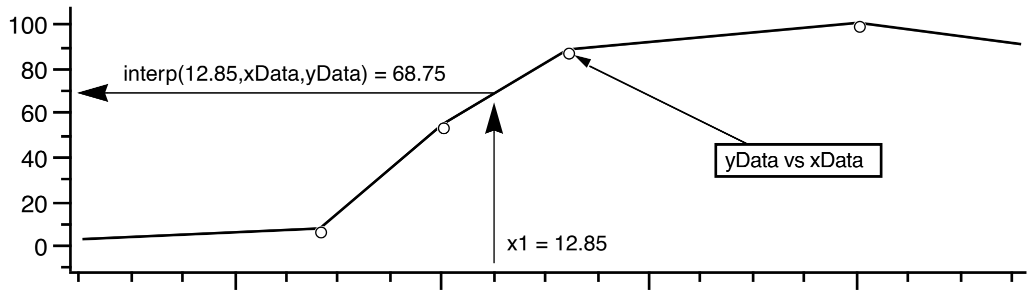
Details

interp returns nonsense if the waves are complex or if *xwaveName* is not monotonic.

The interp function is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

Examples

Examples



See Also

Interpolate2

The **Loess**, **ImageInterpolate**, **Interpolate3D**, and **Interp3DPath** operations.

The **Interp2D**, **Interp3D** and **ContourZ** functions.

Interp2D

`Interp2D(`*`srcWaveName, xValue, yValue`*`)`

The Interp2D function returns a double precision number as the interpolated value for the *xValue*, *yValue* point in the source wave. Returns NaN if the point is outside the source wave domain or if the source wave is complex.

Parameters

srcWaveName is the name of a 2D wave. The wave can not be complex.

xValue is the X-location of the interpolated point.

yValue is the Y-location of the interpolated point.

See Also

The **ImageInterpolate** operation. **Interpolation** on page III-109.

Interp3D

`Interp3D(`*`srcWave, x, y, z`*` [, `*`triangulationWave`*`])`

The Interp3D function returns an interpolated value for location P=(*x*, *y*, *z*) in a 3D scalar distribution *srcWave*.

If *srcWave* is a 3D wave containing a scalar distribution sampled on a regular lattice, the function returns a linearly interpolated value for any P=(*x*, *y*, *z*) location within the domain of *srcWave*. If P is outside the domain, the function returns NaN.

To interpolate a 3D scalar distribution that is not sampled on a regular lattice, *srcWave* is a four column 2D wave where the columns correspond to *x*, *y*, *z*, f(*z*, *y*, *z*), respectively. You must also use a "triangulation" wave for *srcWave* (use `Triangulate3D/out=1` to obtain the triangulation wave). If P falls within the convex domain defined by the tetrahedra in *triangulationWave*, the function returns the barycentric linear interpolation for P using the tetrahedron where P is found. If P is outside the convex domain the function returns NaN.

Examples

```
Make/O/N=(10,20,30) ddd=gnoise(10)
Print interp3D(ddd,1,0,0)
Print interp3D(ddd,1,1,1)
```

```
Make/O/N=(10,4) ddd=gnoise(10)
Triangulate3D/OUT=1 ddd
Print interp3D(ddd,1,0,0,M_3DVertexList)
Print interp3D(ddd,1,1,1,M_3DVertexList)
```

See Also

The **Interpolate3D** operation. **Interpolation** on page III-109.

Interp3DPath

`Interp3DPath `*`3dWave tripletPathWave`*

The Interp3DPath operation computes the trilinear interpolated values of *3dWave* for each position specified by a row of in *tripletPathWave*, which is a 3 column wave in which the first column represents the X coordinate, the second represents the Y coordinate and the third represents the Z coordinate. Interp3DPath stores the resulting interpolated values in the wave W_Interpolated. Interp3DPath is equivalent to calling the Interp3D() function for each row in *tripletPathWave* but it is computationally more efficient.

If the position specified by the *tripletPathWave* is outside the definition of the *3dWave* or if it contains a NaN, the operation stores a NaN in the corresponding output entry.

Both *3dWave* and *tripletPathWave* can be of any numeric type. W_Interpolated is always of type NT_FP64.

See Also

The **ImageInterpolate** operation and the **Interp3D** and **interp** functions. **Interpolation** on page III-109.

Interpolate2

`Interpolate2 [`*`flags`*`] [`*`xWave,`*`] `*`yWave`*

The Interpolate2 operation performs linear, cubic spline and smoothing cubic spline interpolation on 1D waveform or XY data. It produces output in the form of a waveform or an XY pair.

The cubic spline interpolation is based on a routine from "Numerical Recipes in C".

The smoothing spline is based on "Smoothing by Spline Functions", Christian H. Reinsch, *Numerische Mathematic* 10, 177-183 (1967).

For background information, see **The Interpolate2 Operation** on page III-110.

Prior to Igor Pro 7, Interpolate2 was implemented as part of the Interpolate XOP. It is now built-in.

Parameters

xWave specifies the wave which supplies the X coordinates for the input curve. If you omit it, X coordinates are taken from the X values of *yWave*.

yWave specifies the wave which supplies the Y coordinates for the input curve.

Flags

/A=*a*	Controls pre-averaging. Pre-averaging is deprecated - use the smoothing spline (/T=3) instead. If *a* is zero, Interpolate2 does no pre-averaging. If *a* is greater than one, it specifies the number of nodes through which you want the output curve to go. Interpolate2 creates the nodes by averaging the raw input data. Pre-averaging does not work correctly with the log-spaced output mode (/I=2). This is because the pre-averaging is done on linearly-spaced intervals but the input data is log-spaced.
/E=*e*	Controls how the end points are determined for cubic spline interpolation only. *e*=1: Match first derivative (default) *e*=2: Match second derivative (natural)
/F=*f*	*f* is the smoothing factor used for the smoothing spline. *f*=0 is nearly identical to the cubic spline. *f*>0 gives increasing amounts of smoothing as f increases. See **Smoothing Spline Parameters** on page III-113 for details.

/I[=*i*]

Determines at what X coordinates the interpolation is done.

i=0:	Gives output values at evenly-spaced X coordinates that span the X range of the input data. This is the default setting if /I is omitted.
i=1:	Same as *i*=0 except that the X input values are included in the list of X coordinates at which to interpolate. This is rarely needed and is not available if no X destination wave is specified. /I is equivalent to /I=1. Both are not recommended.
i=2:	Gives output values at X coordinates evenly-spaced on a logarithmic scale. Use this if your data is plotted on a logarithmic X axis. This mode ignores any non-positive values in your input X data.
i=3:	Gives output values at X coordinates that you specify by setting the X coordinates of the destination wave before calling Interpolate2. You must create your destination wave or waves before doing the interpolation for this mode.

If you omit /X=*xDest* then the X coordinates come from the X values of the output waveform designated by /Y=*yDest*.

If you include /X=*xDest* then the X coordinates come from the data values of the specified X output wave.

When using /I=3, the number of output points is determined by the destination wave and the /N flag is ignored.

See **Destination X Coordinates from Destination Wave** on page III-114 for further details.

/J=*j*

Controls the use of end nodes with pre-averaging (/A). Pre-averaging is deprecated - use the smoothing spline (/T=3) instead.

j=0:	Turns end nodes off.
j=1:	Creates end nodes by cubic extrapolation.
j=2:	Creates end nodes equal to the first and last data points of the input data set, not counting points that contain NaNs or INFs.

/N=*n*

Controls the number of points in the output wave or waves. *n* defaults to the larger of 200 and the number of points in the source waves. This value is ignored if you /I=3 (X from dest mode).

/S=*s*

s is the estimated standard deviation of the noise of the Y data. It is used for the smoothing spline only. *s* is used as the estimated standard deviation for all points in the Y data.

If neither /S nor /SWAV are present, Interpolate2 arbitrarily assumes an *s* equal to .05 times the amplitude of the Y data.

/SWAV=*stdDevWave*

stdDevWave is a wave containing the standard deviation of the noise of the Y data on a point-by-point basis. It is used for the smoothing spline only. *stdDevWave* must have the same number of points as the Y data wave.

If neither /S nor /SWAV are present, Interpolate2 arbitrarily assumes an *s* equal to .05 times the amplitude of the Y data.

/T=*t*

Controls the type of interpolation performed.

t=1:	Linear interpolation
t=2:	Cubic spline interpolation (default)
t=3:	Smoothing spline interpolation

/X=*xDest* Specifies the X destination wave.

If /X is present the output is an XY pair.

If /X is omitted the output is a waveform.

The X destination wave may or may not exist when Interpolate2 is called except for "X from dest" mode (/I=3) when it must exist. Interpolate2 overwrites it if it exists.

/Y=*yDest* Specifies the Y destination wave name.

If you omit /Y, a default wave name is generated. The name of the default wave is the name of the source Y wave plus "_L" for linear interpolation, "_CS" for cubic spline or "_SS" for smoothing spline.

The Y destination wave may or may not exist when Interpolate2 is called. Interpolate2 overwrites it if it exists.

See Also

The Interpolate2 Operation on page III-110

References

"*Numerical Recipes in C*" for cubic spline.

"Smoothing by Spline Functions", Christian H. Reinsch, *Numerische Mathematic* 10, 177-183 (1967).

Interpolate3D

Interpolate3D [/Z] /RNGX={*x0,dx,nx*}/RNGY={*y0,dy,ny*}/RNGZ={*z0,dz,nz*}
/DEST=*dataFolderAndName*, triangulationWave=*tWave*, srcWave=*sWave*

The Interpolate3D operation uses a precomputed triangulation of *sWave* (see **Triangulate3D**) to calculate regularly spaced interpolated values from an irregularly spaced source. The interpolated values are calculated for a lattice defined by the range flags /RNGX, /RNGY, and /RNGZ. *sWave* is a 4 column wave where the first three columns contain the spatial coordinates and the fourth column contains the associated scalar value. Interpolate3D is essentially equivalent to calling the **Interp3D** function for each interpolated point in the range but it is much more efficient.

Parameters

triangulationWave=*tWave*

Specifies a 2D index wave, *tWave*, in which each row corresponds to one tetrahedron and each column (tetrahedron vertex) is represented by an index of a row in *sWave*. Use **Triangulate3D** with /OUT=1 to obtain *tWave*.

srcWave=*sWave* Specifies a real-valued 4 column 2D source wave, *sWave*, in which columns correspond to *x*, *y*, *z*, f(*x*, *y*, *z*). Requires that the domain occupied by the set of {*x*, *y*, *z*} be convex.

Flags

/DEST=*dataFolderAndName*

Saves the result in the specified destination wave. The destination wave will be created or overwritten if it already exists. *dataFolderAndName* can include a full or partial path with the wave name.

/RNGX={*x0*,*dx*,*nx*} Specifies the range along the X-axis. The interpolated values start at *x0*. There are *nx* equally spaced interpolated values where the last value is at *x0*+(*nx*-1)*dx*. If you would like to interpolate the data for a single plane you can set the appropriate number of values to 1. For example, a YZ plane would have *nx*=1.

/RNGY={*y0*,*dy*,*ny*} Specifies the range along the Y-axis. The interpolated values start at *y0*. There are *nx* equally spaced interpolated values where the last value is at *y0*+(*ny*-1)*dy*. If you would like to interpolate the data for a single plane you can set the appropriate number of values to 1. For example, a XZ plane would have *ny*=1.

/RNGZ={*z0*,*dz*,*nz*}	Specifies the range along the Z-axis. The interpolated values start at *z0*. There are *nz* equally spaced interpolated values where the last value is at *z0*+(*nz*-1)*dz*. If you would like to interpolate the data for a single plane you can set the appropriate number of values to 1. For example, a XY plane would have *nz*=1.
/Z	No error reporting.

Details

The triangulation wave defines a set of tetrahedra that spans the convex source domain. If the requested range consists of points outside the domain, the interpolated values will be set to NaN. The interpolation process for points inside the convex domain consists of first finding the tetrahedron in which the point resides and then linearly interpolating the scalar value using the barycentric coordinate of the interpolated point.

In some cases the interpolation may result in NaN values for points that are clearly inside the convex domain. This may happen when the preceding Triangulate3D results in tetrahedra that are too thin. You can try using Triangulate3D with the flag /OUT=4 to get more specific information about the triangulation. Alternatively you can introduce a slight random perturbation to the input source wave before the triangulation.

Example

```
Function Interpolate3DDemo()
    Make/O/N=(50,4) ddd=gnoise(20)    // First 3 columns store XYZ coordinates
    ddd[][3]=ddd[p][2]                // Fourth column stores a scalar which is set to z
    Triangulate3D ddd                 // Perform the triangulation
    Wave M_3dVertexList
    Interpolate3D /RNGX={-30,1,80}/RNGY={-40,1,80}/RNGZ={-40,1,80}
            /DEST=W_Interp triangulationWave=M_3dVertexList,srcWave=ddd
End
```

See Also

The **Triangulate3D** operation and the **Interp3D** function. **Interpolation** on page III-109.

References

Schneider, P.J., and D. H. Eberly, *Geometric Tools for Computer Graphics*, Morgan Kaufmann, 2003.

inverseErf

`inverseErf(x)`

The inverseErf function returns the inverse of the error function.

Details

The function is calculated using rational approximations in several regions followed by one iteration of Halley's algorithm.

See Also

The **erf**, **erfc**, **dawson**, and **inverseErfc** functions.

inverseErfc

`inverseErfc(x)`

The inverseErfc function returns the inverse of the complementary error function.

Details

The function is calculated using rational approximations in several regions followed by one iteration of Halley's algorithm.

See Also

The **erf**, **erfc**, **erfcw**, **dawson**, and **inverseErf** functions.

ItemsInList

`ItemsInList(`*`listStr`*` [, `*`listSepStr`*`])`

The ItemsInList function returns the number of items in *listStr*. *listStr* should contain items separated by the *listSepStr* character, such as "abc;def;".

Use ItemsInList to count the number of items in a string containing a list of items separated by a single character, such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

If *listStr* is `""` then 0 is returned.

listSepStr is optional. If missing, *listSepStr* is presumed to be ";".

Details

listStr is searched for item strings bound by *listSepStr* on the left and right.

An item can be empty. The lists `"abc;def;;ghi"` and `";abc;def;;ghi;"` have four items (the third item is `""`).

listStr is treated as if it ends with a *listSepStr* even if it doesn't. The search is case-sensitive.

In Igor6, only the first byte of *listSepStr* was used. In Igor7 and later, all bytes are used.

Examples

```
Print ItemsInList("wave0;wave1;wave1#1;")          // prints 3
Print ItemsInList("key1=val1,key2=val2", ",")      // prints 2
Print ItemsInList("1 \t 2 \t", "\t")               // prints 2
Print ItemsInList(";")                             // prints 1
Print ItemsInList(";;")                            // prints 2
Print ItemsInList(";a;")                           // prints 2
Print ItemsInList(";;;")                           // prints 3
```

See Also

The **AddListItem**, **StringFromList**, **FindListItem**, **RemoveListItem**, **RemoveFromList**, **WaveList**, **TraceNameList**, **StringList**, **VariableList**, and **FunctionList** functions.

j

`j`

The j function returns the loop index of the 2nd innermost iterate loop in a macro. Not to be used in a function. iterate loops are archaic and should not be used.

JCAMPLoadWave

`JCAMPLoadWave [`*`flags`*`] [`*`fileNameStr`*`]`

The JCAMPLoadWave operation loads data from the named JCAMP-DX file into waves.

Prior to Igor7, JCAMPLoadWave was implemented as an XOP. It is now a built-in operation.

Parameters

If *fileNameStr* is omitted or is "", or if the /I flag is used, JCAMPLoadWave presents an Open File dialog from which you can choose the file to load.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/A	Automatically assigns arbitrary wave names using "wave" as the base name. Skips names already in use.
/A=*baseName*	Same as /A but it automatically assigns wave names of the form *baseName*0, *baseName*1.
/D	Creates double-precision waves. If omitted, JCAMPLoadWave creates single-precision waves.
/H	Reads header information from JCAMP file. If you include /W, this information is stored in the wave note. If you include /V, it is stored in header variables.
/I	Forces JCAMPLoadWave to display an Open File dialog even if the file is fully specified via /P and *fileNameStr*.
/N	Same as /A except that, instead of choosing names that are not in use, it overwrites existing waves.
/N=*baseName*	Same as /N except that it automatically assigns wave names of the form *baseName0*, *baseName1*.
/O	Overwrite existing waves in case of a name conflict.
/P=*pathName*	Specifies the folder to look in for *fileNameStr*. *pathName* is the name of an existing symbolic path.
/Q	Suppresses the normal messages in the history area.
/R	Reads data from file and creates Igor waves.
/V	Set variables from header information if /H is also present
/W	Stores header information in the wave note if /R and /H are also present.

Details

The /N flag instructs Igor to automatically name new waves "wave", or *baseName* if /N=baseName is used, plus a number. The number starts from zero and increments by one for each wave loaded from the file. If the resulting name conflicts with an existing wave, the existing wave is overwritten.

The /A flag is like /N except that Igor skips names already in use.

Output Variables

JCAMPLoadWave sets the following output variables:

`V_flag`	Number of waves loaded or -1 if an error occurs during the file load.
`S_fileName`	Name of the file being loaded.
`S_path`	File system path to the folder containing the file.
`S_waveNames`	Semicolon-separated list of the names of loaded waves.

S_path uses Macintosh path syntax (e.g., “`hd:FolderA:FolderB:`”), even on Windows. It includes a trailing colon.

When JCAMPLoadWave presents an Open File dialog and the user cancels, V_flag is set to 0 and S_fileName is set to `""`.

In addition, if the /V flag is used, variables are created corresponding to JCAMP-DX labels in the header. See **Variables Set By JCAMPLoadWave** on page II-149 for details.

Example

```
Function LoadJCAMP(pathName, fileName)
   String pathName     // Name of Igor symbolic path or ""
   String fileName     // Full path, partial path or simple file name

   JCAMPLoadWave/P=$pathName fileName
   if (V_Flag == 0)
      Print "No waves were loaded"
      return -1
   endif

   NVAR VJC_NPOINTS
   Printf "Number of points: %d\r", VJC_NPOINTS

   SVAR SJC_YUNITS
   Printf "Y Units: %s\r", SJC_YUNITS

   return 0
End
```

See Also

Loading JCAMP Files on page II-148

JacobiCn

`JacobiCn(`*`x, k`*`)`

The JacobiCn function returns the Jacobian elliptic function cn(x,k) for real x and modulus k with

$$0 < k^2 < 1.$$

The JacobiCn function was added in Igor Pro 7.00.

See Also

JacobiSn

Reference

F. W. J. Olver, D. W. Lozier, R. F. Boisvert, and C. W. Clark, editors, *NIST Handbook of Mathematical Functions*, chapter 22. Cambridge University Press, New York, NY, 2010.

JacobiSn

`JacobiSn(`*`x, k`*`)`

The JacobiSn function returns the Jacobian elliptic function sn(x,k) for real x and modulus k with

$$0 < k^2 < 1.$$

The JacobiSn function was added in Igor Pro 7.00.

See Also

JacobiCn

Reference

F. W. J. Olver, D. W. Lozier, R. F. Boisvert, and C. W. Clark, editors, *NIST Handbook of Mathematical Functions*, chapter 22. Cambridge University Press, New York, NY, 2010.

jlim

`jlim`

The jlim function returns the ending loop count for the 2nd inner most iterate loop. Not to be used in a function. iterate loops are archaic and should not be used.

JointHistogram

`JointHistogram [`*`flags`*`]`*`wave1, wave2`*`[,`*`wave3, wave4`*`]`

The JointHistogram computes 2D, 3D and 4D joint histograms of data provided in the input waves. The input waves must be 1D real numeric waves having the same number of points. The result of the operation is stored in the multi-dimensional wave M_JointHistogram in the current data folder or in the wave specified via the /DEST flag.

This operation was added in Igor Pro 7.00.

Flags

/BINS={*nx*, *ny*, *nz*, *nt*}	Specifies the number of bins along each axis. Set the number of bins for unused axes to zero. If the number of bins is non-zero, then the flags /XBMT, /YBMT, /ZBMT, and /TBMT are overridden.
/C	Sets the output wave scaling so that the values in each axis are centered in the bins. By default, wave scaling of the output wave is set with values at the left bin edges. This flag has no effect on axes where bins are specified by using /XBWV, /YBWV, /ZBWV or /TBWV.
/E	Excludes outliers. This flag is relevant only if there are one or more bin waves specified by using /XBWV, /YBWV, /ZBWV or /TBWV. By default values that might fall below the first bin or above the last bin are folded into the first and last bin respectively. These values (outliers) are excluded from the joint histogram when you use /E. See /P below for the way outliers affect the probability calculation.
/DEST=*destWave*	Specifies the output wave created by the operation. If you omit /DEST then the output wave is M_JointHistogram in the current data folder. It is an error to specify a destination which is the same as one of the input waves. When used in a user-defined function, the JointHistogram operation by default creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-66 for details.
/P=*mode*	Normalizes the histogram to a probability density. Use *mode*=0 to count all points, including possible outliers but excluding non-finite values, in the probability calculation. This is the default setting. Use *mode*=1 if you want to completely exclude outliers from the normalization. When outliers are excluded the output wave sums to 1. When they are included the sum of the output wave is smaller by the ratio of the number of outliers to the total number of points in the histogram.
/W=*weightWave*	Creates a weighted histogram. Instead of adding a single count to the appropriate bin, the corresponding value from *weightWave* is added to the bin. *weightWave* may be any real number type.
/XBMT=*method* /YBMT=*method* /ZBMT=*method* /TBMT=*method*	These flags specify which method is used to set the bins. By default method=0. These flags are overridden by /BINS if a non-zero value is specified for a given axis and by /XBWV, /YBWV, /ZBWV and /TBWV. See **JointHistogram Binning Methods** below for details.
/XBWV=*xBinWave* /YBWV=*yBinWave* /ZBWV=*zBinWave* /TBWV=*tBinWave*	Specifies the exact bins for a corresponding axis. The wave must be 1D real numeric wave with monotonically increasing finite values and must contain a minimum of 3 data points. The values in a bin wave specify the edges of the bins. A bin wave with N points defines n-1 bins. In the case of a 3-point bin wave, the first point corresponds to the minimum value of the first bin, the second point is the boundary between the two bins and the last point is the upper limit of the second bin. These flags override the corresponding bin specification set via /BINS, /XBMT, /YBMT, /ZBMT and /TBMT.
/Z [=*zval*]	Suppresses error reporting. /Z is equivalent to /Z=1 and /Z=0 is equivalent to not using the /Z flag at all.

Details

The input waves must be 1D real numeric waves. If one or more waves contain a non-finite value (a NaN or INF) the corresponding row of all waves are not counted in the joint histogram.

The optional waves that define user-specified bins must be real numeric waves and contain a monotonically increasing values. Using non-finite values in user-specified bin waves may lead to unpredictable results.

JointHistogram Binning Methods

The /XBMT, /YBMT, /ZBMT and /TBMT flags set the binning method for the X, Y, Z and T dimensions respectively.

These flags are overridden by /BINS if a non-zero value is specified for a given axis and by /XBWV, /YBWV, /ZBWV and /TBWV.

The *method* parameter is defined as follows:

method=0:	128 equally spaced bins between the minimum and maximum of the input data. This is the default setting.
method=1:	The number of bins is computed using Sturges' method where *numBins*=1+log2(N). N is the number of data points in each wave. The bins are distributed so that they include the minimum and maximum values.
method=2:	The number of bins is computed using Scott's method where the optimal bin width is given by *binWidth*=3.49*σ*$N^{-1/3}$. σ is the standard deviation of the distribution and N is the number of points. The bins are distributed so that they include the minimum and maximum values.
method=3:	Freedman-Daiconis method where *binWidth*=2*IQR*N-1/3, where IQR is the interquartile distance (see **StatsQuantiles**) and the bins are evenly distributed between the minimum and maximum values.

Bin selection methods are described at: http://en.wikipedia.org/wiki/Histogram

Example: 2D Joint Histogram

```
Make/O/N=(1000) xwave=gnoise(10), ywave=gnoise(5)
JointHistogram/BINS={20,30} xwave,ywave
NewImage M_JointHistogram
```

Example: 2D Joint Histogram using one bins wave

```
Make/O/N=(1000) xwave=gnoise(10), ywave=gnoise(5)
Make/O/N=3 xBinsWave={-8,0,14}
JointHistogram/BINS={0,30}/XBWV=xBinsWave/E xwave,ywave
Display; AppendImage/T M_JointHistogram vs {xBinsWave,*}
```

Example: 3D Joint Histogram

```
Make/O/N=(1000) xwave=gnoise(10), ywave=gnoise(5), zwave=enoise(4)
JointHistogram/BINS={15,15,20,0} xwave,ywave,zwave
NewImage M_JointHistogram
ModifyImage M_JointHistogram plane=10
```

See Also

Histogram, **ImageHistogram**

JulianToDate

`JulianToDate(`*`julianDay`*`, `*`format`*`)`

The JulianToDate function returns a date string containing the day, month, and year. The input *julianDay* is truncated to an integer.

Parameters

julianDay is the Julian day to be converted.

format specifies the format of the returned date string.

format	**Date String**
0	mm/dd/year
1	dd/mm/year
2	Tuesday November 15, 2002
3	year mm dd
4	year/mm/dd

See Also

The **dateToJulian** function.

For more information about the Julian calendar see:
`<http://www.tondering.dk/claus/calendar.html>`.

KillBackground

```
KillBackground
```

The KillBackground operation kills the unnamed background task.

KillBackground works only with the unnamed background task. New code should used named background tasks instead. See **Background Tasks** on page IV-298 for details.

Details

You can not call KillBackground from within the background function itself. However, if you return 1 from the background function, instead of the normal 0, Igor will terminate the background task.

See Also

The **BackgroundInfo**, **CtrlBackground**, **CtrlNamedBackground**, **SetBackground**, and **SetProcessSleep** operations; and **Background Tasks** on page IV-298.

KillControl

```
KillControl [/W=winName] controlName
```

The KillControl operation kills the named control in the top or specified graph or panel window or subwindow.

If the named control does not exist, KillControl does not complain.

Flags

/W=*winName*	Looks for the control in the named graph or panel window or subwindow. If /W is omitted, KillControl looks in the top graph or panel window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

KillDataFolder

`KillDataFolder` `[`**`/Z`**`]` ***`dataFolderSpec`***

The KillDataFolder operation kills the specified data folder and everything in it including other data folders.

However, if *dataFolderSpec* is the name of a data folder reference variable that refers to a free data folder, the variable is cleared and the data folder is killed only if this is the last reference to that free data folder.

Flags

/Z	No error reporting (except for setting V_flag). Does not halt function execution.

Parameters

dataFolderSpec can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

Details

If specified data folder is the current data folder or contains the current data folder then Igor makes its parent the new current data folder.

For legacy reasons, a null data folder is taken to be the current data folder. This can happen when using a $ expression where the string might possibly evaluate to "".

It is legal to kill the root data folder. In this case the root data folder itself is not killed but everything in it is killed.

KillDataFolder generates an error if any of the waves involved are in use. In this case, nothing is killed.

KillDataFolder generates an error if any of the waves involved are in use. In this case, nothing is killed. Execution ceases unless /Z is specified.

The variable V_flag is set to 0 when there is no error, otherwise it is an error code.

Examples

```
KillDataFolder foo          // Kills foo in the current data folder.
KillDataFolder :bar:foo     // Kills foo in bar in current data folder.
String str= "root:foo"
KillDataFolder $str         // Kills foo in the root data folder.
```

See Also

Chapter II-8, **Data Folders** and the **KillStrings**, **KillVariables**, and **KillWaves** operations.

KillFIFO

`KillFIFO` ***`FIFOName`***

The KillFIFO operation discards the named FIFO.

Details

FIFOs are used for data acquisition.

If there is an output or review file associated with the FIFO, KillFIFO closes the file. If the FIFO is used by an XOP, you should call the XOP to release the FIFO before killing it.

See Also

See **FIFOs and Charts** on page IV-291 for information about FIFOs and data acquisition.

KillFreeAxis

`KillFreeAxis` `[`**`/W=`*****`winName`***`]` ***`axisName`***

The KillFreeAxis operation removes a free axis specified by *axisName* from a graph window or subwindow.

Flags

/W=*winName*	Kills the free axis in the named graph window or subwindow. If /W is omitted, it acts on the top graph window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

Only an axis created by **NewFreeAxis** can be killed and only if no traces or images are attached to the axis.

See Also

The **NewFreeAxis** operation.

KillPath

`KillPath` `[`**`/A/Z`**`]` ***`pathName`***

The KillPath operation removes a path from the list of symbolic paths. KillPath is a newer name for the **RemovePath** operation.

Flags

/A	Kills all symbolic paths in the experiment except for the built-in paths. Omit *pathName* if you use /A.
/Z	Does not generate an error if a path to be killed is a built-in path or does not exist. To kill all paths in the experiment, use `KillPath/A/Z`.

Details

You can’t kill the built-in paths “home” and “Igor”.

See Also

The **NewPath** operation.

KillPICTs

`KillPICTs` `[`**`/A/Z`**`] [`***`PICTName`***` [, `***`PICTName`***`]…]`

The KillPICTs operation removes one or more named pictures from the current Igor experiment.

Flags

/A	Kills all pictures in the experiment.
/Z	Does not generate an error if a picture to be killed is in use or does not exist. To kill all pictures in the experiment, use `KillPICTs/A/Z`.

Details

You can not kill a picture that is used in a graph or page layout.

Warning: You *can* kill a picture that is referenced from a graph or layout recreation macro. If you do, the graph or layout can not be completely recreated. Use the Find dialog (Edit menu) to locate references in the procedure window to a named picture you want to kill.

See Also

See **Pictures** on page III-448 for general information on how Igor handles pictures.

KillStrings

`KillStrings [/A/Z] [`*`stringName`*` [, `*`stringName`*`]…]`

The KillStrings operation discards the named global strings.

Flags

/A	Kills all global strings in the current data folder. If you use /A, omit *stringName*.
/Z	Does not generate an error if a global string to be killed does not exist. To kill all global strings in the current data folder, use `KillStrings/A/Z`.

KillVariables

`KillVariables [/A/Z] [`*`variableName`*` [, `*`variableName`*`]…]`

The KillVariables operation discards the named global numeric variables.

Flags

/A	Kills all global variables in the current data folder. If you use /A, omit *variableName*.
/Z	Does not generate an error if a global variable to be killed does not exist. To kill all global variables in the current data folder, use `KillVariables/A/Z`.

KillWaves

`KillWaves [`*`flags`*`] `*`waveName`*` [, `*`waveName`*`]…`

The KillWaves operation destroys the named waves.

Flags

/A	Kills all waves in the current data folder. If you use /A, omit *waveName*s.
/F	Deletes the Igor binary file from which *waveName* was loaded.
/Z	Does not generate an error if a wave to be killed is in use or does not exist.

Details

The memory the waves occupied becomes available for other uses. You can’t kill a wave used in a graph or table or which is reserved by an XOP.

XOPs reserve a wave by sending the OBJINUSE message.

For functions compiled with the obsolete rtGlobals=0 setting, you also can't kill a wave referenced from a user-defined function.

Examples

```
KillWaves/A/Z       // kill waves not in use in current data folder
```

KillWindow

`KillWindow [`*`flags`*`] `*`winName`*

The KillWindow operation kills or closes a specified window or subwindow without saving a recreation macro.

Parameters

winName is the name of an existing window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Flags

/Z	Does not generate an error if the specified window does not exist.

See Also

The **DoWindow** operation.

KMeans

`KMeans [`*`flags`*`]` ***`populationWave`***

The KMeans operation analyzes the clustering of data in *populationWave* using an iterative algorithm. The result of KMeans is a specification of the classes which is saved in the wave M_KMClasses in the current data folder. Optional results include the distribution of class members (W_KMMembers) and the inter-class distances. *populationWave* is a 2D wave in which columns correspond to members of the population and rows contain the dimensional information.

Flags

/CAN	Analyzes the clustering by computing Euclidean distances between the means of the resulting classes. The resulting distances are stored in an NxN square matrix where N is the number of classes. Self distances (along the diagonal) or distances involving classes that did not survive the iterations are filled with NaN. Also saves the wave W_KMDispersion, which contains the sum of the distances between the center of each class and all its members. Distances are evaluated using the method specified by /DIST.
/DEAD=*method*	Specifies how the algorithm should handle "dead" classes, which are those that lose all members in a given iteration. *method*=1: Remove the class if it looses all members. *method*=2: Default; keeps the last value of the mean vector in case the class might get new members in a subsequent iteration. *method*=3: Assigns the class a random mean vector.
/DIST=*mode*	Specifies how the class distances are evaluated. *mode*=1: Distance is evaluated as the sum of the absolute values (also known as Manhattan distance). *mode*=2: Default; distance is evaluated as Euclidean distance.
/INIT=*method*	Specifies the initialization method. *method*=1: Random assignment of members of the population to a class. *method*=2: User-specified mean values (/INW). *method*=3: Default; initialize classes using values of a random selection from the population.
/INW=*iWave*	Sets the initial classes. The number of rows of *iWave* equals the dimensionality of the class and the number of columns of *iWave* is the number of classes. For example, if we want to initialize 5 classes in a problem that involves position in two dimensions then *iWave* must have 2 rows and 5 columns. The number of rows must also match the number of rows in *populationWave*.
/NCLS=*num*	Sets the number of classes in the data. If the initialization method uses specific means (/INIT=2) then the number of columns of *iWave* (see /INW) must match *num*. The default number of classes is 2.

/OUT=*format*	Specifies the format for the results. *format*=1: Output only the specification of the classes in the 2D wave M_KMClasses (default). Each column in M_KMClasses represents a class. The number of rows in M_KMClasses is equal to the number of rows in `populationWave+1`. The last row contains the number of class members. The remaining rows represent the center of the class. For example, if *populationWave* has two rows then the dimensionality of the problem is 2 and M_KMClasses has 3 rows with the first row containing the first components of each class center, the second row containing the second components of each class center and the third row containing the number of elements in each class. *format*=2: Output (in addition to M_KMClasses) the class membership in the wave W_KMMembers. The rows in this 1D wave correspond to sequential members of *populationWave* and the entries correspond to the (zero based) column number in M_KMClasses.
/SEED=*val*	Sets the seed for a new sequence in the pseudo-random number generator that is used by the operation. *val* must be an integer greater than zero. By changing the sequence you may be able to find new solutions or just make the process converge at a different rate.
/TER=*method*	Determines when the iterations stop. *method*=1: User-specified number of iterations (/TERN). *method*=2: Default; continue iterating until no more than a fixed number of elements change classes in one iteration (TERN).
/TERN=*num*	Specifies the termination number. The meaning of the number is determined by /TER above. By default, the termination *method*=2 and the default value of the maximum number of elements that change classes in one iteration is 5% of the size of the population.
/Z	No error reporting. If an error occurs, sets V_flag to -1 but does not halt function execution.

Details

KMeans uses an iterative algorithm to analyze the clustering of data. The algorithm is not guaranteed to find a global optimum (maximum likelihood solution) so the operation provides various flags to control when the iterations terminate. You can determine if the operation iterates a fixed number of times or loops until at most a specified maximum number of elements change class membership in a single iteration. If you are computing KMeans in more than one dimension you should pay attention to the relative magnitudes of the data in each dimension. For example, if your data is distributed on the interval [0,1] in the first dimension and on the interval [0,1e7] in the second dimension, the operation will be biased by the much larger magnitude of values in the second dimension.

Examples

Create data with 3 classes:

```
Make/O/N=(1,128) jack=4+gnoise(1)
jack[0][15,50]+=10
jack[0][60,]+=20
```

Perform KMeans looking for 5 classes:

```
KMeans/init=1/out=1/ter=1/dead=1/tern=1000/ncls=5 jack
Print M_KMClasses
```

```
  M_KMClasses[0][0]= {24.1439,68}
  M_KMClasses[0][1]= {14.1026,36}
  M_KMClasses[0][2]= {4.01537,24}
```

See Also

The **FPClustering** function.

Label

`Label [/W=`*`winName`*`/Z] `*`axisName, labelStr`*

The Label operation labels the named axis with *labelStr*.

Parameters

axisName is the name of an existing axis in the top graph. It is usually one of “left”, “right”, “top” or “bottom”, though it may also be the name of a free axis such as “VertCrossing”.

labelStr contains the text that labels the axis.

Flags

/W=*winName*	Adds axis label in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/Z	No errors generated if the named axis doesn’t exist. Used for style macros.

Details

labelStr can contain escape codes which affect subsequent characters in the text. An escape code is introduced by a backslash character. In a literal string, you must enter two backslashes to produce one. See **Backslashes in Annotation Escape Sequences** on page III-57 for details.

Using escape codes you can change the font, size, style and color of text, create superscripts and subscripts, create dynamically-updated text, insert legend symbols, and apply other effects. See **Annotation Escape Codes** on page III-53 for details.

Some escape codes insert text based on axis properties. See **Axis Label Escape Codes** on page III-57 for details.

The characters “<??>” in an axis label indicate that you specified an invalid escape code or used a font that is not available.

See Also

See **Annotation Escape Codes** on page III-53. See the **Legend** operation about wave symbols.

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

laguerre

`laguerre(`*`n, x`*`)`

The laguerre function returns the Laguerre polynomial of degree *n* (positive integer) and argument *x*. The polynomials satisfy the recurrence relation:

$$(n+1)\mathrm{Laguerre}(n+1,x) = (2n+1-x)\mathrm{Laguerre}(n,x) - n\,\mathrm{Laguerre}(n-1,x),$$

with the initial conditions

$$\mathrm{Laguerre}(0,x) = 1$$

and

$$\mathrm{Laguerre}(1,x) = 1-x.$$

See Also

The **laguerreA**, **laguerreGauss**, **chebyshev**, **chebyshevU**, **hermite**, **hermiteGauss**, and **legendreA** functions.

laguerreA

laguerreA(*n*, *k*, *x*)

The laguerreA function returns the associated Laguerre polynomial of degree *n* (positive integer), index *k* (non-negative integer) and argument *x*. The associated Laguerre polynomials are defined by

$$L_n^k(x) = (-1)^k \frac{d^k}{dx^k}\left[L_{n+k}(x)\right],$$

where $L_{n+k}(x)$ is the Laguerre polynomial.

See Also

The **laguerre** and **laguerreGauss** functions.

References

Arfken, G., *Mathematical Methods for Physicists*, Academic Press, New York, 1985.

laguerreGauss

laguerreGauss(*p*, *m*, *r*)

The laguerreGauss function returns the normalized product of the associated Laguerre polynomials and a Gaussian. This function is typically encountered in solutions to physical problems where it represents the radial solution with an additional factor exp(i*m*ϕ) which is not included in this case. The LaguerreGauss is given by

$$U_{pm}(r) = \sqrt{\frac{2p!}{\pi(m+p)!}}\left(r\sqrt{2}\right)^m L_p^m\left(2r^2\right)\exp\left(-r^2\right).$$

See Also

The **laguerre**, **laguerreA**, and **hermiteGauss** functions.

LambertW

LambertW(*z*, *branch*)

The LambertW function returns the complex value of Lambert's W function for complex *z* and integer index *branch*. The function can be defined through its inverse,

$$z = w\mathrm{e}^w .$$

Since w is multivalued, the branch parameter is used to differentiate between solutions for the equation.

The LambertW function was added in Igor Pro 7.00.

Details

IGOR's LambertW uses complex input and output. You can use LambertW in real expressions but you must make sure that you are not calling the function in a range where its imaginary part is non-zero.

The average accuracy of the function defined by cabs(*z*-w*exp(w)) in the region |real(*z*)|<10, |imag(*z*)|<10 is 5e-14. In general the accuracy decreases with increasing |branch| and with increasing distance from the origin in the z-plane.

IGOR uses a hybrid algorithm to compute the function which requires longer computation times in the presence of numerical instabilities.

References

R.M. Corless, G.H. Gonnet, D.E.G. Hare, D.J. Jeffrey and D.E. Knuth, "On Lambert W Function", Advances in Computational Mathematics 5: 329-359

Layout

`Layout [`*`flags`*`] [`*`objectSpec`*` [, `*`objectSpec`*`]…][as `*`titleStr`*`]`

The Layout operation creates a page layout.

Note: The Layout operation is antiquated and can not be used in user-defined functions. For new programming, use the **NewLayout** operation instead.

Parameters

All of the parameters are optional.

Each *objectSpec* parameter identifies a graph, table, textbox or picture to be added to the layout. An object specification can also specify the location and size of the object, whether the object should have a frame or not, whether it should be transparent or opaque, and whether it should be displayed in high fidelity or not. See **Details**.

titleStr is a string expression containing the layout's title. If not specified, Igor will provide one which identifies the objects displayed in the graph.

Flags

/A=(*rows*,*cols*)	Specifies rows and columns for tiling or stacking.
/B=(*r*,*g*,*b*)	Specifies the background color for the layout. *r*, *g*, and *b* are integers from 0 to 65535. Defaults to white (65535,65535, 65535).
/C=*colorOnScreen*	Obsolete. In ancient times, this flag switched the screen display of the layout between black and white and color. It is still accepted but has no effect.
/G=*g*	Specifies grout, the spacing between tiled objects. Units are points unless /I, /M, or /R are specified.
/HIDE=*h*	Hides (h = 1) or shows (h = 0, default) the window.
/I	Specifies that coordinates are in inches. This affects subsequent /G, /W, and *objectSpec* coordinates. Coordinates are relative to the top/left corner of the paper.
/K=*k*	Specifies window behavior when the user attempts to close it. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. *k*=3: Hides the window. If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.
/M	Specifies that coordinates are in centimeters. This affects subsequent /G, /W, and *objectSpec* coordinates. Coordinates are relative to the top/left corner of the paper.
/P=*orientation*	*orientation* is either Portrait or Landscape (*e.g.*, `Layout/P= Landscape`). This controls the orientation of the page in the layout. See **Details**. If you use the /P flag, you should make it the first flag in the Layout operation. This is necessary because the orientation of the page affects the behavior of other flags, such as /T and /G.
/R	Specifies that coordinates are in percent. This affects subsequent /G, /W, and *objectSpec* coordinates. For /W, coordinates are as a percent of the main screen. For /G and *objectSpec*, coordinates are relative to the top/left corner of the printing part of the page.
/S	Stacks objects.
/T	Tiles objects.

/W=(*left, top, right, bottom*)

Gives the layout window a specific location and size on the screen. Coordinates for /W are in points unless /I or /M are specified.

Details

When you create a new page layout window, if preferences are enabled, the page size is determined by the preferred page size as set via the Capture Layout Prefs dialog. If preferences are disabled, as is usually the case when executing a procedure, the page is set to the factory default size.

If you use the /P flag, you should make it the first flag in the Layout operation. This is necessary because the orientation of the page affects the behavior of other flags, such as /T and /G.

The form of an *objectSpec* is:

objectName [**(*objLeft, objTop, objRight, objBottom*)**] [**/O=*objType***] [**/F=*frame***]
    [**/T=*trans***] [**/D=*fidelity***]

objectName can be the name of an existing graph, table or picture. It can also be the name of an object that does not yet exist. In this case it is called a “dummy object”.

objectSpec can be specified using a string by using the $ operator, but the entire *objectSpec* must be in the string.

Here are some examples of valid usage:

```
Layout Graph0
Layout/I Graph0(1, 1, 6, 5)/F=1
String s = "Graph0"
Layout/I $s
String s = "Graph0(1, 1, 6, 5)/F=1"
Layout/I $s         // Entire object spec is in string.
```

The object’s coordinates are determined as follows:

- If *objectName* is followed by a coordinates specification in (*objLeft*, *objTop*, *objRight*, *objBottom*) form then this sets the object’s coordinates. The units for the coordinates are points unless the /I or /M flag was present in which case the units are inches or centimeters respectively.
- If the object coordinates are not specified explicitly but the Layout/S flag was present then the object is stacked. If the Layout/T flag was present then the object is tiled, and if the Layout/A=(*rows*,*cols*) flag is present, tiling is performed using that number of rows and columns.
- If the object’s coordinates are not determined by these rules then the object is set to a default size and is stacked.

Each object has a type (graph, table, textbox or picture) determined as follows:

O=*objType*	If the *objectName*/O=*objType* flag is present then it determines the object’s type:
	objType=1: Graph.
	objType=2: Table.
	objType=8: Picture.
	objType=32: Textbox.

If there is no /O flag and *objectName* is the name of an existing graph, table or picture, then the object type is graph, table or picture.

If the object’s type is not determined by the above rules and *objectName* contains “Table”, “PICT”, or “TextBox”, then the object type is table, picture or textbox.

If the object’s type is not specified by any of the above rules, it is taken to be a graph type object.

The remaining flags have the following meanings:

/D=*fidelity*	Controls the drawing of the layout object:
	fidelity=0: Low fidelity display.
	fidelity=1: High fidelity display (default).

/F=*frame*	Controls the object frame:
	frame=0: No frame.
	frame=1: Single frame (default).
	frame=2: Double frame.
	frame=3: Triple frame.
	frame=4: Shadow frame.
/T=*trans*	Controls the transparency of the layout object:
	trans=0: Opaque (default).
	trans=1: Transparent. For this to be effective, the object itself must also be transparent. Annotations have their own transparent/opaque settings. Graphs are transparent only if their backgrounds are white. Pictures may have been created transparent or opaque, and Igor cannot make an inherently opaque picture transparent.

See Also

The **NewLayout** and **LayoutInfo** operations. See Chapter II-17, **Page Layouts**.

Layout

`Layout`

Layout is a procedure subtype keyword that identifies a macro as being a page layout recreation macro. It is automatically used when Igor creates a window recreation macro for a layout. See **Procedure Subtypes** on page IV-193 and **Killing and Recreating a Layout** on page II-390 for details.

See Also

See Chapter II-17, **Page Layouts**.

LayoutInfo

`LayoutInfo(`*`winNameStr`*`, `*`itemNameStr`*`)`

The LayoutInfo function returns a string containing a semicolon-separated list of keywords and values that describe an object in the active page of a page layout or overall properties of the layout. The main purpose of LayoutInfo is to allow an advanced Igor programmer to write a procedure which formats or arranges objects.

winNameStr is the name of an existing page layout window or `""` to refer to the top layout.

itemNameStr is a string expression containing one of the following:

- The name (e.g., "Graph0") of a layout object in the active page to get information about that object.
- An object instance (e.g., "Graph0#0" or "Graph0#1") to get information about a particular instance of an object in the active page. This is of use only in the unusual situation when the same object appears in the active page multiple times. "Graph0#0" is equivalent to "Graph0". "Graph0#1" is the second occurrence of Graph0 in the active page.
- An integer object index starting from zero to get information about an object referenced by its position in the active page in the layout. Zero refers to the first object going from back to front in the page.
- The word "Layout" to get overall information about the layout.

Details

In cases 1, 2 and 3 above, where *itemNameStr* references an object, the returned string contains the following keywords, with a semicolon after each keyword-value pair.

Keyword	Information Following Keyword
`FIDELITY`	Object fidelity expressed as a code usable in a ModifyLayout fidelity command.
`FRAME`	Object frame expressed as a code usable in a ModifyLayout frame command.
`HEIGHT`	Object height in points.

Keyword	Information Following Keyword
`INDEX`	Object position in back-to-front order in the active page of the layout, starting from zero.
`LEFT`	Object left position in points.
`NAME`	The name of the object.
`SELECTED`	Zero if the object is not selected or nonzero if it is selected. You can identify the first-selected object by examining the SELECTED code of all objects in the page. The one with the smallest nonzero selected code is the object that was first selected.
`TOP`	Object top position in points.
`TRANS`	Object transparency expressed as a code usable in a ModifyLayout trans command.
`TYPE`	Object type which is one of: Graph, Table, Picture, or Textbox.
`WIDTH`	Object width in points.

In case 4 above, where *itemNameStr* is "Layout", the returned string contains the following keywords, with a semicolon after each keyword-value pair.

Keyword	Information Following Keyword
`BGRGB`	Layout background color expressed as <red>, <green>, <blue> where each color is a value from 0 to 65535.
`MAG`	Layout magnification: 0.25, 0.5, 1.0, or 2.0.
`NUMOBJECTS`	Total number of objects in the active page of the layout.
`NUMSELECTED`	Number of selected objects in the active page of the layout.
`PAGE`	A rectangle defining the part of the paper that is inside the margins, expressed in points. The format is <left>, <top>, <right>, <bottom>.
`CURRENTPAGENUM`	One-based page number of the currently active page. Added in Igor Pro 7.00.
`NUMPAGES`	Total number of pages in the layout. Added in Igor Pro 7.00.
`PAPER`	A rectangle defining the bounds of the paper, expressed in points. The format is <left>, <top>, <right>, <bottom>.
`SELECTED`	A comma-separated list of the names of selected objects in the active page of the layout.
`UNITS`	Units used to display object locations and sizes. This will be one of the following: 0 for points, 1 for inches, 2 for centimeters.

LayoutInfo returns `""` in the following situations:

- *winNameStr* is `""` and there are no layout windows.
- *winNameStr* is a name but there are no layout windows with that name.
- *itemNameStr* is not "Layout" and is not the name or index of an existing object.

Examples

This example sets the background color of all selected graphs in the active page of a particular page layout to the color specified by `red`, `green`, and `blue`, which are numbers from 0 to 65535.

```
Function SetLayoutGraphsBackgroundColor(layoutName,red,green,blue)
   String layoutName       // Name of layout or "" for top layout.
   Variable red, green, blue

   Variable index
   String info
   Variable selected
   String indexStr
   String objectTypeStr
   String graphNameStr

   index = 0
   do
      sprintf indexStr, "%d", index
      info = LayoutInfo(layoutName, indexStr)
      if (strlen(info) == 0)
         break     // No more objects
      endif

      selected = NumberByKey("SELECTED", info)
      if (selected)
         objectTypeStr = StringByKey("TYPE", info)
         if (CmpStr(objectTypeStr,"Graph") == 0)// This is a graph?
            graphNameStr = StringByKey("NAME", info)
            ModifyGraph/W=$graphNameStr wbRGB=(red,green,blue)
            ModifyGraph/W=$graphNameStr gbRGB=(red,green,blue)
         endif
      endif

      index += 1
   while(1)
End
```

See Also

The **Layout** operation. See Chapter II-17, **Page Layouts**.

LayoutMarquee

`LayoutMarquee`

LayoutMarquee is a procedure subtype keyword that puts the name of the procedure in the layout Marquee menu. See **Marquee Menu as Input Device** on page IV-151 for details.

See Also

See Chapter II-17, **Page Layouts**.

LayoutPageAction

`LayoutPageAction [/W=`*`winName`*`]` `[`***`keyword = value`***` [, `***`keyword = value`*** `…]]`

The LayoutPageAction operation adds, deletes, reorders, or adjusts the sizes of layout pages.

The LayoutPageAction operation was added in Igor Pro 7.00.

Parameters

appendPage	Appends a new page.
insertPage=*page*	Inserts a new page before *page*. Page numbers start from 1. Pass 0 for *page* to insert before the first page.
page=*page*	Makes *page* the active page. Page numbers start from 1.
deletePage=*page*	Deletes *page*. This action cannot be undone. Page numbers start from 1.
reorderPages={*anchorPage*, *page1*, ...}	
	Reorders the pages so that *page1* and any others appear before *anchorPage*, in the same order as their appearance in the command. Page numbers start from 1.
size=(*width*, *height*)	Sets the global page dimensions for the layout to width and height, specified in units of points.
size(*page*)=(*width*, *height*)	

Sets the dimensions of *page* to *width* and *height*, specified in units of points.

Using this keyword with *page* set to -1 modifies the global page dimensions for the layout.

margins=(*leftMargin*, *topMargin*, *rightMargin*, *bottomMargin*)

Sets the global page margins for the layout to the specified values, expressed in units of points.

margins(*page*)=(*leftMargin*, *topMargin*, *rightMargin*, *bottomMargin*)

Sets the margins of specified page to these values, expressed in units of points.

Page numbers start from 1.

Passing -1 for page sets the global margins for the layout.

Flags

/W=*winName*	Modifies the named layout. When omitted, the actions affect the top layout.

Details

Page numbers starts from 1. Use *page*=0 to refer to the active page.

The layout as a whole has a size and margins. These are called "global" dimensions and govern all pages by default. You can set the global dimensions using the size and margins keyword without specifying a particular page.

You can override the dimensions for a given page using size(*page*) and margins(*page*) to specify custom dimensions.

Use size(*page*)=(0,0) to revert the specified page to the global layout dimensions. This reverts both the page size and its margins.

See Also

Page Layouts on page II-387, **NewLayout**, **ModifyLayout**

LayoutSlideShow

`LayoutSlideShow [/W=`*`winName`*`]` `[`***`keyword = value`***` [, `***`keyword = value`***` …]]`

The LayoutSlideShow operation starts, stops, or modifies a slideshow that displays the pages of a page layout.

The LayoutSlideShow operation was added in Igor Pro 7.00.

Parameters

autoMode=*a* — Controls whether the presentation will advance between slides automatically (*a*=1) or manually (*a*=0). Use the delay keyword to control the delay between automatic transitions.

delay=*d* — *d* is the number of seconds to wait between slide transitions when running in auto mode.

otherScreenContents=*o*

Controls what is displayed on any additional screens that may be connected.

o=0:	Other screens show the presentation.
o=1:	Other screens show a presenter's view with additional information. Use the presentersView keyword to control the contents of this view.
o=2:	Other screens show a presenter's view with additional information. Use the presentersView keyword to control the contents of this view.

page=*p* — Causes the slideshow to start from page *p*. *p* is a page number starting from 1. This keyword has no effect unless the start keyword is also present.

presentersView=*p*	Controls what is displayed on the screens that show the presenter's view. *p* is a bitfield of flags: Bit 0: Show the next page. Bit 1: Show the current time. Bit 2: Show the elapsed time. **Setting Bit Parameters** on page IV-12 for details about bit settings.
scaleMode=*s*	Specifies how the pages are scaled to fit the screen. *s*=0: No scaling. Pages are drawn at actual size even if they are much larger or smaller than the screen size. *s*=1: All pages are individually scaled to the screen size. *s*=2: All pages are scaled by the same factor so that the largest page fits on the screen. This preserves the relative sizes of the pages.
screen=*s*	Specifies the screen to be used for the main presentation. Use *s*=1 to use the primary screen. Use **IgorInfo** to determine the number of available screens.
start	Starts the slideshow.
stop	Stops the slideshow. You can also stop it by pressing the escape key.
wrapMode=*w*	Controls what happens when the presentation reaches the last page in the slideshow. *w*=0: Advancing to the next page has no effect. *w*=1: Advancing to the next page causes the slideshow to wrap around to the first page. *w*=2: Advancing to the next page causes the slideshow to stop.

Flags

/W= *winName*	*winName* is the name of the desired layout window. If /W is omitted or if *winName* is $"", the top layout window is used.

Details

A layout slide show can be used to present an Igor experiment to others, or to run an information kiosk.

Any changes to the layout window during a slide show are automatically reflected in the slides. For example you could use a background task to update a graph so that the slides always show the latest data.

You can control a running slide show by right-clicking on the slideshow. Alternatively, use the arrow keys or a mouse click to advance to the next slide.

Press the space bar to toggle between automatic and manual advancing of the slides. Press escape to end the slideshow.

Example

```
Function DemoSlideshow()      // Press escape to end the slideshow
   NewLayout
   TextBox/C/N=text0/F=0/A=LB/X=33.57/Y=70.81 "\\Z961"
   LayoutPageAction appendpage
   TextBox/C/N=text0/F=0/A=LB/X=33.57/Y=70.81 "\\Z962"
   LayoutPageAction appendpage
   TextBox/C/N=text0/F=0/A=LB/X=33.57/Y=70.81 "\\Z963"
   LayoutSlideShow autoMode=1,delay=1,page=1,wrapMode=1,start
End
```

See Also

Page Layouts on page II-387, **NewLayout**, **LayoutPageAction**

LayoutStyle

`LayoutStyle`

LayoutStyle is a procedure subtype keyword that puts the name of the procedure in the Style pop-up menu of the New Layout dialog and in the Layout Macros menu. See **Page Layout Style Macros** on page II-409 for details.

See Also

See Chapter II-17, **Page Layouts** and **Page Layout Style Macros** on page II-409.

leftx

`leftx(`*`waveName`*`)`

The leftx function returns the X value of point 0 (the first point) of the named 1D wave. The leftx function is not multidimensional aware. The multidimensional equivalent of this function is **DimOffset**.

Details

Point 0 contains a wave's *first* value, which is usually the leftmost point when displayed in a graph. Leftx returns the value elsewhere called *x0*. The function DimOffset returns any of x0, y0, z0, or t0, for dimensions 0, 1, 2, or 3.

See Also

The **deltax** and **rightx** functions.

For multidimensional waves, see **DimDelta**, **DimOffset**, and **DimSize**.

For an explanation of waves and X scaling, see **Changing Dimension and Data Scaling** on page II-63.

Legend

`Legend [`*`flags`*`] [`*`legendStr`*`]`

The Legend operation puts a legend on a graph or page layout.

Parameters

legendStr contains the text that is printed in the legend.

If *legendStr* is missing or is an empty string (`""`), the text needed for a default legend is automatically generated. Legends are automatically updated when waves are appended to or removed from the graph or when you rename a wave in the graph.

legendStr can contain escape codes which affect subsequent characters in the text. An escape code is introduced by a backslash character. In a literal string, you must enter two backslashes to produce one. See **Backslashes in Annotation Escape Sequences** on page III-57 for details.

Using escape codes you can change the font, size, style and color of text, create superscripts and subscripts, create dynamically-updated text, insert legend symbols, and apply other effects. See **Annotation Escape Codes** on page III-53 for details. However normally you leave it to Igor to automatically manage the legend.

See **Legend Text** on page III-43 for a discussion of what *legendStr* may contain.

Flags

/H=*legendSymbolWidth*

Sets the width in points of the area in which to draw the wave symbols. A value of 0 means "default". This results in a width that is based on the text size in effect when the symbol is drawn. A value of 36 gives a 0.5 inch (36 points) width which is nice in most cases.

/H={*legendSymbolWidth*, *minThickness*, *maxThickness*}

	This is an additional form of the /H flag. The *legendSymbolWidth* parameter works the same as described above.
	The *minThickness* and *maxThickness* parameters allow you to create a legend whose line and marker thicknesses are different from the thicknesses of the associated traces in the graph. This can be handy to make the legend more readable when you use very thin lines or markers for the traces.
	minThickness and *maxThickness* are values from 0.0 to 10.0. Also, setting *minThickness* to 0.0 and *maxThickness* to 0.0 (default) uses the same thicknesses for the legend symbols as for the traces.
/J	Disables the default legend mechanism so that a default legend is not created even if *legendStr* is an empty string ("") or omitted.
	Window recreation macros use /J in case *legendStr* is too long to fit on the same command line as the Legend operation itself. In this case, an AppendText command appears after the Legend command to append *legendStr* to the empty legend. For really long values of *legendStr*, there may be multiple AppendText commands.
/M[=*saMeSize*]	/M or /M=1 specifies that legend markers should be the same size as the marker in the graph.
	/M=0 turns same-size mode off so that the size of the marker in the legend is based on text size.

See the **TextBox** operation for documentation for all other flags.

Examples

The command `Legend` (with no parameters) creates a default legend. A default legend in a layout contains a line for each wave in each of the graphs in the layout, starting from the bottom graph and working toward the front.

The command:

```
Legend/C/N=name ""
```

changes the named existing legend to a default legend.

You can put a legend in a page layout with a command such as:

```
Legend "\s(Graph0.wave0) this is wave0"
```

This creates a legend in the layout that shows the symbol for wave0 in Graph0. The graph named in the command is usually in the layout but it doesn't have to be.

See Also

TextBox, **Tag**, **ColorScale**, **AnnotationInfo**, **AnnotationList**.

Annotation Escape Codes on page III-53.

Legend Text on page III-43.

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

Color as f(z) Legend Example on page II-230 for a discussion of creating a legend whose symbols match the markers in a graph that uses color as f(z).

legendreA

`legendreA(n, m, x)`

The legendreA function returns the associated Legendre polynomial:

$$P_n^m(x)$$

where n and m are integers such that $0 \le m \le n$ and $|x| \le 1$.

References

Arfken, G., *Mathematical Methods for Physicists*, Academic Press, New York, 1985.

limit

`limit(`*`num`*`, `*`low`*`, `*`high`*`)`

The limit function returns *num*, limited to the range from *low* to *high*:

num if *low* <= *num* <= *high*.

low if *num* < *low*.

high if *num* > *high*.

Since all comparisons with NaN return false, limit will not work as expected with NaNs. If a parameter may be NaN, use **numtype** to test it before calling limit.

See Also

SelectNumber, **min**, **max**

LinearFeedbackShiftRegister

`LinearFeedbackShiftRegister` `[`***`flags`***`]`

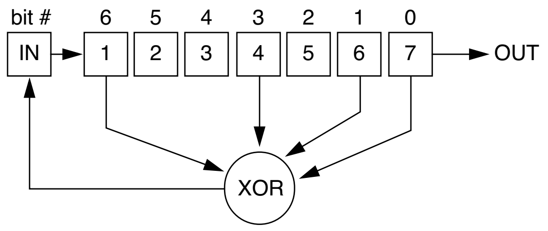
The LinearFeedbackShiftRegister operation implements a, well, linear feedback shift register, or LFSR. A LFSR is a way to produce a sequence of very bad pseudorandom numbers, or a random bit stream (that is, a random sequence of zeroes of ones that over time are nearly equal in number).

If it produces bad random numbers, why would I want to use a LFSR? A properly-configured LFSR will create a “maximal-length sequence”: a LFSR of N bits will produce 2^N-1 numbers in a quasi-random sequence without repeating. That is, it will produce all the N-bit numbers except zero. This gives the sequence good spectral properties for certain applications, and, taking the least-significant bit as the output, it creates a pseudorandom bit stream with nearly equal numbers of zeroes and ones (*nearly* means one more one than zeroes).

The LinearFeedbackShiftRegister operation generates either a wave full of the sequential states of the shift register or a wave full of ones and zeroes representing the least significant bit of the shift register.

Linear Feedback Shift Registers

A LFSR is a shift register with taps. The tap bits are XOR’ed together and the result, after the register is shifted, becomes the new most significant bit. Here is a diagram of a 7-bit LFSR:



Each successive number is generated by shifting the contents of the register (boxes 1-7) to the right, while shifting in the output of the XOR node. The XOR node samples specified bits of the register contents, generating its output ready to be shifted in. Thus, the inputs of the XOR are bits sampled *before* a shift; the output of the XOR becomes the leading bit in the register *after* a shift.

In many applications the output of interest is the stream of bits that appear in the last position. This stream of bits is a pseudorandom sequence of ones and zeroes (or ones and minus ones, or whatever other binary sequence you need).

The bits fed into the XOR node are referred to as *taps*. The taps illustrated here would be specified with the tap list 7,6,4,1. As implemented in Igor Pro, the output tap (tap 7 in the illustration) is the least significant bit, so an alternate way to express the tap list is as the binary number 1001011_2 (77_{10}).

With the right taps, a LFSR produces a maximal-length sequence. The list of sequential states in a maximal-length sequence has length 2^N-1 without repeating a state. That means that every possible N-bit nonzero number appears exactly once in the maximal-length sequence.

Maximal-length tap lists always have an even number of taps.

If you have a tap list that gives a maximal-length sequence, you can generate another tap list from it. If your tap list is (n, A, B, C) the new tap list is (n, n-C, n-B, n-A). This new tap list will generate a bit stream that is the mirror image in time of the bit stream produced by the first tap list.

Flags

/DEST=*wavename*	Specifies a wave to receive the generated sequence. With /MODE=0, the number type of the wave must have at least *nbits* bits for an integer wave, or at least an *nbit* mantissa if it is a floating-point wave. That is, /N=25 requires a double-precision wave or a 32-bit integer wave. /N=18 requires any floating-point wave or a 16- or 32-bit integer wave. If you use an integer wave, we recommend an unsigned integer wave for /N=8, 16, or 32. If /MODE=1 is used, any number type is acceptable. See the Details for what happens if you don't use /DEST. If *wavename* doesn't exist, a suitable integer wave will be made. If *wavename* already exists, LinearFeedbackShiftRegister will use it as-is. The sequence length will be taken from the wave. If the number type of the wave is not suitable, an error is issued. If the sequence length is less than the number of points in your wave, it will be truncated to match.
/FREE	In a user-defined function, makes a free wave. See **Free Waves** on page IV-84 for details.
/INIT=*initialValue*	Sets the initial value of the shift register to *initialValue*. This will also be the first value in the output for /MODE=0, or the least-significant bit of *initialValue* will be the first output for /MODE=1. You can use this initial value to restart a very long sequence from the last state of a previous run. Default is a single 1 bit in the first position (bit *nbits*-1 for /N=*nbits*).
/LEN=*length*	Sets the length of sequence to generate. If the sequence repeats before *length* states are generated, the sequence is terminated early. If *length* is larger than the number of states in a maximal-length sequence, you will get a maximal-length sequence, or a shorter sequence if the initial value is seen again (that is, your sequence is not a maximal-length sequence). You can specify *length* greater than the maximal-length sequence length, but it will be truncated to the maximal length.
/MAX=*index*	An internal table of tap lists gives maximal-length sequences. This table has up to 32 tap lists for each value of *nbits*. You select a tap list by setting index to a number from 0 to 31. For values of *nbits* that do not have 32 maximal-length tap lists, the table repeats. Most *nbits* values have many more than 32 possible maximal-length sequences. For each tap list in the table, another tap list can be accessed using the /MROR flag.
/MODE=*doBitStream*	Sets the output stream format. *doBitStream*=0: Succession of bit register states (default). *doBitStream*=1: Stream of ones and zeroes.
/MROR [=*doMirror*]	Transforms the tap list into its complementary tap list, creating a mirror-image bit stream, when you use /MROR or *doMirror*=1. Specify the tap list using /TAPS, /TAPB, or /MAX.
/N=*nbits*	Determines the number of bits in the shift register. A maximal-length sequence will have 2^{nbits}-1 states. *nbits* must be in the range of 1-32. Note that *nbits* = 1 or 2 is not very interesting.
/STOP=*stopValue*	Terminates the sequence when *stopValue* is the next shift register value. You can use this flag to generate long sequences using multiple calls to LinearFeedbackShiftRegister by storing the initial value of the first call, and setting *stopValue* to that initial value in subsequent calls.
/TAPB=*tapbits*	An alternate way to express the tap list. *tapbits* is a number in which each bit represents a tap, with bit 0 representing the tap with tap number *nbits*.
/TAPS={*t1*, *t2*, …}	Specifies the tap list. Tap numbers are in the range from 1 to *nbits*.

Details

If the /TAPS, /TAPB, or /MAX flags are absent, the maximal-length sequence corresponding to /MAX=0 is generated.

In you omit the /DEST flag, a wave named W_LFSR will be generated for you. W_LFSR is an unsigned integer wave with number type set to the minimum size for the shift register size and /MODE setting. Thus, if you set `/N=10/MODE=0`, W_LFSR will be an unsigned 16-bit integer wave.

Because W_LFSR is an unsigned integer wave, you will need to redimension the wave to a floating-point wave for many purposes. Use the **Redimension** operation or the Redimension Waves item in the Data menu.

Up to /N=18, W_LFSR will initially be created large enough to hold a maximal-length sequence, unless you request a shorter sequence using the /LEN flag. If the initial value is seen again before a maximal-length sequence is generated, it means that the tap list specified was not one that generates a maximal-length sequence, and generation is terminated. The wave is shortened to the generated sequence length.

If you set a register size greater than /N=18 and you do not use /LEN, the generated sequence will stop after 2^{18}-1 (262143) states. Note that beyond some N, it will be impossible to create a wave large enough to hold a maximal-length sequence.

Some tap lists do not generate maximal-length sequences but also do not repeat the initial value. In that case, the generated sequence will be of maximal length but will contain repeated subsequences. The V_flag variable will be set to 0 if the sequence was not a maximal-length sequence, or 1 if it was. If /LEN=*length* values were generated, V_flag is set to 2.

If you specify your own wave using /DEST, the sequence length will be the same as the length of your wave. Your wave will be resized if a shorter sequence is generated.

Generating Long Sequences in Smaller Segments

Very long maximal-length sequences will not fit in the largest wave you can make. It may also be more convenient to make multiple, small fragments of a longer sequence. You can do this using the /INIT, /STOP, and /LEN flags, along with the V_nextValue variable. Here is an example of making 1000-point subsequences from a 16-bit maximal-length sequence:

```
// Start with the first 1000 states, with initial value of 1
LinearFeedbackShiftRegister/N=16/LEN=1000/INIT=1
// Restart using the V_nextValue variable to continue the sequence
// /STOP=1 sets the stopping value to the first initial value
LinearFeedbackShiftRegister/N=16/LEN=1000/INIT=(V_nextValue)/STOP=1
// Continue…
LinearFeedbackShiftRegister/N=16/LEN=1000/INIT=(V_nextValue)/STOP=1
```

Variables

The LinearFeedbackShiftRegister operation returns information in the following variables:

`V_flag`	Set to zero when a nonmaximal-length sequence was detected. This occurs if the initial value is seen again before a maximal-length sequence was generated, or if a maximal-length sequence was generated but the final state was not the same as the initial state. Set to 1 when a maximal-length sequence was generated. Set to 2 when the sequence was limited by /LEN=*length* or by the default limit of 2^{18}-1 (262143) states.
`V_tapValue`	Set to the binary representation of the tap sequence used. That is, you can generate the same sequence using `/TAPB=V_tapValue`. If you use /MROR=1, V_tapValue reflects that setting. It will also give the actual tap value used when you specify the a maximal-length sequence using the /MAX flag.
`V_nextValue`	Set to the next value beyond the last generated register state. This can be used to restart a truncated sequence.

Examples

Generate a 16-bit maximal-length sequence and reprocess the output values to be centered on zero and normalized to a maximum value of 1:

```
LinearFeedbackShiftRegister/N=16
Redimension/D W_LFSR
```

```
W_LFSR -= 2^15
W_LFSR /= 2^15-1
```

Another way to do the same thing that avoids the Redimension operation, which could lead to fragmentation of memory:

```
Make/D/N=(2^16-1) LFSR_output
LinearFeedbackShiftRegister/N=16/DEST=LFSR_output
LFSR_output -= 2^15
LFSR_output /= 2^15-1
```

Make a bit stream with random +1 and -1 instead of 0 and 1:

```
LinearFeedbackShiftRegister/N=16/MODE=1
Redimension/B W_LFSR
W_LFSR = W_LFSR*2-1
```

See Also

If you really need random numbers, we provide high-quality RNG's that return random deviates from a number of distributions. See **enoise**, **gnoise**, and others.

References

A discussion of LFSR's can be found in "Generation of Random Bits" (Section 7.4) in Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992. They refer to "primitive polynomials modulo 2" and do not use the name Linear Feedback Shift Register, but it is the same thing. We use an implementation equivalent to their Method I.

ListBox

`ListBox [/Z] *ctrlName* [*keyword = value* [, *keyword = value* …]]`

The ListBox operation creates or modifies the named control that displays, in the target window, a list from which the user can select any number of items.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the ListBox control to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **Button** and **DefaultGUIControls** for more appearance details.

clickEventModifiers=*modifierSelector*

Selects modifier keys to ignore when processing clicks to start editing a cell or when toggling a checkbox. That is, use this keyword if you want to prevent a shift-click (for instance) from togging checkbox cells. Allows the action procedure to receive mousedown events with those modifiers without interfering actions on the part of the listbox control.

modifierSelector is a bit pattern with a bit for each modifier key; sum these values to get the desired combination of modifiers:

modifierSelector=1:	Control key (Macintosh only)
modifierSelector=2:	Option (Macintosh) or Alt (Windows)
modifierSelector=4:	Context click (right click on Windows, control-click on Macintosh)
modifierSelector=8:	Shift key
modifierSelector=16:	Cmd key (Macintosh) or Ctrl key (Windows)
modifierSelector=32:	Caps lock key

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

col=*c* Sets the left-most visible column (user scrolling will change this). The list is scrolled horizontally as far as possible. Sometimes this won't be far enough to actually make column *c* the first column, but it will at least be visible. Use *c* =1 to put the left edge of column 1 (the *second* column) at the left edge of the list.

colorWave=*cw* Specifies a 3-column (RGB) or 4-column (RGBA) numeric wave. Used in conjunction with planes in selWave to define foreground and background colors for individual cells. Values range from 65535 (full on) to 0.

disable=*d* Sets user editability of the control.

d=0:	Normal.
d=1:	Hide.
d=2:	Draw in gray state; disable control action.

editStyle=*e* Sets the style for cells designated as editable (see selWave, bit 1).

e=0:	Uses a light blue background (default).
e=1:	Draws a frame around the cell with a white background.
e=2:	Combines the frame with the blue background. The background in all cases can be overridden using the colorWave parameter.

focusRing=*fr* Enables or disables the drawing of a rectangle indicating keyboard focus:

fr=0:	Focus rectangle will not be drawn.
fr=1:	Focus rectangle will be drawn (default).

On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.

font="*fontName*" Sets the font used for the list box items, e.g., `font="Helvetica"`.

frame=*f* Specifies the list box frame style.

f=0:	No frame.
f=1:	Simple rectangle.
f=2:	3D well.
f=3:	3D raised.
f=4:	Text well style.

fsize=s Sets list box font size.

fstyle=*fs*	*fs* is a bitwise parameter with each bit controlling one aspect of the font style as follows: Bit 0: Bold Bit 1: Italic Bit 2: Underline Bit 4: Strikethrough See **Setting Bit Parameters** on page IV-12 for details about bit settings.
hScroll=*h*	Scrolls the list to the right by *h* pixels (user scrolling will change this). *h* is the total amount of horizontal scrolling, not an increment from the current scroll position: *h* will be the value returned in the V_horizScroll variable by **ControlInfo**.
keySelectCol=*col*	Sets scan column number *col* when doing keyboard selection. Default is to scan column zero.
listWave=*w*	A 1D or 2D text wave containing the list contents.
mode=*m*	List selection mode specifying how many list selections can be made at a time. *m*=0: No selection allowed. *m*=1: One or zero selection allowed. *m*=2: One and only one selection allowed. *m*=3: Multiple, but not disjoint, selections allowed. *m*=4: Multiple and disjoint selections allowed. When multiple columns are used, you can enable individual cells to be selected using modes 5, 6, 7, and 8 in analogy to *m*=1-4. When using *m*=3 or 4 with multiple columns, only the first column of the selWave is used to indicate selections. Checkboxes and editing mode, however, use all cells even in modes 0-4. Modes 9 and 10 are the same as modes 4 and 8 except they use different selection rules and require testing bit 3 as well as bit 0 in selWave. In modes 4 and 8, a shift click toggles individual cells or rows, but in modes 9 and 10, the Command (*Macintosh*) or Ctrl (*Windows*) key toggles individual cells or rows whereas Shift defines a rectangular selection. T o determine if a cell is selected, perform a bitwise AND with `0x09`.
proc=*p*	Set name of user function proc to be called upon certain events. See discussion below.
pos={*left*,*top*}	Sets the location of top left corner of the list box in pixels.
pos+={*dx*,*dy*}	Offsets the position of the list box in pixels.
row=*r*	*r* is desired top row (user scrolling will change this). Use a value of -1 to scroll to the first selected cell (if any). Combine with selRow to select a row and to ensure it is visible (modes 1 and 2).
selCol=*c*	Defines the selected column when mode is 5 or 6 and no selWave is used. To read this value, use **ControlInfo** and the V_selCol variable.
selRow=*s*	Defines the selected row when mode is 1 or 2; when no selWave is used, it is defined by modes 5 or 6. Use -1 for no selection. To read this value, use **ControlInfo** and the V_value variable.

selWave=*sw*

sw is a numeric wave with the same dimensions as listWave. It is optional for modes 0-2, 5 and 6 and required in all other modes.

In modes greater than 2, *sw* indicates which cells are selected. In modes 1 and 2 use **ControlInfo** to find out which row is selected.

In all modes *sw* defines which cells are editable or function as checkboxes or disclosure controls.

Numeric values are treated as integers with individual bits defined as follows:

Bit 0 (0x01):	Cell is selected.
Bit 1 (0x02):	Cell is editable.
Bit 2 (0x04):	Cell editing requires a double click.
Bit 3 (0x08):	Current shift selection.
Bit 4 (0x10):	Current state of a checkbox cell.
Bit 5 (0x20):	Cell is a checkbox.
Bit 6 (0x40):	Cell is a disclosure cell. Drawn as a disclosure triangle (*Macintosh*) or a treeview expansion node (*Windows*).

In modes 3 and 4 bit 0 is set only in column zero of a multicolumn listbox.

Other bits are reserved. Additional dimensions are used for color info. See the discussion for colorWave. selWave is not required for modes 5 and 6.

setEditCell={*row*,*col*,*selStart*,*selEnd*}

Initiates edit mode for the cell at *row*, *col*. An error is reported if *row* or *col* is less than zero. Nothing happens and no error is reported if *row*, *col* is beyond the limits of the listbox, or if the cell has not been made editable by setting bit 1 of *selWave*.

selStart and *selEnd* set the range of bytes that are selected when editing is initiated; 0 is the start of the text. If there are N bytes in the listbox cell, setting *selStart* or *selEnd* to N or greater moves the start or end of the selection to the point after the last character. Setting *selStart* and *selEnd* to the same value selects no characters and the insertion point is set to *selStart*. Setting *selStart* to -1 always causes all characters to be selected.

size={*width*,*height*}

Sets list box size in pixels.

special={*kind*,*height*,*style*}

Specifies special cell formatting or contents.

kind=0:	Normal text but with specified *height* (if nonzero). Use a *style* of 1 to autocalculate widths based on the entire list contents. In this case, user widths are taken to be minimums and the last is not repeated.
kind=1:	Text taken to be the names of graphs or tables. Images of the graphs or tables are displayed in the cells. Use a *style* of 0 to display just the presentation portion of the graph or 1 to display it entirely. For tables, only the presentation portion is displayed.
kind=2:	Text taken to be the names of pictures. Images are displayed in the cells.
kind=3:	Displays a PNG, TIFF, or JPEG image. You can obtain binary picture data using SavePICT.

For *kind*=1 or 2, *height* may be zero to auto-set cell height to same as width or a specific value.

titleWave=*w*

Specifies a text wave containing titles for the listbox columns, instead of using the list wave dimension labels. Each row is the title for one column; if you have N columns you must have a wave with N rows. Allows more than 31 byte for a title, which is particularly important if you use styled text.

userColumnResize=*u* Enables resizing the list columns using the mouse.

u=0: Columns are not resizable (default). The widths parameter still works, though.

u=1: User can resize columns by dragging the column dividers.

When resizing a column without Option, Alt, or Shift modifiers (a “normal” resizing), any width added to the column is subtracted from the following column (if any).

When resizing while pressing Option (*Macintosh*) or Alt (*Windows*), only columns following the dragged divider will move (the same way table columns are resized).

When pressing Shift, all columns are set to the same width as the column being resized. If the total widths of all columns is less than the width of the listbox, then each column expands to fill the available width.

userdata=*UDStr* Sets the unnamed user data to *UDStr*.

userdata(*UDName*)=*UDStr*

Sets the named user data, *UDName*, to *UDStr*.

userdata+=*UDStr* Appends *UDStr* to the current unnamed user data.

userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current named user data, *UDName*.

widths={*w1*,*w2*,…} Optional list of minimum column widths in screen pixels. If more columns than widths, the last is repeated. If total of widths is greater than list box width then a horizontal scroll bar will appear. If total is less than available width then each expands proportionally.

widths+={*w1*,*w2*,…} Additional column widths. Because only 1000 bytes fit on a command line, lists with many columns may require multiple widths+= parameters to define all the column widths. However if all the widths are the same, widths+= is not needed; just use:

```
ListBox ctrlName widths={sameWidth}
```

win=*winName* Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Flags

/Z No error reporting.

Details

If the list wave has column dimension labels (see **SetDimLabel**), then those will be used as column titles. Note that a 1D wave is subtly different from a 1 column 2D wave. The former does not have any columns and therefore no column dimension labels.

Alternately, use a text wave with the titleWave keyword to specify column titles.

Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.

For instance, to make a title bold use the \f01 escape sequence:

```
SetDimLabel 1, columnNum, $"\\f01My Label", textWave
```

If you can't fit title text within the 31 character limit (styled text can be especially long), use the titleWave keyword with a text wave. The wave must have as many rows as the list wave has columns. When using a title wave, there are no restrictions on the number of or what characters you can use.

This example uses a title wave to add a red up-arrow graph marker to the end of a centered title:

```
Make/O/T/N=(numColumns) columnTitles
columnTitles[colNum]="\\JCThis is the title\\K(65535,0,0)\\k(65535,0,0)\\W523"
ListBox list0 titleWave=columnTitles
```

That's a 51-byte title that results in 19 characters or symbols that you actually see. \JC requests centered text, \K sets the text color (which colors the inside of the graph marker),\k sets the marker stroke color, and \W523 inserts a down-pointing triangular graph marker.

When using modes that allow multiple selections, use Shift to extend or add to the selection.

You can specify individual cells as being editable by setting bit 1 (counting from zero on the right) in selWave. The user can start editing a cell by either clicking in it or, if the cell is selected, by pressing Enter (or Return). When finished, the user can press Enter to accept the changes or can press Escape to reject changes. The user may also press Up or Down Arrow to accept changes and begin editing the next editable cell in a column. Likewise, Tab and Shift-Tab moves to the next or previous column in a row. If bit 2 of selWave is set then a double click will be required rather than a single click. **Note**: in edit mode, Tab and Shift-Tab are used to move left and right because the Left and Right Arrow keys are used to move the text entry cursor left and right.

When the listbox has keyboard focus (either by tabbing to the list box or by clicking in the box), the keyboard arrow keys move a cell selection (or row depending on mode). When not in cell edit mode, Tab and Shift-Tab move the keyboard focus to other objects in the window. The Home, End, Page Up, and Page Down keys affect the vertical scroll bar.

When the listbox has focus, the user may type the first few chars of an entry in the list to select that entry. Only the first column is used. If a match is not found then nothing is done. The search is case insensitive.

Listbox Action Procedure

The action procedure for a ListBox control takes a predefined structure `WMListboxAction` as a parameter to the function:

```
Function ActionProcName(LB_Struct) : ListboxControl
   STRUCT WMListboxAction &LB_Struct
   …
   return 0
End
```

The “`: ListboxControl`” designation tells Igor to include this procedure in the Procedure pop-up menu in the List Box Control dialog.

See **WMListboxAction** for details on the WMListboxAction structure.

Although the return value is not currently used, action procedures should always return zero.

You may see an old format listbox action procedure in old code:

```
Function MyListboxProc(ctrlName,row,col,event) : ListboxControl
   String ctrlName     // name of this control
   Variable row        // row if click in interior, -1 if click in title
   Variable col        // column number
   Variable event      // event code
   …
   return 0            // other return values reserved
End
```

This old format should not be used in new code.

Specifying Cell Color

The background and foreground (text) color of individual cells may be defined by providing colorWave in conjunction with specific planes in selWave. The planes in selWave are taken to be integer indexes into colorWave. The planes are defined by specific dimension labels and not by specific plane numbers. To provide foreground colors, define a plane labeled “foreColors” that contains the desired index values. Likewise define and fill a plane labeled “backColors” for background colors. The value 0 is special and indicates that the default colors should be used. Note that if you have a one column list for which you want to supply colors, the selWave needs to be three dimensional but with just one column. Here is an example:

```
Make/T/N=(5,1) tw= "row "+num2str(p)        // 5 row, 1 col text wave (2D)
Make/B/U/N=(5,1,2) sw                       // 5 row, 1 col, 2 plane byte wave
Make/O/W/U myColors={{0,0,0},{65535,0,0},{0,65535,0},{0,0,65535},{0,65535,65535}}
MatrixTranspose myColors      // above was easier to enter as 3 rows, 5 cols

NewPanel
ListBox lb,mode=3,listWave= tw,selWave= sw,size={200,100},colorWave=myColors
sw[][][1]= p                          // arbitrary index values into plane 1
```

Now, execute the following commands one at a time and observe the results:

```
SetDimLabel 2,1,backColors,sw         // define plane 1 as background colors

SetDimLabel 2,1,foreColors,sw         // redefine plane 1 s foreground colors

sw[][][%foreColors]= 4-p              // change the color index values
```

In the above example, the selWave was defined as unsigned byte. If you need more than 254 colors, you will have to use a larger number size.

The color wave may have four columns instead of three, with the fourth specifying transparency (65535=opaque, 0=transparent). Transparency of the background color is of limited utility as it just shows the white background color through the cell color, resulting in pastel shades. Transparency of the foreground color permits the cell background color to show through the text color.

Checkboxes in Cells

You can cause a cell to contain a checkbox by setting bit 5 in selWave. The title (if any) is taken from listWave and the results (selected/deselected) is bit 4 of selWave. If a checkbox cell is selected then the space bar will toggle the checkbox. (Clicking a checkbox cell does not select it — use the arrow keys.)

Errors

Your listbox may be drawn with a red X and an error code. The error codes are:

Error	Meaning
E1	Too small.
E2	listWave is invalid (missing, not text or no rows).
E3	listWave and selWave do not match in dimensions.
E4	mode > 2 with no selWave.

Event Queue

It is possible for a single user action to produce more than one event. For instance, pressing the up arrow key while editing to select a cell that is not visible generates event codes 4, 8, and 6. Igor calls your action procedure separately for each code.

If your code tests for a nonzero value in eventCode2, and uses it only if it is nonzero, then the event queue method will not break your code. If you have determined empirically when you need to use eventCode2 and you do not test, your code will break.

Scroll Event Warnings

Events 8, 9, and 10 report to you that the listbox has been scrolled vertically or horizontally. These events are envisioned as allowing you to keep two listboxes synchronized (you may find other uses for these events). You might use an action procedure like this one to keep two listboxes (named list0 and list1) in sync:

```
Function ListBoxProc2(LB_Struct) : ListBoxControl
   STRUCT WMListboxAction &LB_Struct

   if (LB_Struct.eventCode == 8)
      String listname
      if (CmpStr(LB_Struct.ctrlName, "list1") == 0)
         listname = "list0"
      else
         listname = "list1"
      endif
      ControlInfo $listname
      if (V_startRow != LB_Struct.row)
         listbox $listname,row=LB_Struct.row
         ControlUpdate $listname
      endif
   endif
```

```
    return 0
End
```

It is very easy to create an infinite cascade of events feeding back between the two listboxes, especially if you use event 10. When this happens, you will see your listboxes jigging up and down endlessly. The test using ControlInfo is intended to make this unlikely.

The slow response of the old-style, nonstructure action procedure can defeat the ControlInfo test by delaying the action procedure execution. If you use events 8, 9, or 10, we recommended that you use the new-style action procedure.

Note on Keystroke Event

In a keystroke event passed to a listbox action procedure the eventCode is 12. A character code is stored in the row field of the WMListboxAction structure. This works only for ASCII characters and a few special characters such as delete (8), forward delete (127) and escape (27). It does not work for non-ASCII characters such as accented characters which, in Igor Pro 7, are represented as UTF-8 and require multiple bytes.

As explained below, many keystroke events are not sent to the listbox action procedure because they are consumed by the internal listbox code in Igor. For this reason the keystroke event for a listbox is of limited use.

The architecture of Igor controls is such that events are passed to an action procedure only after the control has used them. In the case of a keystroke event, that means that other uses of the keystroke may consume the event before the action procedure gets a chance at it. In particular, a listbox editable cell that is actively being edited consumes keystroke events, and the action procedure is not called. The only editing-related events your action procedure will get are event codes 6 and 7.

If an arrow key is pressed, and this results in the selected row or cell changing, your action procedure will not get a keystroke event. Instead, your action procedure will receive event code 4 or 5. If the arrow key causes scrolling to occur, the action procedure will also get event code 8 or 9.

Examples

Here is a simple Listbox example:

```
Make/O/T/N=30 tjack="this is row "+num2str(p)
Make/O/B/N=30 sjack=0
NewPanel /W=(19,61,319,261)
ListBox lb1,pos={42,9},size={137,94},listWave=tjack,selWave=sjack,mode= 3
Edit/W=(367,61,724,306) tjack,sjack
ModifyTable width(tjack)=148
```

Make selections in the list and note changes in the table and vice versa. Edit one of the list text values in the table and note update of the list.

Here is an example using a titleWave and styled text in the title cells. Note that the last title isn't very long when rendered, but requires a 63 character specification.

```
Make/O/T/N=(4,3) ListWave="row "+num2str(p)+" col "+num2str(q)
Make/O/T/N=3 titles                    // three rows to match 3-column ListWave
titles[0] = "\f01Bold Title"
titles[1] = "title with semicolon;"
titles[2] = "Marker in Gray: \K(40000,40000,40000)\k(40000,40000,40000)\W517"
NewPanel /W=(515,542,1011,794)
ListBox list0,pos={1,2},size={391,120},listWave=ListWave
ListBox list0,titleWave=titles
```

An example experiment that lets you easily experiment with ListBox settings is available in "Examples:Feature Demos 2:ListBox Demo.pxp".

See Also

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Setting Bit Parameters on page IV-12 for further details about bit settings.

The **GetUserData** operation for retrieving named user data.

ListBoxControl

`ListBoxControl`

ListBoxControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined listbox control. See **Procedure Subtypes** on page IV-193 for details. See **ListBox** for details on creating a listbox control.

ListMatch

`ListMatch(`*`listStr`*`, `*`matchStr`*` [, `*`listSepStr`*`])`

The ListMatch function returns each list item in *listStr* that matches *matchStr*.

ListStr should contain items separated by the *listSepStr* character, such as "abc;def;".

You may include asterisks in *matchStr* as a wildcard character. Note that matching is case-insensitive. See **StringMatch** for wildcard details.

ListSepStr is optional. If missing, it is taken to be ";".

See Also

The **GrepList**, **StringMatch**, **StringFromList**, and **WhichListItem** functions.

ListToTextWave

`ListToTextWave(`*`listStr`*`, `*`separatorStr`*`)`

The ListToTextWave function returns a free text wave containing the individual list items in *listStr*.

See **Free Waves** on page IV-84 for details on free waves.

The ListToTextWave function was added in Igor Pro 7.00.

Parameters

listStr is an string that contains any number of substrings separated by a common string separator.

separatorStr is the separator string that separates one item in the list from the next. It is usually a single semicolon character but can by any string.

Details

The ListToTextWave function returns a free wave so it can't be used on the command line or in a macro. If you need to convert the free wave to a global wave use MoveWave.

For lists with a large number of items, using ListToWave and then retrieving the substrings sequentially from the returned text wave is much faster than retrieving the substrings using StringFromList. This is because StringFromList must search from the start of the list to the desired substring each time it is called.

The reverse operation, converting the contents of a text wave into a string list, can be accomplished using **wfprintf** like this:

```
WAVE/T tw
String list
wfprintf list, “%s\r”, tw        // Carriage-return separated list
```

Example

```
Function Test(num, separator)
    Variable num
    String separator                        // Usually ";"

    // Build a string list using separator
    String list = ""
    Variable i
    for(i=0; i<num; i+=1)
        list += "item_" + num2str(i) + separator
    endfor

    // Convert to a text wave and print its elements
    Wave/T w = ListToTextWave(list, separator)
    Print numpnts(w)
    for(i=0; i<num; i+=1)
        Print i, w[i]
    endfor
End
```

See Also

ListToWaveRefWave, **WaveRefWaveToList**, **wfprintf**

Using Strings as Lists on page IV-161, **StringFromList**, **MoveWave**, **Free Waves** on page IV-84

ListToWaveRefWave

`ListToWaveRefWave(stringList [, options])`

The ListToWaveRefWave function returns a free wave containing a wave reference for each entry in *stringList* that corresponds to an existing wave.

The ListToWaveRefWave function was added in Igor Pro 7.00.

Parameters

stringList is a semicolon-separated string list of waves specified using full paths or partial paths relative to the current data folder.

options is a bit field that is 0 by default. Set bit 0 to if you want the function to generate an error (see **GetRTError**) if any of the list elements does not specify an existing wave. Other bits are reserved for future use and must be cleared.

Example

```
Function Test()
    NewDataFolder/O/S root:TestDF     // Create empty data folder as current DF
    Make/O aaa                        // aaa will be the first wave in list
    Make/O bbb
    String strList = WaveList("*", ";", "")
    Wave/WAVE wr = ListToWaveRefWave(strList, 0)   // Returns a free wave
    Wave w = wr[0]
    Print GetWavesDataFolder(w, 2)
    Wave w = wr[1]
    Print GetWavesDataFolder(w, 2)
End
```

See Also

WaveRefWaveToList, **ListToTextWave**, **Wave References** on page IV-65

ln

`ln(num)`

The ln function returns the natural logarithm of *num*, -INF if *num* is 0, or NaN if *num* is less than 0. In complex expressions, *num* is complex, and ln(*num*) returns a complex value.

To compute a logarithm base n use the formula:

$$\log_n(x) = \frac{\log(x)}{\log(n)}.$$

See Also

The **log** function.

LoadData

`LoadData [flags] fileOrFolderNameStr`

The LoadData operation loads data from the named file or folder. “Data” means Igor waves, numeric and string variables and data folders containing them. The specified file or folder must be an Igor packed experiment file or a folder containing Igor binary data, such as an Igor unpacked experiment folder or a folder in which you have stored Igor binary wave files.

LoadData loads data objects into memory and they become part of the current Igor experiment, disassociated from the file from which they were loaded.

If loading from a file-system folder, the data (waves, variables, strings) in the folder, including any subfolders if /R is specified, is loaded into the current Igor data folder.

If loading from a packed Igor experiment file, the data in the file, including any packed subdata folders if /R is specified, is loaded into the current Igor data folder.

Use LoadData to load experiment data using Igor procedures. To load experiment data interactively, use the Data Browser (Data menu).

Parameters

If you use a full or partial path for *fileOrFolderNameStr*, see **Path Separators** on page III-401 for details on forming the path.

If *fileOrFolderNameStr* is omitted you get to locate the file (if /D is omitted) or the folder (if /D is present) via a dialog.

Flags

/D	If present, loads from a file-system folder (a directory). If omitted, LoadData loads from an Igor packed experiment file.
/I	Interactive. Forces LoadData to present a dialog.
/J=*objectNamesStr*	Loads only the objects named in the semicolon-separated list of object names.
/L=*loadFlags*	Controls what kind of data objects are loaded with a bit for each data type:

loadFlags	**Bit Number**	**Loads this Object Type**
1	0	Waves
2	1	Numeric Variables
4	2	String Variables

To load multiple data types, sum the values shown in the table. For example, /L=1 loads waves only, /L=2 loads numeric variables only, and /L=3 loads both waves and numeric variables. See **Setting Bit Parameters** on page IV-12 for further details about bit settings.

If no /L is specified, all object types are loaded. This is equivalent to /L=7. All other bits are reserved and should be set to zero.

/O[=*overwriteMode*] If /O alone is used, overwrites existing data objects in case of a name conflict.

overwriteMode is defined as follows:

0:	No overwrite, as if there were no /O.
1:	Normal overwrite. In the event of a name conflict, objects in the incoming file replace the conflicting objects in memory. Incoming data folders completely replace any conflicting data folders in memory.
2:	Mix-in overwrite. In the event of a name conflict, objects in the incoming file replace the conflicting objects in memory but nonconflicting objects in memory are left untouched.

See **Details** for more about overwriting.

/P=*pathName*	Specifies folder to look in for the specified file or folder. *pathName* is the name of an existing symbolic path.
/Q	Suppresses the normal messages in the history area.
/R	Recursively loads subdata folders.
/S=*subDataFolderStr*	Specifies a subdata folder within a packed experiment file to be loaded. See **Details** for more.

/T[=*topLevelName*]	If /T=*topLevelName* is specified, it creates a new data folder in the current data folder with the specified name and places the loaded data in the new data folder. If just /T is specified, it creates a new data folder in the current data folder with a name derived from the name of the unpacked experiment folder, packed experiment file or packed subdata folder being loaded.

Details

If /T is present, LoadData loads the top level data folder and its contents. If /T is omitted, it loads just the contents of the top level data folder and not the data folder itself. This distinction has an analogy in the desktop. You can drag the contents of disk folder A into folder B or you can drag folder A itself into folder B.

If present, /S=*subDataFolderStr* specifies the subdata folder within the packed experiment file from which the load is to start. For example:

```
LoadData/P=Path1/S="Folder A:Folder B" "aPackedExpFile"
```

This starts loading from data folder “Folder B” which is in “Folder A” in the packed experiment file. Note that the string specified by /S must specify each subdata folder until the desired data folder is reached. Since this parameter is specified as a string, you must not use single quotes.

In Igor Pro 7.00 or later, you can include “root:” at the start of *subDataFolderStr*. However, it is not required because, if it is omitted, LoadData takes the path to be relative to the root of the data hierarchy of the experiment being loaded.

/S has no effect if you are loading from a file system folder rather than from a packed experiment file.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like “`hd:Folder1:`” or “`C:\\Folder1\\`”. See **Symbolic Paths** on page II-21 for details.

If /J=*objectNamesStr* is used, then only the objects named in *objectNamesStr* are loaded into the current experiment. For example, `/J="wave0;wave1"` will load only the two named waves, ignoring any other data in the file or folder being loaded.

Assume that you have an experiment that contains 5 runs where each run, stored in a separate data folder in a packed experiment file, consists of data acquired from four channels from an ADC card. Using the /J flag, you can load just one specific channel from each run. This way you can compare that channel’s data from all runs without loading the other channels.

The list of object names used with /J must be semicolon-separated. A semicolon after the last object name in the list is optional. Because the object names exist in a string expression, they should not be quoted. The list is limited to 1000 bytes.

Using `/J=""` acts like no /J at all.

If you load a hierarchy of data folders (using the /R flag) with /J in effect, LoadData will create each data folder in the hierarchy even if it contains none of the named objects. This behavior is necessary to avoid loading a subdata folder without loading its parent, as well as other such complications.

If you do a load of a data folder, overwriting an existing data folder of the same name, the behavior of LoadData depends on whether you use /J. If you do not use /J, the entire data folder and all of its contents are replaced. If you do use /J, just the specified objects in the data folder are replaced, leaving any other preexisting objects in the data folder unchanged.

If you do not use the /O (overwrite) flag or if you use /O=0 and there is a conflict between objects or data folders in the current data folder and objects or data folders in the file or folder being loaded, LoadData will present a dialog to ask you if you want to replace the existing data. However, LoadData can not replace an object with an object of a different type and will refuse to do so.

You can overwrite an object that is in use, such as a wave that is displayed in a graph or table. You can also overwrite a data folder that contains objects that are in use. This is a powerful feature. Imagine that you define a data structure consisting of waves, variables and possibly subdata folders. You can display the data in graphs and tables and you can display these in a layout. You can then overwrite the data with an analogous data structure from a packed experiment file and Igor will automatically update any graphs, tables, or layouts that need to be updated.

Because LoadData can load from a complex packed Igor experiment file or from a complex hierarchy of file-system folders, it does not set the variables normally set by a file loader: S_path, S_fileName, and S_waveNames. The variable V_flag is set to the total number of objects loaded, or to -1 if the user cancelled the open file dialog. To find what objects were created by LoadData, you can use the **CountObjects** and **GetIndexedObjName** functions.

See Also

The **SaveData** operation, **Importing Data** on page II-117, **The Data Browser** on page II-106.

LoadPackagePreferences

`LoadPackagePreferences [/MIS=`*`mismatch`*` /P=`*`pathName`*`]` ***`packageName, prefsFileName, recordID, prefsStruct`***

The LoadPackagePreferences operation loads preference data previously stored on disk by the **SavePackagePreferences** operation. The data is loaded into the specified structure.

Note: The package preferences structure must not use fields of type Variable, String, WAVE, NVAR, SVAR or FUNCREF because these fields refer to data that may not exist when LoadPackagePreferences is called.

The structure can use fields of type char, uchar, int16, uint16, int32, uint32, int64, uint64, float and double as well as fixed-size arrays of these types and substructures with fields of these types.

If the /P flag is present then the location on disk of the preference file is determined by *pathName* and *prefsFileName*. However in the usual case the /P flag will be omitted and the preference file is located in a file named *prefsFileName* in a directory named *packageName* in the Packages directory in Igor's preferences directory.

Note: You must choose a very distinctive name for *packageName* as this is the only thing preventing collisions between your package and someone else's package.

See **Saving Package Preferences** on page IV-237 for background information and examples.

Parameters

packageName is the name of your package of Igor procedures. It is limited to 31 bytes and must be a legal name for a directory on disk. This name must be very distinctive as this is the only thing preventing collisions between your package and someone else's package.

prefsFileName is the name of a preference file to be loaded by LoadPackagePreferences. It should include an extension, typically ".bin".

prefsStruct is the structure into which data from disk, if it exists, will be loaded.

recordID is a unique positive integer that you assign to each record that you store in the preferences file. If you store more than one structure in the file, you would use distinct *recordID*s to identify which structure you want to load. In the simple case you will store just one structure in the preference file and you can use 0 (or any positive integer of your choice) as the *recordID*.

Flags

/MIS=*mismatch*	Controls what happens if the number of bytes in the file does not match the size of the structure: 0: Returns an error. Default behavior if /MIS is omitted. 1: Returns the smaller of the size of the structure and the number of bytes in the file. Does not return an error. Use this if you want to read and update old versions of a preferences structure.
/P=*pathName*	Specifies the directory to look in for the file specified by *prefsFileName*. *pathName* is the name of an existing symbolic path. See **Symbolic Paths** on page II-21 for details. `/P=$<empty string variable>` acts as if the /P flag were omitted.

Details

LoadPackagePreferences sets the following output variables:

`V_flag`	Set to 0 if no error occurred or to a nonzero error code. If the preference file does not exist, V_flag is set to zero so you must use V_bytesRead to detect that case.
`V_bytesRead`	Set to the number of bytes read from the file. This will be zero if the preference file does not exist.
`V_structSize`	Set to the size in bytes of *prefsStruct*. This may be useful in handling structure version changes.

After calling LoadPackagePreferences if V_flag is nonzero or V_bytesRead is zero then you need to create default preferences as illustrated by the example referenced below.

V_bytesRead, in conjunction with the /MIS flag, makes it possible to check for and deal with old versions of a preferences structure as it loads the version field (typically the first field) of an older or newer version structure. However in most cases it is sufficient to omit the /MIS flag and treat incompatible preference data the same as missing preference data.

Example

See the example under **Saving Package Preferences in a Special-Format Binary File** on page IV-237.

See Also

SavePackagePreferences.

LoadPICT

`LoadPICT` `[`***`flags`***`] [`***`fileNameStr`***`][,` ***`pictName`***`]`

The LoadPICT operation loads a picture from a file or from the Clipboard into Igor. Once you have loaded a picture, you can append it to graphs and page layouts.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

If you want to force a dialog to select the file, omit the *fileNameStr* parameter.

If *fileNameStr* is “Clipboard” and /P=*pathName* is omitted, LoadPICT loads its data from the Clipboard rather than from a file.

pictName is the name that you want to give to the newly loaded picture. You can refer to the picture by its name to append it to graphs and page layouts. LoadPICT generates an error if the name conflicts with some other type of object (e.g., wave or variable) or if the name conflicts with a built-in name (e.g., the name of an operation or function).

If you omit *pictName*, LoadPICT automatically names the picture as explained in **Details**.

Flags

/I=*resIndex*	Specifies the resource to load by resource index, starting from 1 (*Macintosh only*).
/M=*promptStr*	Specifies a prompt to use if LoadPICT needs to put up a dialog to find the file.
/N=*resNameStr*	A string that specifies the resource to load by resource name (*Macintosh only*).
/O	Overwrites an existing picture with the same name. If /O is omitted and there is an existing picture with the same name, LoadPICT displays a dialog in which you can resolve the name conflict.

/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/Q	Quiet: suppresses the insertion of picture info into the history area.
/R=*resourceID*	Specifies the resource to load by resource ID (*Macintosh only*).
/Z	Doesn't load the picture, just checks for its existence.

Details

If the picture file is not fully specified then LoadPICT presents a dialog from which you can select the file. "Fully specified" means that LoadPICT can determine the name of the file (from the *fileNameStr* parameter) and the folder containing the file (from the flag /P=*pathName* flag or from the *fileNameStr* parameter). If you want to force a dialog, omit the *fileNameStr* parameter.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-21 for details.

On Macintosh, LoadPICT can load picture data from the file's data or resource fork. Most graphics programs store a picture in the data fork. Some programs may store one or more pictures as resources. If you use /R, /I, or /N, it loads the picture from the specified resource in the file. Otherwise, it loads the picture from the file's data fork.

If you omit *pictName*, LoadPICT automatically names the picture as follows:

On Macintosh, if the picture was loaded from the resource fork of a file (you used /R, /I, or /N) and the resource had a nonempty name, it uses the resource name. If necessary, the name is made legal by replacing illegal characters or shortening it.

If the picture was loaded from a file, LoadPICT uses the file name. If necessary, it makes it into a legal name by replacing illegal characters or shortening it.

Otherwise, LoadPICT uses a name of the form "PICT_*n*".

If the resulting name is in conflict with an existing picture name, Igor puts up a name conflict resolution dialog.

LoadPICT sets the variable V_flag to 1 if the picture exists and fits in available memory or to 0 otherwise.

It also sets the string variable S_info to a semicolon-separated list of values:

Keyword	Information Following Keyword
NAME	Name of the loaded PICT, often "PICT_0", etc.
SOURCE	One of "data fork", "resource fork" or "Clipboard".
RESOURCENAME	Name of the resource the picture was loaded from, or `""` if the source was not the file's resource fork.
RESOURCEID	Resource ID the picture was loaded from, or 0 if the source was not the file's resource fork.

Keyword	Information Following Keyword
TYPE	One of the following types: DIB Encapsulated PostScript Enhanced metafile JPEG PDF PNG SVG TIFF Windows bitmap Windows metafile Unknown type
BYTES	Amount of memory used by the picture.
WIDTH	Width of the picture in pixels.
HEIGHT	Height of the picture in pixels.
PHYSWIDTH	Physical width of the picture in points.
PHYSHEIGHT	Physical height of the picture in points.

See Also

See **Pictures** on page III-448 for general information on how Igor handles pictures.

The **ImageLoad** operation for loading PICT and other image file types into waves, and the **PICTInfo** function.

LoadWave

`LoadWave [flags] [fileNameStr]`

The LoadWave operation loads data from the named Igor binary, Igor text, delimited text, fixed field text, or general text file into waves. LoadWave can load 1D and 2D data from delimited text, fixed field text and general text files, or data of any dimensionality from Igor binary and Igor text files.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If LoadWave can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

If *fileNameStr* is “Clipboard” and /P=*pathName* is omitted, LoadWave takes its data from the Clipboard rather than from a file. This is not implemented for binary loads.

If *fileNameStr* is omitted or is “”, or if the /I flag is used, LoadWave presents an Open File dialog from which you can choose the file to load.

Flags

/A	"Auto-name and go" option (used with /G, /F or /J). This skips the dialog in which you normally enter wave names. Instead it automatically assigns names of the form *wave*0, *wave*1, choosing names that are not already in use. When used with /W, it reads wave names from the file instead of automatically assigning names and /A just skips the wave name dialog. The /B flag can also override names specified by /A.
/A=*baseName*	Same as /A but it automatically assigns wave names of the form *baseName*0, *baseName*1.
/B=*columnInfoStr*	Specifies the name, format, numeric type, and field width for columns in the file. See **Specifying Characteristics of Individual Columns**.
/C	This is used in experiment recreation commands generated by Igor to force experiment recreation to continue if an error occurs in loading a wave.
/D	Creates double precision waves. (Used with /G, /F or /J.) The /B flag can override the numeric precision specified by the /D flag.
/E=*editCmd*	Controls table creation: *editCmd*=1: Makes a new table containing the loaded waves. *editCmd*=2: Appends the loaded waves to the top table. If no table exists, a new table is created. *editCmd*=0: Same as if /E had not been specified (loaded waves are not put in any table).

/ENCG=*textEncoding*

/ENCG={*textEncoding*, *tecOptions*}

Specifies the text encoding of the plain text file being loaded.

This flag was added in Igor Pro 7.00.

This flag is ignored when loading an Igor binary wave file.

See **Text Encoding Names and Codes** on page III-434 for a list of accepted values for *textEncoding*.

For most purposes the default value for *tecOptions*, 3, is fine and you can use /ENCG=textEncoding instead of /ENCG={*textEncoding*, *tecOptions*}.

tecOptions is an optional bitwise parameter that controls text encoding conversion. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

tecOptions is defined as follows:

Bit 0:	If cleared, the presence of null bytes causes LoadWave to consider the text invalid in all byte-oriented text encodings. This makes it easier for LoadWave to identify UTF-16 and UTF-32 text which usually contains null bytes. If set (default), null bytes are allowed in byte-oriented text encodings.
Bit 1:	If cleared LoadWave does not validate text if the specified text encoding is UTF-8. If set (default), LoadWave validates text even if the text encoding is UTF-8.
Bit 2:	If cleared (default) LoadWave validates the text in the specified text encoding except that validation for UTF-8 is skipped if bit 1 is cleared. If set, LoadWave assumes that the text is valid in the specified text encoding and does not validate it. Setting this bit makes LoadWave slightly faster but it is usually inconsequential.
Bit 3:	If cleared (default), LoadWave presents the Choose Text Encoding dialog if the specified text encoding is not valid for the text in the file. If set, LoadWave does not present the Choose Text Encoding dialog if the text is not valid in the specified text encoding and instead returns an error. Set this bit if you are running an automated procedure and do not want it interrupted by a dialog.

See **LoadWave Text Encoding Issues** on page V-453 for further discussion.

/F={*numColumns*, *defaultFieldWidth*, *flags*}

Indicates that the file uses the fixed field file format. Most FORTRAN programs generate files in this format.

numColumns is the number of columns of data in the file.

defaultFieldWidth is the default number of bytes in each column. If the columns do not all have the same number of bytes, you need to use the /B flag to provide more information to LoadWave.

flags is a bitwise parameter that controls the conversion of text to values. The bits are defined as follows:

Bit 0 :	If set, any field that consists entirely of the digit “9” is taken to be blank. A field that consists entirely of the digit “9” except for a leading “+” or “-” is also taken to be blank.

All other bits are reserved and must be cleared.

/G Indicates that the file uses the general text format.

/H	Loads the wave into the current experiment and severs the connection between the wave and the file. When the experiment is saved, the wave copy will be saved as part of the experiment. For a packed experiment this means it is saved in the packed experiment file. For an unpacked experiment this means it is saved in the experiment's home folder. See **Sharing Versus Copying Igor Binary Files** on page II-137.
/I	Forces LoadWave to display an Open File dialog even if the file is fully specified via /P and *fileNameStr*.
/J	Indicates that the file uses the delimited text format.
/K=*k*	Controls how to determine whether a column in the file is numeric or text (only for delimited text and fixed field text files). *k*=0: Deduces the nature of the column automatically. *k*=1: Treats all columns as numeric. *k*=2: Treats all columns as text. This flag as well as the ability to load text data into text waves were added in Igor Pro 3.0. The default for the LoadWave operation is /K=1, meaning that it will treat all columns as numeric. We did this so that existing procedures would behave the same in Igor Pro 3.0 as before. Use /K=0 when you want to load text columns into text waves. /K=2 may have use in a text-processing application. For finer control, the /B flag specifies the format of each column in the file individually.

/L={*nameLine*, *firstLine*, *numLines*, *firstColumn*, *numColumns*}

Affects loading delimited text, fixed field text, and general text files only (/J, /F or /G). /L is accepted no matter what the load type but is ignored for Igor binary and Igor text loads. Line and column numbers start from 0.

nameLine is the number of the line containing column names. For general text loads, 0 means auto. See **Loading General Text Files** on page II-128 for details.

firstLine is the number of the first line to load into a wave. For general text loads, 0 means auto. See **Loading General Text Files** on page II-128 for details.

numLines is the number of lines that should be treated as data. 0 means auto which loads until the end of the file or until the end of the block of data in general text files. The *numLines* parameter can also be used to make loading very large files more efficient. See **Loading Very Large Files**.

firstColumn is the number of the first column to load into a wave. This is useful for skipping columns.

numColumns is the number of columns to load into a wave. 0 means auto, which loads all columns.

/M	Loads data as matrix wave. If /M is used then it ignores the /W flag (read wave names) and follows the /U flags instead. The wave is autonamed unless you provide a specific wave name using the /B flag. The type of the wave (numeric or text) is determined by an assessment of the type of the first loaded column unless you override this using the /K flag or the /B flag. See **The Load Waves Dialog for Delimited Text — 2D** on page II-124 for further information.
/N	Same as /A except that, instead of choosing names that are not in use, it overwrites existing waves.
/N=*baseName*	Same as /N except that it automatically assigns wave names of the form *baseName0*, *baseName1*.

/O	Overwrite existing waves in case of a name conflict.
/P=*pathName*	Specifies the folder to look in for *fileNameStr*. *pathName* is the name of an existing symbolic path.
/Q	Suppresses the normal messages in the history area.

/R={*languageName*, *yearFormat*, *monthFormat*, *dayOfMonthFormat*, *dayOfWeekFormat*, *layoutStr*, *pivotYear*}

Specifies a custom date format for dates in the file. If the /R flag is used, it overrides the date setting that is part of the /V flag.

languageName controls the language used for the alphabetic date components, namely the month and the day-of-week. *languageName* is one of the following:

Default			
Chinese	ChineseSimplified	Danish	Dutch
English	Finnish	French	German
Italian	Japanese	Korean	Norwegian
Portuguese	Russian	Spanish	Swedish

Default means the system language on Macintosh or the user default language on Windows.

yearFormat is one of the following codes:

1:	Two digit year.
2:	Four digit year.

monthFormat is one of the following codes:

1:	Numeric, no leading zero.
2:	Numeric with leading zero.
3:	Abbreviated alphabetic (e.g., Jan).
4:	Full alphabetic (e.g., January).

dayOfMonthFormat is one of the following codes:

1:	Numeric, no leading zero.
2:	Numeric with leading zero.

dayOfWeekFormat is one of the following codes:

1:	Abbreviated alphabetic (e.g., Mon).
2:	Full alphabetic (e.g., Monday).

layoutStr describes which components appear in the date in what order and what separators are used. *layoutStr* is constructed as follows (but with no line break):

```
"<component keyword><separator>
<component keyword><separator>
<component keyword><separator>
<component keyword>"
```

where <component keyword> is one of the following:

```
Year
Month
DayOfMonth
DayOfWeek
```

and <separator> is a string of zero to 15 bytes.

Starting from the end, parts of the layout string must be omitted if they are not used.

Extraneous spaces are not allowed in *layoutStr*. Each separator must be no longer than 15 bytes. No component can be used more than once. Some components may be used zero times.

pivotYear determines how LoadWave interprets two-digit years. If the year is specified using two digits, yy, and is less than *pivotYear* then the date is assumed to be 20yy. If the two digit year is greater than or equal to *pivotYear* then the year is assumed to be 19yy. *pivotYear* must be between 4 and 40.

See **Loading Custom Date Formats** on page V-449 for further discussion of date formats.

/T — Indicates that the file uses the Igor text format.

Although LoadWave is generally thread-safe, it is not thread-safe to load an Igor text file containing an Igor command (e.g., "X <command>").

/U={*readRowLabels*, *rowPositionAction*, *readColLabels*, *colPositionAction*}

These parameters affect loading a matrix (/M) from a delimited text (/J) or a fixed field text (/F) file. They are accepted no matter what the load type is but are ignored when they don't apply.

If *readRowLabels* is nonzero, it reads the first column of data in the file as the row labels for the matrix wave.

rowPositionAction has one of the following values:

0:	The file has no row position column.
1:	Uses the row position column to set the row scaling of the matrix wave.
2:	Creates a 1D wave containing the values in the row position column. The name of the 1D wave will be the same as the matrix wave but with the prefix "RP_".

The *readColumnLabels* and *columnPositionAction* parameters have analogous meanings. The prefix used for the column position wave is "CP_".

See Chapter II-9, **Importing and Exporting Data** for further details.

/V={*delimsStr*, *skipCharsStr*, *numConversionFlags*, *loadFlags*}

These parameters affect loading delimited text (/J) and fixed field text (/F) data and column names. They do not affect loading general text (/G). They are accepted no matter what the load type is but are ignored when they don't apply. These parameters should rarely be needed.

delimsStr is a string expression containing the characters that should act as delimiters for delimited file loads. (When loading data as general text, the delimiters are always tab, comma and space.) The default is "\t," for tab and comma. You can specify the space character as a delimiter, but it is always given the lowest priority behind any other delimiters contained in *delimsStr*. The low priority means that if a line of text contains any other delimiter besides the space character, then that delimiter is used rather than the space character.

skipCharsStr is a string expression containing characters that should always be treated as garbage and skipped when they appear before a number. The default is " $" for space and dollar sign. This parameter should rarely be needed.

numConversionFlags is a bitwise parameter that controls the conversion of text to numbers. The bits are defined as follows:

Bit 0:	If set: dates are dd/mm/yy. If cleared: dates are mm/dd/yy.
Bit 1:	If set: decimal character is comma. If cleared: it is period.
Bit 2:	If set: thousands separators in numbers are ignored when loading delimited text only (LoadWave/J). The thousands separator is the comma in 1,234 or, if comma is the decimal character, the dot in 1.234. Most numeric data files do not use thousands separators and searching for them slows loading down so this bit should usually be 0. This bit has no effect if the thousands separator (e.g., comma) is also a delimiter character as specified by delimsStr.

All other bits are reserved and must be cleared.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

If the /R flag is used to specify the date format, this overrides the setting of bit 0.

loadFlags is a bitwise parameter that controls the overall load. The bits are defined as follows:

Bit 0:	If set, ignores blanks at the end of a column. This is appropriate if the file contains columns of unequal length. Set *loadFlags* = 1 to set bit 0.
Bit 1:	If set, when the /W flag is specified and the line containing column labels starts with one or more space characters, the spaces are taken to be a blank column name. The resulting column will be named Blank. Use this if both the line containing column labels and the lines containing data start with leading spaces in a space-delimited file. Set loadFlags = 2 to set bit 1.
Bit 2:	If set: Disables pre-counting of lines of data. See **Loading Very Large Files** below.
Bit 3:	If set: Disables unescaping of backslash characters in text columns. See **Escape Sequences** below.

All other bits are reserved and must be cleared.

/W Looks for wave names in a file. (With /G, /F, and /J.)

LoadWave cleans up column names to create standard, not liberal, wave names. See **Object Names** on page III-443 and **CleanupName** for details.

Use /W/A to read wave names from the file and then continue the load without displaying the normal wave name dialog.

Details

Without the /G, /F, /J, or /T flags, LoadWave loads Igor Binary files.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-21 for details.

When loading a general text file, the delimiters are always tab, comma and space.

Loading Custom Date Formats

Here are some examples showing custom date formats and how you would specify them using the /R flag: When loading data as delimited text, if you use a date format containing a comma, such as "October 11, 1999", you must use the /V flag to make sure that LoadWave will not treat the comma as a delimiter.

When loading a date format that consists entirely of digits, such as 991011, you must use the LoadWave/B to specify that the data is a date. Otherwise, LoadWave will treat it as a regular number.

Date Format	Specification
October 11, 1999	`/R={English, 2, 4, 1, 1, "Month DayOfMonth, Year", 40}`
Oct 11, 1999	`/R={English, 2, 3, 1, 1, "Month DayOfMonth, Year", 40}`
11 October 1999	`/R={English, 2, 4, 1, 1, "DayOfMonth Month Year", 40}`
11 Oct 1999	`/R={English, 2, 3, 1, 1, "DayOfMonth Month Year", 40}`
10/11/99	`/R={English, 1, 2, 1, 1, "Month/DayOfMonth/Year", 40}`
11-10-99	`/R={English, 1, 2, 2, 1, "DayOfMonth-Month-Year", 40}`
11-Jun-99	`/R={English, 1, 3, 2, 1, "DayOfMonth-Month-Year", 40}`
991011	`/R={English,1,2,2,1,"YearMonthDayOfMonth", 40}`
19991011	`/R={English,2,2,2,1,"YearMonthDayOfMonth", 40}`

Loading Very Large Files

The number of waves (columns) or points (rows) that LoadWave can handle when loading a text file is limited only by available memory.

You can improve the speed and efficiency of loading very large files (i.e., more than 50,000 lines of data) using the *numLines* parameter of the /L flag. Normally this parameter is used to load a section of the file instead of the whole file. However, in delimited, general text and fixed field text loads, the *numLines* parameter also specifies how many rows the waves should initially have. Thus all of the required memory is allocated at the start of the load, rather than increasing the number of wave rows over and over as more lines of data are loaded. When loading very large files, if you know the exact number of lines of data in the file, use the *numLines* parameter of the /L flag. If you don't know the exact number of lines, you can provide a number that is guaranteed to be larger.

If you omit the /L flag or if the *numLines* parameter is zero, and if you are loading a file greater than 500,000 bytes, and if you are running on Windows, LoadWave will automatically count the lines of data in the file so that the entire wave can be allocated before data loading starts. This acts as if you used /L and set *numLines* to the exact correct value. For very large files on Windows, this can speed the loading process considerably. For small files on Windows and for files of any size on Macintosh, it actually makes the load slower. That's why this feature takes effect only on Windows and only for files of greater than 500,000 bytes. You can disable this feature by using the /V flag and setting bit 2 to 1.

Escape Sequences

An escape sequence is a two-character sequence used to represent special characters in plain text. Escape sequences are introduced by a backslash character.

By default, in a text column, LoadWave interprets the following escape sequences: \t (tab), \n (linefeed), \r (carriage-return), \\ (backslash), \" (double-quote) and \' (single-quote). This works well with Igor's **Save** operation which uses escape sequences to encode the first four of these characters.

If you are loading a file that does not use escape sequences but which does contain backslashes, you can disable interpretation of these escape sequences by setting bit 3 of the *loadFlags* parameter of the /V flag. This is mainly of use for loading a text file that contains unescaped Windows file system paths.

Generating Wave Names

The /N flag automatically names new waves "wave" (or *baseName* if =*baseName* is used) plus a digit. The digit starts from zero and increments by one for each wave loaded from the file. If the resulting name conflicts with an existing wave, the existing wave is overwritten.

The /A flag is like /N except that it skips names already in use.

Specifying Characteristics of Individual Columns

The /B=*columnInfoStr* flag provides information to LoadWave for each column in a delimited text (/J), fixed field text (/F) or general text (/G) file. The flag overrides LoadWave's normal behavior. In most cases, you will not need to use it.

columnInfoStr is constructed as follows:

```
"<column info>;<column info>; … ;<column info>;"
```

where <column info> consists of one or more of the following:

C=<number>	The number of columns controlled by this column info specification. <number> is an integer greater than or equal to one.
F=<format>	A code that specifies the data type of the column or columns. <format> is an integer from -2 to 10. The meaning of the <format> is:

-2:	Text. The column will be loaded into a text wave.
-1:	Format unknown. It will deduce the format.
0 to 5:	Numeric.
6:	Date.
7:	Time.
8:	Date/Time.
9:	Octal number.
10:	Hexadecimal number.

The F= flag is used for delimited text and fixed field text files only. It is ignored for general text files.

N=<name>

A name to use for the column. <name> can be a standard name (e.g., wave0) or a quoted liberal name (e.g., 'Heart Rate'). If <name> is '_skip_' (including single quotation marks) then LoadWave will skip the column.

The N= flag works for delimited text, fixed field text and general text files.

T=<numtype>

A number that specifies what the numeric type for the column should be. This flag overrides the LoadWave/D flag. It has no effect on columns whose format is text. <numtype> must be one of the following:

2:	32-bit float.
4:	64-bit float.
8:	8-bit signed integer.
16:	16-bit signed integer.
32:	32-bit signed integer.
72:	8-bit unsigned integer.
80:	16-bit unsigned integer.
96:	32-bit unsigned integer.

W=<width>

The column field width for fixed field files. <width> is an integer greater than or equal to one. Fixed width files are FORTRAN-style files in which a fixed number of bytes is allocated for each column and spaces are used as padding.

The W= flag is used for fixed field text only.

Here is an example of the /B=*columnInfoStr* flag:

```
/B="C=1,F=-2,T=2,W=20,N=Factory; C=1,F=6,W=16,T=4,N=MfgDate;
C=1,F=0,W=16,T=2,N=TotalUnits; C=1,F=0,W=16,T=2,N=DefectiveUnits;"
```

This example is shown on two lines but in a real command it would be on a single line. In a procedure, it could be written as:

```
String columnInfoStr = ""
columnInfoStr += "C=1,F=-2,T=2,W=20,N=Factory;"
columnInfoStr += "C=1,F=6,T=4,W=16,N=MfgDate;"
columnInfoStr += "C=1,F=0,T=2,W=16,N=TotalUnits;"
columnInfoStr += "C=1,F=0,T=2,W=16,N=DefectiveUnits;"
```

Note that each flag inside the quoted string ends with either a comma or a semicolon. The comma separates one flag from the next within a particular column info specification. The semicolon marks the end of a column info specification. The trailing semicolon is required. Spaces and tabs are permitted within the string.

This example provides information about a file containing four columns.

The first column info specification is "`C=1;F=-2,T=2,W=20,N=Factory;`". This indicates that the specification applies to one column, that the column format is text, that the numeric format is single-precision floating point (but this has no effect on text columns), that the column data is in a fixed field width of 20 bytes, and that the wave created for this column is to be named Factory.

The second column info specification is "`C=1;F=6,T=4,W=16,N=MfgDate;`". This indicates that the specification applies to one column, that the column format is date, that the numeric format is double-precision floating point (double precision should always be used for dates), that the column data is in a fixed field width of 16 bytes, and that the wave created for this column is to be named MfgDate.

The third column info specification is "`C=1;F=0,T=2,W=16,N=TotalUnits;`". This indicates that the specification applies to one column, that the column format is numeric, that the numeric format is single-precision floating point, that the column data is in a fixed field width of 16 bytes, and that the wave created for this column is to be named TotalUnits.

The fourth column info specification is the same as the third except that the wave name is DefectiveUnits.

All of the items in a column specification are optional. The default value for each item in the column info specification is as follows:

C=<number>	C=1. Specifies that the column info describes one column.
F=<format>	F=-1. Determines the format as dictated by the /K flag. If /K=0 is used, LoadWave will automatically determine the column format.
N=<name>	N=_auto_. Generates the wave name as it would if the /B flag were omitted.
T=<numtype>	Defaults to T=4 (double precision) if the LoadWave/D flag is used or to T=2 (single precision) if the /D flag is omitted.
W=<width>	W=0. For a fixed width file, LoadWave will use the default field width specified by the /F flag unless you provide an explicit field width greater than 0 using W=<width>.

Taking advantage of the default values, we could abbreviate the example as follows:

```
/B="F=-2,W=20,N=Factory; F=6,T=4,W=16,N=MfgDate;
W=16,N=TotalUnits; W=16,N=DefectiveUnits;"
```

If the file were not a fixed field text file, we would omit the W= flag and the example would become:

```
/B="F=-2,N=Factory;F=6,T=4,N=MfgDate;N=TotalUnits;N=DefectiveUnits;"
```

Here are some more examples and discussion that illustrate the use of the /B=*columnInfoStr* flag.

In this example, the /B flag is used solely to specify the name to use for the waves created from the columns in the file:

```
/B="N=WaveLength; N=Absorbance;"
```

The wave names in the previous example are standard names. If you want to use liberal names, such as names containing spaces or dots, you must use single quotes. For example:

```
/B="N='Wave Number'; N='Reflection Angle';"
```

The name that you specify via N= can not be used if overwrite is off and there is already a wave with this name or if the name conflicts with a macro, function or operation or variable. In these cases, LoadWave generates a unique name by adding one or more digits to the name specified by the N= flag for the column in question. You can avoid the problem of a conflict with another wave name by using the overwrite (/O) flag or by loading your data into a newly-created data folder. You can minimize the likelihood of a name conflict with a function, operation or variable by avoiding vague names.

If you specify the same name in two N= flags, LoadWave will generate an error, so make sure that the names are unique.

Except if the specified name is '_skip_', the N= flag generates a name for one column only, even if the C= flag is used to specify multiple columns. Consider this example:

```
/B="C=10,N=Test;"
```

This ostensibly uses the name Test for 10 columns. However, wave names must be unique, so LoadWave will not do this. It will use the name Test for just the first column and the other columns will receive default names.

You can load a subset of the columns in the file using the /L flag. Even if you do this, the column info specifications that you provide via the /B flag start from the first column in the file, not from the first column to be loaded. For example, if you are using /L to skip columns 0 and 1, you must skip columns 0 and 1 in the column info specification, like this:

```
// Skip column 0 and 1 and name the successive columns
/L={0,0,0,2,0} /B="C=2;N=Column2;N=Column3;"
```

The "C=2;" part accepts default specifications for columns 0 and 1 and the subsequent specifications apply to subsequent columns.

You can achieve the same thing using /B without /L, like this:

```
/B="C=2,N='_skip_';N=Column2;N=Column3;"
```

Also, when loading data into a matrix wave, LoadWave uses only one name. If you specify more than one name, only the first is used. If you are loading data into a matrix and also skipping columns, the explanation above about skipping applies.

In this example, the /B flag solely specifies the format of each column in the file. The file in question starts with a text column, followed by a date column, followed by 3 numeric columns.

```
/B="F=-2; F=6; C=3,F=0"
```

In most cases, it is not necessary to use the F= flag because LoadWave can automatically deduce the formats. The flag is useful for those cases where it deduces the column formats incorrectly. It is also useful to force LoadWave to interpret a column as octal or hexadecimal because LoadWave can not automatically deduce these formats.

The numeric codes (0…10) used by the F= flag are the same as the codes used by the ModifyTable operation. If you create a table using the /E flag, the F= flag controls the numeric format of table columns.

The code -1 is not a real column format code. If you use F=-1 for a particular column, LoadWave will deduce the format for that column from the column text.

In this example, the /B flag is used solely to specify the width of each column in a fixed field file. This file contains a 20 character column followed by ten 16 character columns followed by three 24 character columns.

```
/B="C=1,W=20; C=10,W=16; C=3,W=24"
```

The field widths specified via W= override the default field width specified by the /F flag. If all of the columns in the file have the same field width then you can use just the /F flag.

You can load a subset of the columns in the file using the /L flag. Even if you do this, the column info specifications that you provide via the /B flag start from the first column in the file, not from the first column to be loaded.

LoadWave Text Encoding Issues

This section discusses text encoding issues of interest to advanced users. It assumes that you are familiar with the general topic of text encodings as explained under **Text Encodings** on page III-409.

Since Igor stores all text internally as UTF-8, it must convert text read from a file from the source text encoding to UTF-8. In order to do this it needs to know the source text encoding.

When loading an Igor binary wave file LoadWave ignores the /ENCG=*textEncoding* flag. The loaded wave's text encoding is determined as described under **LoadWave Text Encodings for Igor Binary Wave Files** on page III-424. The rest of this section applied to loading data from plain text files, not from Igor binary wave files.

When loading a text data file you can use the /ENCG=textEncoding flag to tell Igor what that text encoding is. See **Text Encoding Names and Codes** on page III-434 for a list of accepted values for *textEncoding*.

LoadWave uses the text encoding specified by /ENCG and the rules described under **Determining the Text Encoding for a Plain Text File** on page III-417 to determine the source text encoding for conversion of the text file's data to UTF-8. If you omit /ENCG or specify /ENCG=0, the specified text encoding is unknown and does not factor into the determination of the source text encoding. If following the rules does not identify a text encoding that works for converting the file's text to UTF-8, Igor displays the Choose Text Encoding dialog.

If the file contains nothing but ASCII characters, as is often the case, then any byte-oriented text encoding will work and there is no need to use the /ENCG flag.

When you are loading a huge file (e.g., hundreds of megabytes), finding a valid source text encoding may add a noticeable amount to the time it takes to load the file. If you know that the file is either all ASCII or is valid UTF-8, you can tell LoadWave to skip text encoding conversion altogether using an optional parameter, like this:

`/ENCG={1,4}`

"1" tells LoadWave that the text is valid as UTF-8, meaning that it is all ASCII or, if it contains non-ASCII characters, they are properly encoded as UTF-8.

"4" tells LoadWave to assume that the text is valid as UTF-8 and skip all validation and conversion.

In testing with a 200 MB delimited text file containing 1 million rows and 20 columns, we found that using /ENCG={1,4} saved about 10% of the time.

NOTE: If you use this flag but the file is not valid UTF-8 and you are loading data into text wave, the text waves will wind up with invalid data which will result in errors when you use the waves later.

As noted above, if following the rules does not identify a text encoding that works for converting the file's text to UTF-8, Igor displays the Choose Text encoding dialog. If you are loading many files using an unattended, automated procedure, displaying this dialog will cause your procedure to grind to a halt. You can prevent this by using another optional flag, like this:

`/ENCG={1,8}`

If you use this flag and LoadWave can not determine the source text encoding for a file, it will return an error. If you want your procedure to continue with other files you must check for and handle the error using **GetRTError** on page V-270.

Output Variables

LoadWave sets the following variables:

`V_flag`	Number of waves loaded.
`S_fileName`	Name of the file being loaded.
`S_path`	File system path to the folder containing the file.
`S_waveNames`	Semicolon-separated list of the names of loaded waves.

S_path uses Macintosh path syntax (e.g., “`hd:FolderA:FolderB:`”), even on Windows. It includes a trailing colon. If LoadWave is loading from the Clipboard, S_path is set to `""`.

When LoadWave presents an Open File dialog and the user cancels, V_flag is set to 0 and S_fileName is set to `""`.

See Also

The **ImageLoad** operation.

See **Importing Data** on page II-117 for further information on loading waves, including loading multidimensional data from HDF, PICT, TIFF and other graphics files. Check the “More Extensions:File Loaders” folder other file-loader extensions.

Setting Bit Parameters on page IV-12 for further details about bit settings.

Loess

Loess [***flags***] **srcWave =** ***srcWaveName*** [**,** **factors =** ***factorWaveName1***
  [**,** ***factorWaveName2*** …]]

The Loess operation smooths *srcWaveName* using locally-weighted regression smoothing. This algorithm is sometimes classified as a “nonparametric regression” procedure. The regression can be constant, linear, or quadratic. A robust option that ignores outliers is available. See **Basic Algorithm**, **Robust Algorithm**, and **References** for additional details and terminology.

This implementation works with waveforms, XY pairs of waves, false-color images, matrix surfaces, and multivariate data (one dependent data wave with multiple independent variable data waves).

Unlike the **FilterFIR** operation, Loess discards any NaN input values and will not generate a result that is wholly NaN.

Parameters

srcWaveName is the input data to be smoothed. It may be a one-dimensional or a two-dimensional wave, and it may contain NaNs.

When no /DEST flag is specified, Loess will overwrite *srcWaveName* with the smoothed result.

If *srcWaveName* is one-dimensional and no factors are provided, X values are derived from the X scaling of *srcWaveName*.

If *srcWaveName* is two-dimensional, the factors keyword is not permitted and the X and Y values are derived from the X and Y scaling of *srcWaveName*.

Higher dimensions of *srcWaveName* are not supported.

The optional factors parameter(s) provide the independent variable value(s) that correspond to the observed value in *srcWaveName*.

Note: Cleveland et al. (1992) use the term "multiple factors" instead of "multivariate", hence the keyword name "factors" is used to denote these waves.

Use one factors wave when *srcWaveName* is the one-dimensional Y wave of an XY data pair:

```
srcWaveName[i] = someFunction(factorWaveName1[i])
```

Use multiple factors waves when *srcWaveName* contains the values of a multivariate function. "Multivariate" means that *srcWaveName* contains the observed results of a process that combines multiple independent input variables:

```
srcWaveName[i] = someFunction(factorWaveName1[i], factorWaveName2[i],…)
```

A maximum of 10 factors waves is supported.

All factors wave(s) must be numeric, noncomplex, one-dimensional and have the same number points as *srcWaveName*.

Any NaN values in *srcWaveName*[i], *factorWaveName1*[i], *factorWaveName2*[i], … cause all corresponding values to be ignored. Factors waves may contain NaN values only when /DFCT is specified.

Loess does not support NaNs in any of *destFactorWaveName1*, *destFactorWaveName2*,… and the results are undefined.

Flags

/CONF={*confInt*, *ciPlusWaveName* [,*ciMinusWaveName*]}

confInt specifies the confidence interval (a probability value from 0 to 1). *ciPlusWaveName* and the optional *ciMinusWaveName* are the names of new or overwritten output waves to hold the fitted value ± the confidence interval.

Note: /CONF uses large memory allocations, approximately N*N*8 bytes, where N is the number of points in *srcWaveName* (see **Memory Details**).

/DEST=*destWaveName* Specifies the name of the wave to hold the smoothed data. It creates *destWaveName* if it does not already exist or overwrites it if it does. The x (and possibly y) scaling of *destWave* determines the independent (factor) coordinates unless /DFCT={*destFactorWaveName1* [,*destFactorWaveName2*...]} is also specified.

/DFCT Specifies that the /DEST wave's x (and possibly y) scaling determines the independent (factor) coordinates at which to compute the smoothed data.

/DFCT={*destFactorWaveName1* [,*destFactorWaveName2*…]}

Specifies the names of one-dimensional waves providing the independent coordinates at which to compute the smoothed data.

If /DFCT={...} is used, the same number of waves must be specified for /DFCT and for factors = {*factorWaveName1* [, *factorWaveName2*…]}, though their lengths may (and usually will) be different. The length of *destFactorWaveName* waves must be the same as that of the *destWaveName* wave.

All destination factor waves must be numeric, noncomplex, and one-dimensional. The number of destination factor waves must match the number of source factor waves (if specified), or match the dimensionality of *srcWaveName* (one destination factor wave if *srcWaveName* is one-dimensional, two destination factors waves if *srcWaveName* is two-dimensional.)

The values in the destination factor waves may not be NaN.

/E=*extrapolate*	Set *extrapolate* to nonzero to use a slower fitting method that computes values beyond the domain defined by the source factors. This is the “surface” parameter named “direct” in Cleveland et al. (1992). The default is *extrapolate* = 0, which uses the “interpolate” surface parameter, instead.
/N=*neighbors*	Specifies the number of values in the smoothing window. If *neighbors* is even, the next larger odd number is used. When *neighbors* is less than two, no smoothing is done. The default is `0.5*numpnts(`*`srcWaveName`*`)` rounded up to the next odd integer or 3, whichever is larger. Use either /N or /SMTH, but not both.
/NORM [=*norm*]	Set *norm* to 0 when specifying multiple factors and they all have the same scale and meaning, for example multiple factors all in units of meters. The default is *norm* = 1, which normalizes each factor independently when computing the weighting function. This is appropriate when the factors are not dimensionally related, for example one factor measures wavelength and another measures temperature.
/ORD=*order*	Specifies the regression (fitting) order, the *d* parameter in Cleveland (1977): *order*=0: Fits a constant to the locally-weighted neighbors around each point. *order*=1: Fits a line to the locally-weighted neighbors around each point (Lowess smoothing). *order*=2: Default; fits a quadratic (Loess smoothing).
/PASS=*passes*	The number of iterations of local weighting and regression fitting performed. The minimum is 1 and the default is 4. In Cleveland (1977), *passes* corresponds to *r*.
/R [=*robust*]	Set *robust* to nonzero to use a robust fitting method that uses a bisquare weight function instead of the normal tricube weight function. This corresponds to the “symmetric” family in Cleveland et al. (1992). The robust method is less affected by outliers. The default is *robust* = 0, which is the “gaussian” family in Cleveland et al. (1992).
/SMTH=*sf*	Another way to express the number of values in the smoothing window, $0 \leq sf \leq 1$. The default is 0.5. To compute *neighbors* from *sf*, use: *`neighbors`*` = 1+floor(`*`sf`*`*numpnts(`*`srcWaveName`*`)).` Use either /N or /SMTH, but not both.
/TIME=*secs*	*secs* is the number of seconds allowed to complete the calculation before either warning (default) or stopping. If the stop bit (4) of /V=*verbose* is set, the caculcation stops after the alloted time. If the diagnostic bit of /V=*verbose* is also set, warnings about the calculation exceeding the allotted time are printed to the history area. As an example, use /TIME=30/V=6 to abort calculations longer than 30 seconds and print the warning to the history area.

/V [=*verbose*]	Controls how much information to print to the history area. *verbose* is a bitwise parameter with each bit controlling one aspect:

verbose=0:	Prints nothing to the history area (default).
verbose=1:	Prints the number of observations, equivalent number of parameters, and residual standard error.
verbose=2:	Prints diagnostic information and error messages.
verbose=4	Use with /TIME. If this bit is set, calculations that exceed *secs* seconds are aborted.

Set *verbose* to 6 to both limit the time and print diagnostic and error messages.

/V alone is the same as /V=3, which prints all information.

S_info contains all the informational messages regardless of the value of

/Z[=z]	Set *z* to nonzero to prevent an error from stopping execution. Use the V_flag variable to see if the smoothing succeeded.

Basic Algorithm

The basic locally-weighted regression algorithm fits a constant, line, or quadratic to each point of the source data, using data that falls within the given span of neighbors over the factor data. Data outside of the span is ignored (given a weight of zero), and data inside the span is given a weight that depends on the distance of the data from the point being evaluated: data closer to the point being evaluated have higher weights and have a greater affect on the fit.

The basic algorithm uses the “tricube” weighting function to emphasize near values and deemphasize far values. For the one-factor case (simple XY data), the weighting function can be expressed as:

$$w_i = \left(1 - \left|\frac{x - x_i}{\max_q\left(x - x_i\right)}\right|^3\right)^3,$$

where $\max_q(x - x_i)$ is the maximum Euclidean distance of the q factor values within the given span from the factor point (x) whose observation (y) value is being evaluated.

The weights are applied to the factor values in the span to compute the constant, linear, or quadratic regression at x.

When multiple factors are used, the Euclidean distance is computed using one dimension per factor. The default is to normalize each factor’s range by the standard deviation of that factor’s values before computing the Euclidean distances. When factors are dimensionally equal, use the /NORM=0 option to skip this normalization. (See /NORM, about “dimensionally equal”.)

Robust Algorithm

The robust algorithm adds to the basic algorithm a method to identify and remove outliers by rejecting values that exceed a threshold related to the “median absolute deviation” of the basic regression’s residuals. The remaining values are used to compute robust “bisquare” weighting values:

$$r_i = \begin{cases} \left(1 - \left|\dfrac{e_i}{6 \cdot median\left(\left|e_i\right|\right)}\right|^2\right)^2 & for \quad 0 \le \left|e_i\right| < 6 \cdot median\left(\left|e_i\right|\right) \\ 0 & for \quad \left|e_i\right| > 6 \cdot median\left(\left|e_i\right|\right) \end{cases}$$

where e_i is the difference between the observed value and the regression’s fitted value, and $\text{median}(|e_i|)$ is evaluated for all the observed values.

These robust weighting values are multiplied with the original weighting values and a new regression (with new residuals) is computed. This process repeats 4 times by default. Use the /PASS flag to specify a different number of repetitions.

Details

Loess sets the variable V_flag to 0 if the smoothing was successful, or to an error code if not. Unlike other operations, the /Z flag allows execution to continue even if input parameters are in error.

Information printed to the history area when /V is set is always stored in the S_info string, even if /V=0 (the default). S_Info also contains the error message text if V_flag is an error code.

The error messages are described in Cleveland et al. (1992). They are often more dire than they seem.

The error message “Span too small. Fewer data values than degrees of freedom” usually means that the /SMTH or /N values are too small. The error code returned in V_Flag for this case is 1106.

The “Extrapolation not allowed with blending” (V_Flag = 1115) error usually means that the destination factors are trying to compute observations outside of the source factors domain without specifying /E=1. This happens if the /DEST *destWaveName* already exists and has X scaling that extends beyond the X scaling of *srcWaveName*. The solution is either kill the /DEST wave, limit the X scaling to the domain of the source wave, or use /E=1.

Memory Details

Loess requires a lot of memory, especially with the /CONF flag. Even without /CONF, the memory allocations exceed this approximation:

Number of bytes allocated = number of points in *srcWaveName* * 216

With /CONF, Loess can allocate large amounts of memory, approximately N*N*8 bytes, where N is the number of points in *srcWaveName*. The 2GB memory limit of 32-bit addressing limits *srcWaveName* to approximately 10,000 points when using /CONF.

More precisely, the memory allocation may be approximated by this function:

```
Function ComputeLoessMemory(srcPoints, numFactorsWaves, doConfidence)
   Variable srcPoints         // number of points in srcWave, aka N
   Variable numFactorsWaves// 1 or number of factors (independent variables)
   Variable doConfidence      // true if /CONF is specified

   Variable doubles= 9 * srcPoints             // 9 allocated double arrays
   doubles += 5 * numFactorsWaves * srcPoints  // 5 more arrays
   doubles += (1+numFactorsWaves) * srcPoints  // another array
   doubles += (1+numFactorsWaves) * srcPoints  // another array
   doubles += (4+5) * srcPoints                // two more arrays
   if( doConfidence )
      doubles += srcPoints*srcPoints           // one HUGE array
   endif
   Variable bytes= doubles * 8
   return bytes
End

Macro DemoLoessMemory()
   Make/O wSrcPoints={10,100,1000,2000,3000,5000,7500,10000,12500,15000,20000}
   Duplicate/O wSrcPoints, loessMemory, loessMemory3, loessMemoryConf
   SetScale d, 0,0, "Points", wSrcPoints
   SetScale d, 0,0, "Bytes", loessMemory, loessMemory3, loessMemoryConf
   loessMemory= ComputeLoessMemory(wSrcPoints[p],1, 0)// 1 factor (X) no /CONF
   loessMemory3= ComputeLoessMemory(wSrcPoints[p],3, 0)// 3 factors (X,Y,Z) no /CONF
   loessMemoryConf= ComputeLoessMemory(wSrcPoints[p],1, 1)// 1 factor (X)with /CONF
   Display loessMemory vs wSrcPoints; Append loessMemory3 vs wSrcPoints
   ModifyGraph highTrip(bottom)=1e+08, rgb(loessMemory3)=(0,0,65535)
   ModifyGraph lstyle(loessMemory3)=2
   Legend
   Display loessMemoryConf vs wSrcPoints
   AutoPositionWindow
   ModifyGraph highTrip(bottom)=1e+08
End
```

Examples

1-D, factors are X scaling, output in new wave:

```
Make/O/N=200 wv=2*sin(x/8)+gnoise(1)
KillWaves/Z smoothed                     // ensure Loess creates a new wave
Loess/DEST=smoothed srcWave=wv           // 21-point loess.
```

```
Display wv; ModifyGraph mode=3,marker=19
AppendtoGraph smoothed; ModifyGraph rgb(smoothed)=(0,0,65535)
```

1-D, output in existing wave with more points than original data:

```
Make/O/N=100 short=2*cos(x/4)+gnoise(1)
Make/O/N=300 out; SetScale/I x, 0, 99, "" out  // same X range
Loess/DEST=out/DFCT/N=30 srcWave=short
Display short; ModifyGraph mode=3,marker=19
AppendtoGraph out
ModifyGraph rgb(out)=(0,0,65535),mode(out)=2,lsize(out)=2
```

1-D Y vs X wave data interpolated to waveform (Y vs X scaling) with 99% confidence interval outputs:

```
// NOx = f(EquivRatio)
// Y wave
// Note: The next 2 Make commands are wrapped to fit on the page.
Make/O/D NOx = {4.818, 2.849, 3.275, 4.691, 4.255, 5.064, 2.118, 4.602, 2.286, 0.97,
3.965, 5.344, 3.834, 1.99, 5.199, 5.283, 3.752, 0.537, 1.64, 5.055, 4.937, 1.561};

// X wave (Note that the X wave is not sorted)
Make/O/D EquivRatio = {0.831, 1.045, 1.021, 0.97, 0.825, 0.891, 0.71, 0.801,  1.074,
1.148, 1, 0.928, 0.767, 0.701, 0.807, 0.902,   0.997, 1.224, 1.089, 0.973, 0.98, 0.665};

// Interpolate to dense waveform over X range
Make/O/D/N=100 fittedNOx
WaveStats/Q EquivRatio
SetScale/I x, V_Min, V_max, "", fittedNOx
Loess/CONF={0.99,cp,cm}/DEST=fittedNOx/DFCT/SMTH=(2/3) srcWave=NOx, factors={EquivRatio}
Display NOx vs EquivRatio; ModifyGraph mode=3,marker=19
AppendtoGraph fittedNOx, cp,cm       // fit and confidence intervals
ModifyGraph rgb(fittedNOx)=(0,0,65535)
ModifyGraph mode(fittedNOx)=2,lsize(fittedNOx)=2
```

Interpolate X, Y, Z waves as a 3D surface.

```
// Note: The next 3 Make commands are wrapped to fit on the page.
Make/O/D vels= {1769, 1711, 1538, 1456, 1608, 1574, 1565, 1692, 1538, 1505, 1764, 1723,
1540, 1441, 1428, 1584, 1552, 1690, 1673, 1548, 1485, 1526, 1536, 1591, 1671, 1647, 1608,
1562, 1740, 1753, 1590, 1466, 1409, 1429}
Make/O/D ews={8.46279, 3.46303, -1.51508, -6.51483, 16.597, -5.95541, -28.5078, 9.68438,
-6.00159, -21.7557, 14.263, 6.02058, -2.25772, -10.536, -18.7785, 10.7509, -6.07024,
1.77531, 0.767701, -0.235545, -1.24315, 21.7298, 10.3964, 0.133859, -10.1733, -20.4359,
13.7658, -8.88429, 10.8869, 4.91318, -0.0649319, -5.06469, -10.0428, -11.0601}
Make/O/D nss={-38.1732, -15.6207, 6.83407, 29.3865, 3.67947, -1.32028, -6.32004, -
10.3852, 6.43591, 23.3302, -37.1565, -15.6842, 5.88156, 27.4473, 48.9196, 10.0254, -
5.66059, -40.6613, -17.5832, 5.39486, 28.4729, 43.5833, 20.852, 0.26848, -20.4045, -
40.988, 3.0518, -1.9696, -49.1077, -22.1619, 0.292889, 22.8453, 45.3001, 49.8887}

// Evaluate the smoothed function as interpolated image
Make/O/N=(50,50) velsImage
WaveStats/Q ews
SetScale/I x, V_Min, V_Max, "" velsImage    // destination factors
WaveStats/Q nss
SetScale/I y, V_Min, V_Max, "" velsImage    // are X and Y scaling

Loess/DEST=velsImage/DFCT/NORM=0/SMTH=0.75/E/Z srcWave=vels, factors={ews,nss}
// Display source data as a contour with x, y markers.
Display; AppendXYZContour vels vs {ews,nss}
ModifyContour vels xymarkers=1, labels=0
ColorScale

// Display interpolated surface as an image
AppendImage velsImage
ModifyImage velsImage ctab= {*,*,Grays256,0}
ModifyGraph mirror=2
```

References

Cleveland, W.S., Robust locally weighted regression and smoothing scatterplots, *J. Am. Stat. Assoc.*, *74*, 829-836, 1979.

Cleveland, W.S., E. Grosse, and M.-J. Shyu, A Package of C and Fortran Routines for Fitting Local Regression Models, Technical Report, Bell Labs, 54pp, 1992. <http://cm.bell-labs.com/cm/ms/departments/sia/wsc/webpapers.html>.

NIST/SEMATECH, LOESS (aka LOWESS), in *NIST/SEMATECH e-Handbook of Statistical Methods*, <http://www.itl.nist.gov/div898/handbook/pmd/section1/pmd144.htm>, 2005.

See Also

Smooth, **Interpolate2**, **interp**, **MatrixFilter**, **MatrixConvolve**, and **ImageInterpolate**.

log

`log(`*`num`*`)`

The log function returns the log base 10 of *num*.

It returns -INF if *num* is 0, and returns NaN if *num* is less than 0.

To compute a logarithm base n use the formula:

$$\log_n(x) = \frac{\log(x)}{\log(n)}.$$

See Also

The **ln** function.

logNormalNoise

`logNormalNoise(`*`m,s`*`)`

The logNormalNoise function returns a pseudo-random value from the lognormal distribution function whose probability distribution function is

$$f(x,m,s) = \frac{1}{xs\sqrt{2\pi}} \exp\left\{ -\frac{\left[\ln(x) - m\right]^2}{2s^2} \right\},$$

with a mean $\exp\left(m + \frac{1}{2}s^2\right),$

and variance $\exp\left(2m^2 + s^2\right)\left[\exp\left(s^2\right) - 1\right]..$

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-344.

Chapter III-12, **Statistics** for a function and operation overview.

LombPeriodogram

`LombPeriodogram` `[`*`flags`*`]` ***`srcTimeWave, srcAmpWave`*** **`[,`** ***`srcFreqWave`*** **`]`**

The LombPeriodogram is used in spectral analysis of signal amplitudes specified by *srcAmpWave* which are sampled at possibly random sampling times given by *srcTimeWave*. The only assumption about the sampling times is that they are ordered from small to large time values. The periodogram is calculated for either a set of frequencies specified by *srcFreqWave* (slow method) or by the flags /FR and /NF (fast method). Unless you specify otherwise, the results of the operation are stored by default in W_LombPeriodogram and W_LombProb in the current data folder.

Flags

/DESP=*datafolderAndName*

Saves the computed P-values in a wave specified by *datafolderAndName*. The destination wave will be created or overwritten if it already exists. *dataFolderAndName* can include a full or partial path with the wave name.

Creates by default a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-66 for details.

If this flag is not specified, the operation saves the P-values in the wave W_LombProb in the current data folder.

/DEST=*datafolderAndName*

Saves the computed periodogram in a wave specified by *datafolderAndName*. The destination wave will be created or overwritten if it already exists. *datafolderAndName* can include a full or partial path with the wave name (`/DEST=root:bar:destWave`).

Creates by default a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-66 for details.

If this wave is not specified the operation saves the resulting periodogram in the wave W_LombPeriodogram in the current data folder.

/FR=fRes	Use /FR to specify the frequency resolution of the output. This flag is used together with /NF to specify the range of frequencies for which the periodogram is computed. Note that *fRes* is also the lowest frequency in the output.
/NF=numFreq	Use /NF to specify the number of frequencies at which the periodogram is computed. The range of frequencies of the periodogram is then [*fRes*, (*numFreq*-1)**fRes*].
/Q	Quiet mode; suppresses printing results in the history area.
/Z	Do not report any errors.

Details

The LombPeriodogram (sometimes referred to as "Lomb-Scargle" periodogram) is useful in detection of periodicities in data. The main advantage of this approach over Fourier analysis is that the data are not required to be sampled at equal intervals. For an input consisting of N points this benefit comes at a cost of an O(N^2) computations which becomes prohibitive for large data sets. The operation provides the option of computing the periodogram at equally spaced (output) frequencies using /FR and /NF or at completely arbitrary set of frequencies specified by *srcFreqWave*. It turns out that when you use equally spaced output frequencies the calculation is more efficient because certain parts of the calculation can be factored.

The Lomb periodogram is given by

$$LP(\omega)=\frac{1}{2\sigma^2}\left\{\frac{\left[\sum_{i=0}^{N-1}(y_i-\bar{y})\cos\left[\omega(t_i-\tau)\right]\right]^2}{\sum_{i=0}^{N-1}\cos^2\left[\omega(t_i-\tau)\right]}+\frac{\left[\sum_{i=0}^{N-1}(y_i-\bar{y})\sin\left[\omega(t_i-\tau)\right]\right]^2}{\sum_{i=0}^{N-1}\sin^2\left[\omega(t_i-\tau)\right]}\right\}$$

Here yi is the ith point in *srcAmpWave*, ti is the corresponding point in *srcTimeWave*,

$$\bar{y}=\frac{1}{N}\sum_{i=0}^{N-1}y_i,$$

$$\tan(2\omega\tau) = \frac{\sum_{i=0}^{N-1} \sin(2\omega t_i)}{\sum_{i=0}^{N-1} \cos(2\omega t_i)}.$$

and

$$p = 1 - \left\{1 - \exp\left[LP(w)\right]\right\}^{N_{ind}}.$$

In the absence of a Nyquist limit, the number of independent frequencies that you can compute can be estimated using:

$$N_{ind} = -6.362 + 1.193N + 0.00098N^2.$$

This expression was given by Horne and Baliunas derived from least square fitting. Nind is used to compute the P-values as:

$$p = 1 - \left\{1 - \exp\left[LP(w)\right]\right\}^{N_{ind}}.$$

Note that you can invert the last expression to determine the value of LP(w) for any significance level.

See Also

The **FFT** and **DSPPeriodogram** operations.

References

1. J.H. Horne and S.L. Baliunas, *Astrophysical Journal*, 302, 757-763, 1986.
2. N.R. Lomb, *Astrophysics and Space Science*, 39, 447-462, 1976.
3. W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes*, 3rd ed., Section 13.8.

lorentzianNoise

`lorentzianNoise(`*`a`*`,`*`b`*`)`

The function returns a pseudo-random value from a Lorentzian distribution

$$f(x) = \frac{1}{\pi}\frac{(b/2)}{(x-a)^2 + (b/2)^2}.$$

Here *a* is the center and *b* is the full line width at half maximum (FWHM).

See Also

SetRandomSeed, **enoise**, **gnoise**.

Noise Functions on page III-344.

Chapter III-12, **Statistics** for a function and operation overview.

LowerStr

`LowerStr(`*`str`*`)`

The LowerStr function returns a string expression identical to *str* except that all upper-case ASCII characters are converted to lower-case.

See Also

The **UpperStr** function.

Macro

Macro *macroName*([*parameters*]) [:*macro type*]

The Macro keyword introduces a macro. The macro will appear in the Macros menu unless the procedure file has an explicit Macros menu definition. See Chapter IV-4, **Macros** and **Macro Syntax** on page IV-110 for further information.

MacroList

MacroList(*matchStr*, *separatorStr*, *optionsStr*)

The MacroList function returns a string containing a list of the names of user-defined procedures that start with the Proc, Macro, or Window keywords that also satisfy certain criteria. Note that if the procedures need to be compiled, then MacroList may not list all of the procedures.

Parameters

Only macros having names that match *matchStr* string are listed. See **WaveList** for examples.

The first character of *separatorStr* is appended to each macro name as the output string is generated. *separatorStr* is usually ";" for list processing (See **Processing Lists of Waves** on page IV-187 for details on list processing).

optionsStr is used to further qualify the macros. It is a string containing keyword-value pairs separated by commas. Available options are:

KIND:*nk*	Determines the kind of procedure returned. *nk*=1: List Proc procedures. *nk*=2: List Macro procedures. *nk*=4: List Window procedures. *nk* can be the sum of these values to match multiple procedure kinds. For example, use 3 to list both Proc and Macro procedures.
NPARAMS:*np*	Restricts the list to macros having exactly *np* parameters. Omitting this option lists macros having any number of parameters.
SUBTYPE:*typeStr*	Lists macros that have the type *typeStr*. That is, you could use ButtonControl as *typeStr* to list only macros that are action procedures for buttons.
WIN:*windowNameStr*	Lists macros that are defined in the named procedure window. "Procedure" is the name of the built-in procedure window. **Note**: Because *optionsStr* keyword-value pairs are comma separated and procedure window names can have commas in them, the WIN: keyword must be the last one specified.

Examples

To list all Macros with three parameters:

```
Print MacroList("*",";","KIND:2,NPARAMS:3")
```

To list all Macro, Proc, and Window procedures in the main procedure window whose names start with b:

```
Print MacroList("b*",";","WIN:Procedure")
```

See Also

The **DisplayProcedure** operation and the **FunctionList**, **OperationList**, **StringFromList**, and **WinList** functions.

For details on procedure subtypes, see **Procedure Subtypes** on page IV-193, as well as **Button**, **CheckBox**, **SetVariable**, and **PopupMenu**.

magsqr

`magsqr(`*`z`*`)`

The magsqr function returns the sum of the squares of the real and imaginary parts of the complex number *z*, that is, the magnitude squared.

Examples

Assume waveCmplx is complex and waveReal is real.

```
waveReal = sqrt(magsqr(waveCmplx))
```

sets each point of waveReal to the magnitude of the complex points in waveCmplx.

You may get unexpected results if the number of points in waveCmplx differs from the number of points in waveReal because of interpolation. See **Mismatched Waves** on page II-75 for details.

See Also

The **cabs** function.

WaveMetrics provides Igor Technical Note 006, "DSP Support Macros" which uses the magsqr function to compute the magnitude of FFT data, and Power Spectral Density with options such as windowing and segmenting. See the Technical Notes folder. Some of the techniques discussed there are available as Igor procedure files in the "WaveMetrics Procedures:Analysis:" folder.

Make

`Make [`*`flags`*`]` ***`waveName`*** **`[,`** ***`waveName`*** **`]…`**
`Make [`*`flags`*`]` ***`waveName`*** **`[= {`*`n0,n1,…`*`}]…`**
`Make [`*`flags`*`]` ***`waveName`*** **`[= {{`*`n0,n1,…`*`},{`*`n0,n1,…`*`},…}]…`**

The Make operation creates the named waves. Use braces to assign data values when creating the wave.

Flags

/B	Makes 8-bit signed integer waves or unsigned waves if /U is present.
/C	Makes complex waves.
/D	Makes double precision waves.
/DF	Wave holds data folder references. See **Data Folder References** on page IV-72 for more discussion.
/FREE	Creates a free wave. Allowed only in functions and only if a simple name or wave reference structure field is specified. See **Free Waves** on page IV-84 for more discussion.
/I	Makes 32-bit signed integer waves or unsigned waves if /U is present.
/L	Makes 64-bit signed integer waves or unsigned waves if /U is present. Requires Igor Pro 7.00 or later.
/N=*n*	*n* is the number of points each wave will have. If *n* is an expression, it must be enclosed in parentheses: `Make/N=(myVar+1) aNewWave`
/N=(*n1*, *n2*, *n3*, *n4*)	*n1*, *n2*, *n3*, *n4* specify the number of rows, columns, layers and chunks each wave will have. Trailing zeros can be omitted (e.g., `/N=(`*`n1`*`, `*`n2`*`, 0, 0)` can be abbreviated as `/N=(`*`n1`*`, `*`n2`*`))`.
/O	Overwrites existing waves in case of a name conflict. After an overwrite, you cannot rely on the contents of the waves and you will need to reinitialize them or to assign appropriate values.
/R	Makes real value waves (default).
/T	Makes text waves.

/T=*size*	Makes text waves with pre-allocated storage. *size* is the number of bytes preallocated by Igor for each element in each text wave. The waves are not initialized - it is up to you to initialize them. Preallocation can dramatically speed up text wave assignment when the wave has a very large number of points but only when all strings assigned to the wave are exactly the same size as the preallocation size.
/U	Makes unsigned Integer waves.
/W	Makes 16-bit signed integer waves or unsigned waves if /U is present.
/WAVE	Wave holds wave references. See **Wave References** on page IV-65 for more discussion.
/Y=*type*	Specifies wave data type. See details below.

Wave Data Types

As a replacement for the above number type flags you can use /Y=*numType* to set the number type as an integer code. See the **WaveType** function for code values. The /Y flag overrides other type flags but you may still need to use the /C, /T, /DF or /WAVE flags to define the type of an automatic WAVE reference variable when used in user functions.

Details

Unless overridden by the flags, the created waves have the default length, type, precision, units and scaling. The factory defaults are:

Note: The *preferred* precision set by the Miscellaneous Settings dialog only presets the Make Waves dialog checkbox and determines the precision of imported waves. It does not affect the Make operation.

Property	Default
Number of points	128
Precision	Single precision floating point
Type	Real
dimensions	1
x, y, z, and t scaling	offset=0, delta=1 (“point scaling”)
x, y, z, and t units	`""` (blank)
Data Full Scale	0, 0
Data units	`""` (blank)

See Also

The **SetScale**, **Duplicate**, and **Redimension** operations.

MakeIndex

`MakeIndex [/A/C/R] `*`sortKeyWaves, indexWaveName`*

The MakeIndex operation sets the data values of *indexWaveName* such that they give the ordering of *sortKeyWaves*.

For simple sorting problems, MakeIndex is not needed. Just use the **Sort** operation.

Parameters

sortKeyWaves is either the name of a single wave, to use a single sort key, or the name of multiple waves in braces, to use multiple sort keys.

indexWaveName must specify a numeric wave.

All waves must be of the same length and must not be complex.

Flags

/A	Alphanumeric. When *sortKeyWaves* includes text waves, the normal sorting places "wave1" and "wave10" before "wave9". Use /A to sort the number portion numerically, so that "wave9" is sorted before "wave10".
/C	Case-sensitive. When *sortKeyWaves* includes text waves, the ordering is case-insensitive unless you use the /C flag which makes it case-sensitive.
/LOC	Performs a locale-aware sort. When *sortKeyWaves* includes text waves, the text encoding of the text waves' data is taken into account and sorting is done according to the sorting conventions of the current system locale. This flag is ignored if the text waves' data encoding is unknown, binary, Symbol, or Dingbats. This flag cannot be used with the /A flag. See Details for more information. The /LOC flag was added in Igor Pro 7.00.
/R	Reverse the index so that ordering is from largest to smallest.

Details

MakeIndex is used in preparation for a subsequent **IndexSort** operation. If /R is used the ordering is from largest to smallest. Otherwise it is from smallest to largest.

When the /LOC flag is used, the bytes stored in the text wave at each point are converted into a Unicode string using the text encoding of the text wave data. These Unicode strings are then compared using OS specific text comparison routines based on the locale set in the operating system. This means that the order of sorted items may differ when the same sort is done with the same data under different operating systems or different system locales.

When /LOC is omitted the sort is done on the raw text without regard to the waves' text encoding.

See Also

MakeIndex and IndexSort Operations on page III-127.

MandelbrotPoint

MandelbrotPoint(*x*, *y*, *maxIterations*, *algorithm*)

The MandelbrotPoint function returns a value between 0 and *maxIterations* based on the Mandelbrot set complex quadratic recurrence relation z[n] = z[n-1]^2 + c where *x* is the real component of c, *y* is the imaginary component of c and z[0] = 0.

The returned value is the number of iterations the equation was evaluated before |z[n]| > 2 (the escape radius of the Mandelbrot set), or maxIterations, whichever is less.

Parameters

algorithm=0	The "Escape Time" algorithm returns the integer n which is the number of iterations until |z[n]| > 2.
algorithm=1	The "Renormalized Iteration Count Algorithm" algorithm returns a floating point value which is a refinement of the number of iterations n by adding the quantity: `5 - ln( ln( |z[n+4]| ) ) / ln(2)` (which requires four more iterations of the recurrence relation). The returned value is clipped to maxIterations.

See Also

The "MultiThread Mandelbrot Demo" experiment.

References

http://en.wikipedia.org/wiki/Mandelbrot_set

http://linas.org/art-gallery/escape/escape.html

MarcumQ

`MarcumQ(`*`m, a, b`*`)`

The MarcumQ function returns the generalized Q-function defined by the integral

$$Q_m(a,b) = \int_b^{\infty} u\left(\frac{u}{a}\right)^{m-1} \exp\left(-\frac{(a^2+u^2)}{2}\right) I_{m-1}(au)\,du$$

where I_k is the modified Bessel function of the first kind and order k.

Its applications have been primarily in the fields of communication and detection theory. However, an interesting interpretation of its result with m=1 and appropriate parameter scaling is the fractional power of a two-dimensional circular Gaussian function within a displaced circular aperture.

Depending on the input arguments, the MarcumQ function may be computationally intensive but you can abort the calculation at any time.

References

Cantrell, P.E., and A.K. Ojha, Comparison of Generalized Q-Function Algorithms, *IEEE Transactions on Information Theory*, *IT-33*, 591-596, 1987.

Simon, M. K., A New Twist on the Marcum Q-Function and Its Application, *IEEE Communications Letters*, 3, 39-41, 1998.

MarkPerfTestTime

`MarkPerfTestTime` ***`idval`***

Use the MarkPerfTestTime operation for performance testing of user-defined functions in conjunction with `SetIgorOption DebugTimer`. When used between `SetIgorOption DebugTimer, Start` and `SetIgorOption DebugTimer, Stop`, MarkPerfTestTime stores the ID value and the time of the call in a buffer. When `SetIgorOption DebugTimer, Stop` is called the contents of the buffer are dumped to a pair of waves: W_DebugTimerIDs will contain the ID values and W_DebugTimerVals will contain the corresponding times of the calls relative to the very first call. The timings use the same high precision mechanism as the StartMSTimer and StopMSTimer calls.

By default, `SetIgorOption DebugTimer, Start` allocates a buffer for up to 10000 entries. You can allocate a different sized buffer using `SetIgorOption DebugTimer, Start=bufsize`.

See Also

SetIgorOption, **StartMSTimer**, and **StopMSTimer**.

Additional documentation can be found in an example experiment, PerformanceTesting.pxp, and a WaveMetrics procedure file, PerformanceTestReport.ipf.

MatrixCondition

`MatrixCondition(`*`wave2D, mode`*`)`

MatrixCondition returns the estimated reciprocal of the condition number of a 2D square matrix wave2D.

The condition number is the product of the norm of the matrix with the norm of the inverse of the matrix (see details below). The type of norm is determined by the value of the mode parameter. 1-norm is used if mode is 1 and infinity-norm is used otherwise.

The MatrixCondition function was added in Igor Pro 7.00.

Details

The function uses LAPACK routines to estimate the reciprocal condition number by first obaining the norm of the input matrix and then using LU decomposition to obtain the norm of the inverse of the matrix. The estimate returned is

$$reciprocalCon = \frac{1}{\|wave2D\| * \|wave2D^{-1}\|},$$

where the norms are selected by the choice of the mode parameter. The 1-norm of matrix A with elements aij is defined as

$$\|A\|_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}|,$$

and the infinity-norm is defined by

$$\|A\|_\infty = \max_{1 \le i \le m} \sum_{j=1}^{n} |a_{ij}|.$$

The function returns a NaN if there is any error in the input parameters.

References

http://en.wikipedia.org/wiki/Matrix_norm

See Also

MatrixSVD provides a condition number for L2 norm using the ratio of singular values.

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-131 for more about Igor's matrix routines.

MatrixConvolve

`MatrixConvolve [/R=`*`roiWave`*`]` ***`coefMatrix, dataMatrix`***

The MatrixConvolve operation convolves a small coefficient matrix *coefMatrix* into the destination *dataMatrix*.

Flags

/R=*roiWave*	Modifies only data contained inside the region of interest. The ROI wave should be 8-bit unsigned with the same dimensions as *dataMatrix*. The interior of the ROI is defined by zeros and the exterior is any nonzero value.

Details

On input *coefMatrix* contains an *N*x*M* matrix of coefficients where *N* and *M* should be odd. Generally *N* and *M* will be equal. If *N* and *M* are greater than 13, it is more efficient to perform the using the Fourier transform (see **FFT**).

The convolution is performed in place on the data matrix and is acausal, i.e., the output data is not shifted.

Edges are handled by replication of edge data.

When *dataMatrix* is an integer type, the results are clipped to limits of the given number type. For example, unsigned byte is clipped to 0 to 255.

MatrixConvolve works also when both *coefMatrix* and *dataMatrix* are 3D waves. In this case the convolution result is placed in the wave M_Convolution in the current data folder, and the optional /R=*roiWave* is required to be an unsigned byte wave that has the same dimensions as *dataMatrix*.

This operation does not support complex waves.

See Also

MatrixFilter and **ImageFilter** for filter convolutions.

Matrix Math Operations on page III-131 for more about Igor’s matrix routines.

The **Loess** operation.

MatrixCorr

`MatrixCorr` `[`**`/COV`**`]`**`[/DEGC]`** ***`waveA`*** `[,` ***`waveB`***`]`

The MatrixCorr operation computes the correlation or covariance or degree of correlation matrix for the input 1D wave(s).

If we denote elements of *waveA* by $\{x_i\}$ and elements of *waveB* by $\{y_i\}$ then the correlation matrix for these waves is the vector product of the form:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} y_1 & y_2 & y_3 & \cdots & y_n \end{bmatrix}^* = \begin{bmatrix} x_1y_1^* & x_1y_2^* & x_1y_3^* & \cdots & x_1y_n^* \\ x_2y_1^* & x_2y_2^* & x_2y_3^* & & x_2y_n^* \\ x_3y_1^* & x_3y_2^* & x_3y_3^* & & x_3y_n^* \\ \vdots & & & & \\ x_ny_1^* & x_ny_2^* & x_ny_3^* & \cdots & x_ny_n^* \end{bmatrix}$$

where * denotes complex conjugation. If you use the optional *waveB* then the matrix is the cross correlation matrix. *waveB* must have the same length of *waveA* but it does not have to be the same number type.

Flags

The flags are mutually exclusive; only one matrix can be generated at a time.

/COV	Calculates the covariance matrix. The covariance matrix for the same input is formed in a similar way after subtracting from each vector its mean value and then dividing the resulting matrix elements by (n-1) where n is the number of elements of *waveA*. Results are stored in the M_Corr or M_Covar waves in the current data folder.
/DEGC	Calculates the complex degree of correlation. The degree of correlation is defined by: $\deg C = \frac{M\_Covar}{\sqrt{Var(waveA) \cdot Var(waveB)}}$, where *M_Covar* is the covariance matrix and *Var*(*wave*) is the variance of the wave. The complex degree of correlation should satisfy: $0 \le \lvert \deg C \rvert \le 1$.

Examples

The covariance matrix calculation is equivalent to:

```
Variable N=1/(DimSize(waveA,0)-1)
Variable ma=mean(waveA,-inf,inf)
Variable mb=mean(waveB,-inf,inf)
waveA-=ma
waveB-=mb
MatrixTranspose/H waveB
MatrixMultiply waveA,waveB
M_product*=N
```

See Also

Matrix Math Operations on page III-131 for more about Igor's matrix routines.

References

Hayes, M.H., *Statistical Digital Signal Processing And Modeling,* 85 pp., John Wiley, 1996.

MatrixDet

`matrixDet(`*`dataMatrix`*`)`

The MatrixDet function returns the determinant of *dataMatrix*. The matrix wave must be a real, square matrix or else the returned value will be NaN.

Details

The function calculates the determinant using LU decomposition. If, following the decomposition, any one of the diagonal elements is either identically zero or equal to 10^{-100}, the return value of the function will be zero.

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-131 for more about Igor's matrix routines.

MatrixDot

`MatrixDot(`*`waveA`*`, `*`waveB`*`)`

The MatrixDot function calculates the inner (scalar) product for two 1D waves. A 1D wave **A** represents a vector in the sense:

$$\mathbf{A} = \sum \alpha_i \hat{e}_i, \qquad \hat{e}_i \text{ is a unit vector.}$$

Given two such waves **A** and **B**, the inner product is defined as

$$ip = \sum \alpha_i \beta_i .$$

When both *waveA* and *waveB* are complex and the result is assigned to a complex-valued number MatrixDot returns:

$$ipc = \sum \alpha_i^* \beta_i .$$

If the result is assigned to a real number, MatrixDot returns:

$$ip = \left| \sum \alpha_i^* \beta_i \right| .$$

If either *waveA* or *waveB* is complex and the result is assigned to a real-valued number, MatrixDot returns:

$$ip = \left| \sum \alpha_i \beta_i \right| .$$

When the result is assigned to a complex-valued number MatrixDot returns:

$$ipc = \sum \alpha_i \beta_i .$$

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-131 for more about Igor's matrix routines.

MatrixEigenV

`MatrixEigenV [`*`flags`*`] `*`matrixA`*` [, `*`matrixB`*`]`

MatrixEigenV computes the eigenvalues and eigenvectors of a square matrix using LAPACK routines.

Flags for General Matrices

/C	Creates complex valued output for real valued input waves. This is equivalent to converting the input to complex, with zero imaginary component, prior to computing the eigenvalues and eigenvectors. The main benefit of this format is that the output has simple packing of eigenvalues and eigenvectors. See the example in the **Details for General Matrices** section below. The /C flag was added in Igor Pro 7.00.
/B=*balance*	Determines how the input matrix should be scaled and or permuted to improve the conditioning of the eigenvalues. *balance*=0 (default), 1, 2, or 3, corresponding respectively to N, P, S, or B in the LAPACK routines. Applicable only with the /X flag. 0: Do not scale or permute. 1: Permute. 2: Do diagonal scaling. 3: Scale and permute.
/L	Calculates for left eigenvectors.
/O	Overwrites *matrixWave*, requiring less memory.
/R	Calculates for right eigenvectors.
/S=*sense*	Determines which reciprocal condition numbers are calculated. *sense*=0 (default), 1, 2, or 3, corresponding respectively to N, E, V, or B in the LAPACK routines. Applicable only with the /X flag. 0: None. 1: Eigenvalues only. 2: Right eigenvectors. 3: Eigenvalues and right eigenvectors. If sense is 1 or 3 you must compute both left and right eigenvectors. **NOTE**: /S is applicable only with the /X flag or for the generalized eigenvalue problem.
/X	Uses LAPACK expert routines, which require additional parameters (see /B and /S flags). The operation creates additional waves: The W_MatrixOpInfo wave contains in element 0 the ILO, in element 1 the IHI, and in element 2 the ABNRM from the LAPACK routines. The wave W_MatrixRCONDE contains the reciprocal condition numbers for the eigenvalues. The wave W_MatrixRCONDV contains the reciprocal condition number for the eigenvectors.

Flags for Symmetric Matrices

/SYM	Computes the eigenvalues of an NxN symmetric matrix and stores them in the wave W_eigenValues. You must specify this flag if you want to use the special routines for symmetric matrices. The number of eigenvalues is stored in the variable V_npnts. Because W_eigenValues has N points, only the first V_npnts will contain relevant eigenvalues. When using this flag with complex input the matrix is assumed to be Hermitian.
/EVEC	Computes eigenvectors in addition to eigenvalues. Eigenvectors will be stored in the wave M_eigenVectors, which is of dimension NxN. The first V_npnts columns of the wave will contain the V_npnts eigenvectors corresponding to the eigenvalues in W_eigenValues. /EVEC must be preceded by /SYM.

/RNG={*method*,*low*,*high*}

Determines what is computed:

method=0:	Computes all the eigenvalues or eigenvectors (default).
method=1:	Computes eigenvalues for *low* and *high* double precision range.
method=2:	Computes eigenvalues for *low* and *high* integer indices (1 based). For example, to compute the first 3 eigenvalues use: `/RNG={2,1,3}`.

/RNG must be preceded by /SYM.

Flags for Generalized Eigenvalue Problem

These are the same flags as for general (non-symmetric) matrices above except for /O and /X which are not supported for the generalized eigenvalue solution.

Common Flags

/Z	No error reporting (except for setting V_flag).

Details

There are three mutually exclusive branches for the operation. The first is designed for a square matrix input *matrixA*. The operation computes the solution to the problem

$$\mathbf{Ax} = \lambda\mathbf{x},$$

where A is the input matrix, x is an eigenvector and λ is an eigenvalue.

The second branch is designed for symmetric matrices A, i.e., when

$$\mathbf{A} = \mathbf{A}^{\mathbf{T}},$$

where the superscript T denotes a transpose.

The third branch of the operation is designed to solve the generalized eigenvalue problem,

$$\mathbf{Ax} = \lambda\mathbf{Bx},$$

where A and B are square matrices, x is an eigenvector and λ is an eigenvalue.

Each branch of the operation supports its own set of flags as shown above. All branches support input of single and double precision in real or complex waves. If you specify both *matrixA* and *matrixB* then they must have the same number type.

Details for General Matrices

The eigenvalues are returned in the 1D complex wave W_eigenValues. The eigenvectors are returned in the 2D wave M_R_eigenVectors or M_L_eigenVectors.

The calculated eigenvectors are normalized to unit length.

Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having the positive imaginary part first.

If the *jth* eigenvalue is real, then the corresponding eigenvector *u*(*j*)=*M*[][*j*] is the *jth* column of M_L_eigenVectors or M_R_eigenVectors. If the *jth* and (*j*+1)*th* eigenvalues form a complex conjugate pair, then *u*(*j*) = *M*[][*j*] + *i***M*[][*j*+1] and *u*(*j*+1) = *M*[][*j*] - *i***M*[][*j*+1].

Example

```
Function TestMatrixEigenVReal()
   Make/D/N=(2,2)/O eee={{0,1},{-1,0}}
   MatrixEigenV/R eee
   Wave W_eigenvalues, M_R_eigenVectors
   MatrixOP/O firstEV=cmplx(col(M_R_eigenVectors,0),col(M_R_eigenVectors,1))
   MatrixOP/O aa=eee x firstEV - W_eigenvalues[0]*firstEV
   Print aa
End

Function TestMatrixEigenVComplex()
```

```
	Make/D/N=(2,2)/O eee={{0,1},{-1,0}}
	MatrixEigenV/R/C eee
	Wave W_eigenvalues, M_R_eigenVectors
	MatrixOP/O aa=eee x col(M_R_eigenVectors,0) - W_eigenValues[0]*col(M_R_eigenVectors,0)
	Print aa
End
```

Details for Symmetric Matrices

The LAPACK routines that compute the eigenvalues and eigenvectors of symmetric matrices claim to use the Relatively Robust Representation whenever possible. If your matrix is symmetric you should use this branch of the operation (/SYM) for improved accuracy.

Details for Generalized Eigenvalue Problem

Here the right eigenvectors (/R) are solutions to the equation

$$\mathbf{Ax} = \lambda\mathbf{Bx},$$

and the left eigenvectors (/L) are solutions to

$$\mathbf{x^H A} = \lambda\mathbf{x^H B},$$

where the superscript H denotes the conjugate transpose.

When both *matrixA* and *matrixB* are real valued, the operation creates the following waves in the current data folder:

W_alphaValues	Contains the complex alpha values.
W_betaValues	Contains the real-valued denominator such that the eigenvalues are given by $\lambda_j = \dfrac{\alpha_j}{\beta_j}$.
M_leftEigenVectors and M_rightEigenVectors	Real valued waves where columns correspond to eigenvectors of the equation.

Every point in the wave W_alphaValues corresponds to an eigenvalue. When the imaginary part of W_alphaValues[j] is zero, the eigenvalue is real and the corresponding eigenvector is also real e.g., M_rightEigenVectors[][j]. When the imaginary part is positive then there are two eigenvalues that are complex-conjugates of each other with corresponding complex eigenvectors given by

```
cmplx(M_rightEigenVectors[][j],M_rightEigenVectors[][j+1])
```

and

```
cmplx(M_rightEigenVectors[][j],-M_rightEigenVectors[][j+1])
```

When both *matrixA* and *matrixB* are complex the operation creates the complex waves:

W_alphaValues, W_betaValues, M_leftEigenVectors, M_rightEigenVectors

with the ratio W_alphaValues[j]/W_betaValues[j] expressing the generalized eigenvalue. The corresponding M_leftEigenVectors[][j] and M_rightEigenVectors[][j] are the respective left and right eigenvectors. The simplicity of the complex case suggests that when *matrixA* and *matrixB* are real it is best to convert them to complex waves prior to executing matrixEigenV.

Depending on the choice of /S the operation also calculates the reciprocal condition number for the eigenvalues (stored in W_reciprocalConditionE) and the reciprocal condition number of the eigenvectors (stored in W_reciprocalConditionV). Note that a zero entry in W_reciprocalConditionV implies that the eigenvalues could not be ordered.

See Also

Matrix Math Operations on page III-131 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

Symmetric matrices can also be decomposed using the **MatrixSchur** operation and using **MatrixOp** chol.

MatrixFilter

`MatrixFilter [flags] Method dataMatrix`

The MatrixFilter operation performs one of several standard image filter type operations on the destination *dataMatrix*.

Note: The parameters below are also available in ImageFilter. See **ImageFilter** for additional parameters.

Parameters

Method selects the filter type. *Method* is one of the following names:

avg	*n*x*n* average filter.
FindEdges	3x3 edge finding filter.
gauss	*n*x*n* gaussian filter.
gradN, gradNW, gradW, gradSW, gradS, gradSE, gradE, gradNE	3x3 North, NorthWest, West, … pointing gradient filter.
median	*n*x*n* median filter. You can assign values other than the median by specifying the desired rank using the /M flag.
min	*n*x*n* minimum rank filter.
max	*n*x*n* maximum rank filter.
NanZapMedian	*n*x*n* filter that only affects data points that are NaN. Replaces them with the median of the *n*x*n* surrounding points. Unless /P is used, automatically cycles through matrix until all NaNs are gone or until *cols***rows* iterations.
point	3x3 point finding filter `8*center-outer`.
sharpen	3x3 sharpening `filter=(12*center-outer)/4`.
sharpenmore	3x3 sharpening `filter=(9*center-outer)`.
thin	Calculates binary image thinning using neighborhood maps based on the algorithm in *Graphics Gems IV*, p. 465. **Note**: The thin keyword to MatrixFilter will be removed someday. The functionality will be available — just not as a part of MatrixFilter. The /R flag does not apply to the lame duck thin keyword.

Flags

/B=*b*	Specifies value that is considered background. Used with thin. If object is black on white background, use 255. If object is white on a black background, use 0.
/F=*value*	Specifies the value in the ROI wave that marks excluded pixels. *value* is either 0 or 1. This flag was added in Igor Pro 7.00. By default, and for compatibility with Igor Pro 6, *value*=0. Use /F=1 if your ROI wave contains 1 for pixels to be excluded.
/M=*rank*	Assigns a pixel value other than the median when used with the median filter. Valid *rank* values are between 0 and n^2-1 (for the default median *rank*= $n^{2/2}$).
/N=*n*	For any method described above as "*n*x*n*", you can specify that the filtering kernel will be a square matrix of size *n*. In the absence of the /N flag, the default size is 3.
/P=*p*	Filter passes over the data *p* times. The default is one pass.
/R=*roiWave*	Only the data outside the region of interest will be modified. *roiWave* should be an 8-bit unsigned wave with the same dimensions as the data matrix. The exterior of the ROI is defined by zeros and the interior is any nonzero value.
/T	Applies the thining algorithm of Zhang and Suen with the thin parameter. The wave M_MatrixFilter contains the results; the input wave is not overwritten.

Details

This operation does not support complex waves.

See Also

ImageFilter operation for additional options. **Matrix Math Operations** on page III-131 for more about Igor's matrix routines. The **Loess** operation.

References

Heckbert, Paul S., (Ed.), *Graphics Gems IV*, 575 pp., Morgan Kaufmann Publishers, 1994.

Zhang, T. Y., and C. Y. Suen, A fast thinning algorithm for thinning digital patterns, *Comm. of the ACM*, *27*, 236-239, 1984.

MatrixGaussJ

`MatrixGaussJ` *`matrixA, vectorsB`*

The MatrixGaussJ operation solves matrix expression A*x=b for column vector x given matrix A and column vector b. The operation can also be used to calculate the inverse of a matrix.

Parameters

matrixA is a NxN matrix of coefficients and *vectorsB* is a NxM set of right-hand side vectors.

Details

On output, the array of solution vectors *x* is placed in M_x and the inverse of *A* is placed in M_Inverse.

If the result is a singular matrix, V_flag is set to 1 to indicate the error. All other errors result in an alert, and abort any calling procedure.

All output objects are created in the current data folder.

An error is generated if the dimensioning of the input arrays is invalid.

This routine is provided for completeness only and is not recommended for general work (use LU decomposition — see **MatrixLUD**). MatrixGaussJ does calculate the inverse matrix but that is not generally needed either.

See Also

Matrix Math Operations on page III-131 for more about Igor's matrix routines. The **MatrixLUD** operation.

MatrixGLM

`MatrixGLM [/Z] matrixA, matrixB, waveD`

The MatrixGLM operation solves the general Gauss-Markov Linear Model problem (GLM) which minimizes the 2-norm of a vector y

$$\min \|y\|_2 \quad subject\,to \quad d = Ax + By.$$

A is *matrixA* (an NxM wave), B is *matrixB* (an NxP wave), and d is provided by *waveD* which is a 1D wave of N rows. The vectors x and y are the results of the calculation; they are stored in output waves Mat_X and Mat_Y in the current data folder.

Flags

/Z	In the event of an error, MatrixGLM will not return the error to Igor, which would cause procedure execute to abort. Your code should use the V_flag output variable to detect and handle errors.

Details

All input waves must have the same numeric type. Supported types are single-precision and double-precision floating point, both real and complex. The output waves Mat_X and Mat_Y have the same numeric type as the input.

The LAPACK algorithm assumes that M <= N <= M+P and

$$rank(A) = M,$$

$$rank(AB) = N.$$

Under these assumptions there is a unique solution x and a minimal 2-norm solution y, which are obtained using a generalized QR factorization of A and B. If the operation completes successfully the variable V_Flag is set to zero. Otherwise it contains a LAPACK error code.

Output Variables

V_flag	Set to 0 if MatrixGLM succeeds or to a LAPACK error code.

See Also

Matrix Math Operations on page III-131 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

MatrixInverse

`MatrixInverse` `[`***`flags`***`]` ***`srcWave`***

The MatrixInverse operation calculates the inverse or the pseudo-inverse of a square matrix. *srcWave* may be real or complex.

MatrixInverse saves the result in the wave M_Inverse in the current data folder.

Flags

/D	Creates the wave W_W that contains eigenvalues of the singular value decomposition (SVD) for the pseudo-inverse calculation. If one or more of the eigenvalues are small, the matrix may be close to singular.
/G	Calculates only the direct inverse; does not affect calculation of pseudo-inverse. By default, it calculates the inverse of the matrix using LU decomposition. The inverse is calculated using Gauss-Jordan method. The only advantage in using Gauss-Jordan is that it is more likely to flag singular matrices than the LU method.
/O	Overwrites the source with the result.
/P	Calculates the pseudo-inverse of a square matrix using the SVD algorithm. The calculated pseudo-inverse is a unique minimal solution to the problem: $$\min_{\mathbf{X}\in\mathbb{R}^{n\times m}} \left\|\mathbf{AX} - I_m\right\|.$$

Example

```
Make/N=(2,2) mat0={{2,3},{1,7}}
MatrixInverse mat0                    // Creates wave M_inverse
// Check the results
MatrixOp/O mat1=M_inverse x mat0
Print mat1                            // Verify that you got the identity matrix
```

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-131 for more about Igor's matrix routines.

References

See sec. 5.5.4 of:

Golub, G.H., and C.F. Van Loan, *Matrix Computations*, 2nd ed., Johns Hopkins University Press, 1986.

MatrixLinearSolve

`MatrixLinearSolve` `[`***`flags`***`]` ***`matrixA matrixB`***

The MatrixLinearSolve operation solves the linear system *matrixA* *X=*matrixB* where *matrixA* is an N-by-N matrix and *matrixB* is an N-by-NRHS matrix of the same data type.

Flags

/M=*method*	Determines the solution method which best suites input *matrixA*.
	method=1: Uses simple LU decomposition (default). See also LAPACK documentation for SGESV, CGESV, DGESV, and ZGESV.
	Creates the wave W_IPIV that contains the pivot indices that define the permutation matrix P. Row (i) if the matrix was interchanged with row ipiv(i).
	method=2: If *matrixA* is band diagonal, you also have to specify /D. See also LAPACK documentation for SGBSV, CGBSV, DGBSV, and ZGBSV.
	Creates the wave W_IPIV, which contains the pivot indices that define the permutation matrix P. Row (i) if the matrix was interchanged with row ipiv(*i*). Also note that if you are using the /O flag, the overwritten waves may have a different dimensions.
	method=4: For tridiagonal matrix; still expecting full matrix in *matrixA*, but it will ignore the data in the elements outside the 3 diagonals. See also LAPACK documentation for SGTSV, CGTSV, DGTSV, and ZGTSV.
	method=8: Symmetric/hermitian. See also LAPACK documentation for SPOSV, CPOSV, DPOSV, and ZPOSV.
	method=16: Complex symmetric (complex only). See also LAPACK documentation for CSYSV and ZSYSV.
/D={*sub*,*super*}	Specifies a band diagonal matrix. The subdiagonal (*sub*) and superdiagonal (*super*) size must be positive integers.
/L	Uses the lower triangle of *matrixA*. /L and /U are mutually exclusive flags.
/U	Uses the upper triangle of *matrixA*. /U is the default.
/O	Overwrites *matrixA* and *matrixB* with the results of the operation. This will save on the amount of memory needed.
/Z	No error reporting.

Details

If /O is not specified, the operation also creates the n-by-n wave M_A and the n-by-nrhs solution wave M_B.

The variable V_flag is created by the operation. If the operation completes successfully, V_flag is set to zero, otherwise it is set to the LAPACK error code.

See Also

Matrix Math Operations on page III-131 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

MatrixLinearSolveTD

`MatrixLinearSolveTD [/Z]` ***`upperW, mainW, lowerW, matrixB`***

The MatrixLinearSolveTD operation solves the linear system *TDMatrix**X = *matrixB*. In the matrix product on the left hand side, *TDMatrix* is a tridiagonal matrix with upper diagonal *upperW*, main diagonal *mainW*, and lower diagonal *lowerW*. It solves for vector(s) X depending on the number of columns (NRHS) in *matrixB*.

Flags

/Z	No error reporting.

Details

The input waves can be single or double precision (real or complex). Results are returned in the wave M_TDLinearSolution in the current data folder. The wave *mainW* determines the size of the main diagonal

(N). All other waves must match it in size with *upperW* and *mainW* containing one less point and *matrixB* consisting of N-by-NRHS elements of the same data type.

MatrixLinearSolveTD should be more efficient than MatrixLinearSolve with respect to storage requirements.

MatrixLinearSolveTD creates the variable V_flag, which is zero when it finishes successfully.

See Also

Matrix Math Operations on page III-131; the **MatrixLinearSolve** and **MatrixOp** operations.

MatrixLLS

`MatrixLLS [/O/Z/M=`*`method`*`]` ***`matrixA matrixB`***

The MatrixLLS operation solves overdetermined or underdetermined linear systems involving MxN *matrixA*, using either QR/LQ or SV decompositions. Both *matrixA* and *matrixB* must have the same number type. Supported types are real or complex single precision and double precision numbers.

Flags

/M=*method*	Specifies the decomposition method. *method*=0: Decomposition is to QR or LQ (default). Creates the 2D wave M_A, which contains details of the QR/LQ factorization. *method*=1: Singular value decomposition. Creates the 2D wave MA, which contains the right singular vectors stored row-wise in the first `min(m,n)` rows. Creates the 1D wave M_SV, which contains the singular values of *matrixA* arranged in decreasing order.
/O	Overwrites *matrixA* with its decomposition and *matrixB* with the solution vectors. This requires less memory.
/Z	No error reporting.

Details

When the /O flag is not specified, the solution vectors are stored in the wave M_B, otherwise the solution vectors are stored in *matrixB*. Let *matrixA* be *m* rows by *n* columns and *matrixB* be an *m* by NRHS (if NRHS=1 it can be omitted). If $m \geq n$, MatrixLLS solves the least squares solution to an overdetermined system:

$$Minimize\left\|matrixB - matrixA \times \mathbf{X}\right\|.$$

Here the first *n* rows of M_B contain the least squares solution vectors while the remaining rows can be squared and summed to obtain the residual sum of the squares. If you are not interested in the residual you can resize the wave using, for example:

```
Redimension/N=(n,NRHS) M_B
```

If $m<n$, MatrixLLS finds the minimum norm solution of the underdetermined system:

$$matrixA \times \mathbf{X} = matrixB.$$

In this case, the first *m* rows of M_B contain the minimum norm solution vectors while the remaining rows can be squared and summed to obtain the residual sum of the squares for the solution. If you are not interested in the residual you can resize the wave using, for example:

```
Redimension/N=(m,NRHS) M_B
```

Note: Here *matrixB* consists of one or more column vectors B corresponding to one or more solution vectors X that are computed simultaneously. If *matrixB* consists of a single column, M_B is a 2D matrix wave that contains a single solution column.

The variable V_flag is set to 0 when there is no error; otherwise it contains the LAPACK error code.

See Also

Matrix Math Operations on page III-131 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

MatrixLUBkSub

`MatrixLUBkSub` ***`matrtixL, matrixU, index, vectorB`***

The MatrixLUBkSub operation provides back substitution for LU decomposition.

Details

This operation is used to solve the matrix equation Ax=b after you have performed LU decomposition (see **MatrixLUD**). Feed this routine M_Lower, M_Upper and W_LUPermutation from MatrixLUD along with your right-hand-side vector b. The solution vector x is returned as M_x. The array b can be a matrix containing a number of b vectors and the M_x will contain a corresponding set of solution vectors.

Generates an error if the dimensions of the input matrices are not appropriate.

See Also

Matrix Math Operations on page III-131 for more about Igor's matrix routines.

MatrixLUD

`MatrixLUD` `[`***`flags`***`]` ***`matrixA`***

The MatrixLUD operation computes the LU factorization of a matrix. The general form of the factorization/decomposition is expressed in terms of matrix products:

```
M_Pt x srcWave = M_Lower x M_Upper
```

M_Pt, M_Lower and M_Upper are outputs created by MatrixLUD.

M_Pt is the transpose of the permutation matrix, M_Lower is a lower triangular matrix with 1's on the main diagonal and M_Upper is an upper triangular (or trapezoidal) matrix.

The MatrixLUD operation was substantially changed in Igor Pro 7.00. See the /B flag for information about backward compatibility.

Flags

/B	This flag is provided for backward compatibility only; it is not compatible with any other flag. /B makes MatrixLUD behave as it did in Igor Pro 6. This flag is deprecated and will be removed in a future version of Igor. The input is restricted to a 2D real valued, single or double precision square matrix. The outputs (all double precision) are stored in the waves M_Upper, M_Lower and W_LUPermutation in the current data folder. The W_LUPermutation output wave was needed for solving a linear system of equations using the back substitution routine, **MatrixLUBkSub**. For better computation methods see **MatrixLinearSolve**, **MatrixLinearSolveTD** and **MatrixLLS**.
/CMF	Uses Combined Matrix Format where the upper and lower matrix factors are combined into a single matrix saved in the wave M_LUFactors in the current data folder. The upper matrix factor is constructed from the main and from the upper diagonals of M_LUFactors. The lower matrix factor is constructed from the lower diagonals of M_LUFactors and setting the main diagonal to 1.
/MIND	Finds the minimum magnitude diagonal element of M_Upper and store it in V_min. This is useful for investigating the behavior of the determinant of the matrix when it is close to being singular.
/PMAT	Saves the transpose of the permutation matrix in a double precision wave M_Pt in the current data folder. Note that the permutation matrix is orthogonal and so the inverse of the matrix is equal to its transpose.
/SUMP	Computes the sum of the phases of the elements on the main diagonal of M_Upper and store in the variable V_Sum. V_Sum is initialized to NaN and is set only if /SUMP is specified and M_Upper is complex.

Details

The input matrix *srcWave* is an MxN real or complex wave of single or double precision. Use **MatrixLUDTD** if your input is tri-diagonal.

The main results of the factorization are stored in the waves M_Lower, M_Upper and M_Pt. Alternatively the lower and upper factors can be combined and stored in the wave M_LUFactors (see /CMF). The waves M_Lower, M_Upper and M_LUFactors have the same data type as the input wave. M_Pt is always double precision.

When the input matrix *srcWave* is square (NxN), the resulting matrices have the same dimensions (NxN). You can reconstruct the input using the MatrixOp expression:

```
MatrixOp/O rA=(M_Pt^t) x (M_Lower x M_Upper)
```

If the input matrix is rectangular (NxM) the reconstruction depends on the size of N and M. If N<M:

```
MatrixOp/O rA=(M_Pt^t) x (subRange(M_lower,0,N-1,0,N-1) x M_Upper)
```

If N>M:

```
MatrixOp/O rA=(M_Pt^t) x M_lower x subRange(M_Upper,0,M-1,0,M-1)
```

The variable V_flag is set to zero if the operation succeeds and to 1 otherwise (e.g., if the input is singular). When you use the /B flag the polarity of the matrix is returned in the variable V_LUPolarity. The variables V_Sum and V_min are also set by some of the flag options above.

See Also

MatrixLUDTD, **MatrixLUBkSub**, **MatrixLinearSolve**, **MatrixLinearSolveTD**, **MatrixLLS**, **MatrixOp**

Matrix Math Operations on page III-131 for more about Igor's matrix routines.

MatrixLUDTD

`MatrixLUDTD [`*`flags`*`] `*`srcMain, srcUpper, srcLower`*

The MatrixLUDTD operation computes the LU factorization of a tri-diagonal matrix. The general form of the factorization/decomposition is expressed in terms of matrix products:

```
M_Pt x triDiagonalMat = M_Lower x M_Upper
```

`triDiagonalMat` is the matrix defined by the main diagonal specified by *srcMain*, the upper diagonal specified by *srcUpper*, and the lower diagonal specified by *srcLower*.

M_Pt is an output wave created when the /PMAT flag is present. M_Lower and M_Upper are output waves created when the /FM flag is present. M_Pt is the transpose of the permutation matrix, M_Lower is a lower triangular matrix with 1's on the main diagonal and M_Upper is an upper triangular (or trapezoidal) matrix.

Flags

/MIND	Finds the minimum magnitude diagonal element of M_Upper and stores it in V_min. This feature is useful if you want to investigate the behaviour of the determinant of the matrix when it is close to being singular.
/PMAT	Saves the transpose of the permutation matrix in the wave M_Pt in the current data folder. Note that the permutation matrix is orthogonal and so the inverse of the matrix is equal to its transpose.
/SUMP	Computes the sum of the phases of the elements on the main diagonal of M_Upper and store in the variable V_Sum. Note that the variable is initialized to NaN and that it is not set unless this flag is specified and M_Upper is complex.
/FM	The full matrix output is stored in the waves M_Lower and M_Upper in the current data folder.

Details

You specify the tridiagonal matrix using three 1D waves of the same data type (single or double precision real or complex).

If /FM is present the output of the operation consists of two 2D waves and one 1D wave:

M_Lower is a lower triangular matrix with 1's on the main diagonal.

M_Upper is an upper triangular (or trapezoidal) matrix.

W_PIV is 1D wave containing pivot indices.
See code example below for implementation details.

If /FM is omitted the output of the operation consists of five 1D waves:

W_Diagonal is the main diagonal of matrixU.

W_UDiagonal is the first upper diagonal of M_Upper.

W_U2Diagonal is the second diagonal of M_Upper.

W_LDiagonal is the first lower diagonal of M_Lower.

W_PIV is a vector of pivot indices.

In this case M_Lower can be constructed (see below) from W_LDiagonal and the pivot index wave W_PIV.

If you are working with tridiagonal matrices you can take advantage of MatrixOp functionality to reconstruct your outputs. For example:

```
MatrixOp/O M_Upper=Diagonal(W_diagonal)
MatrixOp/O M_Upper=setOffDiag(M_Upper,1,W_UDiagonal)
MatrixOp/O M_Upper=setOffDiag(M_Upper,2,W_U2Diagonal)
```

These commands can be combined into a single command line.

The construction of M_Lower is a bit more complicated and can be accomplished for real data using the following code:

```
Function MakeLTMatrix(W_diagonal,W_LDiagonal,W_PIV)
	Wave W_diagonal,W_LDiagonal,W_PIV

	Variable i,N=DimSize(W_diagonal,0)
	MatrixOp/O M_Lower=setOffDiag(ZeroMat(N,N,4),-1,W_LDiagonal)
	M_Lower=p==q ? 1:M_Lower[p][q]       // Set the main diagonal to 1's
	MatrixOp/O index=W_PIV-1             // Convert from 1-based array
	for(i=1;i<=N-2;i+=1)
		if(index[i]!=i)
			variable j,tmp
			for(j=0;j<=i-1;j+=1)
				tmp=M_Lower[i][j]
				M_Lower[i][j]=M_Lower[i+1][j]
				M_Lower[i+1][j]=tmp
			endfor
		endif
	endfor
End
```

This code is provided for illustration only. In practice you could use the /FM flag so that the operation creates the full lower and upper matrices for you.

The variable V_flag is set to zero if the operation succeeds and to 1 otherwise (e.g., if the input is singular). The variables V_Sum and V_min are also set by some of the flag options above.

See Also

MatrixLUD, **MatrixOp**, **Matrix Math Operations** on page III-131 for more about Igor’s matrix routines.

MatrixMultiply

`MatrixMultiply` ***`matrixA`*** `[`**`/T`**`]`**`,`** ***`matrixB`*** `[`**`/T`**`] [`**`,`** ***`additional matrices`***`]`

The MatrixMultiply operation calculates matrix expression *matrixA*matrixB* and puts the result in a matrix wave named M_product generated in the current data folder. The /T flag can be included to indicate that the transpose of the specified matrix should be used.

If any of the source matrices are complex, then the result is complex.

Parameters

If *matrixA* is an NxP matrix then *matrixB* must be a PxM matrix and the product is an NxM matrix. Up to 10 matrices can be specified although it is unlikely you will need more than three. The inner dimensions must be the same. Multiplication is performed from right to left.

It is legal for M_product to be one of the input matrices. Thus `MatrixMultiply A,B,C` could also be done as:

```
MatrixMultiply B,C
MatrixMultiply A,M_product
```

Details

Supports multiplication of complex matrices.

An error is generated if the dimensioning of the input arrays is invalid.

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-131 for more about Igor’s matrix routines.

MatrixOp

`MatrixOp [/C /FREE /NTHR=`*`n`*` /O /S]` ***`destwave = expression`***

The MatrixOp operation evaluates *expression* and stores the result in *destWave*.

expression may include literal numbers, numeric variables, numeric waves, and the set of operators and functions described below. MatrixOp does not support text waves, strings or structures.

MatrixOp is faster and in some case more readable than standard Igor waveform assignments and matrix operations.

See **Using MatrixOp** on page III-132 for an introduction to MatrixOp.

Parameters

destWave	Specifies a destination wave for the assignment expression. *destWave* is created at runtime. If it already exists, you must use the /O flag to overwrite it or the operation returns an error. When the operation is completed, *destWave* has the dimensions and data type implied by *expression*. In particular, it may be complex if *expression* evaluates to a complex quantity. If *expression* evaluates to a scalar, *destWave* is a 1x1 wave. If you include the /FREE flag then *destWave* is created as a free wave. By default the data type of *destWave* depends on the data types of the operands and the nature of the operations on the right-hand side of the assignment. If *expression* references integer waves only, *destWave* may be an integer wave too but most operations with a scalars convert *destWave* into a double precision wave. See **MatrixOp Data Promotion Policy** on page III-138 for further discussion. Even if *destWave* exists before the operation, MatrixOp may change its data type and dimensionality as implied by *expression*. You can force the number type using MatrixOp functions such as `uint16` and `fp32`.
expression	*expression* is a mathematical expression referencing waves, local variables, global variables and literal numbers together with MatrixOp functions and MatrixOp operators as listed in the following sections. You can use any combination of data types for operands.In particular, you can mix real and complex types in *expression*. MatrixOp determines data types of inputs and the appropriate output data type at runtime without regard to any type declaration such as `Wave/C`.

Operators

`+`	Addition between scalars, matrix addition or addition of a scalar (real or complex) to each element of a matrix.
`-`	Subtraction of one scalar from another, matrix subtraction, or subtracting a scalar from each element of a matrix. Subtraction of a matrix from a scalar is not defined.
`*`	Multiplication between two scalars, multiplication of a matrix by a scalar, or element-by-element multiplication of two waves of the same dimensions.
`/`	Division of two scalars, division of a matrix by a scalar, or element-by-element division between two waves of the same dimensions. Division of a scalar by a matrix is not supported but you can use the `rec` function with multiplication instead.
`x`	Matrix multiplication (lower case x symbol only). This operator *must* be preceded and followed by a space. Matrix multiplication requires that the number of columns in the matrix on the left side be equal to the number of rows in the matrix on the right.
`.`	Generalized form of a dot product. In an expression *a.b* it is expected that *a* and *b* have the same number of points although they may be of arbitrary numeric type. The operator returns the sum of the products of the sequential elements as if both *a* and *b* were 1D arrays.
`^t`	Matrix transpose. This is a postfix operator meaning that ^t appears after the name of a matrix wave.
`^h`	Hermitian transpose. This is a postfix operator meaning that ^h appears after the name of a matrix wave.
&&	Logical AND operator supports all real data types and results in a signed byte numeric token with the value of either 0 or 1. The operation acts on an element by element basis and is performed for each element of the operand waves.
`||`	Logical OR operator supports all real data types and results in a signed byte numeric token with the value of either 0 or 1. The operation acts on an element by element basis and is performed for each element of the operand waves.

MatrixOp does not support operator combinations such as `+=`.

This table shows the precedence of MatrixOp operators:

MatrixOp Operator		Precedence
`^h`	`^t`	**Highest**
`x`	`.`	
`*`	`/`	
`+`	`-`	
`&&`	`||`	**Lowest**

You can use parentheses to force evaluation order.

Operators that have the same precedence associate from right to left. This means that `a* b / c` is equivalent to `a * (b / c)`.

Functions

These functions are available for use with MatrixOp.

`abs(`*w*`)`	Absolute value of a real number or the magnitude of a complex number.
`acos(`*w*`)`	Arc cosine of *w*.

`acosh(`*w*`)`	Inverse hyperbolic cosine of *w*. Added in Igor Pro 7.00.
`asin(`*w*`)`	Arc sine of *w*.
`asinh(`*w*`)`	Inverse hyperbolic sine of *w*. Added in Igor Pro 7.00.
`asyncCorrelation(`*w*`)`	Asynchronous spectrum correlation matrix for a real valued input matrix wave *w*. See `syncCorrelation` for details.
`atan(`*w*`)`	Arc tangent (inverse tangent) of *w*.
`atan2(`*y*`,`*x*`)`	Arc tangent (inverse tangent) of real *y*/*x*.
`atanh(`*w*`)`	Inverse hyperbolic tangent of *w*. Added in Igor Pro 7.00.
`averageCols(`*w*`)`	Returns a (1xcolumns) wave containing the averages of the columns of matrix *w*. This is equivalent to sumCols(*w*)/numRows(*w*). Added in Igor Pro 7.00.
`backwardSub(`*U*`,`*c*`)`	Returns a column vector solution for the matrix equation Ux=c, where U is an (NxN) wave representing an upper triangular matrix and c is a column vector of N rows. If c has additional columns they are ignored. Ideally, U and c should be either SP or DP waves (real or complex). Other numeric data types are supported with a slight performance penalty. This function is typically used in solving linear equations following a matrix decomposition into lower and upper triangular matrices (e.g., Cholesky), with an expression of the form: `MatrixOp/O solVector=backwardSub(U,forwardSub(L,b))` where U and L are the upper and lower triangular factors. Added in Igor Pro 7.00.
`beam(`*w*`,`*row*`,`*col*`)`	When *w* is a 3D wave the beam function returns a 1D array corresponding to the data in the beam defined by w[*row*][*col*][]. In other words, it returns a 1D array consisting of all elements in the specified row and column from all layers. See also **ImageTransform** getBeam. When *w* is a 4D wave it returns a 2D array containing w[*row*][*col*][][]. In other words, it returns a matrix consisting of all elements in the specified row and column from all layers and all chunks. The beam function belongs to a special class in that it does not operate on a layer by layer basis. It therefore does not permit compound expressions in place of any of its parameters. The beam function has the highest precedence.
`bitAnd(`*w1*`,`*w2*`)`	Returns an array of the same dimensions and number type as *w1* where each element corresponds to the bitwise AND operation between *w1* and *w2*. *w2* can be either a single number or a matrix of the same dimensions as *w1*. Both *w1* and *w2* must be real integer numeric types. Added in Igor Pro 7.00.
`bitOr(`*w1*`,`*w2*`)`	Returns an array of the same dimensions and number type as *w1* where each element corresponds to the bitwise OR operation between *w1* and *w2*. *w2* can be either a single number or a matrix of the same dimensions as *w1*. Both *w1* and *w2* must be real integer numeric types. Added in Igor Pro 7.00.

`bitShift(`*`w1`*`,`*`w2`*`)`	Returns an array of the same dimensions and number type as *w1* shifted by the amount specified in *w2*. When *w2* is positive the shifting is to the left. When it is negative the shifting is to the right. *w2* can be either a single number or a matrix of the same dimensions as *w1*. Both *w1* and w2 must be real integer numeric types. Added in Igor Pro 7.00.
`bitXor(`*`w1`*`,`*`w2`*`)`	Returns an array of the same dimensions and number type as *w1* where each element corresponds to the bitwise XOR operation between *w1* and *w2*. *w2* can be either a single number or a matrix of the same dimensions as *w1*. Both *w1* and *w2* must be real integer numeric types. Added in Igor Pro 7.00.
`bitNot(`*`w`*`)`	Returns an array of the same dimensions and number type as *w* where each each element is the bitwise complement of the corresponding element in w. Added in Igor Pro 7.00.
`catCols(`*`w1`*`,`*`w2`*`)`	Concatenates the columns of *w2* to those of *w1*. *w1* and *w2* must have the same number of rows and the same number type. Added in Igor Pro 7.00.
`catRows(`*`w1`*`,`*`w2`*`)`	Concatenates the rows of *w2* to those of *w1*. *w1* and *w2* must have the same number of columns and the same number type. Added in Igor Pro 7.00.
`ceil(`*`z`*`)`	Smallest integer larger than *z*.
`chirpZ(`*`data`*`,`*`A`*`,`*`W`*`,`*`M`*`)`	Chirp Z Transform of the 1D wave data calculated for the contour defined by $z_k = AW^{-k},$ $k = 0,1,...M-1.$ Here both *A* and *W* are complex and the standard z transform for a sequence {x(n)} is defined by $X(z) = \sum_{k=0}^{N-1} x(n) z^{-k}.$ The phase of the output is inverted to match the result of the ChirpZ transform on the unit circle with that of the FFT.
`chirpZf(`*`data`*`,`*`f1`*`,`*`f2`*`,`*`df`*`)`	Chirp Z Transform except that the transform parameters are specified by real-valued starting frequency *f1*, end frequency *f2* and frequency resolution *df*. The transform is confined to the unit circle because both *A* and *W* have unit magnitude.
`chol(`*`w`*`)`	Returns the Cholesky decomposition U of a positive definite symmetric matrix w such that w=U^t x U. Note that only the upper triangle of w is actually used in the computation.
`chunk(`*`w`*`,`*`n`*`)`	Returns chunk *n* from 4D wave *w*. Added in Igor Pro 7.00.

`clip(`*w*`,`*low*`,`*high*`)`

Returns the values in the wave *w* clipped between the *low* and the *high* parameters. If *w* contains NaN or INF values, they are not modified. The result retains the same number type as the input wave *w* irrespective of the range of the *low* and *high* input parameters.

`cmplx(`*re*`,`*im*`)`

Returns a complex token from two real tokens. *re* and *im* must have the same dimensionality.

Added in Igor Pro 7.00.

`col(`*w*`,`*c*`)`

Returns column *c* from matrix wave *w*.

`colRepeat(`*w*`,`*n*`)`

Returns a matrix that consists of *n* identical columns containing the data in the wave *w*. If *w* is a 2D wave, it is treated as if it were a single column containing all the data. Higher dimensions are supported on a layer-by-layer basis. MatrixOp returns an error if n<2.

Added in Igor Pro 7.00.

`conj(`*matrixWave*`)`

Complex conjugate of the input expression.

`const(`*r*`,`*c*`,`*val*`)`

Returns an (*r* x *c*) matrix where all elements are equal to *val*. The data type of the returned matrix is the same as that of *val*. See also `zeroMat` below.

Added in Igor Pro 7.00.

`convolve(`*w1*`,`*w2*`,`*opt*`)`

Convolution of *w1* with *w2* subject to options opt. The dimensions of the result are determined by the largest dimensions of *w1* and *w2* with the number of rows padded (if necessary) so that they are even. Supported options include *opt*=0 for circular convolution and *opt*=4 for acausal convolution.

For fast 2D convolutions where where *w1* is an image and *w2* is a square kernel of the same numeric type, you can use *opt*=-1 or *opt*=-2. When *opt*=-1 the convolution at the boundaries is evaluated using zero padding. When *opt*=-2 the padding is a reflection of *w1* about the boundaries. When working with integer waves the kernel is internally normalized by the sum of its elements. The kernel for floating point waves remain unchanged.

To convolve an image in *w1* with a smaller point spread function in *w2* you can use:

opt=-1 if you want to pad the image boundaries with zeros or

opt=-2 if you want to pad the boundaries by reflecting image values about each boundary.

The negative options are designed for a very optimized convolution calculation which requires that *w1* and *w2* have the same numeric type. If the size of the point spread function is larger than about 13x13 it may become more efficient to compute the convolution using the positive options.

`correlate(`*w1*`,`*w2*`,`*opt*`)`

Correlation of *w1* with *w2* subject to options *opt*. The dimensions of the result are determined by the largest dimensions of *w1* and *w2* with the number of rows padded (if necessary) so that they are even. Supported options include *opt*=0 for circular correlation and *opt*=4 for acausal correlation.

`cos(`*w*`)`

Cosine of *w*.

`cosh(`*w*`)`

Hyperbolic cosine of *w*. Added in Igor Pro 7.00.

`crossCovar(`*w1*`,`*w2*`,`*opt*`)`

Returns the cross-covariance for 1D waves *w1* and *w2*. The options parameter opt can be set to 0 for the raw cross-covariance or to 1 if you want the results to be normalized to 1 at zero offset. If *w1* has N rows and *w2* has M rows then the returned vector is of length N+M-1. The cross-covariance is computed by subtracting the mean of each input followed by correlation and optional normalization. See also **Correlate** with the /NODC flag.

`det(`*`w`*`)`	Returns a scalar corresponding to the determinant of matrix *w*, which must be real.
`diagonal(`*`w`*`)`	Creates a square matrix that has the same number of rows as *w*. All elements are zero except for the diagonal elements whichare taken from the first column in *w*. Use DiagRC if the input is not an existing wave (such as a result from another function).
`diagRC(`*`w`*`,`*`rows`*`,`*`cols`*`)`	2D matrix of dimensions *rows* by *cols*. All matrix elements are set to zero except those of the diagonal which are filled sequentially from elements of *w*. The dimensionality of *w* is unimportant. If the total number of elements in *w* is less than the number of elements on the diagonal then all elements will be used and the remaining diagonal elements will be set to zero.
`e`	Returns the base of the natural logarithm.
`equal(`*`a`*`,`*`b`*`)`	Returns unsigned byte result with 1 for equality and zero otherwise. The dimensionality of the result matches the dimensionality of the largest parameter. Either or both *a* or *b* can be constants (i.e., one row by one column). If *a* and *b* are not constants, they must have the same dimensions. Both parameters can be either real or complex. A comparison of a real with a complex parameter returns zero.
`erf(`*`w`*`)`	Returns the error function (see **erf**) for real values in *w*. Added in Igor Pro 7.00.
`erfc(`*`w`*`)`	Returns the complementary error function (see **erfc**) for real values in *w*. Added in Igor Pro 7.00.
`exp(`*`w`*`)`	Exponential function for *w* which can be real or complex, scalar or a matrix.
`fft(`*`w`*`,`*`options`*`)`	FFT of *w*. *w* must have an even number of rows. *options* contains a binary field flag. Set bit 1 to 1 if you want to disable the zero centering (see /Z flag in the **FFT** operation). Other bits are reserved. MatrixOp does not support wave scaling and therefore it does not produce the same wave scaling changes as the **FFT** operation.
`floor(`*`w`*`)`	Largest integer smaller than *w*. If *w* is complex the function is separatly applied to the real and imaginary parts.
`forwardSub(`*`L`*`,`*`b`*`)`	Returns a column vector solution for the matrix equation Lx=b, where *L* is an (NxN) wave representing a lower triangular matrix and *b* is a column vector of N rows. If *b* has additional columns they are ignored. Ideally, *L* and *b* should be either SP or DP waves (real or complex). Other numeric data types are supported with a slight performance penalty. This function is typically used in solving linear equations following a matrix decomposition into lower and upper triangular matrices (e.g., Cholesky), with an expression of the form: `MatrixOp/O solVector=backwardSub(U,forwardSub(L,b))` where U and L are the upper and lower triangular factors. Added in Igor Pro 7.00.
`fp32(`*`w`*`)`	Converts *w* to 32-bit single precision floating point representation. See also the /NPRM flag below for more information. Added in Igor Pro 7.00.

`fp64(`*w*`)`	Converts *w* to 64-bit single precision floating point representation. See also the /NPRM flag below for more information. Added in Igor Pro 7.00.
`Frobenius(`*w*`)`	Returns the Frobenius norm of a matrix defined as the square root of the sum of the squared absolute values of all elements.
`getDiag(`*w2d*`,`*d*`)`	Returns a 1D wave that contains diagonal *d* of *w2d*. *d*=0 is the main diagonal, *d*>0 correspond to upper diagonals and *d*<0 to lower diagonals. Added in Igor Pro 7.00.
`greater(`*a*`,`*b*`)`	Returns an unsigned byte for the truth of *a* > *b*. Both *a* and *b* must be real but one or both can be constants (see `equal()` above). The dimensionality of the result matches the dimensionality of the largest parameter.
`hypot(`*w1*`,`*w2*`)`	Returns the square root of the sum of the squares of *w1* and *w2*. Added in Igor Pro 7.00.
`ifft(`*w*`,`*options*`)`	IFFT of *w*. *options* is a bitwise parameter defined as follows: Bit 0: Forces the result to be real, like the **IFFT** operation /C flag. Bit 1: Disables center-zero. Bit 2: Swaps the results. See **Setting Bit Parameters** on page IV-12 for details about bit settings. MatrixOp does not support wave scaling and therefore it does not produce the same wave scaling changes as the **IFFT** operation.

`identity(`*n*`,`*m*`)`
`identity(`*n*`)`
	Creates a computational object that is an identity matrix. If you use a single argument *n,* the identity created is an (*n*x*n)* square matrix with 1's for diagonal elements (the remaining elements are set to zero). If you use both arguments, the function creates an (*n*x*m*) zero matrix and fills its diagonal elements with 1's. Note that the identity is created at runtime and persists only for the purpose of the specific operation.

`imag(`*w*`)`	Imaginary part of *w*.
`inf()`	Returns INF. Added in Igor Pro 7.00.
`insertMat(`*s*`,`*d*`,`*r*`,`*c*`)`	Inserts matrix *s* into matrix *d* starting at row r and column c. The waves *s* and *d* must be of the same numeric data type. The inserted range is clipped to the dimensions of the wave *d*. Both *r* and *c* must be non-negative. Added in Igor Pro 7.00.
`int8(`*w*`)`	Converts *w* to 8-bit signed integer representation. See also the /NPRM flag below for more information. Added in Igor Pro 7.00.
`int16(`*w*`)`	Converts *w* to 16-bit signed integer representation. See also the /NPRM flag below for more information. Added in Igor Pro 7.00.
`int32(`*w*`)`	Converts *w* to 32-bit signed integer representation. See also the /NPRM flag below for more information. Added in Igor Pro 7.00.

`integrate(w, opt)`	Returns a running sum of `w`. If *opt*=0 the sum runs over the entire input wave treating the columns of 2D waves as if they were part of one big column. If *opt*=1 the running sum is computed separately for each column of a 2D wave. Added in Igor Pro 7.00.
`intMatrix(w)`	Returns a double-precision matrix of the same dimensions as *w*. Each element of the returned matrix is the sum of all the corresponding elements of *w* that are above and to the left of it, i.e., $out_{ij} = \sum_{m=0}^{i} \sum_{n=0}^{j} w_{mn}.$ The utility of this function is apparent in the following relationship:

$$\sum_{i=x_1}^{x_2} \sum_{j=y_1}^{y_2} w_{ij} = out[x_2, y_2] - out[x_2, y_1 - 1] - out[x_1 - 1, y_2] + out[x_1 - 1, y_1 - 1].$$

	intMatrix was added in Igor Pro 7.00.
`inv(w)`	Returns the inverse of the square matrix *w*. If w is not invertible, the operation returns a matrix of the same dimensions where all elements are set to NaN.
`inverseErf(w)`	Returns the inverse error function (see **inverseErf**) for the real values in *w*. Added in Igor Pro 7.00.
`inverseErfc(w)`	Returns the inverse complementary error function (see **inverseErfc**) for the real values in *w*. Added in Igor Pro 7.00.
`layer(w)`	Returns layer *n* from the 3D wave *w*. *w* can not be a compound expression. Added in Igor Pro 7.00.
`limitProduct(w1,w2)`	Returns a partial element-by-element multiplication of waves *w1* and *w2*. It is assumed that the dimensions of *w1* are greater or equal to that of *w2*. If *w2* is of dimensions NxM then the function returns a matrix of the same dimensions as *w1* with the first NxM elements contain the product of the corresponding elements in *w1* and *w2* and the remaining elements set to zero. The function is designed to be used in filtering applications where the size of the kernel *w2* is much smaller than the size of the input *w1*. Added in Igor Pro 7.00.
`log(w)`	Log base 10 of a token *w* which can be real or complex, scalar or a matrix.
`ln(w)`	Natural logarithm of a token *w2* which can be real or complex, scalar or a matrix.
`mag(w)`	Returns a real valued wave containing the magnitude of each element of *w*. This is equivalent to the `abs` function.
`magSqr(w)`	Returns a real value wave containing the square of a real *w* or the squared magnitude of complex *w*.

`maxAB(a,b)`	Returns the larger of the two real numbers *a* and *b*. `maxAB` does not support NaN or complex inputs. Added in Igor Pro 7.00.
`maxCols(w)`	Returns a (1 x cols) wave containing the maximum values of each column in the wave *w*. If *w* is complex the output is the maximum magnitude of columns of *w*. `maxCols` does not support NaN. Added in Igor Pro 7.00.
`maxVal(w)`	Returns the maximum value of the wave *w*. If *w* is complex it returns the maximum magnitude of *w*. When *w* is a 3D or 4D wave, `maxVal` returns a (1 x 1 x layers x chunks) data token. `maxVal` does not support NaN values.
`mean(w)`	Returns the mean value *w*.
`minVal(w)`	Returns the minimum value of the wave *w*. If *w* is complex the function returns the minimum magnitude of *w*. When *w* is a 3D or 4D wave, `minVal` returns a (1 x 1 x layers x chunks) data token. `minVal` does not support NaN values.
`mod(w,b)`	Returns the remainder after dividing *w* by *b*. *b* can be a scalar or a matrix of the same dimensions as *w*. Added in Igor Pro 7.00.
`nan()`	Returns NaN. Added in Igor Pro 7.00.
`normalize(w)`	Normalized version of a vector or a matrix. Normalization is such that the returned token should have a unity magnitude except if all elements are zero, in which case output is unchanged.
`normalizeCols(w)`	Divides each column of the real wave *w* by the square root of the sum of the squares of all elements of the column.
`normalizeRows(w)`	Divides each row of the real wave *w* by the square root of the sum of the squares of all the elements in that row.
`numCols(w)`	Returns the number of columns in the wave *w*. When *w* is 1D the function returns 1.
`numPoints(w)`	Returns the number of points in a layer of *w*.
`numRows(w)`	Returns the number of rows in *w*.
`numType(w)`	Number the number type of *w*: 0: *w* is a normal number 1: *w* is +/-INF 2: *w* is NaN
`p2Rect(w)`	Converts each element of *w* from polar to rectangular representation.
`phase(w)`	Returns a real valued wave containing the phase of each element of *w* calculated using `phase=atan2(y,x)`.
`Pi`	Returns π.

`powC(w1,w2)`	Complex valued $w1^{w2}$ where *w1* and *w2* can be real or complex.
`powR(x,y)`	Returns *x^y* for real *x* and *y*.
`productCol(w,c)`	Returns the a (1 x 1) wave containing the product of the elements in column *c* of wave *w*. The output is double precision real or complex. Added in Igor Pro 7.00.
`productCols(w)`	Returns a (1 x cols) wave where each entry is the product of all the elements in the corresponding column. The output is double precision real or complex. Added in Igor Pro 7.00.
`productDiagonal(w,d)`	Returns a (1 x 1) wave containing the product of the elements on the specified diagonal of wave w. *d*=0 is the main diagonal, *d*>0 correspond to upper diagonals and *d*<0 to lower diagonals. The output is double precision real or complex. Added in Igor Pro 7.00.
`productRow(w,r)`	Returns the a (1 x 1) wave containing the product of the elements in row *r* of wave *w*. The output is double precision real or complex. Added in Igor Pro 7.00.
`productRows(w)`	Returns a (1 x rows) wave where each entry is the product of all the elements in the corresponding row. The output is double precision real or complex. Added in Igor Pro 7.00.
`r2Polar(w)`	Performs the equivalent of the **r2polar** function on each element of *w,* i.e., each complex number (x+iy) is converted into the polar represenation r,theta with x+iy=r*exp(i*theta)
`real(w)`	Real part of *w*.
`rec(w)`	Reciprocal of each element in *w*.

`redimension(w, nr, nc)`

Returns an (nr x nc) matrix from the data in the wave w. The data in w are moved contiguously (column by column regardless of dimensionality) into the output. If the output size is larger than w the remaining points are set to zero. If w contains more than one layer the new dimensions apply on a layer by layer basis. For example:

```
Make/N=(10,20,30) ddd = p + 10*q + 100*r
MatrixOp/O aa = redimension(ddd,25,1)
```

creates a wave aa with dimensions (25,1,30).

Added in Igor Pro 7.00.

`replace(w,findVal,replacementVal)`

Replace in wave *w* every occurance of *findVal* with *replacementVal*. The wave *w* retains its dimensionality and number type. *replacementVal* is converted to the same number type as *w* which may cause truncation.

`replaceNaNs(w,replacementVal)`

Replaces every occurance of NaN in the wave *w* with *replacementVal*. The wave *w* retains its dimensionality. *replacementVal* is converted to the same number type as *w* which may cause truncation.

`reverseCol(w,c)`	Returns array *w* with column *c* in reverse order. Added in Igor Pro 7.00.

`reverseCols(w)`	Returns array *w* with all columns in reverse order. Added in Igor Pro 7.00.
`reverseRow(w,r)`	Returns array *w* with row *r* in reverse order. Added in Igor Pro 7.00.
`reverseRows(w)`	Returns array *w* with all rows in reverse order. Added in Igor Pro 7.00.
`rotateChunks(w,n)`	Returns a matrix where the last *n* chunks of *w* are moved to chunks [0, *n*-1]. If *n* is negative then the first abs(*n*) chunks are moved to the end of the data. It is an error to pass NaN for *n*. Added in Igor Pro 7.00.
`rotateCols(w,nc)`	Rotates the columns of a 2D wave *w* so that the last *nc* columns are moved to columns [0,*nc* -1] of the data. If *nc* is negative the first abs(*nc*) columns are moved to columns [n-1-*nc* ,n-1]. Here n is the total number of columns. It is an error to pass NaN for *nc* . If *nc* is greater than the number of columns then the effective rotation is mod(*nc* ,actualCols).
`rotateLayers(w,n)`	Returns a matrix where the last *n* layers of *w* are moved to layers [0, *n*-1]. If *n* is negative then the first abs(*n*) layers are moved to the end of the data. It is an error to pass NaN for *n*. Added in Igor Pro 7.00.
`rotateRows(w,nr)`	Rotates the rows of a 2D wave *w* so that the last *nr* rows are moved to rows [0,*nr* -1] of the data. If *nr* is negative the first abs(*nr*) rows are moved to rows [n-1-*nr,* n-1] where n is the total number of rows. It is an error to pass NaN for *nr*. If *nr* is greater than the number of rows then the effective rotation is mod(*nr* ,actualRows).
`round(z)`	Rounds *z* to the nearest integer. The rounding method is “away from zero”.
`row(w,r)`	Returns row *r* from matrix wave *w*. The returned row is a (1xC) wave where C is the number of columns in *w*. To convert it to a 1D wave use `Redimension/N=(C)`. See also **ImageTransform** `getRow`.
`rowRepeat(w,n)`	Returns a matrix that consists of *n* identical rows containing the data in the wave *w*. If *w* is a 2D wave, it is treated as if it were a single column containing all the data. Higher dimensions are supported on a layer-by-layer basis. MatrixOp returns an error if n<2. Added in Igor Pro 7.00.
`scale(w,low,high)`	Returns the values in the wave *w* scaled between the *low* and the *high* parameters. If *w* contains NaN or INF values, they are not modified. The result retains the same number type as that of *w* irrespective of the range of the *low* and *high* input parameters.
`scaleCols(w1,w2)`	Returns a matrix of the same dimensions as *w1* where each column of *w1* is scaled by the value in the corresponding row of the 1D wave *w2*. The number of rows in *w2* must equal the number of columns in *w1*. Added in Igor Pro 7.00.
`scaleRows(w1,w2)`	Returns a matrix of the same dimensions as *w1* where each row of *w1* is scaled by the value in the corresponding row of the 1D wave *w2*. The number of rows in *w2* must equal the number of rows in *w1*. Added in Igor Pro 7.00.

`setCol(`*`w2d`*`,`*`c`*`,`*`w1d`*`)`	Returns the data in the *w2d* with the contents of *w1d* stored in column *c*. *w1d* must have at least as many elements as the number of rows of *w2d*. *w2d* and *w1d* must be either both real or both complex. Added in Igor Pro 7.00.
`setNaNs(`*`w`*`,`*`mask`*`)`	Returns the data in the wave *w* with NaNs stored where the mask wave is non-zero. The wave *w* can be of any numeric type. *mask* must have the same dimensions as *w* and must be real. It is usually the result of another expression. For example, to set all values in the destination to NaN where *w* is greater than 5: `MatrixOp/o ou = setNaNs(w,greater(w,5))` Added in Igor Pro 7.00.
`setOffDiag(`*`w`*`,`*`d`*`,`*`w1`*`)`	Returns the data in the *w* with the contents of *w1* stored in diagonal *d*. *d*=0 is the main diagonal of w*, d*>0 correspond to upper diagonals and *d*<0 to lower diagonals. *w* and *w1* can either be both real or both complex. Added in Igor Pro 7.00.
`setRow(`*`w2d r`*`,`*`w1d`*`)`	Returns the data in the *w2d* with the contents of *w1d* stored in row *r*. *w1d* must have at least as many elements as the number of columns in w2d. *w2d* and *w1d* must be either both real or both complex. Added in Igor Pro 7.00.
`sgn(`*`w`*`)`	Returns the sign of each element in *w*. It returns -1 for negative numbers and 1 otherwise. It does not accept complex numbers.
`shiftVector(`*`w`*`, `*`n`*`, `*`val`*`)`	
	Shifts the element of a 1D row-vector *w* by *n* elements and fills the displaced elements with *val,* which must match the data type of *w* and should be expressed as `cmplx(`*`a`*`,`*`b`*`)` for complex *w*.
`sin(`*`z`*`)`	Sine of *z*.
`sinh(`*`z`*`)`	Hyperbolic sine of *z*. Added in Igor Pro 7.00.
`sqrt(`*`z`*`)`	Square root of *z*.
`subRange(`*`w`*`,`*`rs`*`,`*`re`*`,`*`cs`*`,`*`ce`*`)`	
	Returns a contiguous subset of the wave *w* from starting row *rs* through ending row *re* and from starting column *cs* through ending column *ce*. This is similar to Duplicate/R except that dimension scaling and labels not preserved. Added in Igor Pro 7.00.
`subtractMean(`*`w`*`,`*`opt`*`)`	Computes the mean of the real wave *w* and returns the values of the wave minus the mean value (*opt*=0). Computes the mean of each column and subtracts it from that column (*opt*=1). Subtracts the mean of each row from row values (*opt*=2).
`subWaveC(`*`w`*`,`*`r`*`,`*`c`*`,`*`count`*`[,`*`stride`*`])`	

Returns a subset of the data that is sampled along columns of the wave *w*, containing count elements starting with the element at row *r* and column *c*. By default *stride*=1 and the sampling is continuous.

You can specify a negative stride to sample backwards from the starting element.

The operation returns an error if the sampling would exceed the array bounds in either direction.

For example:

```
Make/O/N=(22,33) ddd=x

MatrixOP/O/P aa=subWaveC(ddd,4,5,10,2)
// aa={4,6,8,10,12,14,16,18,20,0}

MatrixOP/O/P aa=subWaveC(ddd,4,5,5,-4)
// aa={4,0,18,14,10}
```

`subWaveR(`*`w`*`,`*`r`*`,`*`c`*`,`*`count`*`[,`*`stride`*`])`

Returns a subset of the data that is sampled along rows of the wave w, containing count elements starting with the element at row r and column c. By default stride=1 and the sampling is continuous.

You can specify a negative stride to sample backwards from the starting element.

The operation returns an error if the sampling would exceed the array bounds in either direction.

Examples:

```
Make/O/N=(10,20) ddd=y

// Forward sampling across right boundary
MatrixOP/O/P aa=subWaveR(ddd,4,15,6,2)
// aa={15,17,19,1,3,5}

// Reverse sampling across left boundary
MatrixOP/O/P aa=subWaveR(ddd,2,3,5,-1)
// aa={3,2,1,0,19}
```

`sum(`*`z`*`)` Returns the sum of all the elements in expression *z*.

`sumBeams(`*`w`*`)` Returns an (n x m) matrix containing the sum over all layers of all the beams of the 3D wave *w*:

$$out_{ij} = \sum_{k=0}^{nLayers-1} w_{ijk}.$$

A beam is a 1D array in the Z-direction.

sumBeams is a non-layered function which requires that *w* be a proper 3D wave and not the result of another expression.

`sumCols(`*`w`*`)` Returns a (1 x m) matrix containing the sums of the m columns in the nxm input wave *w*:

$$out_j = \sum_{i=0}^{nRows-1} w_{ij}.$$

`sumRows(`*`w`*`)`

Returns an (n x 1) matrix containing the sums of the n rows in the nxm input wave *w*:

$$out_i = \sum_{j=0}^{nCols-1} w_{ij} .$$

`sumSqr(`*`w`*`)`

Sum of the squares of all elements in *w*.

`syncCorrelation(`*`w`*`)`

Synchronous spectrum correlation matrix for a real valued input matrix wave *w*. See also `asyncCorrelation`.

The correlation matrix is computed by subtracting from each column of *w* its mean value, multiplying the resulting matrix by its transpose, and finally dividing all elements by (nrows-1) where nrows is the number of rows in *w*.

`tan(`*`w`*`)`

Tangent of *w*.

`tanh(`*`w`*`)`

Hyperbolic tangent of *w*. Added in Igor Pro 7.00.

`tensorProduct(`*`w1`*`,`*`w2`*`)`

Returns a 2D matrix that is the tensor product of the 2D matrices *w1* and *w2*. For example, the tensor product of two (2 x 2) matrices is given by:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \otimes \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{pmatrix}$$

Added in Igor Pro 7.00.

`Trace(`*`w`*`)`

Returns a real or complex scalar which is the sum of the diagonal elements of *w*. If *w* is not a square matrix, the sum is over the elements for which the row and column indices are the same.

`transposeVol(`*`w`*`,`*`mode`*`)`

For 3D wave *w*, transposeVol returns a transposed 3D wave depending on the value of the *mode* parameter:

mode=1:	output=w[p][r][q]
mode=2:	output=w[r][p][q]
mode=3:	output=w[r][q][p]
mode=4:	output=w[q][r][p]
mode=5:	output=w[q][p][r]

transposeVol is a non-layered function which requires that *w* be a proper 3D wave and not the result of another expression.

`triDiag(`*`w1`*`,`*`w2`*`,`*`w3`*`)`

Returns a tri-diagonal matrix where *w1* is the upper diagonal, *w2* the main diagonal and *w3* the lower diagonal. If *w2* has *n* points than *w1* and *w3* are expected to have *n*-1 points. The waves can be of any numeric type and the returned wave has a numeric type that accommodates the input.

`uint8(`*`w`*`)`

Converts *w* to 8-bit unsigned integer representation. See also the /NPRM flag below for more information. Added in Igor Pro 7.00.

`uint16(`*`w`*`)`

Converts *w* to 16-bit unsigned integer representation. See also the /NPRM flag below for more information. Added in Igor Pro 7.00.

`uint32(`*`w`*`)`

Converts *w* to 32-bit unsigned integer representation. See also the /NPRM flag below for more information. Added in Igor Pro 7.00.

`varCols(`*w*`)`	Returns a (1 x cols) wave where each element contains the variance of the corresponding column in *w*.
`waveIndexSet(`*w1*`,`*w2*`,`*w3*`)`	Returns a matrix of the same dimensions as *w1* with values taken either from *w1* or from *w3* depending on values in *w2* using: $$out[i][j]=\begin{cases} w1[i][j] & if\ w2[i][j]<0 \\ w3[w2[i][j]] & otherwise \end{cases}.$$ *w1* and *w2* must have the same number of rows and columns. *w1* and *w3* must match in number type. *w2* cannot be unsigned. Values from *w2* are used as point number indices into *w3* which is treated like a 1D wave regardless of its actual dimensionality. An index value from *w2* is out-of-bounds if it is greater than or equal to the number of points in *w3*. In this case, the output value is taken from *w1* as if the index value were negative.
`waveMap(`*w1*`,`*w2*`)`	Returns an array of the same dimensions as *w2* containing the values w1[*w2*[*i*][*j*]]. The data type of the output is the same as that of *w1*. Values of *w2* are taken as 1D integer indices into the *w1* array. See also **IndexSort**.
`waveChunks(`*w*`)`	Returns the number of chunks in the wave *w*. Added in Igor Pro 7.00.
`waveLayers(`*w*`)`	Returns the number of layers in the wave *w*. Added in Igor Pro 7.00.
`wavePoints(`*w*`)`	Returns the number of points in the wave *w*. Added in Igor Pro 7.00.
`within(`*w*`,`*low*`,`*high*`)`	Returns an array of the same dimensions as *w* with the value 1 where the corresponding element of w is between low and high (low <= w[i][j] < high). Added in Igor Pro 7.00. All parameters must be real. It is an error to pass a NaN as either *low* or *high*. It is also an error if *low* >= *high*. If *w* contains NaNs, the corresponding outputs are 0.
`zeroMat(`*r*`,`*c*`,`*nt*`)`	Returns an (*r* x *c*) matrix of number type *nt* where all entries are set to zero. See **WaveType** for supported types. See also `const` above. Added in Igor Pro 7.00.

Wave Parameters

MatrixOp was designed to work with 2D waves (matrices) but also works with 1D, 3D and 4D waves. A 1D wave is treated like a 1-column matrix. 3D and 4D waves are treated on a layer-by-layer basis, as if each layer were a matrix>

You can reference subsets of waves in *expression*. Only two types of subsets are supported: those that evaluation to a single element, which are treated as scalars, and those that evaluate to one or more layers. For example:

`wave1d[`*a*`]`	Scalar
`wave2d[`*a*`][`*b*`]`	Scalar
`wave3d[`*a*`][`*b*`][`*c*`]`	Scalar
`wave3d[][][`*a*`]`	Layer *a* from 3D wave
`wave3d[][][`*a*`,`*b*`]`	Layers *a* through *b* from 3D wave
`wave3d[][][`*a*`,`*b*`,`*c*`]`	Layers *a* through *b* stepping by *c* from 3D wave

You can pass waves of any dimensions as parameters to MatrixOp functions. For example:

```
Make/O/N=128 wave1d = x
MatrixOp/O outWave = powR(wave1d,2)
```

MatrixOp does not allow using the same 3D wave on both sides of the assignment:

```
MatrixOp/O wave3D = wave3D + 3    // Not allowed
```

See **MatrixOp Wave Data Tokens** on page III-134 for further discussion.

Flags

/C	Provides a complex wave reference for *destWave*. If omitted, MatrixOp creates a real wave reference for *destWave*. The wave reference allows you to refer to the output wave in a subsequent statement of a user-defined function.
/FREE	Creates *destWave* as a free wave. Allowed only in functions and only if a simple name or wave reference structure field is specified. Requires Igor Pro 6.1 or later. For advanced programmers only. See **Free Waves** on page IV-84 for more discussion.
/NTHR=*n*	Sets the number of threads used to compute the results for 3D waves. Each thread computes the results for a single layer of the input. By default (/NTHR omitted) the calculations are performed by the main thread only. If *n*=0 the operation uses as many threads you have processors on your computer. If *n*>0, n specifies the number of threads to use. More threads may or may not improve performance.

/NPRM	Use /NPRM to restrict the automatic promotion of numeric data types in MatrixOp expressions.
	By default, MatrixOp promotes numeric data types so that operations result in reasonable accuracy. In some situations you may want to keep the results as a particular data type even at the risk of truncation or overflow. If you include the /NPRM flag, MatrixOp creates the destination wave using the highest precision data type in the expression. For example, an expression A=B+C where B is 16-bit wave and C is an 8-bit wave results in a 16-bit wave A.
	Unsigned number types can result only when all operands are unsigned.
	/NPRM is ignored when data promotion is required. For example:
	`Make/B/U wave2`
	`MatrixOp/O/NPRM wave1 = -wave2`
	You can use MatrixOp functions such as int8, int16, etc., to precisely control the number type of any token.

/O	Overwrites *destWave* if it already exists.
/S	Preserves the dimension scaling, units and wave note of a pre-existing destination wave in a MatrixOp/O command.

Details

MatrixOp has the general form:

```
MatrixOp [flags] destWave = expression
```

destWave specifies the wave created by MatrixOp or overwritten by MatrixOp/O.

From the command line, *destWave* can be a simple wave name, a partial data folder path or a full data folder path. In a user-defined function it can be a simple wave name or, if /O is present, a wave reference pointing to an existing wave.

expression is a mathematical expression that consists of one or more data tokens combined with the built-in MatrixOp functions and MatrixOp operators listed above. MatrixOp does not support the p, q, r, s, or x, y, z, t symbols that are used in waveform assignment statements.

Data tokens include waves, variables and literal numbers.

You can use any combination of data types for operands. In particular, you can mix real and complex types in *expression*. MatrixOp determines data types of inputs and the appropriate output data type at runtime without regard to any type declaration such as `Wave/C`.

See **Using MatrixOp** on page III-132 for more information.

Examples

In addition to these examples, see **MatrixOp Optimization Examples** on page III-139.

The following matrices are used in these examples:

```
Make/O/N=(3,3) r1=x, r2=y
```

Matrix addition and matrix multiplication by a scalar:

```
MatrixOp/O outWave = r1+r2-3*r1
```

Using the matrix `Identity` function:

```
MatrixOp/O outWave = Identity(3) x r1
```

Create a persisting identity matrix for another calculation:

```
MatrixOp/O id4 = Identity(4)
```

Using the `Trace` function:

```
MatrixOp/O outWave = (Trace(r1)*identity(3) x r1)-3*r1
```

Using matrix inverse function `Inv()` with matrix multiplication:

```
MatrixOp/O outWave = Inv(r2) x r2
```

Using the determinant function `Det()`:

```
MatrixOp/O outWave = Det(r1)+Det(r2)
```

Using the Transpose postfix operator:

```
MatrixOp/O outWave = r1^t+(r2-r1)^t-r2^t
```

Using a mix of real and complex data:

```
Variable/C complexVar = cmplx(1,2)
MatrixOp/O outWave = complexVar*r2 - Cmplx(2,4)*r1
```

Hermitian transpose operator:

```
MatrixOp/O outWave = Trace(complexVar*r2)^h -Trace(cmplx(2,4)*r1)^h
```

In-place operation and conversion to complex:

```
MatrixOp/O r1 = r1*cmplx(1,2)
```

Image filtering using 2D spatial filter filterWave:

```
MatrixOp/O filteredImage=IFFT(FFT(srcImage,2)*filterWave,3)
```

Positive shift:

```
Make/O w={0,1,2,3,4,5,6}
MatrixOp/O w=shiftVector(w,2,77)
Print w
// w[0]= {77,77,0,1,2,3,4}
```

Negative shift:

```
Make/O w={0,1,2,3,4,5,6}
MatrixOp/O w=shiftVector(w,(-2),77)
Print w
// w[0]= {2,3,4,5,6,77,77}
```

References

syncCorrelation and asyncCorrelation:

Noda, I., Determination of Two-Dimensional Correlation Spectra Using the Hilbert Transform, *Applied Spectroscopy 54*, 994-999, 2000.

ChirpZ:

Rabiner, L.R., and B. Gold, *The Theory and Application of Digital Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1975.

See Also

Using MatrixOp on page III-132

Matrix Math Operations on page III-131 for more about Igor's matrix routines

FastOp

MatrixRank

`matrixRank(`*`matrixWaveA`*` [, `*`conditionNumberA`*`])`

The matrixRank function returns the rank of *matrixWaveA* subject to the specified condition number.

The matrix is not considered to have full rank if its condition number exceeds the specified *conditionNumberA*.

If the optional parameter *conditionNumberA* is not specified, Igor Pro uses the value 10^{20}.

matrixRank supports real and complex single precision and double precision numeric wave data types.

The value of *conditionNumberA* should be large enough but taking into account the accuracy of the numerical representation given the numeric data type.

If there are any errors the function returns NaN.

See Also

Matrix Math Operations on page III-131 for more about Igor's matrix routines.

MatrixSchur

`MatrixSchur [/Z]` ***`srcMatrix`***

The MatrixSchur operation computes for an NxN nonsymmetric srcMatrix, the eigenvalues, the real Schur form A and the matrix of Schur vectors V.

The Schur factorization has the form: S = V x A x (V^T), where V^T is the transpose (use V^H if S is complex) and x denotes matrix multiplication.

Flags

/Z	No error reporting.

Details

The operation creates:

M_A	Upper triangular matrix containing the Schur form A.
M_V	Unitary matrix containing the orthogonal matrix V of the Schur vectors.
W_REigenValues W_IEigenValues	Waves containing the real and imaginary parts of the eigenvalues when *srcMatrix* is a real wave. If *srcMatrix* is complex, the eigenvalues are stored in W_eigenValues.

The variable V_flag is set to 0 when there is no error; otherwise it contains the LAPACK error code.

Examples

You can test this operation for an N-by-N source matrix:

```
Make/D/C/N=(5,5) M_S=cmplx(enoise(1),enoise(1))
MatrixSchur M_S
MatrixOp/O unitary=(M_V^h) x M_V              // Check unitary
MatrixOp/O diff=abs(M_S-M_V x M_A x (M_V^H))   // Check decomposition
```

See Also

Matrix Math Operations on page III-131 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

MatrixSolve

`MatrixSolve` ***`method, matrixA, vectorB`***

The MatrixSolve operation was superseded by MatrixLLS and is included for backward compatibility only.

Used to solve matrix equation Ax=b using the method of your choice. Choices for *method* are:

method	**Solution Method**
GJ	Gauss Jordan.
LU	LU decomposition.
SV	Singular Value decomposition.

Details

The array b can be a matrix containing a number of b vectors and the output matrix M_x will contain a corresponding set of solution vectors.

V_flag is set to zero if success, 1 if singular matrix using GJ or LU and 1 if SV fails to converge.

For normal problems you should use LU. GJ is provided only for completeness and has no practical use.

When using SV, singular values smaller than 10^{-6} times the largest singular value are set to zero before back substitution.

Generates an error if the dimensions of the input matrices are not appropriate.

See Also

The **MatrixLLS** operation. **Matrix Math Operations** on page III-131 for more about Igor's matrix routines.

MatrixSVBkSub

`MatrixSVBkSub` ***`matrixU, vectorW, matrixV, vectorB`***

The MatrixSVBkSub operation does back substitution for SV decomposition.

Details

Used to solve matrix equation Ax=b after you have performed an SV decomposition.

Feed this routine the M_U, W_W and M_V waves from **MatrixSVD** along with your right-hand-side vector b. The solution vector x is returned as M_x.

The array b can be a matrix containing a number of b vectors and the M_x will contain a corresponding set of solution vectors.

Generates an error if the dimensions of the input matrices are not appropriate.

See Also

Matrix Math Operations on page III-131 for more about Igor's matrix routines.

MatrixSVD

`MatrixSVD` `[`***`flags`***`]` ***`matrixWave`***

The MatrixSVD operation uses the singular value decomposition algorithm to decompose an MxN matrixWave into a product of three matrices. The default decomposition is into MxM wave M_U, min(M,N) wave W_W and NxN wave M_VT.

Flags

/B	Use this flag for backwards compatibility with Igor Pro 3. This option applies only to real valued input waves. Note that no other flag can be combined with /B. Here the decomposition is such that: U*W*V^T = *matrixWave* U: MxN column-orthonormal matrix. W: NxN diagonal matrix of positive singular values. V: NxN orthonormal matrix.
/DACA	Replaces the standard LAPACK algorithm with one that is based on a divide and conquer approach. For a typical 1000x1000 matrix this provides a 6x speed improvement. Added in Igor Pro 7.00.

/INVW	Saves the inverse of the elements in W_W. The results are then stored in wave W_InvW.
/O	Overwrites *matrixWave* with the first columns of U. Use this flag to if you need to conserve memory. See also related settings of /U and /V.
/PART =*nVals*	Performs a partial SVD computing only *nVals* singular values (stored in W_W) and the associated vectors in the matrix M_U and M_V. If you use this flag the operation ignores all other flags except /PDEL. The partial SVD is computed using the Power method of Nash and Shlien. The /PART flag was added in Igor Pro 7.00.
/PDEL=*del*	Sets the convergence threshold which defaults to 1e-6. Larger positive values result in faster execution but may lead to less accurate results. The /PDEL flag was added in Igor Pro 7.00.
/U =*UMatrixOptions*	*UMatrixOptions* can have the following values: 0: All columns of U are returned in the wave M_U (default). 1: The first min(m,n) columns of U are returned in the wave M_U. 2: The first min(m,n) columns of U overwrite matrixWave (/O must be specified). 3: No columns of U are computed.
/V=*VMatrixOptions*	*VMatrixOptions* can have the following values: 0: All rows of V^T are returned in the wave M_VT (default). 1: The first min(m,n) rows of V^T are returned in the wave M_VT. 2: The first min(m,n) rows of V^T are overwritten on *matrixWave* (/O must be specified) 3: No rows of V^T are computed.
/Z	No error reporting.

Details

The singular value decomposition is computed using LAPACK routines. The diagonal elements of matrix W are returned as a 1D wave named W_W. If /B is used W_W will have N elements. Otherwise the number of elements in W_W is min(M,N).

The matrix V is returned in a matrix wave named M_V if /B is used otherwise the transpose V^T is returned in the wave M_VT.

All output objects are created in the current data folder.

The variable V_flag is set to zero if the operation succeeds. It is set to 1 if the algorithm fails to converge.

The variable V_SVConditionNumber is set to the condition number of the input matrix. The condition number is the ratio of the largest singular value to the smallest.

Example

```
Make/O/D/N=(10,20) A=gnoise(10)
MatrixSVD A
MatrixOp/O diff=abs(A-(M_U x DiagRC(W_W,10,20) x M_VT))
Print sum(diff,-inf,inf)
```

References

J.C. Nash and S.Shlien "Simple Algorithms for the Partial Singular Value Decomposition", The Comp. J. (30) No. 3 1987.

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-131 for more about Igor's matrix routines and for background references with details about the LAPACK libraries.

MatrixTrace

`matrixTrace(`*`dataMatrix`*`)`

The matrixTrace function calculates the trace (sum of diagonal elements) of a square matrix. *dataMatrix* can be of any numeric data type.

If the matrix is complex, it returns the sum of the magnitudes of the diagonal elements.

See Also

Matrix Math Operations on page III-131 for more about Igor's matrix routines.

MatrixTranspose

`MatrixTranspose [/H]` ***`matrix`***

The MatrixTranspose operation Swaps rows and columns in *matrix*.

Does not take complex conjugate if data are complex. You can do that as a follow-on step.

Swaps row and column labels, units and scaling.

This works with text as well as numeric waves. If the matrix has zero data points, it just swaps the row and column scaling.

Flags

/H	Computes the Hermitian conjugate of a complex wave.

See Also

The **MatrixOp** operation for more efficient matrix operations.

Matrix Math Operations on page III-131 for more about Igor's matrix routines.

max

`max(`*`num1, num2`*` [, `*`num3, ... num200`*`])`

The max function returns the greatest value of *num1*, *num2*, ... *num200*.

If any parameter is NaN, the result is NaN.

Details

In Igor7 or later, you can pass up to 200 parameters. Previously max was limited to two parameters.

See Also

min, **limit**, **WaveMin**, **WaveMax**

mean

`mean(`*`waveName`*` [, `*`x1, x2`*`])`

The mean function returns the arithmetic mean of the wave for points from x=*x1* to x=*x2*.

Details

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

The wave values from *x1* to *x2* are summed, and the result divided by the number of points in the range.

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with `pnt2x(`*`waveName`*`,`*`pointNumber1`*`)`, and a similar expression for *x2*.

If the points nearest to *x1* or *x2* are not within the point range of 0 to numpnts(*waveName*)-1, mean limits them to the nearest of point 0 or point numpnts(*waveName*)-1.

If any values in the point range are NaN, mean returns NaN.

The function returns NaN if the input wave has zero points.

Unlike the area function, reversing the order of *x1* and *x2* does *not* change the sign of the returned value.

Examples

```
Make/O/N=100 data; SetScale/I x 0,Pi,data
data=sin(x)
Print mean(data,0,Pi)          // the entire point range, and no more
Print mean(data)               // same as -infinity to +infinity
Print mean(data,Inf,-Inf)      // +infinity to -infinity
```

The following is printed to the history area:

```
Print mean(data,0,Pi)          // the entire point range, and no more
  0.630201
Print mean(data)               // same as -infinity to +infinity
  0.630201
Print mean(data,Inf,-Inf)      // +infinity to -infinity
  0.630201
```

See Also

Variance, **WaveStats**, **median**

The figure “Comparison of area, faverage and mean functions over interval (12.75,13.32)”, in the **Details** section of the **faverage** function.

median

`median(waveName [, *x1*, *x2*])`

The median function returns the median value of the wave for points from x=*x1* to x=*x2*.

The median function was added in Igor Pro 7.00.

Details

If you omit *x1* and *x2*, they default to -INF and +INF, respectively.

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x1*. To use point indexing, replace *x1* with "pnt2x(waveName,pointNumber1)", and a similar expression for *x2*.

If the points nearest to *x1* or *x2* are outside the point range of 0 to numpnts(waveName)-1, median limits them to the nearest of point 0 or point numpnts(waveName)-1.

If the wave contains NaNs they are skipped.

The function returns NaN if the input wave has zero non-NaN points.

See Also

mean, **Variance**, **StatsMedian**, **StatsQuantiles**, **WaveStats**

MeasureStyledText

MeasureStyledText [/W=*winName* /A=*axisName* /F=*fontName* /SIZE=*fontSize* /STYL=*fontStyle*] *styledTextStr*

The MeasureStyledText operation takes as input a string optionally containing style codes such as are used in graph annotations. It sets various variables with information about the dimensions of the string.

Flags

/W=*winName*	Takes default text information from the window *winName*.
/A=*axisName*	Takes default text information from the axis named *axisName*. If the /W flag is used, the axis should be in that window (the window should also be a graph). If the /W flag is not used, MeasureStyledText looks at the top graph window.
/F=*fontNameStr*	The name of the default font.
/SIZE=*size*	Sets default font size.

/STYL=*fontStyle*	Sets default font style: Bit 0: Bold Bit 1: Italic Bit 2: Underline Bit 4: Strikethrough See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Parameters

styledTextStr	The text to be measured. The text can contain escape codes to set the font, size, style, color and other properties. See **Annotation Escape Codes** on page III-53 for details.

Details

In the absence of formatting codes within the text that set the font, font size and font style, some mechanism must be provided that sets them. The /W flag tells MeasureStyledText to look at a particular window and get defaults from that window.

The /A flag specifies that the defaults should come from a graph's axis of the given name. MeasureStyledText will look for the axis in the window named by /W, or in the top graph window in the absence of the /W flag.

The /F, /SIZE and /STYL flags set defaults that override any defaults from a window or axis. If you don't use any flags, the defaults are Igor's overall defaults.

Variables

The MeasureStyledText operation returns information in the following variables:

V_width	The width in points of the text.
V_height	The height in points of the text.

See Also

Annotation Escape Codes on page III-53 for a list of text formatting codes.

Menu

`Menu` ***`menuNameStr`*** `[`**`, hideable, dynamic, contextualmenu`**`]`

The Menu keyword introduces a menu definition. You can use this to create your own menu, or to add items to a built-in Igor menu.

Use the optional *hideable* keyword to make the menu hideable using **HideIgorMenus**.

Use the optional dynamic keyword to cause Igor to re-evaluate the menu definition when the menu is used. This is helpful when the menu item text is provided by a user-defined function. See **Dynamic Menu Items** on page IV-120.

Use the optional contextualmenu keyword for menus invoked by **PopupContextualMenu**/N.

See Chapter IV-5, **User-Defined Menus** for further information.

min

`min(`*`num1`*`,` ***`num2`*** `[`**`,`** ***`num3`*****`, ...`** ***`num200`***`]`**`)`**

The min function returns the least value of *num1*, *num2*, ... *num200*.

If any parameter is NaN, the result is NaN.

Details

In Igor7 or later, you can pass up to 200 parameters. Previously min was limited to two parameters.

See Also

max, **limit**, **WaveMin**, **WaveMax**

MLLoadWave

`MLLoadWave [`*`flags`*`] fileNameStr`

The MLLoadWave operation loads data from the named Matlab MAT file into single 1D waves (vectors), multi-dimensional waves (matrices), numeric variables or string variables.

For background information, including configuration instructions, see **Loading Matlab MAT Files** on page II-144.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If LoadWave can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

If *fileNameStr* is omitted or is "" or the /I flag is used, MLLoadWave displays an Open File dialog in which you locate the file to be loaded.

Flags

/A[=*name*]	Assign wave names using "wave" or *name*, if present, as the name or base name. Skips names already in use.
/B	This flag is obsolete and is ignored. Previously it was required to tell MLLoadWave the byte order of the data in the file. MLLoadWave now determines the byte order automatically.
/C	Loads columns from a Matlab matrix into an Igor 1D wave. Use /R to load rows.
/E	Skips empty Matlab matrices.
/G	Tells Igor to make numeric and string variables global when called from a macro. When called from a user-defined function or from the command line, variables are always created as globals.
/I	Interactive. Displays the Open File dialog to get the path to the file.
/M=m	m =1: Loads an entire Matlab matrix into an Igor 1D wave. This is the default if you omit /M.
	m =2: Loads an entire Matlab matrix into an Igor matrix.
	m =3: Loads an entire Matlab matrix into a transposed Igor matrix.
	/M by itself is equivalent to /M=1.
/N[=*name*]	Assign wave names using "wave" or *name*, if present, as the name or base name. Overwrites existing waves if the name is already in use.
/O	Overwrites existing waves and variables in case of a name conflict. If /O is omitted, MLLoadWave chooses names that don’t conflict with existing objects.
/P=*pathName*	Specifies the folder to look in for the specified file or folder. *pathName* is the name of an existing Igor symbolic path.
/Q	Be quiet. Suppresses normal diagnostic messages.
/R	Loads rows from a Matlab matrix into an Igor 1D wave. Use /C to load columns.

/S=s	Controls how Matlab string data is loaded: *s*=1 Skips Matlab string matrices. *s*=2 Loads Matlab string matrices into Igor string variables. This is the default if /S is omitted. *s*=3 Loads Matlab string matrices into Igor text waves. /S by itself is equivalent to /S=1.
/T	Displays the loaded waves in a new table.
/V	Skips Matlab numeric variables (numeric matrices with one element).
/Y=y	Specifies the number type of the numeric waves to be created. The allowed codes for y are: 2: Single-precision floating point 4: Double-precision floating point 32: 32-bit signed integer 16: 16-bit signed integer 8: 8-bit signed integer 96: 32-bit signed integer 80: 16-bit signed integer 72: 8-bit signed integer
/Z	Interactive load. Displays a dialog presenting options for each Matlab matrix in the file.

Details

If neither /A, /A[=name], /N, or N[=name] is used then the waves names are taken from the matrix name, as stored in the Matlab file.

When loading 1D waves, the /N flag instructs MLLoadWave to automatically name new waves "wave" (or *baseName* if /N=*baseName* is used) plus a number. The number starts from zero and increments by one for each wave loaded from the file. When loading multi-dimensional waves, *name* is used without an appended number.

The /A flag is like /N except that MLLoadWave skips names already in use.

If a given matrix is to be loaded into a single Igor wave, MLLoadWave uses the name without appending any digits. For example, if you have a 5x3 matrix in a file and you tell MLLoadWave to load it as a matrix using the name "mat", MLLoadWave will name the matrix "mat". However, if you tell MLLoadWave to load the matrix as 3 1D waves, it will use "mat0", "mat1" and "mat2".

If the name that MLLoadWave would use when creating a wave or variable is in use for an object of the same type and if you use the overwrite flag, then it will overwrite the existing object. If you do not tell MLLoadWave to overwrite, it will choose a non-conflicting name. If the conflict is with an object of a different type or with an operation or function, MLLoadWave will also choose a non-conflicting name.

When loading Matlab strings into Igor, you can tell MLLoadWave to create Igor string variables or Igor text waves. For example, if you have a 2x8 string matrix, MLLoadWave can create two string variables (/S=2) or one text wave (/S=3) containing two elements.

When loading Matlab string data into an Igor wave, the Igor wave will be of dimension one less than the Matlab data set. This is because each element in a Matlab string data set is a single byte whereas each element in an Igor string wave is a string (any number of bytes).

MLLoadWave loads numeric matrices with one element into Igor numeric variables. It loads all other numeric matrices into Igor waves.

When called from a macro, MLLoadWave creates local numeric and string variables unless you use the /G flag which tells it to create global variables. When called from the command line or from a user-defined function, MLLoadWave always creates global variables. Macros should be avoided in new programming.

For a discussion of how MLLoadWave handles 3D and 4D Matlab data, see Numeric Data Loading Modes.

The /Z flag instructs MLLoadWave to load each Matlab object (matrix, vector, variable, string) step by step. MLLoadWave presents a dialog for each Matlab object in the file. You can choose to load or skip the object. If you omit the /Z flag, MLLoadWave will load all objects in the file without presenting any dialogs.

When running MLLoadWave using the Mathworks-supplied libraries, MLLoadWave can handle data from any platform supported by the Mathworks libraries, presumably all platforms on which Matlab runs. When running MLLoadWave without having installed the Mathworks-supplied libraries, MLLoadWave can load VAX F (single precision) and G (double precision) but not D (extended precision) floating point data.

Output Variables

MLLoadWave sets the following output variables:

`S_path`	File system path to the folder containing the file. This is a system file path (e.g., "hd:FolderA:FolderB:"), not an Igor symbolic path. The path uses Macintosh path syntax, even on Windows, and has a trailing colon.
`S_fileName`	Name of the loaded file.
`V_flag`	Number of waves created.
`V_flag1`	Number of Matlab data sets (2D, 3D, or 4D) loaded.
`V_flag2`	Number of waves created.
`V_flag3`	Number of numeric variables created.
`V_flag4`	Number of string variables created.
`S_waveNames`	Semicolon-separated list of the names of loaded waves.

Prior to MLLoadWave 5.50, the variables V_Flag1, V_Flag2, V_Flag3 and V_Flag4 were named V1_Flag, V2_Flag, V3_Flag and V4_Flag.

See Also

Symbolic Paths on page II-21

See **Loading Matlab MAT Files** on page II-144 for background information, including configuration instructions.

mod

`mod(`*`num, div`*`)`

The mod function returns the remainder when *num* is divided by *div*.

The mod function may give unexpected results when *num* or *div* is fractional because most fractional numbers can not be precisely represented by a finite-precision floating point value.

See Also

trunc, **gcd**

ModDate

`ModDate(`*`waveName`*`)`

The ModDate function returns the modification date/time of the wave.

Details

The returned value is a double precision Igor date/time value, which is the number of seconds from 1/1/1904. It returns zero for waves created by versions of Igor prior to 1.2, for which no modification date/time is available.

See Also

The **Secs2Date** and **Secs2Time** functions.

Modify

`Modify`

We recommend that you use **ModifyGraph**, **ModifyTable**, **ModifyLayout**, or **ModifyPanel** rather than Modify. When interpreting a command, Igor treats the Modify operation as ModifyGraph, ModifyTable, ModifyLayout or ModifyPanel, depending on the target window. This does not work when executing a user-defined function.

ModifyBrowser

`ModifyBrowser [/M]` `[`***`keyword = value`***` [, `***`keyword = value`***` …]]`

The ModifyBrowser operation modifies the state of the Data Browser according to the specified keywords.

Documentation for the ModifyBrowser operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "ModifyBrowser"
```

ModifyCamera

`ModifyCamera` `[`***`flags`***`]` **`[`*`keywords`*`]`**

The ModifyCamera operation modifies the properties of a camera window.

Documentation for the ModifyCamera operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "ModifyCamera"
```

ModifyContour

`ModifyContour` `[/W=`***`winName`***`]`***`contourInstanceName, keyword=value`***
    `[, `***`keyword=value`***`…]`

The ModifyContour operation modifies the number, Z value and appearance of the contour level traces associated with *contourInstanceName*.

contourInstanceName is a name derived from the name of the wave that provides the Z data values. It is usually just the name of the wave, but may have #1, #2, etc. added to it in the unlikely event that the same Z wave is contoured more than once in the same graph.

contourInstanceName can also take the form of a null name and instance number to affect the instanceth contour plot. That is,

```
ModifyContour ''#1
```

modifies the appearance of the second contour plot in the top graph, no matter what the contour plot names are. Note: Two single quotes, not a double quote.

The number of contour level traces and their Z values are set by the *autoLevels*, *manLevels*, and *moreLevels* keywords, described in the **Parameters** section. Normally, you will use either *autoLevels* or *manLevels,* and then optionally generate additional levels using *moreLevels*.

Parameters

Each parameter has the syntax

```
keyword = value
```

and is applied to all of the contour level traces associated with *contourInstanceName*.

To modify an individual contour level trace, use **ModifyGraph**.

autoLevels= {*minLevel*, *maxLevel*, *numLevels*}

Controls automatic determination of contour levels.

If *numLevels* is zero, no automatic levels are generated. If it is nonzero, it specifies the desired number of automatic contour levels.

minLevel specifies the minimum contour level and *maxLevel* specifies the maximum contour level. The values that you specify are an approximate guide for Igor to use in determining the actual levels.

However, if *minLevel* or *maxLevel* is * (asterisk symbol), Igor uses the minimum or maximum value of the Z data for the corresponding contour level.

Using the autoLevels keyword cancels the effect of any previous autoLevels or manLevels keyword.

When you first append a contour plot to a graph, default contour levels are generated by the default setting `autoLevels={*,*,11}`.

boundary=*b*

Draws an outline around the XY domain of the contour data. For a matrix, this draws a rectangle showing the minimum and maximum X and Y values. For XYZ triples, the outline is a polygon enclosing the outside edges of the Delaunay Triangulation. Like the contour lines, the boundary is drawn using a graph trace, whose name is usually something like "`contourInstanceName = boundary`".

b=0:	Hides the data boundary (default).
b=1:	Shows the data boundary.

cIndexFill= *matrixWave*

Sets contour fills to use a color index wave when automatic fill is on (see the fill keyword).

cIndexFill works the same as the cIndexLines keyword which controls the colors of the contour level traces.

See **Contour Fills** on page II-286 for more information.

cIndexFill was added in Igor Pro 7.00.

cIndexLines= *matrixWave*

Sets the Z value mapping mode such that contour line colors are determined by doing a lookup in the specified matrix wave.

matrixWave is a 3 column wave that contains red, green, and blue values from 0 to 65535. (The matrix can actually have more than three columns. Any extra columns are ignored.)

The color for a the contour line at Z=*z* is determined by finding the RGB values in the row of *matrixWave* whose scaled X index is *z*. In other words, the red value is *matrixWave*(*z*)[0], the green value is *matrixWave*(*z*)[1] and the blue value is *matrixWave*(*z*)[2].

If *matrixWave* has default X scaling, where the scaled X index equals the point number, then row 0 contains the color for Z=0, row 1 contains the color for Z=1, etc.

If you use cIndexLines, you must not use ctabLines or rgbLines in the same command.

cTabFill= {*zMin*, *zMax*, *ctName*, *mode*}

Sets contour fills to use a color table when automatic fill is on (see the fill keyword).

cTabFill works the same as the ctabLines keyword which controls the colors of the contour level traces.

See **Contour Fills** on page II-286 for more information.

cTabFill was added in Igor Pro 7.00.

ctabLines={*zMin*, *zMax*, *ctName*, *mode*}

Sets the Z value mapping mode such that contour line colors are determined by doing a lookup in the specified color table. *zMin* is mapped to the first color in the color table. *zMax* is mapped to the last color. Z values between the min and max are linearly mapped to the colors between the first and last in the color table.

You can enter * (an asterisk) for *zMin* and *zMax*, which uses the minimum and maximum Z values of the data. The default is {*,*,Rainbow}.

Set parameter *mode* to 1 to reverse the color table; zero or missing does not reverse the color table.

ctName can be any color table name returned by the **CTabList** function, such as Grays or Rainbow (see **Image Color Tables** on page II-305) or the name of a 3 column or 4 column color table wave (see **Color Table Waves** on page II-311).

A color table wave name supplied to `ctabLines` must not be the name of a built-in color table (see **CTabList**). A 3 column or 4 column color table wave must have values that range between 0 and 65535. Column 0 is red, 1 is green, and 2 is blue. In column 3 a value of 65535 is opaque, and 0 is fully transparent.

If you use ctabLines, you must not use cIndexLines or rgbLines in the same command.

equalVoronoiDistances=*e*

Normally the x range and y range of the data are each normalized to a 0-1 range separately to generate the Voronoi triangulation. Voronoi triangulation is a distance-based ("nearest neighbor") algorithm that may benefit from scaling the X and Y ranges together to avoid numerical problems that occur when the triangles become very thin because of widely differing x and y ranges.

e=0:	The x and y ranges are scaled individually to the 0-1 range (default).
e=1:	The x and y ranges are scaled so that that maximum range of x or y is scaled to the 0-1 range, and the other is proportionally smaller. For example, if yMax-yMin = 1000 and xMax-xMin = 5, then the y range is scaled to 0-1 and the y range is scaled to 5/1000 = 0 - 0.005.

The equalVoronoiDistances keyword is allowed only for XYZ contour plots.

fill=*f*

Controls the automatic filling of contour levels.

f=0:	Turns automatic fill off. Default.
f=1:	Turns automatic fill on.

See **Contour Fills** on page II-286 for more information.

fill was added in Igor Pro 7.00.

interpolate=*i*	XYZ contours can be interpolated to increase the apparent resolution, resulting in smoother contour lines. This keyword is allowed only for XYZ contours, created by **AppendXYZContour**. *i*=0: Linear interpolation (default). This means that only the original Delaunay triangulation generates contour lines. *i*=1: Four times the resolution generates a smoother set of contour lines. As expected, this takes longer than Linear interpolation. *i*=2: Sixteen times the resolution generates a much smoother set of contour lines. This is rather slow. The interpolate parameter can be up to 8. Each time you increase *i* by one, you quadruple the apparent resolution and get smoother contour lines at the expense of computation time. Values of *i* greater than two are impractical because of the computation time required.
labelBkg=(*r*, *g*, *b*)	Sets the background color for all contour level labels to the specified color. *r*, *g*, and *b* are values from 0 to 65535.
labelBkg=*b*	Controls the background color of contour labels. *b*=0: Uses each label's individual background color, as set via the Modify Annotation dialog. *b*=1: Makes all contour level labels transparent. *b*=2: Uses the plot area background color as the label background color (default). *b*=3 Uses the window background color as the label background color.
labelDigits=*d*	*d* is the number of digits after the decimal point when using labelFormat=3 or labelFormat=5.
labelFont=*fontName*	Default; specifies the font to use for contour level labels. If you pass "" for *fontName*, it will use the graph font (set via the Modify Graph dialog) for contour labels.
labelFormat=*l*	Controls the formatting of contour labels. See the **printf** operation for a discussion of formatting. *l*=0: Uses general format that is suitable for most data. This is equivalent to "%<sigDigits>g". *l*=1: Uses integer format, equivalent to "%<sigDigits>d". This rounds fractional values. *l*=3: Uses fixed point format, equivalent to "%<decimalDigits>f". *l*=5 Uses exponential format, equivalent to "%<decimalDigits>e".
labelFSize=*s*	Specifies the font size of contour labels in points. For example, use labelSize=12 for 12 point type. The default value is 0, which chooses the size automatically based on the size of the graph.
labelFStyle=*n*	*n* is a bitwise parameter with each bit controlling one aspect of the font style for the contour level labels. The default is 0, plain text. Bit 0: Bold Bit 1: Italic Bit 2: Underline Bit 4: Strikethrough See **Setting Bit Parameters** on page IV-12 for details about bit settings.

labelHV=*hv*	Specifies the contour label orientation. If *hv* is 3, 4, 5, or 6, the contour label's text rotates whenever it is redrawn, usually when the underlying contour data changes, the graph is resized, or the label is reattached to a new contour trace point. *hv*=0: Horizontal contour level labels. *hv*=1: Vertical contour level labels. *hv*=2: Horizontal or vertical contour level labels, depending on the slope of the contour line. *hv*=3: Tangent to the contour line. *hv*=4: Tangent to the contour line, snaps to vertical or horizontal if within 2 degrees of vertical or horizontal (default). *hv*=5: Perpendicular to the contour line. *hv*=6: Perpendicular to the contour line, snaps to vertical or horizontal if within 2 degrees of vertical or horizontal.
labelRGB=(*r*, *g*, *b*)	Sets the text color for all contour level labels. *r*, *g*, and *b* are values from 0 to 65535. The default is black, labelRGB=(0,0,0).
labels=*l*	Controls the display of contour labels. *l*=0: Hides contour level labels. *l*=1: Leaves any contour level labels in place but stops updating them and stops generation of new labels. *l*=2: Generates or updates labels for the existing contour levels and window size when the command executes, but disables further updating of labels when window size or contour plot changes. This is the recommended setting if updating the labels takes long enough to annoy you. *l*=3: Default; generates labels for all contour levels whenever the contoured data changes but not when the window size changes. If you resize the graph, the labels may overlap or be too sparse. *l*=4: Generates labels for all contour levels whenever the contoured data, contour levels, axis range, or the graph size changes. (Actually, there are too many causes to list here. If all this update annoys you, use labels=2 “update once, now”.)
labelSigDigits=*d*	*d* is the number of significant digits when labelFormat=0 is used.
logLines= 1 or 0	0 sets the default linearly-spaced contour line colors. 1 turns on logarithmically-spaced line colors. This requires that the contour levels values be greater than 0 to display correctly. Affects line color only when the cIndexLines or ctabLines parameter is used. logLines does not affect the contour levels. To assign logarithmically-spaced contour levels, use the moreLevels parameter and disable autoLevels, for example:

```
ModifyContour ''#0, autoLevels={*,*,0} // No auto levels
ModifyContour ''#0, moreLevels=0
ModifyContour ''#0, moreLevels={1e-07,1e-06,1e-05,1e-04}
```

manLevels= {*firstLevel*, *increment*, *numLevels*}

Explicitly specifies contour levels. ModifyContour will generate *numLevels* contour levels, evenly spaced starting from *firstLevel* and stepping by *increment*.

manLevels cancels the effect of any previous manLevels or autoLevels settings.

manLevels= *manLevelsWave*

Explicitly specifies contour levels. ModifyContour will generate contour levels at the values in *manLevelsWave*.

manLevels cancels the effect of any previous manLevels or autoLevels settings.

moreLevels= {*level*, *level* …}

Explicitly specifies contour levels. ModifyContour will generate a contour trace for each of the listed levels. The maximum number of levels that you can specify in a single command is the 50. However, you can concatenate any number of ModifyContour moreLevels commands. moreLevels adds levels in addition to any specified by manLevels or autoLevels. It does not override other parameters.

moreLevels=0: Removes all levels generated by previous moreLevels settings.

nullValue=*zValue*

This keyword only affects the behavior of the **ContourZ** function. It is allowed only for XYZ contours, created by **AppendXYZContour**.

By default, ContourZ treats data outside the domain of the contour as NaN and so returns NaN if you ask for a contour value outside that domain.

The nullValue keyword allows you to change the default behavior to make ContourZ treat values outside the domain as the specified zValue.

nullValueAuto

This keyword only affects the behavior of the **ContourZ** function. It is allowed only for XYZ contours, created by **AppendXYZContour**.

nullValueAuto acts like nullValue=*zValue* with *zValue* automatically set to the minimum value in the Z wave minus 1.

See the nullValue keyword for details.

To turn nullValueAuto off and return the contour to the default state, execute:

```
ModifyContour <contourInstanceName>, nullValue=NaN
```

perturbation=*p*

Enable or disable perturbation (alteration) of the x and y values by a miniscule amount to improve the natural neighbor triangulation of XYZ contours.

p=0:	Disables perturbation, preserving the original x and y values unchanged.
p=1:	Enables x/y perturbation (default). The values are shifted by random values less than +/-0.000005 times the x and y domain extents.

You can observe the perturbed x/y coordinates in the triangulation trace added by ModifyContour triangulation=1.

The perturbation keyword is allowed only for XYZ contour plots.

rgbFill=(*r*, *g*, *b*)

Specifies red, green, and blue values for all contour fills. *r*, *g*, and *b* are values from 0 to 65535.

If you use rgbFill, you must not use cIndexFill or ctabFill in the same command.

rgbLines=(*r*, *g*, *b*)

Specifies red, green, and blue values for all contour lines. *r*, *g*, and *b* are values from 0 to 65535.

If you use rgbLines, you must not use cIndexLines or ctabLines in the same command.

triangulation=*t*

Draws the Delaunay Triangulation. As part of the XYZ contouring algorithm, the XY domain is subdivided into triangles in a process called Delaunay Triangulation. Like the contour lines, the triangulation is drawn using a graph trace, whose name is usually something like "contourInstanceName =triangulation".

The triangulation keyword is allowed only for XYZ contours, created by **AppendXYZContour**.

t=0:	Hides the Delaunay triangulation (default).
t=1:	Shows the Delaunay triangulation.

update=*u*	Sets the type of updating of contour traces when the data or contour settings change.
	u=0: Turns off dynamic updates, which might be advisable if updates take a long time.
	u=1: Updates the contours only once, or until you next execute an update=1 command.
	u=2: Updates are automatic (default).
	u=3 Marks the contour plot as having been updated once (*u*=1) already. This option is used in recreation macros to prevent an extra redraw of a graph saved with *u*=1 update mode in effect.
	If you use it in a command, the result is similar to *u*=0, but the Modify Contour Appearance dialog will automatically select "update once, now" from the Update Contours pop-up menu.
xymarkers=*x*	Controls the visibility of XY markers.
	x=0: Hides markers showing XY coordinates of the Z data (default).
	x=1: Displays markers showing XY coordinates of Z data. Initially, this uses marker number zero. You can change this using the Modify Trace Appearance dialog.

Flags

/W=*winName*	Applies to contours in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line.
	When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

See Also

AppendMatrixContour and **AppendXYZContour**.

References

Watson, David F., *nngridr - An Implementation of Natural Neighbor Interpolation*, Dave Watson Publisher, Claremont, Australia, 1994.

ModifyControl

`ModifyControl` `[/Z]` ***`ctrlName`*** `[`***`keyword`*** **`=`** ***`value`*** `[`**`,`** ***`keyword`*** **`=`** ***`value`*** `…]]`

The ModifyControl operation modifies the named control. ModifyControl works on any kind of existing control. To modify multiple controls, use **ModifyControlList**.

Parameters

ctrlName specifies the name of the control to be created or changed. The control must exist.

Keywords

The following keyword=value parameters are supported:

`activate`	`appearance`	`bodywidth`	`disable`	`fColor`	`focusRing`
`font`	`fSize`	`fStyle`	`help`	`labelBack`	`noproc`
`pos`	`proc`	`rename`	`size`	`title`	`userdata`
`valueBackColor`	`valueColor`	`win`			

For details on these keywords, see the documentation for **SetVariable** on page V-729.

The following keywords are not supported:

mode	popmatch	popvalue	value	variable

Flags

/Z	No error reporting.

Details

Use ModifyControl to move, hide, disable, or change the appearance of a control without regard to its kind

Example

Here is a **TabControl** procedure that shows and hides all controls in the tabs appropriately, without knowing what kind of controls they are.

The “trick” here is that all controls that are to be shown within particular tab *n* have been assigned names that end with “_tab*n*” such as “_tab0” and “_tab1”:

```
Function TabProc(ctrlName,tabNum) : TabControl
    String ctrlName
    Variable tabNum

    String curTabMatch= "*_tab"+num2istr(tabNum)

    String controls= ControlNameList("")
    Variable i, n= ItemsInList(controls)
    for(i=0; i<n; i+=1)
        String control= StringFromList(i, controls)
        Variable isInATab= stringmatch(control,"*_tab*")
        if( isInATab )
            Variable show= stringmatch(control,curTabMatch)
            ControlInfo $control                    // gets V_disable
            if( show )
                V_disable= V_disable & ~0x1         // clear the hide bit
            else
                V_disable= V_disable | 0x1          // set the hide bit
            endif
            ModifyControl $control disable=V_disable
        endif
    endfor
    return 0
End

// Action procedures which enable or disable the buttons
Function Tab1CheckProc(ctrlName,enableButton) : CheckBoxControl
    String ctrlName
    Variable enableButton

    ModifyControl button_tab1, disable=(enableButton ? 0 : 2 )
End

Function Tab0CheckProc(ctrlName,enableButton) : CheckBoxControl
    String ctrlName
    Variable enableButton

    ModifyControl button_tab0, disable=(enableButton ? 0 : 2 )
End

// Panel macro that creates a TabControl using TabProc
Window TabbedPanel() : Panel
    PauseUpdate; Silent 1            // building window...
    NewPanel /W=(381,121,614,237) as "Tab Demo"
    TabControl tab, pos={12,9},size={205,91},proc=TabProc,tabLabel(0)="Tab 0"
    TabControl tab, tabLabel(1)="Tab 1",value= 0
    Button button_tab0, pos={54,39},size={110,20},disable=2
    Button button_tab0, title="Button in Tab0"
    Button button_tab1, pos={54,63},size={110,20},disable=1
    Button button_tab1, title="Button in Tab1"
    CheckBox check1_tab1, pos={51,41}, size={117,14}, disable=1, value= 1
    CheckBox check1_tab1, proc=Tab1CheckProc, title="Enable Button in Tab 1"
    CheckBox check0_tab0, pos={51,73}, size={117,14}, proc=Tab0CheckProc
    CheckBox check0_tab0, value= 0, title="Enable Button in Tab 0"
EndMacro
```

Run TabbedPanel to create the panel. Then click on “Tab 0” and “Tab 1” to run TabProc.

See Also

See Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

Related functions **ModifyControlList** and **ControlNameList**.

The **Button**, **Chart**, **CheckBox**, **GroupBox**, **ListBox**, **PopupMenu**, **SetVariable**, **Slider**, **TabControl**, **TitleBox**, and **ValDisplay** controls.

ModifyControlList

`ModifyControlList [/Z] `*`listStr`*` [, keyword = `*`value`*`]…`

The ModifyControlList operation modifies the controls named in the *listStr* string expression. ModifyControlList works on any kind of existing control.

Parameters

listStr is a semicolon-separated list of names in a string expression. The expression can be an explicit list of control names such as "button0;checkbox1;" or it can be any string expression such as a call to the ControlNameList string function:

```
ModifyControlList ControlNameList("",";","*_tab0") disable=1
```

The controls must exist.

Keywords

The following keyword=value parameters are supported:

activate	appearance	bodywidth	disable	fColor	focusRing
font	fSize	fStyle	help	labelBack	noproc
pos	proc	rename	size	title	userdata
valueBackColor	valueColor	win			

For details on these keywords, see the documentation for **SetVariable** on page V-729.

The following keywords are not supported:

mod	popmatch	popvalue	value	variable

Flags

/Z	No error reporting.

Details

Use ModifyControlList to move, hide, disable, or change the appearance of multiple controls without regard to their kind.

If *listStr* contains the name of a nonexistent control, an error is generated.

if *listStr* is `""` (or any list element in *listStr* is `""`), it is ignored and no error is generated.

Example

Here is the **TabControl** procedure example from **ModifyControl** rewritten to use ModifyControlList. It shows and hides all controls in the tabs appropriately, without knowing what kind of controls they are, but the code is simpler. This method does not, however, preserve the enable bit when a control is hidden.

The “trick” here is that all controls that are to be shown within particular tab *n* have been assigned names that end with “_tab*n*” such as “_tab0” and “_tab1”:

```
// Action procedure
Function TabProc2(ctrlName,tabNum) : TabControl
	String ctrlName
	Variable tabNum

	String controlsInATab= ControlNameList("",";","*_tab*")

	String curTabMatch= "*_tab"+num2istr(tabNum)
	String controlsInCurTab= ListMatch(controlsInATab, curTabMatch)
	String controlsInOtherTabs=ListMatch(controlsInATab,"!"+curTabMatch)
```

```
   ModifyControlList controlsInOtherTabs disable=1          // hide
   ModifyControlList controlsInCurTab disable=0             // show

   return 0
End

// Panel macro that creates a TabControl using TabProc2():
Window TabbedPanel2() : Panel
   PauseUpdate; Silent 1                  // building window…
   NewPanel /W=(35,208,266,374) as "Tab Demo"
   TabControl tab,pos={12,9},size={205,140},proc=TabProc2
   TabControl tab,tabLabel(0)="Tab 0"
   TabControl tab,tabLabel(1)="Tab 1",value= 0
   Button button_tab0,pos={26,43},size={110,20},title="Button in Tab0"
   Button button2_tab0,pos={26,74},size={110,20},title="Button in Tab0"
   Button button3_tab0,pos={26,106},size={110,20},title="Button in Tab0"
   Button button_tab1,pos={85,43},size={110,20},title="Button in Tab1"
   Button button2_tab1,pos={85,75},size={110,20},title="Button in Tab1"
   Button button3_tab1,pos={84,108},size={110,20},title="Button in Tab1"
   ModifyControlList ControlNameList("",";","*_tab1") disable=1
EndMacro
```

Run TabbedPanel2 and then click on "Tab 0" and "Tab 1" to run TabProc2.

See Also

See Chapter III-14, **Controls and Control Panels** for details about control panels and controls.

Related functions **ModifyControl** and **ControlNameList**.

The **Button**, **Chart**, **CheckBox**, **GroupBox**, **ListBox**, **PopupMenu**, **SetVariable**, **Slider**, **TabControl**, **TitleBox**, and **ValDisplay** controls.

ModifyFreeAxis

ModifyFreeAxis [**/W=*winName***] ***axisName*, master=*mastName***
  [**, hook=*funcName***]

The ModifyFreeAxis operation designates the free axis (created with **NewFreeAxis**) to follow a controlling axis from which it gets axis range and units information. The free axis updates whenever the controlling axis changes. The axis limits and units can be modified by a user hook function.

Parameters

axisName is the name of the free axis (which must have been created by **NewFreeAxis**).

masterName is the name of the master axis controlling *axisName*.

funcName is the name of the user function that modifies the limits and units properties of the axis. If *funcName* is `$""`, the named hook function is removed.

Flags

/W=*winName*	Modifies *axisName* in the named graph window or subwindow. If /W is omitted the command affects the top graph window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The free axis can also be designated to call a user-defined hook function that can modify limits and units properties of the axis. The hook function must be of the following form:

```
Function MyAxisHook(info)
   STRUCT WMAxisHookStruct &info

   <code to modify graph units or limits>
   return 0
End
```

where `WMAxisHookStruct` is a built-in structure with the following members:

`WMAxisHookStruct` Structure Members

Member	Description
`char win[MAX_WIN_PATH+1]`	Host (sub)window.
`char axName[MAX_OBJ_NAME+1]`	Name of the axis.
`char mastName[MAX_OBJ_NAME+1]`	Name of controlling axis or nil.
`char units[MAX_UNITS+1]`	Axis units. User modifiable.
`double min, max`	Axis range minimum and maximum values. User modifiable.

The constants used to size the `char` arrays are internal to Igor and are subject to change in future versions.

The hook function is called when refreshing axis range information (generally early in the update of a graph). Your hook must never kill a graph or an axis.

Example

This example demonstrates how to program a free axis hook function, whose most important task is to change the values of info.min and info.max to alter the axis range of the free axis. The example free axis displays Fahrenheit values for data in Celsius.

```
Function CentigradeAndFahrenheit()
	Make/O/N=20 temperatures = -2+p/3+gnoise(0.5)  // sample data
	Display temperatures// default left axis will indicate data's centigrade range
	String graphName = S_name
	Label/W=$graphName left "°C"
	ModifyGraph/W=$graphName zero(left)=1
	Legend/W=$graphName

	// make a right axis whose range will be Fahrenheit
	NewFreeAxis/R/O/W=$graphName fahrenheit
	ModifyGraph/W=$graphName freePos(fahrenheit)={0,kwFraction},lblPos(fahrenheit)=43
	Label/W=$graphName fahrenheit "°F"

	ModifyFreeAxis/W=$graphName fahrenheit, master=left, hook=CtoF_FreeAxisHook
	// NOTE master=left part which makes the "free" axis
	// actually a "slave" to the left ("master") axis.
End

Function CtoF_FreeAxisHook(info)
	STRUCT WMAxisHookStruct &info

	GetAxis/Q/W=$info.win $info.mastName    // get master axis range in V_min, V_Max
	Variable minF = V_min*9/5+32
	Variable maxF = V_max*9/5+32

//	SetAxis/W=$info.win $info.axName, minF, maxF
//	SetAxis here is fruitless. These values get overwritten by Igor
//	after reading info.min and info.max, which we now set:
	info.min = minF           // new min for free axis
	info.max= maxF            // new max for free axis
	return 0
End
```

See Also

The **SetAxis**, **KillFreeAxis**, and **NewFreeAxis** operations.

The **ModifyGraph (axes)** operation for changing other aspects of a free axis.

ModifyGizmo

`ModifyGizmo` `[`***`flags`***`]` ***`keyword`*** `[=`***`value`***`]`

The ModifyGizmo operation changes Gizmo properties.

Documentation for the ModifyGizmo operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "ModifyGizmo"
```

ModifyGraph *(general)*

`ModifyGraph [/W=`*`winName`*`/Z]` ***`key=value`*** `[,` ***`key=value`***`]…`

The ModifyGraph operation modifies the target or named graph. This section of ModifyGraph relates to general graph window settings.

Parameters

expand=*e*	Specifies the onscreen expansion (or magnification) factor of a graph. *e* may be zero or 0.125 to 8 times expansion. Graph magnification affects only base graphs (not subwindowed graphs), and it affects only the onscreen display; it has **no** effect on graph exporting or printing. When magnification changes, the graph window will automatically resize except for negative values, which are used in recreation macros where the size is already correct.
frameInset= *i*	Specifies the number of pixels by which to inset the frame of the graph subwindow.
frameStyle= *f*	Specifies the frame style for a graph subwindow. *f*=0: None. *f*=1: Single. *f*=2: Double. *f*=3: Triple. *f*=4: Shadow. *f*=5: Indented. *f*=6: Raised. *f*=7: Text well. The last three styles are fake 3D and will look good only if the background color of the enclosing space and the graph itself is a light shade of gray.
gfMult=*f*	Multiplies font and marker size by *f* percent. Clipped to between 25% and 400%; it is applied after all other font and marker size calculations.
gFont=*fontStr*	Specifies the name of the default font for the graph, overriding the normal default font. The normal default font for a subgraph is obtained from its parent while a base graph uses the value set by the **DefaultFont** operation.
gfSize=*gfs*	Sets the default size for text in the graph. Normally, the default size for text is proportional to the graph size; gfSize will override that calculation as will the gfRelSize method. Use a value of -1 to make a subgraph get its default font size from its parent.
gfRelSize=*pct*	Specifies the percentage of the graph size to use in calculating a default size for text in the graph. This overrides the normal method for setting default font size as a function of graph size. When used, the default marker size is set to one third the font size. Use a value of 0 to revert to the default method.
gmSize=*gms*	Sets the default size for markers in the graph. Use a value of -1 to make a subgraph get its default marker size from its parent.
height=*heightSpec*	Sets the height for the graph area. See the **Examples**.

swapXY=*s* — Sets the orientation of the X and Y axes.

s=0:	Normal orientation of X and Y axes.
s=1:	Swap X and Y values to plot Y coordinates versus the horizontal axes and X coordinates versus the vertical axes. The effect is similar to mirroring the graph about the lower-left to upper-right diagonal.

useComma=*uc* — Controls the decimal separator used in tick mark labels.

uc=0:	Use period as decimal separator and comma as thousands separator (default) when displaying numbers in graph labels and annotations.
uc=1:	Use comma as decimal separator and period as the thousands separator. This does not alter the presentation of numbers in \\{*expression*} constructs in annotations.

UIControl=*f* — Disables certain aspects of the user interface for graphs. The UIControl keyword, added in Igor Pro 7.00, is for use by advanced Igor programmers who want to disable user actions.

This is a bitwise setting. **Setting Bit Parameters** on page IV-12 for details about bit settings.

f is defined as follows:

Bit 0:	Disable axis click. Prevents moving or otherwise modifying an axis.
Bit 1:	Disable cursor click. Prevents moving a graph cursor.
Bit 2:	Disable trace drag. Prohibits the click-and-hold action to offset a trace on the graph.
Bit 3:	Disable marquee. When set, you can't make a marquee on the graph, which in turn prevents changing the range of the graph using the marquee.
Bit 4:	Disable draw mode.
Bit 5:	Disable double click. Prohibits any double-click action. In general, double-clicks in a graph bring up dialogs to modify the graph's appearance.
Bit 6:	Disable clicks on annotations. Prevents modification of annotations.
Bit 7:	Disable tool tips.
Bit 8:	Disable contextual menus.
Bit 9:	Disable marquee menu. With this set, you can still have a marquee and use it for, i.e., selecting some portion of the graph, but you can't use the maruqee menu to change the graph's range. Note that if bit 3 is set, this bit is moot.
Bit 10:	Disable mouse wheel events. This will prevent axis scaling using the mouse wheel.
Bit 11:	Disable option-drag. Prevents offsetting the graph by holding down the option (Macintosh) or Alt (Windows) key and then dragging in the plot area.

To disable items in the Graph menu use **SetIgorMenuMode**.

useLongMinus=*m* — Uses a normal (*m*=0; default) or long dash (*m*=1) for the minus sign.

width=*widthSpec* — Sets the width of the graph area. See the examples.

Flags

/W=*winName*	Modifies the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/Z	Does not generate an error if the indexed trace, named wave, or named axis does not exist in a style macro.

Examples

The following code creates a graph where all the text expands and contracts directly in relation to the window size:

```
Make jack=sin(x/8);display jack
ModifyGraph mode=4,marker=8,gfRelSize= 5.0
TextBox/N=text0/A=MC "Some \\Zr200big\\]0 and \\Zr050small\\]0\rtext"
```

The *widthSpec* and *heightSpec*s set the width and height mode for the top graph. The following examples illustrate how to specify the various modes.

`ModifyGraph width=0, height=0`	Set to auto height, width mode. The width, height of horizontal and vertical axes are automatically determined based on the overall size of the graph and other factors such as axis offset setting and effect of exterior textboxes. This is the normal, default mode.

`Variable n=72*5`
`ModifyGraph width=n`
	Five inches as points absolute width mode, horizontal axis width constrained to `n` points.

`ModifyGraph height=n`	Absolute height mode, `n` is in points. The height of the vertical axes is constrained to `n` points.

`Variable n=2`
`ModifyGraph`
`width={perUnit,n,bottom}`
	Per unit width mode. The width of the horizontal axes is `n` points times the range of the bottom axis.

`ModifyGraph height={Aspect,n}`	Aspect height mode, `n = aspect ratio`. The height of the vertical axes is `n` times the width of the horizontal axes.

`ModifyGraph`
`width={Plan,n,bottom,left}`
	Plan width mode. The width of the horizontal axes is `n` times the height of the vertical axes times range of the bottom axis divided by the range of the left axis.

ModifyGraph *(traces)*

`ModifyGraph` `[`***`/W=winName/`***`Z]` ***`key`*** `[`**`(`*`traceName`*`)`**`]` **`=`** ***`value`***
   `[`**`,`** ***`key`*** `[`**`(`*`traceName`*`)`**`]` **`=`** ***`value`***`]…`

This section of ModifyGraph relates to modifying the appearance of wave “traces” in a graph. A trace is a representation of the data in a wave, usually connected line segments.

Parameters

Each *key* parameter may take an optional *traceName* enclosed in parentheses. Usually *traceName* is simply the name of a wave displayed in the graph, as in “`mode(myWave)=4`”. If “(*traceName*)” is omitted, all traces in the graph are affected. For instance, “`ModifyGraph lSize=0.5`” sets the lines size of all traces to 0.5 points.

For multiple trace instances, *traceName* is followed by the “#” character and instance number. For example, “`mode(myWave#1)=4`”. See **Instance Notation** on page IV-19.

A string containing a trace name can be used with the $ operator to specify *traceName.* For example, `String MyTrace="myWave#1"; mode($MyTrace)=4`.

Though not shown in the syntax, the optional “(*traceName*)” may be replaced with “[*traceIndex*]”, where *traceIndex* is zero or a positive integer denoting the trace to be modified. “[0]” denotes the first trace appended to the graph, “[1]” denotes the second trace, etc. This syntax is used for style macros, in conjunction with the /Z flag.

For certain modes and certain properties, you can set the conditions at a specific point on a trace by appending the point number in square brackets after the trace name. For more information, see the **Customize at Point** on page V-532.

The parameter descriptions below omit the optional "(*traceName*)". When using ModifyGraph from a user-defined function, be careful not to pass wave references to ModifyGraph. ModifyGraph expects trace names, not wave references. See **Trace Name Parameters** on page IV-82 for details.

arrowMarker=0

arrowMarker={*aWave*, *lineThick*, *headLen*, *headFat*, *posMode* [, *barbSharp*=*b*, *barbSide*=*s*, *frameThick*=*f*]}

Draws arrows instead of conventional markers at each data point in a wave. Arrows are not clipped to the plot area and will be drawn wherever a data point is within the plot area.

aWave contains arrow information for each data point. It is a two (or more) column wave containing arrow line lengths (in points) in column 0 and angles (in radians measured counterclockwise) in column 1. Zero angle is a horizontal arrow pointing to the right. If an arrow is below the minimum length of 4 points, a default marker is drawn.

You can change arrow markers into standard meteorological wind barbs by adding a column to *aWave* and giving it a column label of windBarb. Values are integers from 0 to 40 representing wind speeds up to 4 flags. Use positive integers for clockwise barbs and negative for the reverse. Use NaN to suppress the drawing. See **Wind Barb Plots** on page II-258 for an example.

Additional columns may be supplied in *aWave* to control parameters on a point by point basis. These optional columns are specified by dimension label and not by specific column numbers. The labels are *lineThick*, *headLen*, and *headFat* that correspond to the same parameters listed above.

lineThick is the line thickness in points.

headLen is the arrow head length in points.

headFat controls the arrow fatness. It is the width of the arrow head divided by the length.

posMode specifies the arrow location relative to the data point.

posMode=0:	Start at point.
posMode=1:	Middle on point.
posMode=2:	End at point.

In addition to the wave specification, *aWave* can also be the literal `_inline_` to draw lines and arrows between points on the trace (see **Examples**). If *aWave* is `_inline_`, *posMode* values are:

posMode=0:	Arrow at start.
posMode=1:	Arrow in middle.
posMode=2:	Arrow at end.
posMode=3:	Arrow in middle pointing backwards.

You can also enable inline mode even if *aWave* is not `_inline_` by setting posMode to values between 4 and 7. These are the same as modes 0-3 above.

Optional parameters must be specified using *keyword* = *value* syntax and can only be appended after *posMode* in any order.

barbSharp is the continuously variable barb sharpness between -1.0 and 1. 0:

barbSharp=1:	No barb; lines only.
barbSharp=0:	Blunt (default).
barbSharp=-1:	Diamond.

barbSide specifies which side of the line has barbs relative to a right-facing arrow:

barbSide=0:	None.
barbSide=1:	Top.
barbSide=2:	Bottom.
barbSide=3:	Both (default).

frameThick specifies the stroke outline thickness of the arrow in points. The default is *frameThick* = 0 for solid fill.

aWave can contain columns with data for each optional parameter using matching column names.

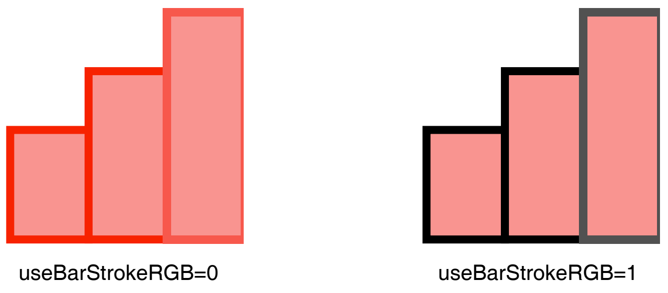
barStrokeRGB=(*r*,*g*,*b*)

Specifies a separate color for bar strokes (outlines) if useBarStrokeRGB is 1. *r*, *g* and *b* specify the amount of red, green and blue in the color of the stroked lines as an integer from 0 to 65535. The default is black (0,0,0).

Applies only to Histogram Bars drawing mode (mode=5).

The bar fill color continues to be set with the rgb=(*r*,*g*,*b*), zColor={...}, usePlusRGB, plusRGB=(*r*,*g*,*b*), useNegRGB, and negRGB=(*r*,*g*,*b*) parameters.

Use barStrokeRGB and useBarStrokeRGB to put a differently-colored outline around Histogram Bars:



cmplxMode=*c* Display method for complex waves.

c=0:	Default mode displays both real and imaginary parts (imaginary part offset by dx/2).
c=1:	Real part only.
c=2:	Imaginary part only.
c=3:	Magnitude.
c=4:	Phase (radians).

cmplxMode=0 does not work when the trace is a subrange of a multi-dimensional wave.

column=*n* Changes the displayed column from a matrix. Out of bounds values are clipped.

gaps=*g* Controls treatment of NaNs:

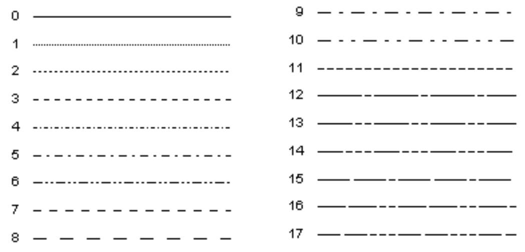
g=0:	No gaps (ignores NaNs).
g=1:	Gaps (shows NaNs as gaps).

gradient=<*parameters*>

Controls color gradients for graph trace fills. See **Gradient Fills** on page III-441 for details.

gradientExtra=<*parameters*>

	Controls color gradient details for graph trace fills. See **Gradient Fills** on page III-441 for details.
hBarNegFill=*n*	Fill kind for negative areas if useNegPat is true. *n* is the same as for the hbFill keyword.
hbFill=*n*	Sets the fill pattern. *n*=0: No fill. *n*=1: Erase. *n*=2: Solid black. *n*=3: 75% gray. *n*=4: 50% gray. *n*=5: 25% gray. *n*>=6: See **Fill Patterns** on page III-441.
hideTrace=*h*	Removes a trace from the graph display. *h*=0: Shows the trace if it is hidden. *h*=1: Hides the trace and removes it from autoscale calculations. *h*=2: Hides the trace. When using *h*=1 to hide a graph trace, the hidden trace symbol and following text in annotations are also hidden. The amount of hidden text is the lesser of: the remaining text on the same line or the text up to but not including another trace symbol `"\s(traceName)"`.
lHair=*lh*	Sets the hairline factor for traces printed on a PostScript® printer.
live=*lv*	Turns Live Mode off (*lv*=0) or on (*lv*=1).
logZColor=*lzc*	Controls the interpretation of the zColor parameter. *lzc*=0: Sets the default linearly-spaced zColors. *lzc*=1: Turns on logarithmically-spaced zColors. This requires that the zWave values be greater than 0 to display correctly. Affects trace line color only when the zColor parameter is used with a color table or color index wave - it has no effect if rgb=(*r*,*g*,*b*) parameter or zColor={...,directRGB} are used.
lOptions=*options*	*options* is a bitwise parameter: Bit 0: If set, dashed lines use round end caps. If cleared they use square end caps. All other bits are reserved and must be cleared. See **Setting Bit Parameters** on page IV-12 for details about bit settings.
lSize=*l*	Sets the line thickness, which can be fractional or zero, which hides the line.
lSmooth=*ls*	Sets the smoothing factor for traces printed on a PostScript® printer.
lStyle=*s*	Sets trace line style or dash pattern. *s*=0 for solid lines. *s*=1 to s=17 for various dashed line styles.



marker=*n* n =0 to 62 designates various markers if mode=3 or 4.

You can also create custom markers. See the **SetWindow** markerHook keyword.

See **Markers** on page II-223 for a table of marker values.

mask={*maskwave*,*mode*,*value*} or 0

Specifies individual points for display by comparing values in *maskWave* with *value* as specified by *mode*.

mode=0:	Exclude if equal.
mode=1:	Include if equal.
mode=2:	Include if bitwise AND is true.
mode=3:	Include if bitwise AND is false.

maskwave can be specified using subrange notation. The length of *maskwave* (or subrange) must match the size of specified trace's wave (or subrange.) Bitwise modes should be used with integer waves with the intent of using one mask wave with multiple traces. See **Examples**.

mode=*m* Sets trace display mode.

m=0:	Lines between points.
m=1:	Sticks to zero.
m=2:	Dots at points.
m=3:	Markers.
m=4:	Lines and markers.
m=5:	Histogram bars.
m=6:	Cityscape.
m=7:	Fill to zero.
m=8:	Sticks and markers.

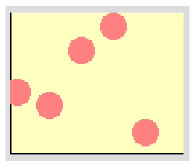
mrkStrokeRGB=(*r*,*g*,*b*)

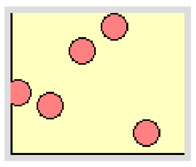
Specifies the color for marker stroked lines if useMrkStrokeRGB = 1. *r*, *g*, and *b* values are the amount of red, green, and blue in the color of the lines as an integer from 0 to 65535. The default is black (0,0,0).

The marker fill color continues to be set with the rgb=(*r*,*g*,*b*) or zColor={…} parameters.

Applies only to the nontext and nonarrow marker modes.

Use mrkStrokeRGB and useMrkStrokeRGB to put a colored outline around filled markers, such as marker=19:


useMrkStrokeRGB=0


useMrkStrokeRGB=1

Note: The stroke color of unfilled markers such as marker 8 is also affected by mrkStrokeRGB, but their fill color is only affected by the opaque parameter (and the opaque fill color is always white, so if you want a color-filled marker, don't use unfilled markers).

mrkThick=*t* Sets the thickness of markers in points, which can be fractional.

msize=*m* Specifies the marker size in points.

m=0:	Autosize markers.
m>0:	Sets marker size.

m can be fractional, which will only make a difference when the graph is printed because fractional points can not be displayed on the screen.

mskip=*n* Puts a marker on only every *n*th data point in Lines and Markers mode (mode=4). Useful for displaying many data points when you want to identify the traces with markers. The maximum value for *n* is 32767.

muloffset={*mx*,*my*} Sets the display multiplier for X (*mx*) and Y (*my*). The effective value for a given X or Y data point then becomes *muloffset***data*+*offset*. A value of zero means "no multiplier" — not multiply by zero.

negRGB=(*r*, *g*, *b*) Specifies the color for negative areas if useNegRGB is 1. *r*, *g*, and *b* specify the amount of red, green, and blue in the color of the trace as an integer from 0 to 65535.

offset={*x*,*y*} Sets the display offset in horizontal (X) and vertical (Y) axis units.

opaque=*o* Displays transparent (*o*=0) or opaque (*o*=1) markers.

patBkgColor= 0, 1, 2 or (r,g,b)

Specifies the background color for fill patterns.

0, the default, is white, 1 is graph background, 2 is transparent (does not work when exporting in the Enhanced Metafile or Windows Metafile formats).

Use (r,g,b) for a specific RGB color.

plotClip=*p* *p* =1 clips the trace by the operating system (not by Igor) to the plot rectangle. This trims overhanging markers and thick lines. On Windows, this may not be supported for certain printers or by certain applications when importing.

plusRGB=(*r*, *g*, *b*) Specifies the color for positive areas if usePlusRGB is 1. *r*, *g*, and *b* specify the amount of red, green, and blue in the color of the trace as an integer from 0 to 65535.

quickdrag=*q* Controls dragging of traces.

q=0:	Normal traces.
q=1:	Traces that can be instantly dragged without the normal one second delay. See the **Quickdrag** section below.
q=2:	Causes the mouse cursor to change to 4 arrows when over the trace and a reduced search is used.

rgb=(*r*,*g*,*b*) Specifies the amount of red, green, and blue (*r*, *g*, and *b)* in the color of the trace as an integer from 0 to 65535.

textMarker={<*char* or *wave*>,*font*,*style*,*rot*,*just*,*xOffset*,*yOffset*} or 0

Uses the specified character or text from the specified wave in place of the marker for each point in the trace.

If the first parameter is a quoted string or a string expression of the form `""+strexpr` in a user function, ModifyGraph uses the first three bytes of the string as the marker for all points. Three bytes are supported mainly for non-ASCII characters but can be used for 3 separate single-byte characters. Otherwise, it interprets the first parameter as the name of a wave. If the wave is a text wave, it uses the value of each point in the text wave as the marker for the corresponding point in the trace. If the wave is a numeric wave, the value for each point is converted into text and the result is used as the marker for the corresponding point in the trace.

xOffset and *yOffset* are offsets in fractional points. Each marker will be drawn offset from the location of the corresponding point in the trace by these amounts.

style is a font style code as used with the ModifyGraph fstyle keyword.

rot is a text rotation between -360 and 360 degrees.

just is a justification code as used in the **DrawText** operation except the X and Y codes are combined as y*4+x. Use 5 for centered.

The font size is 3*marker size. Note that marker size and color can be dynamically set via the zColor and zmrkSize keywords.

toMode=*t* Modifies the behavior of the display modes as determined by the mode parameter.

t=0:	Fill to zero.
t=1:	Fill to next trace. Applies to Sticks to zero (mode=1), histogram bars (mode=5), and fill to zero (mode=7).
t=2:	Add the current trace's Y values to the next trace's Y values. Works with all display modes.
t=3:	Stack on next and is the same as *t*=2 except that the added value is clipped to zero. Works with all display modes.
t=-1:	This mode is used only with category plots and means "keep with next" (i.e., put in the same subcategory as the next trace). It is used for special effects only.

For modes 1, 2 and 3, both Y-waves must have the same number of points and must use the same X values. Igor uses the X values from the first wave for both Y-

useBarStrokeRGB=*u*

If *u*=1 then bar stroked lines use the color specified by the barStrokeRGB keyword.

Applies only to Histogram Bars drawing mode (mode=5).

The bar fill color continues to be set with the rgb=(*r*,*g*,*b*), zColor={...}, usePlusRGB, plusRGB=(*r*,*g*,*b*), useNegRGB, and negRGB=(*r*,*g*,*b*) parameters.

If *u*=0 then the bar stroked line colors are set with the rgb=(*r*,*g*,*b*) or zColor={...} parameters, just like the bar fill color.

useMrkStrokeRGB=*u*

If *u* =1 then marker stroked lines use the color specified by the mrkStrokeRGB keyword. The marker fill color continues to be set with the rgb=(*r*,*g*,*b*) or zColor={…} parameters.

Applies only to the nontext and nonarrow marker modes.

If *u*=0 then the marker stroked line colors are set with the rgb=(*r*,*g*,*b*) or zColor={…} parameters, just like the marker fill color.

useNegPat=*u* If *u*=1, negative fills use the mode specified by the hBarNegFill keyword. Applies to the fill-to-zero, fill-to-next and histogram bar modes.

useNegRGB=*u* If *u* =1, negative fills use the color specified by the negRGB keyword. Applies to the fill-to-zero, fill-to-next and histogram bar modes.

usePlusRGB=*u* If *u* =1, positive fills use the color specified by the plusRGB keyword. Applies to the fill-to-zero, fill-to-next and histogram bar modes.

userData={*udName*, *doAppend*, *data*}

Attaches arbitrary data to a trace. You should specify a trace name (userData(<traceName>)={...}). Otherwise copies of the data will be attached to every trace, which is most likely not what you intend.

Use the GetUserData function to retrieve the data, with the trace name as the object ID.

udName: The name of your user data. Use $"" for unnamed user data.

doAppend=0: Do not append. Any pre-existing data is replaced.

doAppend=1: Append the data. Data is added to the end of any pre-existing data.

data: A string expression containing the data you wish to attach to the trace.

zColor={*zWave*,*zMin*,*zMax*,*ctName* [,*reverseMode* [,*cWave*]]} or 0

Dynamically sets color based on the values in *zWave* and color table name or mode specified by *ctName*.

zWave may be a subrange expression such as `myZWave[2,9]` when *zWave* has more points than the trace, in which case `myZWave[2]` provides the Z value for the first point of the trace, and autoscaled *zMin* or *zMax* is determined over only the *zWave* subrange.

If a value in the *zWave* is NaN then a gap or missing marker will be observed. If a value is out of range it will be replaced with the nearest valid value. See also the zColorMax and zColorMin keywords.

ctName can be the name of a built-in color table such as returned by the **CTabList** function, such as Grays or Rainbow, for color table mode, `ctableRGB` for color table wave mode, `cindexRGB` for color index wave mode, or `directRGB` for direct color wave mode.

zColor for Built-in Color Table Mode

This mode uses *zWave* to select a color from a built-in color table specified by *ctName*. See **Image Color Tables** on page II-305 for details.

zWave contains values that are used to select a color from the built-in color table specified by *ctName*.

zMin is the *zWave* value that maps to the first entry in the color table. Use * for *zMin* to autoscale it to the smallest value in zWave.

zMax is the *zWave* value that maps to the last entry in the color table. Use * for *zMax* to autoscale it to the largest value in zWave.

ctName is the name of a built-in color table such as Grays or Rainbow. See the **CTabList** function for a list of built-in color tables.

Set *reverseMode* to 1 to reverse the color table lookup or to 0 to use the normal lookup. If you omit *reverseMode* or specify -1, the reverse mode is unchanged.

cWave must be omitted.

Normally the colors from the color table are linearly distributed between *zMin* and *zMax*. Use `logZColor=1` to distribute them logarithmically.

```
// Example zColor command using built-in color table
ModifyGraph zColor(data)={zWave,*,*,Rainbow}
```

zColor for Color Table Wave Mode

This mode is like Built-in Color Table except that the colors are stored in a color table wave that you have created. A color table wavey can be a 3 column RGB wave or a 4 column RGBA wave. See **Color Table Waves** on page II-311 for details.

zWave contains values that are used to select a color from the color table wave specified by *cWave*.

zMin is the *zWave* value that maps to the first entry in the color table wave. Use * for *zMin* to autoscale it to the smallest value in zWave.

zMax is the *zWave* value that maps to the last entry in the color table wave. Use * for *zMax* to autoscale it to the largest value in zWave.

ctName is `ctableRGB`.

Set *reverseMode* to 1 to reverse the color table lookup or to 0 to use the normal lookup. If you omit *reverseMode* or specify -1, the reverse mode is unchanged.

cWave is a reference to your color table wave.

Normally the colors from the color table are linearly distributed between *zMin* and *zMax*. Use `logZColor=1` to distribute them logarithmically.

Example `ctableRGB zColor` command:

```
ColorTab2Wave Rainbow   // Creates M_Colors wave
Rename M_Colors, MyColorTableWave
```

```
ModifyGraph zColor(data)={zWave,*,*,ctableRGB,0,MyColorTableWave}
```

zColor for Color Index Wave Mode

This mode is like Color Table Wave except that the values in *zWave* represent X indices with respect to *cWave*. You must create the RGB or RGBA color index wave and set its X scaling appropriately. See **Color Index Wave** on page II-285 for details.

zWave contains values that are used to select a color from the color index wave specified by *cWave*.

zMin and *zMax* are not used and should be set to `*`.

ctName is `cindexRGB`.

Set *reverseMode* to 1 to reverse the color table lookup or to 0 to use the normal lookup. If you omit *reverseMode* or specify -1, the reverse mode is unchanged. Normally the zWave values select the color from the row of *cWave* whose X value is closest to the zWave value. *reverseMode*=1 reverses the colors.

cWave is a reference to your color index wave.

Normally the colors from the color index wave are linearly distributed between the minimum and maximum X values of the color index wave. Use `logZColor`=1 to distribute them logarithmically.

```
// Example cindexRGB zColor command
zColor(data)={myZWave,*,*,cindexRGB,0,M_colors}
// M_colors is generated by ColorTab2Wave
```

zColor for Direct Color Wave Mode

In direct color mode, *zWave* is an RGB or RGBA wave that directly specifies the color for each point in the trace. If *zWave* is 8-bit unsigned integer, then color component values range from 0 to 255. For other numeric types, color component values range from 0 to 65535. See **ColorTab2Wave**, which generates RGB waves, and **Direct Color Details** on page II-313.

zWave is an RGB or RGBA wave that directly specifies the color for each point of the trace.

zMin and *zMax* are not used and should be set to `*`.

ctName is `directRGB`.

reverseMode is not applicable and should be omitted or set to 0.

cWave must be omitted.

```
// Example directRGB zColor command
zColor(data)={zWaveRGB,*,*,directRGB}
```

Turning zColor Off

zColor = 0 turns the zColor modes off.

zColorMax=(*red*, *green*, *blue*)

Sets the color of the trace for zColor={*zWave*, …} values greater than the zColor's *zMax*. Also turns on zColorMax mode.

The *red*, *green*, and *blue* color values are in the range of 0 to 65535.

zColorMax=1, 0, or NaN

Turns zColorMax mode off, on, or transparent. These modes affect the color of zColor={*zWave*, …} values greater than the zColor's *zMax*.

1:	Turns on zColorMax mode. The color of the affected trace pixels is black or the last color set by zColorMax=(*red*, *green*, *blue*).
0:	Turns off zColorMax mode (default). The color of the affected trace pixels is the last color in the zColor's *ctname* color table.
NaN:	Transparent zColorMax mode. Affected trace pixels are not drawn.

zColorMin=(*red*, *green*, *blue*)

Sets the color of the trace for zColor={*zWave*, …} values less than the zColor's *zMin*. Also turns zColorMin mode on.

The *red*, *green*, and *blue* color values are in the range of 0 to 65535.

zColorMin=1, 0, or NaN

Turns zColorMin mode off, on, or transparent. These modes affect the color of zColor={*zWave*, …} values less than the zColor's *zMin*.

1:	Turns on zColorMin mode. The color of the affected image pixels is black or the last color set by zColorMin=(*red*, *green*, *blue*).
0:	Turns off zColorMin mode (default). The color of the affected trace pixels is the first color in the zColor's *ctname* color table.
NaN:	Transparent zColorMin mode. Affected trace pixels are not drawn.

zmrkNum={*zWave*} or 0

Dynamically sets the marker number for each point to the corresponding value in *zWave*. The values in *zWave* are the marker numbers (as used with the marker keyword). If a value in the *zWave* is NaN then no marker will be drawn at the corresponding point. If a value is out of range it will be replaced with the nearest valid value.

zmrkNum=0 turns this mode off.

zmrkSize={*zWave*,*zMin*,*zMax*,*mrkmin*,*mrkmax*} or 0

Dynamically sets marker size based on values in *zWave*. Use * or a missing parameter for *zMin* and *zMax* to autoscale. *mrkmin* and *mrkmax* can be fractional. If a value in the *zWave* is NaN then a gap or missing mark will be observed. The marker size is clipped to 20 on the high end and 1 on the low end. If a value is out of range it will be replaced with the nearest valid value.

zmrkSize = 0 turns this mode off.

zpatNum={*zWave*} or 0

Dynamically sets the positive fill type/pattern number for each point to the corresponding value in *zWave*. The values in *zWave* are the pattern numbers (as used with the hbFill keyword). If a value in the *zWave* is NaN then the corresponding point will not be drawn. If a value is out of range it will be replaced with the nearest valid value.

zpatNum=0 turns this mode off.

Flags

/W=*winName*	Modifies the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the Command Line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/Z	Does not generate an error if the indexed trace, named wave, or named axis does not exist in a style macro.

Details

Live Mode (live=1) improves graph update performance when one or more of the waves displayed in the graph is frequently modified, for example, if the waves are being acquired from a data acquisition system. Live Mode traces do not autoscale the axes.

Waves supplied with zmrkSize, zmrkNum, and zColor may use **Subrange Display Syntax** on page II-250.

Quickdrag

Quick drag mode (quickdrag=1) is a special purpose mode for creating cross hair cursors using a package of Igor procedures. (See the Cross Hair Demo example experiment.) Normally you would have to click and hold on a trace for one second before entering drag mode. When quickdrag is in effect, there is no delay. If a trace is in quickdrag mode it should also be set to live mode. With this combination you can click a trace and immediately drag it to a new XY offset. In addition to quick drag mode, the cross hair package relies on Igor to store information about the drag in a string variable if certain conditions are in effect. The string variable name (that you have to create) is S_TraceOffsetInfo, which must reside in a data folder that has the same name as the graph (not title!) which in turn must reside in root:WinGlobals:. If these conditions are met, then after a trace is dragged, information will be stored in the string using the following key-value format: `GRAPH:<name of graph>;XOFFSET:<x offset value>;YOFFSET:<y offset value>;TNAME:<trace name>;`

Customize at Point

You can customize the appearance of individual points on a trace in a graph for bar, marker, dot and lines to zero modes using `key(tracename[pnt])=value` syntax. The point number must be a literal number and the trace name is not optional. To turn off a customization, use `key(tracename[-pnt-1])=value` where value is not important but must match the syntax for the keyword. The offset of -1 is needed because point numbers start from zero.

Although the syntax is allowed for all trace modifiers, it has meaning only for the following: rgb, marker, msize, mrkThick, opaque, mrkStrokeRGB, barStrokeRGB, hbFill, patBkgColor and lSize.

Note that useBarStrokeRGB and useMrkStrokeRGB are not needed. The act of using barStrokeRGB or mrkStrokeRGB is enough to customize the point. But as a convenience, since these are generated by the modify graph dialog, they are ignored if used with [pnt] syntax.

Also note that legend symbols can use [pnt] syntax like so:

```
\s(<tracename>[pnt])
```

Automatically generated legends automatically include symbols for customized points.

For example:

```
Make/O/N=10 jack=sin(x); Display jack
ModifyGraph mode=5,hbFill=6,rgb=(0,0,0)
ModifyGraph hbFill(jack[2])=7,rgb(jack[2])=(0,65535,0)
ModifyGraph rgb(jack[3])=(65535,0,0)
Legend/C/N=text1/F=0/A=MC
```

Examples

Arrow markers.

```
Make/N=10 wave1= x; Display wave1
Make/N=(10,2) awave
awave[][0]= p*5                 // length
awave[][1]= pi*p/9              // angle
ModifyGraph mode=3,arrowMarker(wave1)={awave,1,10,0.3,0}

// Now add an optional column to control headLen
Redimension/N=(-1,3) awave
awave[][2]= 7+p                 // will be head length

// Note: nothing changes until the following is executed
SetDimLabel 1,2,headLen,awave
```

Create meteorological wind barb symbols.

```
Make/O/N=50 jack= floor(x/10),jackx= mod(x,10)
Display jack vs jackx
Make/O/N=(50,3) jackbarb
jackbarb[][0]= 40               // length of stem
jackbarb[][1]= 45*pi/180        // angle (45deg)
jackbarb[][2]= p                // wind speed code
SetDimLabel 1,2,windBarb,jackbarb
ModifyGraph mode=3,arrowMarker(jack)={jackbarb,1,10,0.5,0}
ModifyGraph margin(top)=62,margin(right)=84
```

See also **Wind Barb Plots** on page II-258.

Inline arrows and barb sharpness.

```
Make/O/N=20 wavex=cos(x/3),wavey=sin(x)
Display wavey vs wavex
ModifyGraph mode=3,arrowMarker={_inline_,1,20,.5,0,barbSharp= 0.2}
```

Use direct color mode to individually color each point in a trace:

```
Make jack=sin(x/8)
Make/N=(128,3)/B/U jackrgb
Display jack
ModifyGraph mode=3,marker=19
jackrgb= enoise(128)+128
ModifyGraph zColor(jack)={jackrgb,*,*,directRGB}
```

Use masking.

```
Make/N=100 jack= (p&1) ? sin(x/8) : cos(x/8)
Display jack

Make/N=100 mjack= (p&1) ? 0 : NaN        // just to show NaN can be used
ModifyGraph mask(jack)={mjack,0,NaN}

// now switch which points are shown
mjack= (p&1) ? NaN : 0
```

See Also

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

ModifyGraph *(axes)*

`ModifyGraph [/W=`*`winName`*`/Z]` ***`key`*** **`[(`*`axisName`*`)] =`** ***`value`***
  `[,` ***`key`*** **`[(`*`axisName`*`)] =`** ***`value`*** **`]…`**

This section of ModifyGraph relates to modifying the appearance of axes in a graph.

Parameters

Each *key* parameter may take an optional *axisName* enclosed in parentheses.

axisName is "left", "right", "top", "bottom" or the name of a free axis such as "vertCrossing". For instance, "`ModifyGraph axThick(left)=0.5`" sets the axis thickness for only the left axis.

If "(*axisName*)" is omitted, all axes in the graph are affected. For instance, "`ModifyGraph standoff=0`" disables axis standoff for all axes in the graph.

The parameter descriptions below omit the optional "(*axisName*)".

axisClip= *c* Specifies one of three clipping modes for traces.

c=0:	Clips traces to a plot rectangle as defined by the pair of axes used by a given trace (default).
c=1:	Plots traces on an axis with a restricted range (as set by axisEnab) to extend to the full range of the normal plot rectangle.
c=2:	Traces extend outside the normal plot rectangle to the full extent of the graph area.

axisEnab={*lowFrac*,*highFrac*}

Restricts the length of an axis to a subrange of normal. The axis is drawn from *lowFrac* to *highFrac* of graph area height (vertical axis) or width (horizontal axis). For instance, {0.1,0.75} specifies that the axis is drawn from 10% to 75% of the graph area height/width, instead of the normal 0% to 100%. AxisEnab is discussed in **Creating Split Axes** on page II-259 and **Creating Stacked Plots** on page II-253.

axisOnTop=*t* Specifies drawing level of axis and associated grid lines.

t=0:	Draws axis before traces and images (default).
t=1:	Draws the axis after all traces and images.

axOffset=*a*	Specifies the distance from default axis position to actual axis position in units of the width of a zero character (0) in a tick mark label. Unlike margin, axOffset adjusts to changes in the size of the graph.
axThick=*t*	Specifies the axis thickness in points.
barGap=*fraction*	Sets the fraction of the width available for bars to be used as gap between bars. barGap sets the gap between bars within a single category while catGap sets the gap between categories.
btLen=*p*	Sets the length of major ("big") tick marks to *p* points. If *p* is zero, it uses the default length. *p* may be fractional.
btThick=*p*	Sets the thickness of major ("big") tick marks to *p* points. If *p* is zero, it uses the default thickness. *p* may be fractional.

catGap=*fraction* The value for catGap is the fraction of the category width to be used as gap. The gap is divided equally between the start and end of the category width. A value of 0.2 would use 20% of the available space for the gap and leave 80% of the available space for the bars.

catGap sets the gap between categories while barGap sets the gap between bars within a single category.

dateFormat={*languageName*, *yearFormat*, *monthFormat*, *dayOfMonthFormat*, *dayOfWeekFormat*, *layoutStr*, *commonFormat*}

Sets the custom date format used in the active graph.

Note: Use a custom date format only if you turn it on via a ModifyGraph dateInfo command. The last parameter to the ModifyGraph dateInfo command must be -1 to turn on the custom date format.

Parameters are the same as for the LoadWave/R flag except for the last one.

commonFormat selects the common date format to use in the Modify Axis dialog. The legal values correspond to the choices in the Common Format pop-up menu of the Modify Axis dialog. They are:

Value	Date Format	Value	Date Format
1	mm/dd/yy	16	mm/yy
2	mm-dd-yy	17	mm.yy
3	mm.dd.yy	18	Abbreviated month and year
4	mmddyy	19	Full month and year
6	dd/mm/yy	21	mm/dd
7	dd-mm-yy	22	dd.mm
8	dd.mm.yy	23	Abbreviated month and day
9	ddmmyy	24	Full month and day
11	yy/mm/dd	26	Abbreviated date without day of week
12	yy-mm-dd	27	Abbreviated date with day of week
13	yy.mm.dd	28	Full date without day of week
14	yymmdd	29	Full date with day of week

If the *commonFormat* parameter is negative, then it will select the Use Custom Format radio button in the Modify Axis dialog rather than Use Common Format and will then use the absolute value of *commonFormat* to determine which item to select in the Common Format pop-up menu.

dateInfo={*sd*,*tm*,*dt*} Controls formatting of date/time axes.

sd=0:	Show date in the date&time format.
sd=1:	Suppress date.
tm=0:	12 hour (AM/PM) time.
tm=1:	24 hour (military) time.
tm=2:	Elapsed time.
dt=-1:	Custom date as specified via the dateFormat keyword.
dt=0:	Short dates (2/22/90).
dt=1:	Long dates (Thursday, February 22, 1990).
dt=2:	Abbreviated dates (Thurs, Feb 22, 1990).

font="*fontName*" Sets the axis label font, e.g., `font(left)="Helvetica"`.

freePos(*freeAxName*)=*p*

Sets the position of the *free* axis relative to the edge of the plot area to which the axis is anchored. *p* is in points. i.e., if the axis was made via /R=*axName* then the axis is placed *p* points from the right edge of the plot area. Positive is away from the central plot area. *freeAxName* may not be any of the standard axes: “left”, “bottom”, “right” or “top”.

freePos(*freeAxName*)={*crossAxVal*,*crossAxName*}

Positions the *free* axis so it will cross the perpendicular axis *crossAxName* where it has a value of *crossAxVal*. *freeAxName* may not be any of the standard axis names “left”, “bottom”, “right”, or “top”, though *crossAxName* may.

You can position a free axis as a fraction of the distance across the plot area by using `kwFraction` for *crossAxName*. *crossAxVal* must then be between 0 and 1; any values outside this range are clipped to valid values.

fsize=*s* — Autosizes (*s*=0) tick mark labels and axis labels.

If *s* is between 3 and 99 then the labels are fixed at *s* points.

fstyle=*f* — *f* is a bitwise parameter with each bit controlling one aspect of the font style for the axis and tick mark labels as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

ftLen=*p* — Sets the length of 5th (or emphasized minor) tick marks to *p* points. If *p* is zero, it uses the default length. *p* may be fractional.

ftThick=*p* — Sets the thickness of 5th (or emphasized minor) tick marks to *p* points (fractional). If *p* is zero, it uses the default thickness.

grid=*g* — Controls grid lines.

g=0:	Grid off.
g=1:	Grid on.
g=2:	Grid on major ticks only.

gridEnab={*lowFrac*,*highFrac*}

Restricts the length of axis grid lines to a subrange of normal. The grid is drawn from *lowFrac* to *highFrac* of graph area height (if axis is horizontal) or width (if axis is vertical).

gridHair=*h* — Sets the grid hairline thickness (*h* =0 to 3; 0 for thicker lines, 3 for thinner; default is 2). If *h*=0, the thickness of grid lines on major tick marks is the same as the axis thickness, half for a minor tick and one tenth for a subminor tick (log axis only). As *h* increases these thicknesses decrease by a factor of 2^*h*. If you want to see the effect of different values of gridHair, you will need to print a sample graph because you generally can’t see the effect of thin lines on the screen. Also see the example experiment “Examples:Graphing Techniques:Graph Grid Demo”.

gridStyle=*g*	Sets the grid style to various combinations of solid and dashed lines. In the following discussion, major, minor and subminor refer to grid lines the corresponding tick marks. Subminor ticks are used only on log axes when there is a small range and sufficient room (they correspond to hundredths of a decade). The different grid styes are solid, dotted, dashed, and blank. The possible grids are as follows:

g=0:	Same as mode 1 if graph background is white else uses mode 5.
g=1:	Major dotted, minor and subminor dashed.
g=2:	All dotted.
g=3:	Major solid, minor dotted, subminor blank.
g=4:	Major and minor solid, subminor dotted.
g=5:	All solid.

Also see the example experiment “Examples:Graphing Techniques:Graph Grid Demo”.

highTrip=*h*	If the extrema of an axis are between its lowTrip and its highTrip then tick mark labels use fixed point notation. Otherwise they use exponential (scientific or engineering) notation.
lblLatPos=*p*	Sets a lateral offset for the axis label. This is an offset parallel to the corresponding axis. *p* is in points. Positive is down for vertical axes and to the right for horizontal axes.
lblMargin=*l*	Specifies the distance from the edge of graph to a label in points.
lblPos=*p*	Sets the distance from an axis to the corresponding axis label in points. If *p*=0, it automatically picks an appropriate distance. This setting is used only if the given graph edge has at least one free axis. Otherwise, the lblMargin setting is used to position the axis label.
lblPosMode= *m*	Affects the meaning and usage of lblPos, lblLatPos, and lblMargin parameters. Mainly for use when you have multiple axes on a side and you need axis labels to be properly positioned even as you make graph windows dramatically larger or smaller.

m=0:	Default compatibility mode (Margin or Axis absolute depending on presence of free axis).
m=1:	Margin absolute.
m=2:	Margin scaled.
m=3:	Axis absolute.
m=4:	Axis scaled.

The absolute modes are measured in points whereas scaled modes have similar values but automatically expand or contract as the axis font height changes. Mode 0 is the default and results in no change relative to previous versions of Igor Pro that used lblMargin unless a given side used a free axis in which case it used lblPos in absolute mode. The margin modes measure relative to an edge of the graph while the axis modes measure relative to the position of the axis. When using stacked axes, use either margin modes. With multiple nonstacked axes, use Axis scaled if the graph edge is not using a fixed margin or use axis absolute if it is.

lblRot=*r*	Rotates the axis label by *r* degrees. *r* is a value from -360 to 360. Rotation is counterclockwise and starts from the label's normal orientation.
linTkLabel=*tl*	*tl*=1 attaches the data units with any exponent or prefix to each tick label on a normal axis. *tl*=0 removes them.

log=*l*	Controls axis log mode. *g*=0: Normal axis. *g*=1: Log base 10. *g*=2: Log base 2.
logHTrip=*h*	Same as highTrip but for log axes.
logLabel=*l*	Sets the maximum number of decades in a log axis before minor tick labels are suppressed.
logLTrip=*l*	Same as lowTrip but for log axes.
loglinear=*l*	Switches to a linear tick method (*l*=1) on a log axis if the number of decades of ranges is less than 2. It switches to a linear tick exponent method if the number of decades is greater than five.
logTicks=*t*	Sets the maximum number of decades in log axis before minor ticks are suppressed.
lowTrip=*l*	If the extrema of an axis are between its lowTrip and its highTrip then tick mark labels use fixed point notation. Otherwise they use exponential (scientific or engineering) notation.

manminor={*number*, *emphasizeEvery*}

Specifies how to draw minor ticks in manual tick mode. There will be *number* ticks between each major (labeled) tick. You will usually want to set this to 4 to make 5 divisions, or 9 to make 10 divisions. A medium-sized tick (an emphasized minor tick) will be drawn every *emphasizeEvery* minor tick.

manTick={*cantick*, *tickinc*, *exp*, *digitsrt* [, *timeUnit*]}

Turns on manual tick mode. The tick from which all other ticks are calculated is the cononic tick (*cantick*). The numerical spacing between ticks is set by *tickinc*. *cantick* and *tickinc* are multiplied by 10*exp*. The number of digits to the right of the decimal point displayed in the tick labels is set by *digitsrt*.

The optional parameter *timeUnit* is used with Date/Time axes to specify the units of tickinc. In this case, tickinc must be an integer. The value of *timeUnit* is one of the following keywords:

```
second, minute, hour, day, week, month, year
```

On a date/time axis, the *exp* and *digitsrt* keywords are ignored, but must be present. You can set them to zero.

manTick=0	Turns off manual tick mode.
margin=*m*	Sets a fixed margin from the edge of the window to the axis in points. Used principally to make axes of multiple graphs on a page line up when “stacked”. You can use the left, right, bottom, and top axis names (even if an axis with that name doesn’t exist) to adjust the graph plot area. See **Types of Axes** on page II-215. *m*=0: Sets “automatic” margin size (dependent on the length and height of tick marks and labels). *m*=-1: Sets the margin to “none”, or 0. The axis is drawn at the graph window’s edge.
minor=*m*	Disables (*m*=0) or enables (*m*=1) minor ticks.

mirror=*m*	Controls axis mirroring. *m*=1: Right axis mirroring left or top mirroring bottom. *m*=2: Mirror axis without tick marks. *m*=3: Mirror axis with tick marks and tick labels. *m*=0: No mirroring.
mirrorPos=*pos*	Specifies the position of the mirror axis relative to the normal position. *pos* is a value between 0 and 1.
noLabel=*n*	Controls axis labeling. *n*=0: Normal labels. *n*=1: Suppresses tick mark labels. *n*=2: Suppresses tick mark labels and axis labels.
notation=*n*	Uses engineering (*n*=0) or scientific (*n*=1) notation for tick mark labels. Affects tick mark labels displayed exponentially. See highTrip and lowTrip. Does not affect log axes.
nticks=*n*	Specifies the approximate number of ticks marks (*n*) on axis.
prescaleExp=*exp*	Multiplies axis range by 10^*exp* for tick labeling and *exp* is subtracted from the axis label exponent. In other words, the exponent is moved from the tick labels to the axis label. (This affects the display only, not the source data.)
sep=*s*	Specifies the minimum number of screen points (s) between minor ticks.
standoff=*s*	Suppresses (*s*=0) or enables (*s*=1) axis standoff. Axis standoff prevents waves or markers from covering the axis.
stLen=*p*	Sets the length of minor (“small”) tick marks to *p* points. If *p* is zero, it uses the default length. *p* may be fractional.
stThick=*p*	Sets the thickness of minor (“small”) tick marks to *p* points. If *p* is zero, it uses the default thickness. *p* may be fractional.
tick=*t*	Sets tick position. *t*=0: Outside axis. *t*=1: Crossing axis. *t*=2: Inside axis. *t*=3: None. In a category plot, adding 4 to the usual values for the tick keyword will place the tick marks in the center of each category rather than at the edges.
tickEnab={*lowTick*,*highTick*}	Restricts axis ticking to a subrange of normal. Ticks are drawn and labelled only if they fall within this inclusive numerical range.
tickExp=*te*	*te*=1 forces tick labels to exponential notation when labels have units with a prefix. *te*=0 turns this off.
tickUnit=*tu*	Suppresses (*tu* =1) or turns on (*tu* =0) units labels attached to tick marks.
tickZap={[*v1* [,*v2* [,*v3*]]]}	

Suppresses drawing of the tick mark label for values given in the list. This is useful when you have crossing axes to prevent tick mark labels from overlapping. The list may contain zero, one, two or three values. The values must be exact to suppress the label.

tkLblRot=*r*	Rotates the tick mark labels by *r* degrees. *r* is a value from -360 to 360. Rotation is counterclockwise and starts from the label's normal orientation.
tlOffset=*o*	Offsets the tick mark labels by *o* fractional points relative to the default tick mark label position. Positive is away from the axis.
ttLen=*p*	Sets the length of subminor ("tiny") tick marks to *p* points. If *p* is zero, it uses the default length. *p* may be fractional. Subminor ticks are used only in log axes.
ttThick=*p*	Sets the thickness of subminor ("tiny") tick marks to *p* points. If *p* is zero, it uses the default thickness. *p* may be fractional.

userticks={*tickPosWave*, *tickLabelWave*}

Draws axes with purely user-defined tick mark positions and labels. *tickPosWave* is a numeric wave containing the desired positions of the tick marks, and *tickLabelWave* is a text wave containing the labels. See **User Ticks from Waves** on page II-241 for an example.

The tick mark labels can be multiline and use styled text. For more details, see **Fancy Tick Mark Labels** on page II-270.

tickPosWave need not be monotonic. Igor will plot a tick if a value is in the range of the axis. Both linear and log axes are supported.

Graph margins will adjust to accommodate tick labels. This will not prevent overlap between labels, which you will need fix yourself.

useTSep=*t*	*t*=1 displays a thousand's separator character between every group of three digits in the tick mark label (e.g., "1,000" instead of "1000"). The default is *t*=0.
zapLZ=*t*	Removes (*t*=1) leading zeros from tick mark labels. For example 0.5 becomes .5 and -0.5 becomes -.5. Default is *t*=0.
zapTZ=*t*	Removes (*t*=1) trailing zeros from tick mark labels. The the radix point will also be removed if all digits are zero. Default is *t*=0.
zero=*z*	Controls the zero line.

z=0:	A zero line at x=0 or y=0.
	The line style is set to *z*-1. See **ModifyGraph (traces)** on page V-522, lStyle keyword, for details on line styles.
z=1:	No zero line.

zeroThick=*zt*	Sets the thickness of the zero line in points, from 0.0 to 5.0 points. *zt*=0.0 means the zero line thickness automatically follows the thickness of the axis; this is the default. You can use 0.1 for a thin zero line thickness.
ZisZ=*t*	*t*=1 uses the single digit 0 as the zero tick mark label (if any) regardless of the number of digits used for other labels. Default is *t*=0.

Flags

/W=*winName*	Modifies the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/Z	Does not generate an error if the named axis does not exist in a style macro.

Details

With the prescaleExp parameter, you can force tick and axis label scaling to values different from the defaults. For example, if you have data whose X scaling ranges from 9pA to 120pA and you display this on a log axis, the tick marks will be labelled 10pA and 100pA. But if you really want the tick marks labeled 10 and 100 with pA in the axis label, you can set the prescaleExp to 12. To see this, execute the following commands:

```
Make/O jack=x
Display jack
SetScale x,9e-12,120e-12,"A",jack
ModifyGraph log(bottom)=1
```

then execute:

```
ModifyGraph prescaleExp(bottom)=12
```

The tickExp parameter applies to units that do not traditionally use SI prefix characters. For example, one usually speaks of 10^{-3} Torr and not mTorr. To see how this feature works, execute the following example commands:

```
Make/O jack=x
Display jack
SetScale x,1E-7,1E-5,"Torr",jack
ModifyGraph log(bottom)=1
```

then execute:

```
ModifyGraph tickExp(bottom)=1
```

at this point, the tick mark labels have Torr in them. If you want to eliminate the units from the tick marks, execute:

```
ModifyGraph tickUnit(bottom)=1
```

and if you now want Torr in the label string, use the \U escape in the label string:

```
Label bottom "\\U"
```

To see the effect of linTkLabel, execute these commands:

```
Make/O jack=x
Display jack
SetScale x,1E-7,1E-5,"Torr",jack
```

then execute:

```
ModifyGraph linTkLabel(bottom)=1
```

and then try:

```
ModifyGraph tickExp(bottom)=1
```

and finally:

```
ModifyGraph tickUnit(bottom)=1
```

ModifyGraph *(colors)*

ModifyGraph [/W=winName/Z] ***key*** **[(*axisName*)] = (*r*,*g*,*b*)**
[, *key* [(*axisName*)] = (*r*,*g*,*b*)]…

This section of ModifyGraph relates to modifying the use of colors in a graph.

Parameters

Most (but not all) of the *key* parameters may take an optional *axisName* enclosed in parentheses. *axisName* is “left”, “right”, “top”, “bottom” or the name of an free axis such as “vertCrossing”.

Where the parameter descriptions indicate an “(*axisName*)”, it may be omitted to change all axes in the graph.

r, g, and *b* are each an integer from 0 to 65535 where (0, 0, 0) is black and (65535, 65535, 65535) is white.

Parameter Specification	Object Colored
`alblRGB(`*`axisName`*`)=(`*`r,g,b`*`)`	Axis labels
`axRGB(`*`axisName`*`)=(`*`r,g,b`*`)`	Axis
`cbRGB=(`*`r,g,b`*`)`	Control bar background
`gbRGB=(`*`r,g,b`*`)`	Graph background
`gbGradient=<`*`parameters`*`>`	Controls color gradients for graph background. See **Gradient Fills** on page III-441 for details.
`gbGradientExtra=<`*`parameters`*`>`	Controls color gradient details for graph background. See **Gradient Fills** on page III-441 for details.
`gridRGB(`*`axisName`*`)=(`*`r,g,b`*`)`	Axis grid lines
`tickRGB(axisName )=(`*`r,g,b`*`)`	Axis Tick marks
`tlblRGB(`*`axisName`*`)=(`*`r,g,b`*`)`	Axis Tick labels
`wbRGB=(`*`r,g,b`*`)`	Window background
`wbGradient=<`*`parameters`*`>`	Controls color gradients for window background. See **Gradient Fills** on page III-441 for details.
`wbGradientExtra=<`*`parameters`*`>`	Controls color gradient details for window background. See **Gradient Fills** on page III-441 for details.

Flags

/W=*winName*	Modifies the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/Z	Does not generate an error if the named axis does not exist in a style macro.

Details

On Windows, use maximum white to set the control bar background color to track the 3D Objects color in the Appearance Tab of the Display Properties control panel:

```
ModifyGraph cbRGB=(65535,65535,65535)
```

See Also

See **Instance Notation** on page IV-19.

ModifyImage

`ModifyImage [/W=`*`winName`*`] `*`imageInstance, keyword = value`*
`    [, `*`keyword = value`*`]…`

The ModifyImage operation changes properties of the given image in the top graph (or the specified graph if /W is used). *imageInstance* is the name of the image to be altered. This name is usually simply the name of the matrix wave containing the image data. If the same matrix wave is displayed more than once, you must append #0, #1 etc. to the name to distinguish which is which.

imageInstance can also take the form of a null name with an instance number to affect the instanceth image. That is,

```
ModifyImage ''#1
```

modifies the appearance of the second image that was appended to the top graph, no matter what the image names are. Note: two single quotes are used, not a double quote.

Parameters

Here are the keyword-value pairs. These apply to false color images in which the data in the matrix is used as an index into a color table. They do not apply to direct color images in which the data in the matrix specifies the color directly.

cindex=*matrixWave* Sets the Z value mapping mode such that image colors are determined by doing a lookup in the specified matrix wave.

matrixWave is a 3 column wave that contains red, green, and blue values from 0 to 65535. (The matrix can actually have more than three columns. It ignores any extra columns.)

The color at Z=z is determined by finding the RGB values in the row of *matrixWave* whose scaled X index is z. In other words, the red value is *matrixWave*(z)[0], the green value is *matrixWave*(z)[1] and the blue value is *matrixWave*(z)[2].

If *matrixWave* has default X scaling, where the scaled X index equals the point number, then row 0 contains the color for Z=0, row 1 contains the color for Z=1, etc.

If you use cindex, you should not use ctab in the same command.

ctab={*zMin*, *zMax*, *ctName*, *reverse*}

Sets the z mapping mode by which values in the matrix are mapped linearly into the color table specified by *ctName*.

zMin and *zMax* set the range of z values to map. Omit *zMin* or *zMax* to leave as is or use * to autoscale.

The color table name can be missing if you want to leave it as is.

ctName can be any color table name returned by the CTabList function, such as Grays or Rainbow (see **Image Color Tables** on page II-305) or the name of a 3 column or 4 column color table wave (**Color Table Waves** on page II-311).

A color table wave name supplied to `ctab` must not be the name of a built-in color table (see **CTabList**). A 3 column or 4 column color table wave must have values that range between 0 and 65535. Column 0 is red, 1 is green, and 2 is blue. In column 3 a value of 65535 is opaque, and 0 is fully transparent.

Set *reverse* to 1 to reverse the color table. Setting it to 0 or omitting it leaves the color table unreversed.

ctabAutoscale=*autoBits*

Sets the range of data used for autoscaling ctab * values.

Bit 0:	Autoscales only the XY subset being displayed.
Bit 1:	Autoscales only the current plane being displayed.

If neither bit is set (if *autoBits* = 0, the default), then all of the data in the image wave is used to autoscale the *'d *zMin*, *zMax* values for ctab.

eval={*value*, *red*, *green*, *blue*, [*alpha*]}

If the red, green, and blue values are in the valid range for a color value (0 to 65535) the explicit value-color pair is added (or updated if value already exists). If the color values are out of range (-1 is suggested) then the value is removed from the list if it is present (no error if it is not).

alpha is optional: a value of 65535 is opaque, and 0 is fully transparent.

explicit=1 *or* 0

Turns explicit (monochrome) mode on (1) or off (0). Meant to be used with unsigned byte data but will do the best it can for other types. If value of data is equal to one of the defined explicit values then its defined color is used otherwise the pixel will be blank. The default predefined values are:

255:	black
0:	white

You can add, change, or delete explicit values with the eval keyword.

imCmplxMode=*m*

Sets complex data display mode.

m=0:	Magnitude (default).
m=1:	Real only.
m=2:	Imaginary only.
m=3:	Phase in radians.

interpolate= *mode*

mode = 1 turns on smoothing of the boundaries between pixels. Since this is implemented via system graphics calls and not by Igor actually doing the interpolation, it will not affect EPS or EMF export on Windows and will not affect EPS export on Mac. Although this may create a more esthetically pleasing display, it is not clear that it is appropriate for scientific data.

mode = -1 forces pixels to be drawn as individual rectangles. This is sometimes needed when a third-party program improperly interpolates PDF or EPS exported images.

log= 1 or 0

0 sets the default linearly-spaced false-image colors.

1 turns on logarithmically-spaced false-image colors. This requires that the image values be greater than 0 to display correctly.

Affects the image colors for color table and color index images only (see **Color Table Details** on page II-308 and **Indexed Color Details** on page II-312).

lookup= *waveName*

Specifies an optional 1D wave that can be used to modify the mapping of scaled z values into the color table specified with the ctab parameter. Values should range from 0.0 to 1.0. A linear ramp from 0 to 1 would have no effect while a ramp from 1 to 0 would reverse the image. Used to apply gamma correction to grayscale images or for special effects. Use a NULL wave (`$""`) to remove the option.

maxRGB=(*red*, *green*, *blue*, [*alpha*])

Sets the color of image values greater than the ctab *zMax* or greater than the cindex of the *matrixWave* maximum X scaling value. Also turns max color mode on.

The *red*, *green,* and *blue* color values are in the range of 0 to 65535.

alpha is optional: a value of 65535 is opaque, and 0 is fully transparent.

maxRGB=1 *or* 0 *or* NaN

Turns max color mode off, on, or transparent. These modes affect the display of image values greater than the ctab *zMax* or greater than the cindex of the *matrixWave* maximum X scaling value.

1:	Turns on max color mode. The color of the affected image pixels is black or the last color set by maxRGB=(*red*, *green*, *blue*).
0:	Turns off max color mode (default). The color of the affected image pixels is the last color table or color index color.
NaN:	Transparent max color mode. The affected image pixels are not drawn.

minRGB=(*red*, *green*, *blue*, [*alpha*])

Sets the color of image values less than the ctab *zMin* or less than the cindex of the *matrixWave* minimum X scaling value. Also turns min color mode on.

The *red*, *green,* and *blue* color values are in the range of 0 to 65535.

alpha is optional: a value of 65535 is opaque, and 0 is fully transparent.

minRGB=1 *or* 0 *or* NaN

Turns min color mode off, on, or transparent. These modes affect the display of image values less than the ctab *zMin* or less than the cindex of the *matrixWave* minimum X scaling value.

1:	Turns on min color mode. The color of the affected image pixels is black or the last color set by minRGB=(*red*, *green*, *blue*).
0:	Turns off min color mode (default). The color of the affected image pixels is the first color table or color index color.
NaN:	Transparent min color mode. The affected image pixels are not drawn.

plane=*p* Determines which part of a 3D or 4D image wave to display.

The meaning of *p* depends on the nature of the image wave. If the size of the layer dimension of the image wave is exactly three then the wave is treated as RGB data with R, G, and B data in the three layers. If the size of the layer dimension is exactly four, then the wave is treated as RGBA data, with A in the fourth layer. Otherwise each layer of the wave is treated as a separate grayscale image.

Plane=p With RGB Data

If the wave is 3D, plane=*p* has no effect.

If the wave is 4D, each chunk contains a different set of R, G and B layers and *p* selects which chunk to display.

Plane=p With Grayscale Data

If the wave is 3D, *p* selects which layer to display.

If the wave is 4D, plane=*p* acts as if all of the chunks were combined into a virtual 3D wave and *p* selects which layer of this virtual 3D wave to display.

rgbMult=*m* If *m* is non-zero, direct color values (3 plane RGB) are multiplied by *m*. This would typically be used for 10, 12 or 14 bit integers in a 16 bit word. For example, if your image data is 14 bits, use rgbMult=4.

Flags

/W=*winName* Directs action to a specific window or subwindow rather than the top graph window. When omitted, action will affect the active window or subwindow.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

See Also

AppendImage and **RemoveImage**.

ModifyLayout

ModifyLayout [*flags*] *key* [(*objectName*)] =*value* [, *key* [(*objectName*)] =*value*]…

The ModifyLayout operation modifies objects in the top layout or in the layout specified by the /W flag.

Parameters

Each *key* parameter may take an optional *objectName* enclosed in parentheses. If "(*objectName*)" is omitted, all objects in the layout are affected.

Though not shown in the syntax, the optional "(*objectName*)" may be replaced with "[*objectIndex*]", where *objectIndex* is zero or a positive integer denoting the object to be modified. "[0]" denotes the first object appended to the layout, "[1]" denotes the second object, etc. This syntax is used for style macros, in conjunction with the /Z flag.

The parameter descriptions below omit the optional "(*objectName*)".

The "units", "mag" and "bgRGB" keywords apply to the layout as a whole, not to a specific object and do not accept an *objectName*.

bgRGB=(*r*,*g*,*b*)	Specifies the background color for the layout. *r*, *g,* and *b* are integers from 0 to 65535.
columns=*c*	Specifies the number of columns for a table object.
fidelity=*f*	Controls the drawing of layout objects. *f*=0: Low fidelity. *f*=1: High fidelity.
frame=*f*	Specifies the type of frame enclosing the object. *f*=0: No frame. *f*=1: Single frame (default). *f*=2: Double frame. *f*=3: Triple frame. *f*=4: Shadow frame.
gradient=<*parameters*>	Controls color gradients for layout pages. See **Gradient Fills** on page III-441 for details.
gradientExtra=<*parameters*>	Controls color gradient details for layout pages. See **Gradient Fills** on page III-441 for details.
height=*h*	Sets the height of the object.
left=*l*	*l* is the horizontal coordinate of the left edge of the object relative to the left edge of the paper.
mag=*m*	Sets the layout magnification where *m*=0.25, 0.5, 1, or 2.
rows=*r*	Specifies the number of rows for table object.
top=*t*	*t* is the vertical coordinate of the top edge of the object relative to the top edge of the paper.
trans=*t*	Controls the transparency of the layout object: *t*=0: Opaque (default). *t*=1: Transparent. For this to be effective, the object itself must also be transparent. Annotations have their own transparent/opaque settings. Graphs are transparent only if their backgrounds are white. PICTs may have been created transparent or opaque, and Igor cannot make an opaque PICT transparent.
units=*u*	Sets dimension units in the layout info panel and in the Modify Objects dialog. *u*=0: Points. *u*=1: Inches. *u*=2: Centimeters.
width=*w*	Sets the object width.

Flags

/I	Dimensions in inches.
/M	Dimensions in centimeters.
/W=*winName*	*winName* is the name of the page layout window to be modified. If /W is omitted or if *winName* is `$""`, the top page layout is modified.
/Z	Does not generate an error if the indexed or named object does not exist in a style macro.

The /I and /M flags affect the units of the parameters for the left, top, width and height keywords only. If neither /I nor /M is present then the parameters for the left, top, width and height keywords are points.

Details

Note that the units keyword affects only the units used in the layout info panel and in the Modify Objects dialog. It has nothing to do with the units used for the left, top, width and height keywords. Those units are points unless the /I or /M flags is present.

See Also

NewLayout, **AppendLayoutObject** and **RemoveLayoutObjects**.

ModifyPanel

`ModifyPanel [/W=`*`winName`*`] `*`keyword = value`*` [, `*`keyword = value`*` …]`

The ModifyPanel operation modifies properties of the top or named control panel window or subwindow.

Parameters

keyword is one of the following:

cbRGB=(*r*,*g*,*b*)	Specifies the background color of the entire control panel or the graph's control bar area. *r*, *g*, and *b* are values from 0 to 65535.
fixedSize=*f*	Controls the resizing of the panel window. *f*=0: Panel can be resized (default). *f*=1: Panel cannot be resized by adjusting the size box or frame (nor maximized on Windows), but the window can be minimized (on Windows) and the MoveWindow operation can still change the size. The `fixedSize` keyword overrides any previous size limit set using the `SetWindow sizeLimit` command. If you try to use `SetWindow sizeLimit` on a window with `fixedSize=1`, Igor generates an error.
frameInset= *i*	Specifies the number of pixels by which to inset the frame of the panel subwindow. Mostly useful for overlaying panels in graphs to give a fake 3D frame a better appearance.
frameStyle= *f*	Specifies the frame style for a panel subwindow. *f*=0: None. *f*=1: Single. *f*=2: Indented. *f*=3: Raised. *f*=4: Text well. The last three styles are fake 3D and will look good only if the background color of the enclosing space and the panel itself is a light shade of gray.

noEdit= *e*	Sets the editability of the panel. *e*=0: Editable (default). *e*=1: Not editable. For a panel window, the Panel menu item is not present and the ShowTools command is ignored. For a panel subwindow, it can not be activated by clicking.

Flags

/W= *winName*	Modifies the control panel in the named graph or control panel window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

On Windows, set *r*, *g*, and *b* = 65535 (maximum white) to set the background color of the control panel to track the 3D Objects color in the Appearance Tab of the Display Properties control panel.

See Also

The **NewPanel** operation.

Controls in Graphs on page III-392.

ModifyTable

`ModifyTable [/W=`*`winName`*`/Z] `*`key`*` [(`*`columnSpec`*`)] =`*`value`*` [, `*`key`*` [(`*`columnSpec`*`)] =`*`value`*`]…`

The ModifyTable operation modifies the appearance the top or named table window or subwindow.

Parameters

Many of the parameter keywords take an optional *columnSpec* enclosed in parentheses. Usually *columnSpec* is simply the name of a wave displayed in the table. All table columns are affected when you omit `(`*`columnSpec`*`)`.

More precisely, column specifications are wave names for waves in the current data folder or data folder paths leading to waves in any data folder optionally followed by the suffixes .i, .l, .d, .id or .ld to specify dimension indices, dimension labels, data values, dimension indices and data values, or dimension labels and data values of the wave. For example, `ModifyTable font(myWave.i)="Helvetica"`. If the wave is complex, the column specification may be followed by `.real` or `.imag` suffixes.

One additional *columnSpec* is Point, which refers to the first column containing the dimension index numbers. If multidimensional waves are displayed in the table, this column may have the title "Row", "Column", "Layer", "Chunk" or "Element", but the *columnSpec* for this column is always Point. See **Column Names** on page II-176 for details.

Though not shown in the syntax, the optional `(`*`columnSpec`*`)` may be replaced with `[`*`columnIndex`*`]`, where *columnIndex* is zero or a positive integer denoting the column to be modified. `[0]` denotes the Point column, `[1]` denotes the first column appended to the table, `[2]` denotes the second appended column, etc. This syntax is used for style macros, in conjunction with the /Z flag.

You can use a range of column numbers instead of just a single column number, for example `[0,3]`.

The parameter descriptions below omit the optional (*columnSpec*).

alignment=*a*	Sets the alignment of table cell text. *a*=0: Left aligned. *a*=1: Center aligned. *a*=2: Right aligned.

autosize={*mode*, *options*, *padding*, *perColumnMaxSeconds*, *totalMaxSeconds*}

Autosizes the specified column or columns.

mode=0:	Sets width of each data column from a given multidimensional wave individually.
mode=1:	Sets width of all data columns from a given multidimensional wave the same.

options is a bitwise parameter. Usually 0 is the best choice.

Bit 0:	Ignores column names.
Bit 1:	Ignores horizontal indices.
Bit 2:	Ignores data cells.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

padding specifies extra padding for each column in points. Use -1 to get the default amount of padding (16 points).

perColumnMaxSeconds specifies the maximum amount of time to spend autosizing a single column. Use 0 to get the default amount of time (one second).

totalmaxSeconds specifies the maximum amount of time for autosizing the entire table. Use 0 to get the default amount of time (ten seconds).

digits=*d* Specifies the number of digits after decimal point or, for hexadecimal and octal columns, the number of total digits.

elements=(*row*, *col*, *layer*, *chunk*)

Selects the view of a multidimensional wave in the table. The values given to *row*, *col*, *layer*, and *chunk* specify how to change the view.

-1:	No change from current view.
-1:	Display this dimension vertically.
-3:	Display this dimension horizontally.
≥0:	For waves with 3 or 4 dimensions, display this element of the other dimensions.

See **ModifyTable Elements Command** on page II-197 for a detailed discussion of

entryMode=*m* Queries or sets the table's entry line mode.

m=0:	Just queries.
m=1:	Accepts any entry that was started if possible.
m=2:	Cancels any entry that was started if possible.

If *m* is 0 then the entry line state is not changed but is returned via V_flag as follows:

0:	No entry is in progress.
-1:	An entry is in progress and is valid.
Other:	An entry is in progress and is invalid.

If *m* is 1 then the entry is accepted if it is valid and its state is returned via V_flag as follows:

0:	No entry is in progress.
-1:	The entry was accepted.
Other:	The entry is invalid and was not accepted.

If *m* is 2 then the entry is cancelled if possible and its state is returned via V_flag as follows:

0:	No entry is in progress.
-1:	The entry was cancelled.

font="*fontName*" Sets font used in the table, e.g., `font="Helvetica"`.

format=*f* Sets the data format for the table.

f=0:	General.
f=1:	Integer.
f=2:	Integer with thousands (e.g., "1,234").
f=3:	Fixed point (e.g., "1234.56").
f=4:	Fixed point with thousands (e.g., "1,234.56").
f=5:	Exponential (scientific only).
f=6:	Date format.
f=7:	Time format (always 24 hour time).
f=8:	Date&time format (date followed by time).
f=9:	Octal.
f=10:	Hexadecimal.

You cannot apply date or date&time formats to a wave that is not double-precision (see **Date, Time, and Date&Time Units** on page II-64). To avoid this error, use **Redimension** to change the wave to double-precision.

frameInset= *i* Specifies the number of pixels by which to inset the frame of the table subwindow.

frameStyle= *f* Specifies the frame style for a table subwindow.

f=0:	None.
f=1:	Single.
f=2:	Double.
f=3:	Triple.
f=4:	Shadow.
f=5:	Indented.
f=6:	Raised.
f=7:	Text well.

The last three styles are fake 3D and will look good only if the background color of the enclosing space and the table itself is a light shade of gray.

horizontalIndex=*h* Controls what is displayed in the horizontal index row when multidimensional waves are displayed.

h=0:	Displays dimension labels if the multidimensional wave's label column is displayed, otherwise displays numeric indices (default).
h=1:	Always displays numeric indices for multidimensional waves.
h=2:	Always displays dimension labels for multidimensional waves.

The horizontal index row appears below the row of column names if the table contains a multidimensional wave. Use horizontalIndex to override the default behavior in order to display labels for the horizontal dimension while displaying numeric indices for the vertical dimension or vice versa.

horizontalIndex controls the horizontal index row only. To control what is displayed vertically, use **AppendToTable** to append a numeric index or dimension label column.

rgb=(*r*, *g*, *b*)	Sets color of text. *r*, *g*, and *b* are red, green, and blue components of the color and range from 0 to 65,535. Default is black: (0,0,0).

selection=(*firstRow*, *firstCol*, *lastRow*, *lastCol*, *targetRow*, *targetCol*)

Sets the selected cells in the table.

If any of the parameters have the value -1 then the corresponding part of the selection is not changed.

Otherwise they set the first and last selected cell and the target cell. Row and column values are 0 or greater. The Point column can not be selected.

The proposed parameters are clipped to avoid invalid combinations, such as the last selected row being before the first selected row.

With one exception, it does not support selecting unused cells. Therefore the proposed selection is clipped to prevent this. The exception is that, if the parameters call for selecting the first cell in the first unused column, then this is permitted.

showFracSeconds=*s* Shows (*s*=1) or hides (*s*=0; default) fractional seconds.

showParts=*parts* Specifies what elements of the table should be visible. Other elements are hidden.

parts is a bitwise parameter specifying what to show.

bit 0:	Entry line and other top line controls.
bit 1:	Name row.
bit 2:	Horizontal index row.
bit 3:	Point column.
bit 4:	Horizontal scroll bar.
bit 5:	Vertical scroll bar.
bit 6:	Insertion cells.
bit 7:	Insertion cells.

All other bits are reserved and must be set to zero except that you can pass -1 to indicate that you want to show all parts of the table.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Presentation tables in subwindows in graphs and page layouts do not have an entry line or scroll bars and therefore never show these items.

See **Parts of a Table** on page II-170 and **Showing and Hiding Parts of a Table** on page II-172 for further information.

sigdigits=*d* *d* is the number of significant digits when the numeric format is general.

size=*s* Font size, e.g., `size=14`.

style=*n* *n* is a bitwise parameter with each bit controlling one aspect of the column's font style as follows:

Bit 0:	Bold
Bit 1:	Italic
Bit 2:	Underline
Bit 4:	Strikethrough

For example, bold underlined is $2^0 + 2^2 = 1 + 4 = 5$. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

title="*title*" Sets the title of a column to *title*.

topLeftCell=(*row*, *column*)

	Scrolls the table contents so that the cell identified by (*row*, *column*) is the top left visible data cell, or as close as possible. If *row* is -1 then the table's vertical scrolling is not changed. If *column* is -1 then the table's horizontal scrolling is not changed. If they are positive, *row* and *column* are zero-based numbers which are clipped to valid values before being used. *row*=0 refers to the first row of data in the table, *column*=0 refers to the first column of data. The Point column can not be scrolled horizontally.
trailingZeros=*t*	Shows trailing zeros (*t*=1). This affects the general numeric format only.
width=*w*	Sets column width to *w* points. You will not always get the exact number of points that you request. This is because a column must have an even number of screen pixels, so that grid lines look good. Igor will modify your requested number of points to meet this requirement.

Flags

/W= *winName*	Modifies the named table window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/Z	No errors generated if the indexed or specified column does not exist in a style macro.

Examples

```
ModifyTable size(myWave)=14       // change font size of myWave column
ModifyTable width(Point)=0                 // hide Point column
ModifyTable style(cmplxWave.imag)=32       // condensed= bit 5 = 2^5 = 32
```

See Also

See **Column Names** on page II-176 and **ModifyTable Elements Command** on page II-197.

ModifyWaterfall

`ModifyWaterfall` `[`***`/W=winName`***`]` ***`keyword = value`*** `[,` ***`keyword = value`*** `…]`

The ModifyWaterfall operation modifies the properties of the waterfall plot in the top or named graph.

Parameters

keyword is one of the following:

angle= *a*	Angle in degrees from horizontal of the angled Y axis (*a* =10 to 90).
axlen= *len*	Relative length of angled Y axis. *len* is a fraction between 0.1 and 0.9.
hidden= *h*	Controls the hidden line algorithm. *h*=0: Turns hidden lines off. *h*=1: Uses painter's algorithm. *h*=2: True hidden. *h*=3: Hides lines with bottom removed. *h*=4: Hides lines using a different color for the bottom. When specified, the top color is the normal color for lines and the bottom color is set using `ModifyGraph negRGB=(r,g,b)`. Hidden lines are active only when the mode is lines between points.

Flags

/W= *winName*	Modifies waterfall plot in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

Painter's algorithm draws the traces from back to front and erases hidden lines while modes 2, 3 and 4 detect which line segments are hidden and suppresses the drawing of these segments.

See Also

Waterfall Plots on page II-255.

The **NewWaterfall** and **ModifyGraph** operations.

ModuleName

`#pragma ModuleName =` ***`modName`***

The ModuleName pragma assigns a name, which must be unique, to a procedure file so that you can use static functions and Proc Pictures in a global context, such as in the action procedure of a control or on the Command Line.

Using the ModuleName pragma involves at least two steps. First, within the procedure file assign it a name using `#pragma ModuleName=`*`modName`*, and then access objects in the named file by preceding the object name with the name of the module and the `#` character, such as or example: `ModName#StatFuncName()`.

See Also

The **Regular Modules** on page IV-222, **Static**, **Picture**, and #**pragma**.

MoveDataFolder

`MoveDataFolder` ***`sourceDataFolderSpec, destDataFolderPath`***

The MoveDataFolder operation removes the source data folder (and everything it contains) and places it at the specified location with the original name.

Parameters

sourceDataFolderSpec can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

destDataFolderPath can be a partial path (relative to the current data folder) or an absolute path (starting from root).

Details

MoveDataFolder generates an error if a data folder of the same name already exists at the destination.

Examples

Move data folder `foo` into data folder `bar`:

```
MoveDataFolder foo,root:bar:
```

Move data folder `foo` into data folder `bar`:

```
MoveDataFolder foo,:bar:
```

See Also

See the **DuplicateDataFolder** operation. Chapter II-8, **Data Folders**.

MoveFile

`MoveFile` `[`***`flags`***`][`***`srcFileStr`***`] [`**`as`** ***`destFileOrFolderStr`***`]`

The MoveFile operation moves or renames a file on disk. A file is renamed by "moving" it to the same folder it is already in using a different name.

Parameters

srcFileStr can be a full path to the file to be moved or renamed (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*.

If Igor can not determine the location of the file from *srcFileStr* and *pathName*, it displays an Open File dialog allowing you to specify the source file.

destFileOrFolderStr is interpreted as the name of (or path to) an existing folder when /D is specified, otherwise it is interpreted as the name of (or path to) a possibly existing file.

If *destFileOrFolderStr* is a partial path, it is relative to the folder associated with *pathName*.

If /D is specified, the source file is moved inside the folder using the source file's name.

If Igor can not determine the location of the destination file from *pathName*, *srcFileStr*, and *destFileOrFolderStr*, it displays a Save File dialog allowing you to specify the destination file (and folder).

If you use a full or partial path for either *srcFileStr* or *destFileOrFolderStr*, see **Path Separators** on page III-401 for details on forming the path.

Folder paths should not end with single Path Separators. See the **Details** section for **MoveFolder**.

Flags

/D	Interprets *destFileOrFolderStr* as the name of (or path to) an existing folder (or directory). Without /D, *destFileOrFolderStr* is the name of (or path to) a file. If *destFileOrFolderStr* is not a full path to a folder, it is relative to the folder associated with *pathName*.
/I [=*i*]	Specifies the level of interactivity with the user. /I=0: Interactive only if *srcFileStr* or *destFileOrFolderStr* is not specified or if the source file is missing. (Same as if /I was not specified.) /I=1: Interactive even if *srcFileStr* is specified and the source file exists. /I=2: Interactive even if *destFileOrFolderStr* is specified. /I=3: Interactive even if *srcFileStr* is specified and the source file exists. Same as /I only.
/M=*messageStr*	Specifies the prompt message in the Open File dialog. If /S is not specified, then *messageStr* will be used for both Open File and for Save File dialogs. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/O	Overwrite existing destination file, if any. Without /O, the user is asked if replacing the existing file is to be allowed.
/P=*pathName*	Specifies the folder to look in for the source file, and the folder into which the file is copied. *pathName* is the name of an existing symbolic path. Using /P means that both *srcFileStr* and *destFileOrFolderStr* must be either simple file or folder names, or paths relative to the folder specified by *pathName*.
/S=*saveMessageStr*	Specifies the prompt message in the Save File dialog.
/Z[=*z*]	Prevents procedure execution from aborting if it attempts to move a file that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort. /Z=0: Same as no /Z. /Z=1: Moves a file only if it exists. /Z alone is equivalent to /Z=1. /Z=2: Moves a file if it exists or displays a dialog if it does not exist.

Variables

The MoveFile operation returns information in the following variables:

`V_flag`	Set to zero if the file was moved, to -1 if the user cancelled either the Open File or Save File dialogs, and to some nonzero value if an error occurred, such as the specified file does not exist.
`S_fileName`	Stores the full path to where the file was moved from. If an error occurred or if the user cancelled, it is set to an empty string.
`S_path`	Stores the full path where the file was moved to. If an error occurred or if the user cancelled, it is set to an empty string.

Examples

Rename a file, using full paths:

```
MoveFile "HD:folder:aFile.txt" as "HD:folder:bFile.txt"
```

Rename a file, using a symbolic path:

```
MoveFile/P=myPath "aFile.txt" as "bFile.txt"
```

Move a file into a subfolder (the subfolder must exist):

```
MoveFile/D "Macintosh HD:folder:aFile.txt" as ":subfolder"
```

Move a file into an unrelated folder (the subfolder must exist):

```
MoveFile/D "Macintosh HD:folder:afile.txt" as "Server:archive"
```

Move a file from one folder to another and rename it:

```
MoveFile "Macintosh HD:folder:afile.txt" as "Server:archive:destFile.txt"
```

Move user-selected file into a particular folder:

```
MoveFile/D as "C:My Data:Selected Files Folder"
```

Move user-selected file in any folder as `bFile.txt` in same folder:

```
MoveFile as "bFile.txt"
```

Move user-selected file in any folder as `bFile.txt` in any folder:

```
MoveFile/I=2 as "bFile.txt"
```

See Also

The **Open**, **MoveFolder**, **CopyFolder**, **NewPath**, and **CreateAliasShortcut** operations. The **IndexedFile** function. **Symbolic Paths** on page II-21.

MoveFolder

`MoveFolder` `[`***`flags`***`][`***`srcFolderStr`***`]` `[`**`as`** ***`destFolderStr`***`]`

The MoveFolder operation moves or renames a folder on disk. A folder is renamed by “moving” it into the same folder it is already in, but with a different name.

Warning: *The `MoveFolder` command can destroy data* by overwriting another folder and its contents!

If you overwrite an existing folder on disk, MoveFolder will do so only if permission is granted by the user. The default behavior is to display a dialog asking for permission. The user can alter this behavior via the Miscellaneous Settings dialog’s Misc category.

If permission is denied, the folder will not be moved and V_Flag will return 1088 (Command is disabled) or 1275 (You denied permission to overwrite a folder). Command execution will cease unless the /Z flag is specified.

Parameters

srcFolderStr can be a full path to the folder to be moved or renamed (in which case /P is not needed), a partial path relative to the folder associated with *pathName*, or the name of a folder within the folder associated with *pathName*.

If the location of the source folder cannot be determined from *srcFolderStr* and *pathName*, it displays a Select Folder dialog allowing you to specify the source.

If /P=*pathName* is given, but *srcFolderStr* is not, then the folder associated with *pathName* is moved or renamed.

destFolderStr specifies the final location of the folder or, if /D is used, the parent of the final location of the folder.

destFolderStr can be a full path to the output (destination) folder (in which case /P is not needed), or a partial path relative to the folder associated with *pathName*.

If the location of the destination folder cannot be determined from *destFolderStr* and *pathName*, it displays a Save Folder dialog allowing you to specify the destination.

If you use a full or partial path for either file, see **Path Separators** on page III-401 for details on forming the path.

Flags

/D	Interprets *destFolderStr* as the name of (or path to) an existing folder (or “directory”) to move the source folder into. Without /D, it interprets *destFolderStr* as the name of (or path to) the moved folder. If *destFolderStr* is not a full path to a folder, it is relative to the source folder.
/I [=*i*]	Specifies the level of interactivity with the user. /I=0: Interactive only if *srcFolderStr* or *destFolderStr* is not specified or if the source folder is missing. (Same as if /I was not specified.) /I=1: Interactive even if *srcFolderStr* is specified and the source folder exists. /I=2: Interactive even if *destFolderStr* is specified. /I=3: Interactive even if *srcFolderStr* is specified and the source folder exists. Same as /I only.
/M=*messageStr*	Specifies the prompt message in the Open File dialog. If /S is not used, then *messageStr* will be used for both Open File and for Save File dialogs. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/O	Overwrite existing destination folder, if any. This deletes the existing destination folder. When /O is specified, the source folder can’t be moved into an existing folder without specifying the name of the moved folder in *destFolderStr*.
/P=*pathName*	Specifies the folder for relative paths in *srcFolderStr* and *destFolderStr*. *pathName* is the name of an existing symbolic path. If *srcFolderStr* is omitted, the folder associated with *pathName* is moved. If *destFolderStr* is omitted, the source folder is moved into the folder associated with *pathName*. Using /P means that *srcFolderStr* (if specified) and *destFolderStr* must be either simple folder names or paths relative to the folder specified by *pathName*.
/S=*saveMessageStr*	Specifies the prompt message in the Save File dialog.
/Z[=*z*]	Prevents procedure execution from aborting if it attempts to move a folder that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort. /Z=0: Same as no /Z. /Z=1: Moves a folder only if it exists. /Z alone is equivalent to /Z=1. /Z=2: Moves a folder if it exists or displays a dialog if it does not exist.

Variables

The MoveFolder operation returns information in the following variables:

`V_flag`	Set to zero if the file was moved, to -1 if the user cancelled either the Open File or Save File dialogs, and to some nonzero value if an error occurred, such as the specified file does not exist.
`S_fileName`	Stores the full path to the folder that was moved, with a trailing colon. If an error occurred or if the user cancelled, it is set to an empty string.
`S_path`	Stores the full path of the moved folder, with a trailing colon. If an error occurred or if the user cancelled, it is set to an empty string.

Details

You can use only /P=*pathName* (omitting *srcFolderStr*) to specify the source folder to be moved.

A folder path should not end with single Path Separators. For example:

```
MoveFolder "Macintosh HD:folder" as "Macintosh HD:Renamed Folder:"
MoveFolder "Macintosh HD:folder:" as "Macintosh HD:Renamed Folder"
MoveFolder "Macintosh HD:folder:" as "Macintosh HD:Renamed Folder:"
```

will do weird, unexpected things (and probably damaging things when /O is also used). Instead, use:

```
MoveFolder "Macintosh HD:folder" as "Macintosh HD:Renamed Folder"
```

Beware of PathInfo and other command which return paths with an ending path separator. (They can be removed with the **RemoveEnding** function.)

A folder may not be moved into one of its own subfolders.

Conversely, the command:

```
MoveFolder/O/P=myPath "afolder"
```

which attempts to overwrite the folder associated with myPath with a folder that is inside it (namely “afolder”) is not allowed. Instead, use:

```
MoveFolder/O/P=myPath "::afolder"
```

On Windows, renaming or moving a folder never updates the value of any Igor Symbolic Paths that point to a moved folder:

```
// Create a folder
NewPath/O/C myPath "C:\\My Data\\My Work"

// Move the folder
MoveFolder/P=myPath as "C:\\My Data\\Moved"

// Display the path's value
PathInfo myPath               // (or use the Path Status dialog)
Print S_Path
• C:My Data:My Work
```

You can use PathInfo to determine if a folder referred to by an Igor symbolic path exists and where it is on the disk. Use NewPath/O to reset the path’s value.

On the Macintosh, however, renaming or moving a folder on the same volume does alter the value of symbolic path. This is because MoveFolder uses a Mac OS alias to keep track of the folder. A folder renamed or moved on the same volume retains the original “volume refnum” and “directory ID” stored in the alias mechanism, so that the alias (and hence Igor’s symbolic path) remains pointing to the moved folder. After moving the folder, using the unchanged volume refnum and directory ID (in PathInfo or when you use /P=pathName) returns the updated path.

Moving the folder to a different volume actually creates a new folder with new volume refnum and directory IDs, and symbolic paths pointing to or into the moved folder aren’t updated. They will be pointing at a deleted folder (they’re probably invalid).

Examples

Rename a folder (“move” it to the same folder):

```
MoveFolder "Macintosh HD:folder" as "Macintosh HD:Renamed Folder"
```

Rename a folder referred to by only a path:

```
NewPath/O myPath "Macintosh HD:folder"
MoveFolder/P=myPath as "::Renamed Folder"
```

Move a folder from one volume to another. This moves “Macintosh HD:My Folder” inside “Server:My Folder” if “Server:My Folder” already exists:

```
MoveFolder "Macintosh HD:My Folder" as "Server:My Folder"
```

Move a folder from one volume to another. This overwrites “Server:My Folder” (if it existed) with the moved “Macintosh HD:My Folder”:

```
MoveFolder/O "Macintosh HD:My Folder" as "Server:My Folder"
```

Move user-selected folder in any folder as “Renamed Folder” into a user-selected folder (possibly the same one):

```
MoveFolder as "Renamed Folder"
```

Move user-selected file in any folder as “Moved Folder” in any folder:

```
MoveFolder/I=3 as "Moved Folder"
```

See Also

MoveFile, **CopyFolder**, **IndexedDir**, **PathInfo**, and **RemoveEnding**. **Symbolic Paths** on page II-21.

MoveString

MoveString *sourceString*, *destDataFolderPath* [***newname***]

The MoveString operation removes the source string variable and places it in the specified location optionally with a new name.

Parameters

sourceString can be just the name of a string variable in the current data folder, a partial path (relative to the current data folder) and variable name or an absolute path (starting from root) and variable name.

destDataFolderPath can be a partial path (relative to the current data folder) or an absolute path (starting from root).

Details

An error is issued if a variable or wave of the same name already exists at the destination.

Examples

```
MoveString :foo:s1,:bar:       // Move string s1 into data folder bar
MoveString :foo:s1,:bar:ss1    // Move string s1 into bar with new name ss1
```

See Also

The **MoveVariable**, **MoveWave**, and **Rename** operations; andChapter II-8, **Data Folders**.

MoveSubwindow

MoveSubwindow [**/W=*winName***] ***key* = (*values*)**[**, *key* = (*values*)**]…

The MoveSubwindow operation moves the active or named subwindow to a new location within the host window. This command is primarily for use by recreation macros; users should use layout mode for repositioning subwindows.

Parameters

fguide=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the frame guide name(s) to which the outer frame of the subwindow is attached inside the host window.

The frame guides are identified by the standard names or user-defined names as defined by the host. Use * to specify a default guide name.

When the host is a graph, additional standard guides are available for the outer graph rectangle and the inner plot rectangle (where traces are plotted).

See **Details** for standard guide names.

fnum=(*left*, *top*, *right*, *bottom*)

Specifies the new location of the subwindow. The location coordinates of the subwindow sides can have one of two possible meanings:

When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.

When any value is greater than 1, coordinates are taken to be fixed locations measured in points, or pixels for control panels, relative to the top left corner of the host frame.

pguide=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the guide name(s) to which the plot rectangle of the graph subwindow is attached inside the host window.

Guides are identified by the standard names or user-defined names as defined by the host. Use * to specify a default guide name.

See **Details** for standard guide names.

Flags

/W= *winName*	Moves the subwindow in the named window or subwindow. When omitted, action will affect the active subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

When moving an exterior subwindow, only the fnum keyword may be used. The values are the same as the **NewPanel** /W flag for exterior subwindows.

The names for the built-in guides are as defined in the following table:

	Left	**Right**	**Top**	**Bottom**
Subwindow Frame	FL	FR	FT	FB
Outer Graph Rectangle	GL	GR	GT	GB
Inner Plot Rectangle	PL	PR	PT	PB

The frame guides apply to all window and subwindow types. The graph rectangle and plot rectangle guide types apply only to graph windows and subwindows.

See Also

The **MoveWindow** operation. Chapter III-4, **Embedding and Subwindows** for further details and discussion.

MoveVariable

MoveVariable *sourceVar*, *destDataFolderPath* [***newname***]

The MoveVariable operation removes the source numeric variable and places it in the specified location optionally with a new name.

Parameters

sourceVar can be just the name of a numeric variable in the current data folder, a partial path (relative to the current data folder) and variable name or an absolute path (starting from root) and variable name.

destDataFolderPath can be a partial path (relative to the current data folder) or an absolute path (starting from root).

Details

An error is issued if a variable or wave of the same name already exists at the destination.

Examples

```
MoveVariable :foo:v1,:bar:          // Move v1 into data folder bar
MoveVariable :foo:v1,:bar:vv1       // Move v1 into bar with new name vv1
```

See Also

The **MoveString**, **MoveWave**, and **Rename** operations; and Chapter II-8, **Data Folders**.

MoveWave

`MoveWave` ***`sourceWave, destDataFolderPath`*** `[`***`newname`***`]`

The MoveWave operation removes the source wave and places it in the specified location optionally with a new name.

Parameters

sourceWave can be just the name of a wave in the current data folder, a partial path (relative to the current data folder) and wave name or an absolute path (starting from root) and wave name.

destDataFolderPath can be a partial path (relative to the current data folder) or an absolute path (starting from root).

Details

An error is issued if a variable or wave of the same name already exists at the destination.

Examples

```
MoveWave :foo:w1,:bar:        // Move wave w1 into data folder bar
MoveWave :foo:w1,:bar:ww1     // Move w1 into bar with new name ww1
```

See Also

The **MoveString**, **MoveVariable**, and **Rename** operations; and Chapter II-8, **Data Folders**.

MoveWindow

`MoveWindow` `[`***`flags`***`]` ***`left, top, right, bottom`***

The MoveWindow operation moves the target or specified window to the given coordinates.

Flags

/C	Moves Command window instead of the target window.
/F	*Windows*: Moves the Igor Pro application “frame” and the frame is then adjusted so that no part is offscreen. *Macintosh*: Moves nothing.
/I	Coordinates are in inches.
/M	Coordinates are in centimeters.
/P=*procedureTitleAsName*	Moves the specified procedure window instead of the target window.
/W=*winName*	Moves the named window.

Details

Note that neither *winName* nor *procedureTitleAsName* is a string but is the actual window name or procedure window title. If the procedure window’s title (procedure windows don’t have names) has a space in it, use $ and quotes:

```
MoveWindow/P=$"Log Histogram" 0,0,600,400
```

If /W, /F, /C, and /P are omitted, MoveWindow moves the target window.

The coordinates are in points if neither /I nor /M is used.

In Igor Pro 7.00 or later, to move the window without changing its size, pass -1 for both *right* and *bottom*.

You can use the MoveWindow operation to minimize, restore, or maximize a window by specifying 0, 1, or 2 for all of the coordinates, respectively, as follows:

```
MoveWindow 0, 0, 0, 0      // Minimize target window.
MoveWindow 1, 1, 1, 1      // Restore target window.
MoveWindow 2, 2, 2, 2      // Maximize target window.
```

On Macintosh, “maximize” means to move and resize the window so that it fills the screen. “Minimize”means to minimize to the dock.

If the window size has been constrained by `SetWindow sizeLimit`, those limits are silently applied to the size set by MoveWindow.

See Also

The **MoveSubwindow** and **DoWindow** operations.

MultiTaperPSD

`MultiTaperPSD [`*`flags`*`]` ***`srcWave`***

The MultiTaperPSD operation estimates the power spectral density of *srcWave* using Slepian (DPSS) tapers.

The MultiTaperPSD operation was added in Igor Pro 7.00.

Flags

/A	Uses Thomson's adaptive algorithm. In this case the operation also creates the wave W_MultiTaperDF that contains the effective degrees of freedom. For each frequency of the PSD the algorithm is expected to converge within few iterations. When it fails to converge, the operation prints in the history the total number of frequencies where it did not converge while the actual output contains the last iteration estimate.
/dB	Scale the PSD results as `10*log10(spectralEst(f))`.
/dbF=*f0*	Scale the PSD results as `10*log10(spectralEst(f)/spectralEst(`*`f0`*`))` where *f0* must be in the range [0,0.5/DimDelta(*srcWave*,0)].
/DEST=destWave	Saves the PSD estimate in a wave specified by destWave. The destination wave is created or overwritten if it already exists. Creates a wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-66 for details. If you omit /DEST the operation saves the resulting spectral estimate in the wave W_MultiTaperPSD in the current data folder.
/F	Computes F-test statistic for each output frequency. The results are stored in the wave W_MultiTaperF. If /DEST is also used then the F-test results are stored in the same data folder as destWave. Otherwise W_MultiTaperF is created in the current data folder. The statistic is a variance ratio, of the background and the power at the specific frequency. Since the PSDs of the background and the line are assumed to be distributed as Chi-squared with 2 and 2*nTapers-2 degrees of freedom respectively, the relevant critical value for computing confidence intervals can be obtained from: `StatsInvFCdf(percentSignificance/100,2,2*nTapers-2)`
/NOR=*N*	Sets the normalization factor that is used to multiply each element of the output. For example, if you want to normalize the output such that the sum of the PSD estimate matches the variance of the input use /NOR=2/(np*np) where np is the number of points in *srcWave*.
/NTPR=*nTapers*	Specifies the number of Slepian tapers to be used. If you do not specify a number of tapers, the operation uses 2*nw(twice the time-bandwidth product).
/NW=*nw*	Specifies the time-bandwidth product. This value should typically be in the range [2,6]. Given a time-bandwidth product nw it is recommended to use no more than 2**nw* tapers in order to maximize variance efficiency.
/Q	Quiet mode; suppresses printing in the history area.
/R=[*startPoint*,*endPoint*]	Calculates the PSD estimate for a specified input range. *startPoint* and *endPoint* are expressed in terms of point numbers of the source wave.

/R=(*startX*,*endX*)	Calculates the PSD estimate for a specified input range. *startX* and *endX* are expressed in terms of X values. Note that this option converts your X specifications to point numbers and some roundoff may occur.
/Z	Do not report errors.

Details

The MultiTaperPSD operation estimates the PSD of *srcWave* by computing a set of discrete prolate spheroidal functions (Slepian **DPSS**) and using them as optimal window functions. The window functions/tapers are applied to the input signal and squares of the resulting Fourier transforms are weighted together to produce the PSD estimate.

srcWave must be a real-valued numeric wave of single or double precision and must not contain any INFs or NaNs.

The mean value of the input is subtracted prior to multiplication by the tapers. Like DSPPeriodogram, the MultiTaperPSD operation leaves the normalization to the user.

The default PSD estimate is calculated by combining the Fourier transforms of the tapered signal with weights from the DPSS calculation. You can use the /A flag to improve the PSD estimate for increasing tapers. Thomson's adaptive algorithm is reasonably efficient and also provides an estimate of the effective degrees of freedom as a function of frequency.

The operation sets the variable V_Flag to zero if successful or to a -1 if it encounters an error. If you are using Thomson's adaptive algorithm (/A) V_Flag is set to the number of frequencies at which the algorithm failed to converge.

See Also

FFT, **DSPPeriodogram**, **DPSS**, **ImageWindow**, **Hanning**, **LombPeriodogram**

Demos

See the “MultiTaperPSD Demo” example experiment.

References

D.J. Thomson: "Spectrum Estimation and Harmonic Analysis", Proc. IEEE 70 (9) 1982 pp. 1055.

D. Slepian, "Prolate Spheroidal Wave Functions, Fourier Analysis, and Uncertainty -- V: the Discrete Case", Bell System Tech J. Vol 57 (5) May-June 1978.

Lees, J. M. and J. Park (1995). Multiple-taper spectral analysis: A stand-alone C-subroutine: Computers & Geosciences: 21, 199-236.

MultiThread

MultiThread ***wave* = *expression***

In user-defined functions, the MultiThread keyword can be inserted in front of wave assignment statements to speed up execution on multiprocessor computer systems.

Warning: Misuse of this keyword can result in a performance penalty or even a crash. Be sure to read **Automatic Parallel Processing with MultiThread** on page IV-303 before using MultiThread.

The expression must be thread-safe. This means that if it calls a function, the function must be thread-safe. This goes for both built-in and user-defined functions.

Not all built-in functions are thread-safe. Use the Command Help tab in the Igor Help Browser to see which functions are thread-safe.

User-defined functions are thread-safe if they are defined using the **ThreadSafe** keyword. See **ThreadSafe Functions** on page IV-97 for details.

See Also

Automatic Parallel Processing with MultiThread on page IV-303.

Waveform Arithmetic and Assignments on page II-69.

The “MultiThread Mandelbrot Demo” experiment.

MultiThreadingControl

`MultiThreadingControl` *`keyword`* `[=`*`value`*`]`

The MultiThreadingControl operation allows you to control how automatic multithreading works with those IGOR operations that support it. Automatic multithreading is described below under Details.

For most purposes you will not need to use this operation.

The MultiThreadingControl operation was added in Igor Pro 7.00.

Keywords

getMode	Writes the current mode value into the variable V_autoMultiThread.
getThresholds	Creates the wave W_MultiThreadingArraySizes in the current data folder. See **Automatic Multithreading Thresholds** below for details.
setMode=*m*	Sets the mode for automatic multithreading. The mode controls the circumstances in which automatic multithreading is enabled.

m=0:	Disables automatic multithreading unconditionally.
m=1:	Enables automatic multithreading based on operation-specific thresholds for operations called from the main thread only. This is the default setting.
m=4:	Enables automatic multithreading based on operation-specific thresholds for operations called from the main thread and from user-created explicit threads.
m=8:	Enables automatic multithreading unconditionally - regardless of thresholds or the type of the calling thread.

You can not combine modes by ORing. The only valid values for *m* are those shown above.

setThresholds=*tWave*	Sets the thresholds for automatic multithreading. See **Automatic Multithreading Thresholds** below for details.

Details

Some IGOR operations and functions have internal code that can execute calculations in parallel using multiple threads. These operations are marked as "Automatically Multithreaded" in the Command Help pane of the Igor Help Browser.

Running on multiple threads reduces the time required for number-crunching tasks on multi-processor machines when the benefit of using multiple processors exceeds the overhead of running in multiple threads. This is usually the case only for large-scale jobs.

By default Igor uses automatic multithreading in operations that support it when the number of calculations exceeds a threshold value. This is called "automatic multithreading" to distinguish it from the explicit multithreading that you can instruct Igor to do. Explicit multithreading is described under **ThreadSafe Functions and Multitasking** on page IV-308. You don't need to do anything to benefit from automatic multithreading.

By default automatic multithreading is enabled for operations called from the main thread and disabled for operations called from explicit threads that you create (mode=1). You can change this using the setMode keyword described above.

The state of automatic multithreading is not saved with the experiment. It is initialized to mode=1 with default thresholds every time you start IGOR.

Automatic Multithreading Thresholds

Executing these commands

```
MultiThreadingControl getThresholds
Edit W_MultiThreadingArraySizes.ld
```

creates a wave named W_MultiThreadingArraySizes and displays it in a table. This shows you the current threshold for each operation that supports automatic multithreading. The wave includes dimension labels so you can see which row represents which operation's threshold.

The meaning of a given threshold value depends on the operation. For most operations the threshold is in terms of the number of points in the input wave. For some operations the threshold depends on the complexity of the calculation. For example, the threshold for the CurveFit operation takes the complexity of the fitting function into account.

You can change the threshold for a given operation by setting the data for the appropriate row of W_MultiThreadingArraySizes and passing it back to the MultiThreadingControl operation using the setThresholds keyword. For example:

```
W_MultiThreadingArraySizes[%ICA]=5000      // Set the ICA threshold
MultiThreadingControl setThresholds=W_MultiThreadingArraySizes  // Apply
```

You should not change any aspect of the wave other than the threshold values.

Examples

```
MultiThreadingControl setMode=0         // Disable automatic multithreading
MultiThreadingControl setMode=8         // Always multithread regardless of wave size
```

See Also

Automatic Parallel Processing with TBB on page IV-302

Automatic Parallel Processing with MultiThread on page IV-303

ThreadSafe Functions on page IV-97, **ThreadSafe Functions and Multitasking** on page IV-308

NameOfWave

`NameOfWave(`*`wave`*`)`

The NameOfWave function returns a string containing the name of the specified wave.

In a user-defined function that has a parameter or local variable of type WAVE, NameOfWave returns the actual name of the wave identified by the WAVE reference. It can also be used with wave reference functions such as **WaveRefIndexedDFR**.

NameOfWave does not return the full data folder path to the wave. Use **GetWavesDataFolder** for this information.

A null wave reference returns a zero-length string. This might be encountered, for instance, when using WaveRefIndexedDFR in a loop to act on all waves in a data folder, and the loop has incremented beyond the highest valid index.

Examples

```
Function ZeroWave(w)
   Wave w
   w = 0
   Print "Zeroed the contents of", NameOfWave(w)
End
```

See Also

See **WAVE**; the **GetWavesDataFolder** and **WaveRefIndexed** functions; and **Wave Reference Functions** on page IV-186.

NaN

`NaN`

The NaN function returns the “Not a Number” value according to the IEEE standards.

Comparison operators do not work with NaN parameters because, by definition, NaN compared to anything, even another NaN, is false. Use **numtype** to test if a value is NaN.

NeuralNetworkRun

`NeuralNetworkRun [/Q/Z] Input=`*`testWave`*`, WeightsWave1=`*`w1`*`, WeightsWave2=`*`w2`*

The NeuralNetworkRun operation uses the interconnection weights generated by NeuralNetworkTrain, and saved in the waves M_Weights1 and M_Weights2, to execute the network for a given input. The input can contain a single run represented by a 1D wave or M runs represented by M columns of a 2D wave. The output of the calculation is saved in the wave W_NNResults or M_NNResults depending on the dimensionality of the input wave. The structure of the network is completely specified by the two weights waves and must match the number of rows in the input wave.

Flags

/Q	Suppresses printing information in the History area.
/Z	No error reporting.

Parameters

Input=*testWave*	Specifies the input to the neural network. *testWave* must be a single or double precision wave containing entries in the range [0,1] and have the correct number of rows to match the weights. Execute the network for multiple runs by using a 2D input wave where each column corresponds to a single run. For a 2D input, the result will be stored in M_NNResults with a corresponding column structure.
WeightsWave1=*w1*	Specifies the interconnection weights between the input and the hidden layer.
WeightsWave2=*w2*	Specifies the interconnection weights between the hidden layer and the output.

See Also

The **NeuralNetworkTrain** operation.

NeuralNetworkTrain

`NeuralNetworkTrain [/Q/Z] [`*`keyword = value`*`]…`

The NeuralNetworkTrain operation trains a three-layer neural network. The training produces two 2D waves that store the interconnection weights between the network neurodes. Once you obtain the weights, you can use them with NeuralNetworkRun.

Flags

/Q	Suppresses printing information in the History area.
/Z	No error reporting.

Parameters

keyword is one of the following:

Input=*inWave*	Specifies the input patterns for training. *inWave* is a 2D wave where each row corresponds to a single training event and each column corresponds to the input values. The number of rows in *inWave* (the number of training sets) and in the output wave must be equal. *inWave* must be single or double precision and all entries must be in the range [0,1].
Iterations=*num*	Specifies the number of iterations. Default is 10000.
MinError=*val*	Terminates training when the total error drops below *val* (default is 1e-8). The total error is normalized, and is defined as the sum of the squared errors divided by the number of training sets times outputs.
Momentum=*val*	Specifies a coefficient for the back-propagation algorithm. This coefficient adds to the change in a particular weight a contribution proportional to the error in a previous iteration. Default momentum is 0.075.
NHidden=*num*	Specifies the number of hidden neurodes. You do not need to use the Structure keyword with NHidden because the network is completely specified by the training waves and NHidden.
NReport=*num*	Specifies over how many iterations (default is 1000) to print the global RMS error to the history area. Ignored with /Q.

Output=*outWave*	Specifies the expected outputs corresponding to the entries in the input wave. The number of rows in *outWave* (the number of training sets) and in the input wave must be equal. *outWave* must be single or double precision and all entries must be in the range [0,1].
LearningRate=*val*	Sets the network learning rate, which is used in the backpropagation calculation. Default is 0.15.
Restart	Allows specification of your own set of weights as the starting values. Use this to run the training and feed the output weights of one training session as the input for the next.
Structure={*Ni*, *Nh*, *No*}	Specifies the structure of the network. *Ni* is the number of neurodes at the input, *Nh* is the number of hidden neurodes, and *No* is the number of output neurodes. Structure is unnecessary when using NHidden is because the remaining numbers are determined by the sizes of the input and output waves.
WeightsWave1=*w1*	Specifies the weights for propagation from the first layer to the second. The 2D wave must be double precision and the dimensions must match the specified neurodes with the same numbers of rows and inputs and with matching numbers of columns and hidden neurodes.
WeightsWave2=*w2*	Specifies the weights for propagation from the second to the third layer. The 2D wave must be double precision and the dimensions must match the specified neurodes with the same numbers of rows and hidden neurodes and with matching numbers of columns and outputs.

Details

NeuralNetworkTrain is the first half of the implementation of a three-layer neural network in which both in inputs and outputs are taken as normalized quantities in the range [0,1]. Network training is based on back-propagation to iteratively minimize the error between the output and the expected output for any given training set. Training creates in two 2D waves that contain the interconnection weights between the neurodes. M_Weights1 contains the weights between the input layer and the hidden layer and M_Weights2 contains the weights between the hidden layer and the output layer. During the iteration stage, global error information can be printed in the history area.

The algorithm computes the output of the *k*th neurode by

$$V_k = \left[1 + \exp\left(-\sum_{i=1}^{n} w_i s_i\right)\right]^{-1},$$

where w_i is the weight corresponding to input *i*, s_i is the signal corresponding to that input, and *n* is the number of inputs connected to the neurode.

The total error is defined as the sum (over all training sets and all outputs) of the squared differences between the network outputs and the expected values. The sum is normalized by the product of the number of training sets and the number of outputs. The history reports (see NReport parameter) the square root of the total error (RMS error). The square root of the error computed at the end of the last iteration is stored in the variable V_rms.

See Also

The **NeuralNetworkRun** operation.

NewCamera

`NewCamera` `[`***`flags`***`] [`***`keywords`***`]`

The NewCamera operation creates a new camera window.

Documentation for the NewCamera operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "NewCamera"
```

NewDataFolder

`NewDataFolder` `[`**`/O/S`**`]` ***`dataFolderSpec`***

The NewDataFolder operation creates a new data folder of the given name.

Parameters

dataFolderSpec can be just a data folder name, a partial path (relative to the current data folder) with name or a full path (starting from root) with name. If just a data folder name is used then the new data folder is created in the current data folder. If a full or partial path is used, all data folders except for the last in the path must already exist.

Flags

/O	No error if a data folder of the same name already exists.
/S	Sets the current data folder to dataFolderSpec after creating the data folder.

Examples

```
NewDataFolder foo         // Creates foo in the current data folder
NewDataFolder :bar:foo    // Creates foo in bar in current data folder
NewDataFolder root:foo    // Creates foo in the root data folder
```

See Also

Chapter II-8, **Data Folders**.

NewFIFO

`NewFIFO` ***`FIFOName`***

The NewFIFO operation creates a new FIFO.

Details

Useless until channel info is added with **NewFIFOChan**.

An error is generated if a FIFO of same name already exists. *FIFOName* needs to be unique only among FIFOs. You can not overwrite a FIFO.

See Also

FIFOs are used for data acquisition. See **FIFOs and Charts** on page IV-291 and the **NewFIFOChan** operation for more information.

NewFIFOChan

`NewFIFOChan` `[`***`flags`***`]` ***`FIFOName, channelName, offset, gain, minusFS, plusFS, unitsStr`*** `[,` ***`vectPnts`***`]`

The NewFIFOChan operation creates a new channel for the named FIFO.

Parameters

channelName must be unique for the specified FIFO.

The *offset*, *gain*, *plusFS*, *minusFS* and *unitsStr* parameters are used when the channel's data is displayed in a chart or transferred to a wave. If given, *vectPnts* must be between 1 and 65535.

Flags

The flags define the type of data to be stored in the FIFO channel:

/B	8-bit signed integer. Unsigned if /U is present.
/C	Complex.
/D	Double precision IEEE floating point.
/I	32-bit signed integer. Unsigned if /U is present.
/S	Single precision IEEE floating point (default).
/U	Unsigned integer data.

/W	16-bit signed integer. Unsigned if /U is present.
/Y=*type*	Specifies wave data type. See details below.

Wave Data Types

As a replacement for the above number type flags you can use /Y=*numType* to set the number type as an integer code. See the **WaveType** function for code values. Do not use /Y in combination with other type flags.

Details

You can not invoke NewFIFOChan while the named FIFO is running.

If you provide a value for *vectPnts*, you will create a channel capable of holding a vector of data rather than just a single data value. When such a channel is used in a Chart, it is displayed as an image using one of the built-in color tables.

Igor scales values in the FIFO channel before displaying them in a chart or transferring them to a wave as follows:

```
scaled_value = (FIFO_value - offset) * gain
```

Igor uses the *plusFS* and *minusFS* parameters (plus and minus full scale) to set the default display scaling for charts.

The *unitsStr* parameter is limited to a maximum of three bytes.

When you transfer a channel's data to a wave, using the **FIFO2Wave** operation, Igor stores the *plusFS* and *minusFS* values and the *unitsStr* in the wave's Y scaling.

See Also

FIFOs are used for data acquisition. See **FIFOs and Charts** on page IV-291 and the **NewFIFO** and **FIFO2Wave** operations for more information.

The **Chart** operation for displaying FIFO data.

NewFreeAxis

NewFreeAxis[***flags***] ***axisName***

The NewFreeAxis operation creates a new free axis that has no controlling wave.

Parameters

axisName is the name for the new free axis.

Flags

/L/R/B/T	Specifies whether to attach the free axis to the Left, Right, Bottom, or Top plot edge, respectively. The Left edge is used by default.
/O	Replaces *axisName* if it already exists, which means any existing axis is marked as truly free.
/W=*winName*	Draws in the named graph window. *winName* may also be the name of a subwindow. *winName* must not conflict with other axis names except when using the /O flag. If /W is omitted, it creates a new axis in the active graph window or subwindow.

Details

A truly free axis does not use any scaling or units information from any associated waves (which need not exist.) You can set the properties of a free axis using **SetAxis** or **ModifyFreeAxis**.

Example

Copy this function to your Procedure window and compile:

```
Function axhook(s)
    STRUCT WMAxisHookStruct &s

    Variable t= s.max
    s.max= s.min
    s.min= t
    return 0
End
```

Now execute this code on the Command line:

```
Make jack=x
Display jack
NewFreeAxis fred
ModifyFreeAxis fred, master=left, hook=axhook
```

See Also

The **SetAxis**, **KillFreeAxis**, and **ModifyFreeAxis** operations.

NewFreeDataFolder

`NewFreeDataFolder()`

The NewFreeDataFolder function creates a free data folder and then returns its data folder reference.

Recommended for advanced programmers only.

Details

Free data folders are those that are not a part of the normal data folder hierarchy and can not be located by name.

See Also

Chapter II-8, **Data Folders**, **Free Data Folders** on page IV-88 and **Data Folder References** on page IV-72.

NewFreeWave

`NewFreeWave(`*`type`*`, `*`numPoints`*`)`

The NewFreeWave function creates a free 1D wave of the given type and number of points and then returns its wave reference.

Recommended for advanced programmers only.

Details

NewFreeWave creates a free wave named '_free_'.

You can also create free waves using **Make**/FREE and **Duplicate**/FREE. These are preferable for creating multidimensional free waves and also fine for general use.

The type parameter can be either a code as documented for **WaveType** or can be 0x100 to create a data folder reference wave or 0x200 to create a wave reference wave.

You can redimension free waves as desired but, for maximum efficiency, you should create the wave with the desired type and total number of points and then use the /E=1 flag with **Redimension** to simply reshape without moving data.

A free wave is automatically discarded when the last reference to it disappears.

See Also

Free Waves on page IV-84, **Make**, **Duplicate**.

NewGizmo

`NewGizmo` `[`*`flags`*`]`

The NewGizmo operation creates a new Gizmo display window.

Documentation for the NewGizmo operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "NewGizmo"
```

NewImage

`NewImage` `[`*`flags`*`]` ***`matrix`***

The NewImage operation creates a new image graph much like “`Display;AppendImage matrix`” except the graph is prepared using a style more appropriate for images. Rather than using preferences, NewImage provides several discrete styles to choose from.

Parameters

matrix is usually an MxN matrix containing image data. See **AppendImage** for details.

Flags

/F	By default, the image is flipped vertically to correspond to normal image orientation. if /F is present then the image is not flipped.
/G=*g*	Controls treatment of three-plane images as direct (RGB) color. *g*=1: Suppresses the autodetection of three-plane images as direct (RGB) color. *g*=1: Same as no /G flag (default).
/HIDE=*h*	Hides (h = 1) or shows (h = 0, default) the window.
/HOST=*hcSpec*	Embeds the new image plot in the host window or subwindow specified by *hcSpec*. When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/K=*k*	Specifies window behavior when the user attempts to close it. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. *k*=3: Hides the window. If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.
/N=*name*	Requests that the created graph have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen graph name. Use `DoWindow/K` *`name`* to ensure that *name* is available.
/S=*s*	Specifies one of several window styles. *s*=0: Fills entire window with image. No axes. However, this can result in the lower-right corner not being visible due to the target icon or grow icon (*Macintosh*). *s*=1: Like *s*=0 but insets image to avoid corner icon. *s*=2: Provides minimalist axes (default).

Details

The graph is sized to make the image pixels a multiple of the screen pixels with the graph size constrained to be not too small and not too large.

If *matrix* appears to fit Igor's standard monochrome category, then explicit mode is set (See ModifyImage explicit). To be considered monochrome the wave must be unsigned byte and contain only values of 0, 64 or 255.

Once the graph is created it is a normal graph and has no special properties other than the settings it was created with. Specifically, it will not autosize itself if the dimensions of *matrix* are changed. NewImage is just a shortcut for creating a graph window with a style appropriate for images.

This operation is limited in scope by design. If you need to specify the position, size or title, then use the operations Display and AppendImage.

If the styles provided are not what you desire, touch up an image graph to meet your needs and then use Capture Graph Prefs from the Graphs menu. Then use "`Display;AppendImage`" rather than NewImage.

See Also

The **Display**, **DoWindow**, **AppendImage**, and **ModifyImage** operations.

NewLayout

`NewLayout [`*`flags`*`] [as `*`titleStr`*`]`

The NewLayout operation creates a page layout.

Unlike the Layout operation, NewLayout can be used in user-defined functions. Therefore, NewLayout should be used in new programming instead of Layout.

NewLayout just creates the layout window. Use **AppendLayoutObject** to add objects to the window.

Parameters

The optional *titleStr* parameter is a string expression containing the layout's title. If not specified, Igor will provide one which identifies the objects displayed in the graph.

Flags

/B=(*r*,*g*,b)	Specifies the background color for the layout. *r*, *g*, and *b* are integers from 0 to 65535. Defaults to white (65535, 65535, 65535).
/C=*colorOnScreen*	Obsolete. In ancient times, this flag switched the screen display of the layout between black and white and color. It is still accepted but has no effect.
/HIDE=*h*	Hides (h = 1) or shows (h = 0, default) the window.
/K=*k*	Specifies window behavior when the user attempts to close it. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. *k*=3: Hides the window. If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.
/N=*name*	Requests that the layout have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen layout name. Use `DoWindow/K name` to ensure that *name* is available. If /N is not used, a name of the form "Layout*n*", where *n* is some integer, is assigned. In a function or macro, the assigned name is stored in the S_name string. This is the name you can use to refer to the page layout window from a procedure. Use the **RenameWindow** operation to rename the window.
/P=*orientation*	Sets the orientation of the page in the layout to either Portrait or Landscape (e.g., `Layout/P=Landscape`). See **Details**.
/W=(*left*,*top*,*right*,*bottom*)	Gives the layout window a specific location and size on the screen. Coordinates for /W are in points.

Details

When you create a new page layout window, if preferences are enabled, the page size is determined by the preferred page size as set via the Capture Layout Prefs dialog. If preferences are disabled, as is usually the case when executing a procedure, the page is set to the factory default size.

See Also

AppendLayoutObject, **DoWindow**, **RemoveLayoutObjects**, and **ModifyLayout**.

NewMovie

`NewMovie [`*`flags`*`] [as `*`fileNameStr`*`]`

The NewMovie operation opens a movie file in preparation for adding frames. It creates QuickTime movies on Macintosh and AVI movies on Windows. Prior to Igor Pro 7, QuickTime was an option on Windows.

Parameters

The file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If

NewMovie can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/A	This flag is obsolete. On Windows prior to Igor Pro 7, /A specified AVI rather than QuickTime. Now AVI is always used on Windows.
/CTYPE=*typeStr*	Specifies the compression codec to use. /CTYP is supported on Macintosh only. It is ignored on Windows but may be supported in the future. *typeStr* is a 4 character case-sensitive string that specifies a compression codec. If you omit /CTYP or if *typeStr* is "", the default is "rpza" unless you provide a file name extension of either "mp4" or "m4v" in which case the "mp4v" Apple MPEG4 Compressor is used. Common compressors in order of increasing file size when used with the FM Modulation Movie example experiment include "smc ", "avc1", "png ", "rpza", "h263", "icod", "jpeg", "mp4v" and "rle ". Be sure to test on a small example first. If the type you specify is invalid or not available on your machine, the resulting file will be small and not playable. /CTYP was added in Igor Pro 7.00.
/F=*frameRate*	Frames per second between 1 and 60. Defaults to 10.
/I	Presents a system-provided dialog in which you can change the compression settings. The selections you make become the new default settings but only until you quit Igor Pro. As of Igor Pro 7, /I is ignored on Macintosh due to changes by Apple. Use the /CTYP flag instead.
/L[=*flatten*]	/L is obsolete as of Igor Pro 7 and is ignored. Movies are always created flattened.
/O	Overwrite existing file, if any.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/PICT=*pictName*	Uses the specified picture (see **Pictures** on page III-448) rather than the top graph.
/S=*soundWave*	Creates and defines sound track. The specified wave can be either a full-range 16 bit or 8 bit integer type. Floating point waves can also be used and are assumed to contain values from -128 to +127. The wave's time/point, as determined by its X scaling, must be between 1.5625e-5 to 2e-4 which correspond to sampling rates of 5000 to 64000 hertz. The duration should match the duration of a video frame. As of Igor Pro 7, /S is no longer supported on Macintosh due to changes by Apple.
/Z	No error reporting; an error is indicated by nonzero value of the output variable V_flag. If the user clicks the cancel button in the Save File dialog, V_flag is set to -1.

Details

If either the path or the file name is omitted then NewMovie displays a Save File dialog to let you create a movie file. If both are present, NewMovie creates the file automatically.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-21 for details.

There can be only one open movie at a time.

The target window at the time you invoke NewMovie must be a graph (unless the /PICT flag is present) and the graph size should remain constant while adding frames to the movie. The graph and optional sound wave are used to determine the size and sound properties only; they do not specify the first frame.

In Igor7 or later, the target window at the time you call NewMovie is remembered and is used by AddMovieFrame even if it is not the target window when you call AddMovieFrame.

The /PICT flag allows you to create a movie from a page layout in conjunction with the SavePICT/P=_PictGallery_ method. See **SavePICT** on page V-704. This allows creation of a movie from a source other than a graph, page layout or Gizmo window, but is rarely needed.

See Also

Movies on page IV-230.

The **AddMovieFrame**, **AddMovieAudio**, **CloseMovie**, **PlayMovie**, **PlayMovieAction** and **SavePICT** operations.

NewNotebook

`NewNotebook [`*`flags`*`] [as `*`titleStr`*`]`

The NewNotebook operation creates a new notebook document.

Parameters

The optional *titleStr* is a string containing the title of the notebook window.

Flags

/HOST=*hcSpec*	Embeds the new notebook in the host window or subwindow specified by *hcSpec*. The host window or subwindow must be a control panel. Graphs and page layouts are not supported as hosts for notebook subwindows. When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy. See **Notebooks as Subwindows in Control Panels** on page III-86 for more information.
/ENCG=*textEncoding*	

textEncoding specifies the text encoding for the new notebook. This determines the text encoding used for later saving the notebook to a file.

See **Text Encoding Names and Codes** on page III-434 for a list of accepted values for textEncoding.

This flag was added in Igor Pro 7.00.

This flag is relevant for plain text notebooks only and has no effect for formatted notebooks because formatted text notebooks can contain multiple text encodings. See **Plain Text File Text Encodings** on page III-417 and **Formatted Text Notebook File Text Encodings** on page III-421 for details.

If you omit /ENCG or pass 0 (unknown) for *textEncoding*, the notebook's text encoding is determined by the default text encoding - see **The Default Text Encoding** on page III-415 for details.

For most purposes, UTF-8 (*textEncoding*=1) is recommended. Other values are available for compatibility with software that requires a specific text encoding. This includes Igor Pro 6 which uses MacRoman (*textEncoding*=2), Windows-1252 (*textEncoding*=3) or Shift-JIS (*textEncoding*=4) depending on the operating system and localization.

This flag has an optional form that allows you to control whether the byte order mark is written when the notebook is later saved to disk. It applies to Unicode text encodings also. The form is:

```
/ENCG = {textEncoding, writeBOM }
```

If you use the simpler form or omit /ENCG entirely, the notebook's writeBOM property defaults to 1.

See **Byte Order Marks** on page III-420 for background information.

/F=*format* Specifies the format of the notebook:

format=0:	Normal with dialog (default).
format=1:	Kills with no dialog.
format=-1:	Disables killing.

/K=*k* Specifies window behavior when the user attempts to close it.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.
k=3:	Hides the window.

If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.

/N=*winName* Sets the notebook's window name to *winName*.

/OPTS=*options* Sets special options. *options* is a bitwise parameter interpreted as follows:

Bit 0:	Hide the vertical scroll bar.
Bit 1:	Hide the horizontal scroll bar.
Bit 2:	Set the write-protect icon initially to on.
Bit 3:	Sets the changeableByCommandOnly bit. When set, the user can not make any modifications.

All other bits are reserved and must be set to zero.

If /OPTS is omitted, all bits default to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/V=*visible* Specifies whether the notebook window is visible (*visible*=1; default) or invisible (*visible*=0).

/W=(*left*,*top*,*right*,*bottom*)	
	Sets window location. Coordinates are in points for normal notebook windows.
	When used with the /HOST flag, the specified location coordinates can have one of two possible meanings:
	When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.
	When any value is greater than 1, coordinates are taken to be fixed locations measured in points relative to the top left corner of the host frame.

Details

A notebook has a file name, a window name, *and* a window title. In the simplest case these will all be the same.

The file name is the name by which the operating system identifies the notebook once it is saved to disk. When you initially create a notebook, it is not associated with any file. However it still has a file name. This is the name that will be used when the file is saved to disk.

The window name is the name by which Igor identifies the window and therefore the name you specify in operations that act on the notebook.

The window title is what appears in the window's title bar. If you omit the title, NewNotebook uses a default title that is the same as the window name.

If you specify the window name and the notebook format and omit the window title, this is the simplest case. NewNotebook creates the document with no user interaction. The file name, window name and window title will all be the same. For example:

```
NewNotebook/N=Notebook1/F=0
```

If you omit the window name, NewNotebook chooses a default name (e.g., "Notebook0") and presents the standard New Notebook dialog.

If you omit the format or specify a format of -1 (either plain or formatted text), NewNotebook presents the standard New Notebook dialog. For example:

```
NewNotebook/N=Notebook1        // no format specified
```

See Also

The **Notebook** and **OpenNotebook** operations, and Chapter III-1, **Notebooks**.

Notebooks as Subwindows in Control Panels on page III-86.

NewPanel

`NewPanel [`*`flags`*`] [as `*`titleStr`*`]`

The NewPanel operation creates a control panel window or subwindow, which may contain Igor controls and drawing objects.

Flags

/EXT=*e*	Creates an exterior subwindow in combination with /HOST. *e* specifies the host window side location:	
	e=0:	Right.
	e=1:	Left.
	e=2:	Bottom.
	e=3:	Top.
/FG=(*gLeft*, *gTop*, *gRight*, *gBottom*)		

	Specifies the frame guide to which the outer frame of the subwindow is attached inside the host window.
	The standard frame guide names are FL, FR, FT, and FB, for the left, right, top, and bottom frame guides, respectively, or user-defined guide names as defined by the host. Use * to specify a default guide name.
	Guides may override the numeric positioning set by /W.
/FLT[=*f*]	/FLT or /FLT=1 makes the panel a floating panel.
	/FLT=2 makes it a floating panel with no close box.
	/FLT=0 is the same as omitting /FLT and creates a regular (non-floating) control panel.
	You must execute the following after the NewPanel command:
	`SetActiveSubwindow _endfloat_`
	See **Floating Panels** below for further information.
/HIDE=*h*	Hides (h = 1) or shows (h = 0, default) the window.
/HOST=*hcSpec*	Embeds the new control panel in the host window or subwindow specified by *hcSpec*.
	When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/I	Sets coordinates to inches.
/K=*k*	Specifies window behavior when the user attempts to close it.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.
k=3:	Hides the window.

If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.

Exterior subwindows never display a dialog when killed.

/M	Sets coordinates to centimeters.
/N=*name*	Requests that the created panel have this name, if it is not in use. If it is in use, then *name*0, `name`1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen panel name. Use `DoWindow/K` *`name`* to ensure that *name* is available.
	Note that a function or macro with the same name will cause a name conflict.
/NA= *n*	Sets panel no-activate mode.

n=0:	Normal (default).
n=1:	Button click doesn’t activate window but click outside of any control does.
n=2:	No activation even if click is outside controls. Title bar clicks still activate.

/W=(*left*,*top*,*right*,*bottom*)

Sets the initial coordinates of the panel window. The coordinates are in points unless /I or /M are used before /W.

When used with the /HOST flag, the specified location coordinates of the sides can have one of two possible meanings:

When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.

When any value is greater than 1, coordinates are taken to be fixed locations measured in points relative to the top left corner of the host frame.

When the subwindow position is fully specified using guides (using the /HOST or /FG flags), the /W flag may still be used although it is not needed.

Details

If /N is not used, NewPanel automatically assigns to the panel a window name of the form "Panel*n*", where *n* is some integer. In a function or macro, the assigned name is stored in the S_name string. This is the name you can use to refer to the panel from a procedure. Use the **RenameWindow** operation to rename the panel.

On Windows there are special considerations relating to screen resolution and control panels. See **Control Panel Resolution on Windows** on page III-405 for details.

Floating Panels

Floating control panels float above all other windows except dialogs. Because floating panels cover up other windows, you should use them sparingly and you should take care to make them small and unobtrusive.

Floating panels are not resizable by default. To allow panel resizing use

```
ModifyPanel fixedSize=0
```

Because floating panels always act as if they are on top, the standard rules for target windows and keyboard focus do not apply.

Normally, a floating panel is never the target window and control procedures will need to explicitly designate the target. But a newly-created floating panel is the default target and will remain so until you execute

```
SetActiveSubwindow _endfloat_
```

It also becomes the default target when the tools are showing and in any non-Operate mode. Similarly, a floating panel with tools not in Operate mode has keyboard focus. To avoid confusion, do not attempt to work on other windows when a floating panel is the default target.

When working with a floating panel, you can show or hide tools or create a recreation macro by Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the panel.

A floating panel does not have keyboard focus. However, a floating panel gains keyboard focus when a control that needs focus is clicked. Focus remains until you press Enter or Escape for a text entry in a setvariable, press Tab until no control has the focus, or until you click outside a focusable control.

On Macintosh, if a floating panel has focus and you activate another window, focus will leave the panel. However on Windows, if a floating panel has focus and you activate another window, the activate sequence will be fouled up leaving the windows in an indeterminate state. Consequently, it is important that you always finish any keyboard interaction started in a floating panel before moving on to other windows. If this can cause confusion, you should not use controls such as SetVariable and ListBox in a floating panel.

On Macintosh, floating panels are hidden when dialogs are up or when Igor Pro is not the front application.

Exterior Subwindows

Exterior subwindows are automatically positioned along the designated side of a host graph, table or panel window. You can designate fixed sizes or automatic size with minima. Subwindows are stacked beside the designated side in their creation order with the first one closest.

Subwindow dimensions have various meanings depending on their location. Interior values are taken to be additional grout, exterior values are taken to be sizes. For left or right panels, top is taken to be the minimum height and bottom, if not zero, is height. For top and bottom, left is taken to be the minimum width and right, if not zero, is width. Zero values default to 50 for width and height or size of host.

Exterior subwindows are nonresizable by default. Use `ModifyPanel fixedSize=0` to allow manual resizing. If you resize a panel, the original window dimensions are lost. You can also use **MoveSubwindow** to resize the subwindow.

Unlike normal subwindows, exterior subwindows have a tools palette. Click in the window and then choose the Show Tools or Hide Tools menu item.

Exterior subwindows have hook functions independent of the host window.

Examples

In a new experiment, execute these commands on the command line to create two exterior subwindows:

```
Display
// Create panel on right with min height of 200 points, width of 100.
NewPanel/HOST=Graph0/EXT=0/W=(0,200,100,0)
// Create another panel on right with grout of 10 and height= width= 100.
NewPanel/HOST=Graph0/EXT=0/W=(10,0,100,100)
```

Now try resizing and moving the graph.

For a demonstration of how the various exterior panels work, copy the following code to the procedure window in a new experiment:

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

   switch( ba.eventCode )
      case 2:                                  // mouse up
         ControlInfo/W=$ba.win ckUseRect
         Variable useR= V_Value
         ControlInfo/W=$ba.win popSide
         Variable side= V_Value-1
         ControlInfo/W=$ba.win ckResizeable
         Variable resizeable= V_Value
         WAVE w=root:epsizes
         if( useR )
            NewPanel/HOST=$ba.win/EXT=(side)/W=(w[0],w[1],w[2],w[3])
         else
            NewPanel/HOST=$ba.win/EXT=(side)
         endif
         if( resizeable )
            ModifyPanel fixedSize=0 // default is 1 for floating and exterior sw
         endif
         break
   endswitch

   return 0
End

Window ExSwTest() : Graph
   PauseUpdate; Silent 1              // building window...
   Display /W=(803,377,1158,591)
   Button bNewSW,pos={35,21},size={181,30},proc=bpNewExSw,title="Exterior Subwindow"
   SetVariable svLeft,pos={118,82},size={96,15},title="left"
   SetVariable svLeft,limits={0,100,1},value= epsizes[0],bodyWidth= 76
   SetVariable svTop,pos={120,97},size={94,15},title="top"
   SetVariable svTop,limits={0,100,1},value= epsizes[1],bodyWidth= 76
   SetVariable svRight,pos={112,113},size={102,15},title="right"
   SetVariable svRight,limits={0,100,1},value= epsizes[2],bodyWidth= 76
   SetVariable svBottom,pos={103,129},size={111,15},title="bottom"
   SetVariable svBottom,limits={0,100,1},value= epsizes[3],bodyWidth= 76
   CheckBox ckUseRect,pos={70,62},size={61,14},title="Use Rect:",value= 0
   PopupMenu popSide,pos={73,149},size={78,20},title="Side"
   PopupMenu popSide,mode=1,popvalue="Right",value= #"\"Right;Left;Bottom;Top\""
   CheckBox ckResizeable,pos={76,176},size={65,14},title="Resizeable",value= 0
EndMacro

Function test()
   Make/O/N=4 epsizes=0
   Execute "ExSwTest()"
End
```

After compiling the procedures, execute `test()` on the command line. You can now experiment with different sides and size values.

See Also

Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

The **ModifyPanel** operation.

NewPath

`NewPath` `[`***`flags`***`]` ***`pathName`*** `[`***`, pathToFolderStr`***`]`

The NewPath operation creates a new symbolic path name that can be used as a shortcut to refer to a folder on disk.

Parameters

pathToFolderStr is a string containing the path to the folder for which you want to make a symbolic path. *pathToFolderStr* can also point to an alias (*Macintosh*) or shortcut (*Windows*) for a folder.

If you use a full path for *pathToFolderStr*, see **Path Separators** on page III-401 for details on forming the path. If you use a partial path or just a simple name for *pathToFolderStr*, and you use the /C flag, a new folder is created relative to the Igor Pro 7 folder. No dialog is presented.

If you omit *pathToFolderStr*, you get a chance to select a folder or create a new folder from a dialog.

Flags

/C	Create the folder specified by *pathToFolderStr* if it does not already exist.
/M=*messageStr*	Specifies the prompt message in the dialog. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/O	Overwrites the symbolic path if it exists.
/Q	Suppresses printing path information in the history.
/Z	Doesn’t generate an error if the folder does not exist.

Details

Symbolic paths help to isolate your experiments from specific file system paths that contain files created or used by Igor. By using a symbolic path, if the actual location or name of the folder changes, you won’t need to change all of your commands. Instead, you need only to change the symbolic path so that it points to the changed folder location.

NewPath sets the variable V_flag to zero if the operation succeeded or to nonzero if it failed. The main use for this is to determine if the user clicked Cancel when you use NewPath to display a choose-folder dialog.

On the Macintosh, pressing Command-Option as the Choose Folder dialog comes up will allow you to choose package folders and folders inside packages.

Examples

```
NewPath Path1, "hd:IgorStuff:Test 1"                // Macintosh
NewPath Path1, "C:IgorStuff:Test 1"                 // Windows
```

creates the symbolic path named Path1 which refers to the specified folder (the path’s “value”). You can then refer to this folder in many Igor operations and dialogs by using the symbolic path name Path1.

Windows Note:

You can use either the colon or the backslash character to separate folders. However, the backslash character is Igor’s escape character in strings. This means that you have to double each backslash to get one backslash like so:

```
NewPath stuff, "C:\\IgorStuff\\Test 1"
```

Because of this complication, it is recommended that you use Macintosh path syntax even on Windows. See **Path Separators** on page III-401 for details.

See Also

The **PathInfo** operation; especially if you need to preset a starting path for the dialog.

KillPath

NewWaterfall

`NewWaterfall` `[`***`flags`***`]` ***`mwave`*** `[`**`vs {`*`wavex,wavez`*`}`**`]`

The NewWaterfall operation creates a new waterfall plot window or subwindow using each column in the 2D matrix wave, *mwave*, as a waterfall trace.

You can manually set x and z scaling by specifying *wavex* and *wavez* to override the default scalings. Either *wavex* or *wavez* may be omitted by using a "*".

Flags

/FG=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the frame guide to which the outer frame of the subwindow is attached inside the host window.

The standard frame guide names are FL, FR, FT, and FB, for the left, right, top, and bottom frame guides, respectively, or user-defined guide names as defined by the host. Use * to specify a default guide name.

Guides may override the numeric positioning set by /W.

/HIDE=*h* Hides (h = 1) or shows (h = 0, default) the window.

/HOST=*hcSpec* Embeds the new waterfall plot in the host window or subwindow specified by *hcSpec*.

When identifying a subwindow with *hcSpec*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

/I Sets window coordinates to inches.

/K=*k* Specifies window behavior when the user attempts to close it.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.
k=3:	Hides the window.

If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.

/M Sets window coordinates to centimeters.

/N=*name* Requests that the created waterfall plot window have this name, if it is not in use. If it is in use, then *name*0, *name*1, etc. are tried until an unused window name is found. In a function or macro, S_name is set to the chosen name. Use `DoWindow/K` *`name`* to ensure that *name* is available.

/PG=(*gLeft*, *gTop*, *gRight*, *gBottom*)

Specifies the inner plot rectangle of the waterfall plot subwindow inside its host window.

The standard plot rectangle guide names are PL, PR, PT, and PB, for the left, right, top, and bottom plot rectangle guides, respectively, or user-defined guide names as defined by the host. Use * to specify a default guide name.

Guides may override the numeric positioning set by /W.

/W=(*left*,*top*,*right*,*bottom*)

Specifies window size. Coordinates are in points unless /I or /M is specified before /W.

When used with the /HOST flag, the specified location coordinates of the sides can have one of two possible meanings:

When all values are less than 1, coordinates are assumed to be fractional relative to the host frame size.

When any value is greater than 1, coordinates are taken to be fixed locations measured in points relative to the top left corner of the host frame.

When the subwindow position is fully specified using guides (using the /HOST, /FG, or /PG flags), the /W flag may still be used although it is not needed.

Details

The X and Z axes are always at the bottom and left, whereas the Y axis runs at a default 45 degrees along the right-hand side. The angle and length of the Y axis can be changed using the ModifyWaterfall operation. Other features of the graph can be changed using normal graph operations.

Each column from *mwave* is plotted in (and clipped by) a rectangle defined by the X and Z axes with the rectangle displaced along the angled Y axis as a function of the y value.

Except when hidden lines are active, the traces are drawn from back to front.

To modify certain properties of a waterfall plot, you need to use the ModifyWaterfall operation. For other properties, use the usual axis and trace dialogs.

See Also

Waterfall Plots on page II-255.

The **ModifyWaterfall** and **ModifyGraph** operations.

norm

`norm(`*`srcWave`*`)`

The norm function evaluate the norm of *srcWave*. It returns:

$$\sqrt{\sum abs\left(w[i]\right)^2}$$

This function does not support text waves.

See Also

MatrixOp

NormalizeUnicode

`NormalizeUnicode(`*`sourceTextStr, normalizationForm`*`[, `*`options`*`])`

The NormalizeUnicode function normalizes the UTF-8-encoded text in sourceTextStr using the specified normalization form. The output text encoding is UTF-8.

NormalizeUnicode was added in Igor Pro 7.00. Most users will have no need for this function and can ignore it.

As explained under Details, in Unicode there are sometimes multiple ways to spell what appears visually to be the same word. This can cause problems when comparing text. Two strings that appear to represent the same word and which you consider equivalent may be spelled differently, causing a comparison operation to indicate that they are unequal. The NormalizeUnicode function converts *sourceTextStr* to a normalized form, which aides comparison.

Parameters

sourceTextStr is the text that you want to normalize. It must be encoded as UTF-8.

normalizationForm specifies the normalization form to use. These forms are described at http://unicode.org/reports/tr15/#Norm_Forms. The allowed values are:

0:	NFD (Canonical Decomposition)

1:	NFC (Canonical Decomposition, followed by Canonical Composition)
2:	NFKD (Compatibility Decomposition)
3:	NFKC (Compatibility Decomposition, followed by Canonical Composition)

options is a bitwise parameter, with the bits defined as follows:

Bit 0: If cleared, in the event of an error, a null string is returned and an error is generated. Use this if you want to abort procedure execution if an error occurs.

If set, in the event of an error, a null string is returned but no error is generated. Use this if you want to detect and handle an error yourself. You can test for null using strlen as shown in **String Variable Text Encoding Error Example** on page III-428.

All other bits are reserved and must be cleared.

Details

The Unicode standard specifies that some sequences of code points represent essentially the same character. There are two types of equivalence: canonical equivalence and compatibility.

Sequences of code points defined as canonically equivalent are assumed to have the same appearance and meaning when printed or displayed. For example, the code point U+006E (LATIN SMALL LETTER N) followed by U+0303 (COMBINING TILDE) is defined by Unicode to be canonically equivalent to the single code point U+00F1 (LATIN SMALL LETTER N WITH TILDE). The former is called "decomposed" while the later is called "precomposed".

Sequences that are defined as compatible are assumed to have possibly distinct appearances, but the same meaning in some contexts. Thus, for example, the code point U+FB00 (LATIN SMALL LIGATURE FF) is defined to be compatible, but not canonically equivalent, to the sequence U+0066 U+0066 (two Latin "f" letters). Sequences that are canonically equivalent are also compatible, but the opposite is not necessarily true.

Text searching and sorting routines in Igor do not do any form of Unicode normalization. As a consequence, searching for the precomposed form of small letter n with tilde (U+00F1) in a string that contains the decomposed form (U+006E U+0303) will not result in a match. To get the desired result, you would need to first pass both the target string and the string to be searched through NormalizeUnicode using the same value for the *normalizationForm* parameter.

Example

```
Function TestNormalizeUnicode()
	String precomposed = "Ni" + "\u00F1" + "o"
	String decomposed = "Ni" + "\u006E\u0303" + "o"
	String precomposedTarget = "\u00F1"
	String decomposedTarget = "\u006E\u0303"
	Variable foundPos

	// SUCCESSFUL TESTS
	// Searching the precomposed string for the precomposed target is successful.
	foundPos = strsearch(precomposed, precomposedTarget, 0)
	Print foundPos                // Prints 2

	// Likewise, searching the decomposed string for the decomposed target is successful.
	foundPos = strsearch(decomposed, decomposedTarget, 0)
	Print foundPos                // Prints 2

	// UNSUCCESSFUL TESTS
	// Searching the precomposed string for the decomposed target fails.
	foundPos = strsearch(precomposed, decomposedTarget, 0)
	Print foundPos                // Prints -1

	// Likewise, searching the decomposed string for the precomposed target fails.
	foundPos = strsearch(decomposed, precomposedTarget, 0)
	Print foundPos                // Prints -1

	// USING NormalizeUnicode() FUNCTION
	Variable normForm = 2         // Could use 0-3 and the results would be the same.
```

```
    String precomposedNorm = NormalizeUnicode(precomposed, normForm)
    String decomposedNorm = NormalizeUnicode(decomposed, normForm)
    String precomposedTargetNorm = NormalizeUnicode(precomposedTarget, normForm)
    String decomposedTargetNorm = NormalizeUnicode(decomposedTarget, normForm)

    // Now, searching either precomposedNorm or decomposedNorm for either
    // precomposedTargetNorm or decomposedTargetNorm will give a match.
    Print strsearch(precomposedNorm, precomposedTargetNorm, 0)   // Prints 2
    Print strsearch(decomposedNorm, precomposedTargetNorm, 0)    // Prints 2
    Print strsearch(precomposedNorm, decomposedTargetNorm, 0)    // Prints 2
    Print strsearch(decomposedNorm, decomposedTargetNorm, 0)     // Prints 2
End
```

See Also

Text Encodings on page III-409, **String Variable Text Encoding Error Example** on page III-428

http://en.wikipedia.org/wiki/Unicode_equivalence

http://unicode.org/reports/tr15/#Norm_Forms

note

`note(`*`waveName`*`)`

The note **function** returns a string containing the note associated with the specified wave.

See Also

To create a wave note, use the **Note** *operation*.

Note

`Note [/K/NOCR]` ***`waveName`*** **`[,`** ***`str`*** **`]`**

The Note operation appends *str* to the wave note for the named wave.

Parameters

str is a string expression.

Flags

/K	Kills existing note for specified wave.
/NOCR	Appends note without a preceding carriage return (\r character). No effect when used with /K.

Examples

```
Note/K wave0        // remove existing note
Note wave0, "This is the first line of the note"
Note wave0, "This is the second line of the note"
Note/K wave0, "This is now the only line of the note"
```

See Also

To get the contents of a wave note, use the **note** *function*.

Notebook

`Notebook` ***`winName, keyword=value`*** **`[,`** ***`keyword=value`*** **`]…`**

The Notebook operation sets various properties of the named notebook window. Notebook also inserts text and graphics. See Chapter III-1, **Notebooks**, for general information on notebooks.

Notebook returns an error if the notebook is open for read-only. Keywords that don't materially change the notebook, including findText, findPicture, selection, visible, magnification, userKillMode, showRuler and rulerUnits, are still permitted. See **Notebook Read/Write Properties** on page III-10 for further information.

Parameters

winName is either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-87 for details on host-child specifications.

If *winName* is an hcSpec, the host window or subwindow must be a control panel. Graphs and page layouts are not supported as hosts for notebook subwindows.

The parameters to the Notebook operation are of the form *keyword*=*value* where *keyword* says what to do and *value* is a parameter or list of parameters. Igor limits the parameters that you specify to legal values before applying them to the notebook.

The parameters are classified into related groups of *keyword*s.

See Also

To create or modify a notebook action special character, see **NotebookAction**.

To create a notebook subwindow in a control panel, see **Notebooks as Subwindows in Control Panels** on page III-86.

Notebook (*Document Properties*)

Notebook document property parameters

This section of Notebook relates to setting the document properties of the notebook.

adopt=*a* — Adopts a notebook if it is a file saved to disk. Adopting a notebook makes it part of the packed experiment file, which becomes more self-contained; if you send the experiment to a colleague you will not need to send a notebook file.

a=0:	Checks only whether the notebook is adoptable. Sets V_flag to 0 if the notebook is already adopted or to 1 if it is adoptable.
a=1:	Checks only whether the notebook is adoptable. Sets V_flag to 0 if the notebook is already adopted or to 1 if it is adoptable.

backRGB=(*r*,*g*,*b*) — Sets background color. *r*, *g*, and *b* are integers from 0 to 65535.

changeableByCommandOnly=*c*

This changeableByCommandOnly property is used to prevent manual modifications to the notebook but allow modifications using commands.

c=0:	Turn changeableByCommandOnly off.
c=1:	Turn changeableByCommandOnly on.

See **Notebook Read/Write Properties** on page III-10 for details.

defaultTab=*dtw* — *dtw* is the default tab width in points.

magnification=*m* — Specifies the desired magnification in percent (between 25 and 500). Otherwise, *m* can be one of these special values:

m=1:	Default magnification.
m=2:	Default magnification. In Igor Pro 6 this specified the no-longer-supported Fit Width mode.
m=3:	Default magnification. In Igor Pro 6 this specified the no-longer-supported Fit Page mode.

pageMargins={*left*, *top*, *right*, *bottom*}

Sets page margins in points. *left*, *top*, *right*, and *bottom* are distances from the respective edges of the physical page.

rulerUnits=*r* — Sets the units for the ruler:

r=0:	Points.
r=1:	Inches.
r=2:	Centimeters.

showRuler=*s* — Hides (*s*=0) or shows (*s*=1) the ruler.

startPage=*sp* — Sets the starting page number for printing.

statusWidth=*sw* — As of Igor7, because of changes to the layout of notebook windows, this keyword does nothing.

In Igor6 it set the width in points of the status area on the left of the horizontal scroll bar.

userKillMode=*k* — Specifies window behavior when the user attempts to close it.

k=0:	Normal with dialog (default).
k=1:	Clicking the close button kills the notebook with no dialog.
k=2:	Clicking the close button does nothing.
k=3:	Clicking the close button hides the notebook with no dialog.

writeBOM=*w*	Sets the document's writeBOM property which determines if Igor writes a byte order mark when saving the notebook. This applies to plain text notebooks only and is ignored for formatted text notebooks.
	w=-1: Does not change writeBOM flag.
	w=0: Sets writeBOM to false.
	w=1: Sets writeBOM to true.
	See **Byte Order Marks** on page III-420 for details.
	This keyword was added in Igor Pro 7.00.
writeProtect=*wp*	The write-protect property is used to prevent inadvertent manual changes to the notebook.
	wp=0: Turn write-protect off.
	wp=1: Turn write-protect on.
	See **Notebook Read/Write Properties** on page III-10 for details.

Notebook (*Headers and Footers*)

Notebook headers and footers

You can turn headers and footers on and off and position headers and footers using the keywords in this section.

There is currently no way to set the content of headers and footers except manually through the Document Settings dialog. You may be able to use stationery files to create files with specific headers and footers.

footerControl={*defaultFooter*, *firstFooter*, *evenOddFooter*}

defaultFooter is 1 to turn the default footer on, 0 to turn it off.

firstFooter is 1 to turn the first page footer on, 0 to turn it off.

evenOddFooter is 1 to turn different footers for even and odd pages on, 0 to use the same footer for even and odd pages.

footerPos=*pos* *pos* is the position of the footer relative to the bottom of the page in points.

headerControl={*defaultHeader* , *firstHeader* , *evenOddHeader*}

defaultHeader is 1 to turn the default header on, 0 to turn it off.

firstHeader is 1 to turn the first page header on, 0 to turn it off.

evenOddHeader is 1 to turn different headers for even and odd pages on, 0 to use the same header for even and odd pages.

headerPos=*pos* *pos* is the position of the header relative to the top of the page in points.

Notebook (*Miscellaneous*)

Notebook miscellaneous parameters

This section of Notebook relates to setting miscellaneous properties of the notebook.

autoSave=*v*	Controls auto-save mode. *v*=0: Notebook subwindow contents will not be saved in recreation macros. *v*=1: Notebook subwindow contents will be saved in recreation macros (default). This affects notebook subwindows in control panels only. Use autoSave=0 if you do not want the notebook's contents to be saved and restored when the control panel is recreated. Otherwise the notebook subwindow's contents will be restored when recreated.
frameInset= *i*	Specifies the number of pixels by which to inset the frame of a notebook subwindow. Does not affect a normal notebook window. This keyword was added in Igor Pro 7.00.
frameStyle= *f*	Specifies the frame style for a notebook subwindow. Does not affect a normal notebook window. *f*=0: None. *f*=1: Single. *f*=2: Double. *f*=3: Triple. *f*=4: Shadow. *f*=5: Indented. *f*=6: Raised. *f*=7: Text well. The last three styles are fake 3D and will look best if the background color behind the subwindow is a light shade of gray. This keyword was added in Igor Pro 7.00.
status={*messageStr*, *flags*}	Sets the message in the status area at the bottom left of the notebook window. *flags* is interpreted bitwise. Message is erased when: Bit 0: Selection changes. Bit 1: Window is activated. Bit 2: Window is deactivated. Bit 3: Document is modified. If all bits are zero, the message stays until a new message comes along. All other bits are reserved for future use and should be zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.
updating={*flags*, *r*}	Sets parameters related to the updating of special characters.

flags is interpreted bitwise:

Bit 0:	Suppress automatic periodic updating of date and time special characters. By default this bit is set so date and time special characters are updated only when the user explicitly requests it or during printing when they appear in headers and footers.
Bit 1:	Allow manual updating of special characters via the specialUpdate keyword or via the Special menu. By default this is cleared so manual updating is not allowed.

All other bits are reserved for future use and should be zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

r is the update rate in seconds for updating date and time special characters.

These settings have no effect on the updating of special characters in headers or footers. These characters are always automatically updated when the document is printed.

We recommend that you leave automatic updating off (set bit 0 of the *flags* parameter to 1) so that updating occurs only via the specialUpdate keyword or via the Special menu.

visible=*v* Sets notebook visibility.

v=0:	Hides notebook.
v=1:	Shows notebook but does not make it top window.
v=2:	Shows notebook and makes it top window.

Notebook (*Paragraph Properties*)

Notebook paragraph property parameters

This section of Notebook relates to setting the paragraph properties of the current selection in the notebook.

The margins, spacing, justification, tabs and rulerDefaults keywords provide control over paragraph properties which are governed by rulers. These keywords, in conjunction with the ruler and newRuler keywords, allow you to set paragraph properties. They are allowed for formatted text notebooks only, not for plain text notebooks.

The ruler keywords are described in detail below. Before we get to the detail, you should understand the different things you can do with rulers.

There are four things you can do with a ruler:

modify it	(analogous to manually adjusting a ruler).
redefine it	(analogous to the Redefine Ruler dialog).
create it	(analogous to the Define New Ruler dialog).
apply it	(analogous to selecting a ruler name from Ruler pop-up menu).

Igor's behavior in response to ruler keywords depends on the order in which the keywords appear.

To modify the ruler(s) for the selected paragraph(s), use the margins, spacing, justification, tabs and rulerDefaults keywords *without* using the newRuler or ruler keywords. For example:

```
Notebook Notebook0 tabs={36,144,288},justification=1
```

To redefine an existing ruler, invoke the ruler=*rulerName* keyword *before* any other keywords. For example:

```
Notebook Notebook0 ruler=Ruler1,tabs={36,144,288},justification=1
```

Unlike redefining the ruler manually, when you redefine an existing ruler using ruler=*rulerName*, it does not apply the ruler to the selected text. However, it does update any text governed by the redefined ruler.

To create a new ruler, invoke the newRuler=*rulerName* keyword *before* any other keywords. For example:

```
Notebook Notebook0 newRuler=Ruler1,tabs={36,144,288},justification=1
```

Unlike creating it manually, when you create a new ruler using newRuler=*rulerName*, it does not apply the new ruler to the selected text. If you do not set a particular ruler property when creating a new ruler, the property will be the same as for the Normal ruler. If the specified ruler already exists, newRuler=*rulerName* overwrites the existing ruler.

To apply an existing ruler to the selected text, invoke the ruler=*rulerName* keyword without any other keywords. For example:

```
Notebook Notebook0 ruler=ruler1
```

You and Igor will get confused if you mix ruler keywords with other types of keywords in the same command. It is alright, however to put a selection keyword at the start of the command. Mixing will not cause a crash or any drastic problem but it will likely produce results that you don't understand.

To keep things clear, follow these rules:

- If you use ruler=*rulerName* or newRuler=*rulerName*, put them before any other ruler keywords.
- Do not mix ruler keywords with other kinds, except that it is alright to use the selection keyword at the start of the command.

justification=*j* Sets text justification:

j=0:	Left aligned.
j=1:	Center aligned.
j=2:	Right aligned.
j=3:	Fully justified.

margins={*indent*,*left*,*right*}

indent sets the indentation of first line from left page margin.

left sets the paragraph's left margin in points measured from the left page margin.

right sets the paragraph's right margin in points measured from the left page margin.

newRuler=*rulerName*

Creates a new ruler with the specified name. If a ruler with this name already exists, it is overwritten.

ruler=*rulerName* Applies the named ruler to the selected text or to redefine the named ruler, as explained above.

rulerDefaults={*"fontName"*, *fSize*, *fStyle*, (*r*,*g*,*b*)}

"fontName" sets the ruler's text font, e.g., `"Helvetica"`.

fSize sets the ruler's text size.

fStyle sets the ruler's text style.

(*r*,*g*,*b*) sets the ruler's text color. *r*, *g*, and *b* are integers from 0 to 65535.

You can only use rulerDefaults if you are redefining an existing ruler, using ruler=*rulerName*, or you are creating a new ruler using newRuler=*rulerName*.

spacing={*spaceBefore*,*spaceAfter*,*lineSpace*}

spaceBefore sets the extra space before paragraph in points.

spaceAfter sets the extra space after paragraph in points.

lineSpace sets the extra space between lines of a paragraph in points.

tabs={*tabSpec*} *tabSpec* is list of tab stops in points added to special values that change the tab stop type.

Tab stops have two parts: the tab stop position and the tab type. Each integer in the list of tabs encodes both of these parts as follows:

The low 11 bits contains the tab stop position in points.

The next two bits are reserved for future use and must be zero.

The high three bits are used to contain the tab type as follows:

left tab	0	
center tab	1	add 1*8192 to tab stop position.
right tab	2	add 2*8192 to tab stop position.
decimal tab	3	add 3*8192 to tab stop position.
comma tab	4	add 4*8192 to tab stop position.

Tabs Example

The following puts a left tab at 1 inch, a center tab at 3 inches and a decimal tab at 5 inches:

```
Notebook Notebook1 tabs={1*72, 3*72 + 8192, 5*72 + 3*8192}
```

Notebook (*Selection*)

Notebook selection parameters

This section of Notebook relates to selecting a range of the content of the notebook.

findPicture={*graphicNameStr*, *flags*}

Searches for the picture containing the named graphic (*Macintosh only*) or the next picture if you pass `""`. Sets V_flag to 1 if the picture was found or to 0 if not found.

flags is a bitwise parameter interpreted as follows:

Bit 0:	Show selection after the find.

All other bits are reserved for future use. Set bit 0 by setting *flags* = 1.

The search is always forward from the end of the current selection to the end of the document.

findSpecialCharacter={*specialCharacterNameStr*, *flags*}

Searches for the special character with the specified name or the next special character if you pass `""`. Selects the special character if it is found.

Sets V_flag to 1 if the special character was found or to 0 if not. Sets S_name to the name of the found special character or to `""` if it was not found.

flags is a bitwise parameter interpreted as follows:

Bit 0:	Show selection after the find.

All other bits are reserved for future use. Set bit 0 by setting *flags* = 1.

If *specialCharacterNameStr* is empty (`""`), the search proceeds from the end of the current selection to the end of the document. Otherwise the search always covers the entire document.

findText={*textToFindStr*, *flags*}

Searches for the specified text. Sets V_flag to 1 if the text was found or to 0 if not found.

textToFindStr is a string expression for the text you want to find. If the text contains a carriage return, Igor considers only the part of the text before the carriage return.

flags is a bitwise parameter interpreted as follows:

Bit 0:	Show selection after the find.
Bit 1:	Do case-sensitive search.
Bit 2:	Search for whole words.
Bit 3:	Wrap around.
Bit 4:	Search backward.

All other bits are reserved and must be set to zero.

To set bit 0 and bit 3, use $2^0+2^3 = 9$ for *flags*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

If you are searching forward, the search starts from the end of the current selection. If you are searching backward, the search starts from the start of the current selection.

If you specify `""` as the text to search for, it “finds” the current selection. This displays the current selection using `findText={"", 1}`.

selection={*selStart*, *selEnd*}

selStart and *selEnd* are locations within the document. You can specify these document locations by using the following expressions:

(*paragraph*, *pos*)	*paragraph* and *pos* are numeric expressions. *paragraph* is a paragraph number from 0 to *n*-1 where *n* is the number of paragraphs in the document. *pos* is a byte position from 0 to *n* where *n* is the number of bytes in the paragraph. Position 0 is to the left of the first character in the paragraph. Position *n* is to the right of the last character in the paragraph.
startOfFile	Start of the document.
endOfFile	End of the document.
startOfParagraph	Start of current *selStart* paragraph.
endOfParagraph	End of current *selStart* paragraph.
startOfNextParagraph	Start of paragraph after current *selStart* paragraph.
endOfNextParagraph	End of paragraph after current *selStart* paragraph.
startOfPrevParagraph	Start of paragraph before current *selStart* paragraph.
endOfPrevParagraph	End of paragraph before current *selStart* paragraph.
endOfChars	Just before the carriage return of current *selStart* paragraph.
startOfPrevChar	Start of the character before the character at the current selection start or selection end. This moves the selection start or selection end like pressing the left arrow key. Added in Igor Pro 7.00.
startOfNextChar	Start of the character after the character at the current selection start or selection end. This moves the selection like pressing the right arrow key. Added in Igor Pro 7.00.

Igor clips the specified locations to legal values. It also sets the V_flag variable to 0 if the *selStart* location that you specified was valid, to 1 if the start paragraph was out of bounds and to 2 if the start position was out of bounds. You can use the startOfNextParagraph keyword to step through the document one paragraph at a time. When V_flag is nonzero, you are at the end of the document.

The terms next and prev are relative to the paragraph containing the start of the selected text before the selection keyword was invoked.

The selection keyword just sets the selection. If you also want to scroll the selected text into view you must also use the findText keyword as shown in the examples.

Selection Examples

Following are some examples of setting the selection:

```
// select all text in notebook
Notebook Notebook1 selection={startOfFile, endOfFile}

// move selection to the start of the notebook and display the selection
Notebook Notebook1 selection={startOfFile,startOfFile}, findText={"",1}

// move selection to the end of the notebook and display the selection
Notebook Notebook1 selection={endOfFile,endOfFile}, findText={"",1}

// select all of paragraph 3
Notebook Notebook1 selection={(3,0), (4,0)}

// select all of paragraph 3 and display the selection
Notebook Notebook1 selection={(3,0), (4,0)}, findText={"",1}

// select all of current paragraph except for trailing CR, if any
Notebook Notebook1 selection={startOfParagraph, endOfChars}
```

```
// select the first occurrence of "Hello" in the document and display the selection
Notebook Notebook1 selection={startOfFile,startOfFile}, findText={"Hello",1}
```

```
// select the first picture in the document
Notebook Notebook1 selection={startOfFile,startOfFile}, findPicture={"",1}
```

See Also

The **GetSelection** operation to “copy” the selection.

Notebook (*Text Properties*)

Notebook text property parameters

This section of Notebook relates to setting the text properties of the current selection in the notebook.

font="*fontName*"	*"fontName"* is the name of the font. Use `"default"` to specify the paragraph's ruler font. If you specify an unavailable font, it does nothing. This is so that, when you share procedures with a colleague, using a font that the colleague does not have will not cause your procedures to fail. The downside of this behavior is that if you misspell a font name you will get no error message.
fSize=*fontSize*	Text size from 3 to 32000 points. Use -1 to specify the paragraph's ruler size.
fStyle=*fontStyle*	A binary coded integer with each bit controlling one aspect of the text style as follows: Bit 0: Bold Bit 1: Italic Bit 2: Underline Bit 4: Strikethrough Use -1 to specify the paragraph's ruler style. To set bit 0 and bit 1 (bold italic), use $2^0+2^1 = 3$ for *fontStyle*. See **Setting Bit Parameters** on page IV-12 for details about bit settings.
textRGB=(*r*,*g*,*b*)	Specifies text color. *r*, *g*, and *b* are integers from 0 to 65535. (0, 0, 0) specifies black. (65535, 65535, 65535) specifies white.
vOffset=*v*	Sets the vertical offset in points (positive offset is down, negative is up). Use this to create subscripts and superscripts. vOffset is allowed for formatted text files only, not for plain text files.

Notebook (*Writing Graphics*)

Writing notebook graphics parameters

This section of Notebook relates to inserting graphics at the current selection in the notebook.

These graphics keywords are allowed for formatted text files only, not for plain text files.

convertToPNG=*x*	Converts all pictures in the current selection to cross-platform PNG format. If the picture is already PNG, it does nothing. *x* is the resolution expansion factor, an integer from 1 to 16 times screen resolution. *x* is clipped to legal limits.
frame=*f*	Sets the frame used for the picture and insertPicture keywords. *f*=0: No frame (default). *f*=1: Single frame. *f*=2: Double frame. *f*=3: Triple frame. *f*=4: Shadow frame.

insertPicture={*pictureName*, *pathName*, *filePath*, *options*}

Inserts a picture from a file specified by *pathName* and *filePath*. The supported graphics file formats are listed under **Inserting Pictures** on page III-13.

pictureName is the special character name (see **Special Character Names** on page III-13) to use for the inserted notebook picture or $"" to automatically assign a name.

pathName is the name of an Igor symbolic path created via **NewPath** or $"" to use no path.

filePath is a full path to the file to be loaded or a partial path or simple file name relative to the specified symbolic path.

If *pathName* and *filePath* do not fully specify a file, an Open File dialog is displayed from which the user can choose the file to be inserted.

options is a bitwise parameter interpreted as follows:

Bit 0:	If set, an Open File dialog is displayed even if the file is fully specified by *pathName* and *filePath*.
Bit 1:	Determines what to do in the event of a name conflict. If set, the existing special character with the conflicting name is overwritten. If cleared, a unique name is created and used as the special character name for the inserted picture.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The variable V_flag is set to 1 if the picture was inserted or to 0 otherwise, for example, if the user canceled from the Open File dialog.

The string variable S_name is set to the special character name of the picture that was inserted or to "" if no picture was inserted.

The string variable S_fileName is set to the full path of the file that was inserted or to "" if no picture was inserted.

picture={*objectSpec*, *mode*, *flags* [, *expansion*]}

Inserts a picture based on the specified object.

objectSpec is usually just an object name, which is the name of a graph, table, page layout, Gizmo plot, or picture from Igor's picture gallery (Misc→Pictures). See further discussion below.

mode controls what happens when you insert a picture of a graph, table or page layout window. It does not affect insertions of pictures from the picture gallery.

mode specifies the format of the graph, table, or page layout picture as follows:

mode	**Macintosh**	**Windows**
-9	SVG	SVG
-8	PDF	8X Enhanced metafile
-7	TIFF	TIFF
-6	JPEG	JPEG
-5	PNG	PNG
-4	4X PNG	Device-independent bitmap
-2	PDF	8X Enhanced metafile
-1	PDF	8X Enhanced metafile
0	PDF	8X Enhanced metafile
1	1X PDF	8X Enhanced metafile
2	2X PDF	8X Enhanced metafile
4	4X PDF	8X Enhanced metafile
8	8X PDF	8X Enhanced metafile

Modes -6, -7, -8, and -9 require Igor Pro 7.00 or later.

Mode 0 is recommended unless you are concerned about cross-platform compatibility in which case you must use mode -5 (PNG) or mode -9 (SVG).

If *objectSpec* names a Gizmo window, only modes -5, -6, or -7 are allowed.

Modes -2 through 8 are supported for backward compatibility. In previous versions of Igor, they selected other formats that are now obsolete.

See Chapter III-5, **Exporting Graphics (Macintosh)**, Chapter III-6, **Exporting Graphics (Windows)**, and **Metafile Formats** on page III-96 for further discussion of these formats.

flags is a bitwise parameter interpreted as follows:

Bit 0:	0 for black and white, 1 for color.

All other bits are reserved and must be set to zero.

For color, set *flags* = 2^0 = 1.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

expansion is optional and requires Igor Pro 7.00 or later. On Macintosh, it affects modes -5, -6, -7, and -8 only. On Windows, it affects modes -5, -6, and -7 only.

expansion sets the expansion factor over screen resolution. *expansion* must be an integer between 1 and 8 and is usually 1, 2, 4 or 8. The default value is 1.

scaling={*h*, *v*}	Sets the horizontal(*h*) and vertical (*v*) scaling for the selected picture or the picture and insertPicture keywords. *h* and *v* are in percent.

When using the picture keyword, you may include a coordinate specification after the object name in *objectSpec*. For example:

```
Notebook Notebook1 picture={Layout0(100, 50, 500, 700), 1, 1}
```

The coordinates are in points. A coordinate specification of (0, 0, 0, 0) behaves the same as no coordinate specification at all.

If the object is a graph, the coordinate specification determines the width and height of the graph. If you omit the coordinate specification, Igor takes the width and height from the graph window.

If the object is a layout, the coordinate specification identifies a section of the layout. If you omit the coordinate specification, Igor selects a section of the layout that includes all objects in the layout plus a small margin.

For any other kind of object, Igor ignores the coordinate specification if it is present.

The scaling and frame keywords affect the selected picture, if any. If no picture is selected, they affect the insertion of a picture using the picture or insertPicture keywords. For example, this command inserts a picture of Graph0 with 50% scaling and a double frame:

```
Notebook Test1 scaling={50, 50}, frame=2, picture={Graph0, 1, 1}
```

If no picture is selected and no picture is inserted, scaling and frame have no effect.

InsertPicture Example

```
Function InsertPictureFromFile(nb)
   String nb           // Notebook name or "" for top notebook

   if (strlen(nb) == 0)
      nb = WinName(0, 16, 1)
   endif

   if (strlen(nb) == 0)
      Abort "There are no notebooks"
   endif

   // Display Open File dialog to get the file to be inserted
   Variable refNum     // Required for Open but not really used
   String fileFilter = "Graphics Files:.eps,.jpg,.png;All Files:.*;"
   Open /D /R /F=fileFilter refNum
   String filePath = S_fileName
```

```
	if (strlen(filePath) == 0)
		Print "You cancelled"
		return -1
	endif

	Notebook $nb, insertPicture={$"", $"", filePath, 0}
	if (V_flag)
		Print "Picture inserted"
	else
		Print "No picture inserted"
	endif

	return 0
End
```

Save notebook pictures to files

The savePicture keyword is allowed for formatted text files only, not for plain text files.

savePicture={*pictureName*, *pathName*, *filePath*, *options*}

Saves a picture from a formatted text notebook to a file specified by *pathName* and *filePath*.

pictureName is the special character name (see **Special Character Names** on page III-13) of the picture to be saved or $"" to save the selected picture in which case one picture and one picture only must be selected in the notebook.

pathName is the name of an Igor symbolic path created via **NewPath** or $"" to use no path.

filePath is a full path to the file to be written or a partial path or simple file name relative to the specified symbolic path.

If *pathName* and *filePath* do not fully specify a file, a Save File dialog is displayed in which the user can specify the file to be written.

options is a bitwise parameter interpreted as follows:

Bit 0:	If set, a Save File dialog is displayed even if the file is fully specified by *pathName* and filePath.
Bit 1:	If set, a file with the same name is overwritten if it exists. If cleared, a Save File dialog is displayed if the specified file already exists.
Bit 2:	If set then the leaf name specified by *filePath* is ignored and a name is automatically generated based on the picture name.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The variable V_flag is set to 1 if the picture was written or to 0 otherwise, for example, if the user canceled from the Save File dialog.

The string variable S_name is set to the special character name of the picture that was saved or to "" if no picture was saved.

The string variable S_fileName is set to the full path of the file that was written or to "" if no picture was written.

Notebook (*Writing Special Characters*)

Writing special character parameters

This section of Notebook relates to inserting special characters at the current selection in the notebook. To insert a notebook action, see **NotebookAction**.

The special characters are page break, short date, long date, abbreviated date and time. They act in some respects like a single character but have special properties. You can insert the special characters using the specialChar keyword.

The specialChar keyword is allowed for formatted text files only, not for plain text files.

Other special characters are allowed in headers and footers only and you can not insert them in a document using the specialChar keyword. These are window title, page number and total pages.

The special characters other than page break character are dynamic and update periodically.

specialChar={*type*, *flags*, *optionsStr*}

type is the special character type as follows:

1:	Page break.
2:	Short date.
3:	Long date.
4:	Abbreviated date.
5:	Time.

flags is reserved for future use. You should pass 0 for *flags*.

optionsStr is reserved for future use. You should pass "" for *optionsStr*.

specialUpdate=*flags*

Updates special characters in the notebook.

flags is interpreted bitwise:

Bit 0:	0 to update all special characters.
Bit 1:	1 to update special characters in the selected text.

If 1, updates regardless of whether updating is enabled or not.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The specialUpdate keyword can update pictures of graphs, tables, and page layouts that were created from windows in the current experiment.

See Also

Chapter III-1, **Notebooks**. The **NewNotebook**, **NotebookAction**, and **OpenNotebook** operations; the **SpecialCharacterInfo** and **SpecialCharacterList** functions.

Notebook (*Accessing Contents*)

Accessing Notebook Contents

getData=*mode* Causes Igor to return the contents of the notebook in the S_value variable. The contents are binary data in a private Igor format encoded as text. The only use for this keyword is to transfer data from one notebook to another by calling getData followed by setData.

Causes Igor to return the contents of the notebook in the S_value variable. The contents are binary data in a private Igor format encoded as text. The only use for this keyword is to transfer data from one notebook to another by calling getData followed by setData.

mode=1:	Stores in S_value plain text or formatted text data, depending on the type of the notebook, from the entire notebook.
mode=2:	Stores in S_value plain text data, regardless of the type of the notebook, from the entire notebook.
mode=3:	Stores in S_value plain text or formatted text data, depending on the type of the notebook, from the notebook selection only.
mode=4:	Stores in S_value plain text data, regardless of the type of the notebook, from the notebook selection only.

See the Notebook In Panel example experiment for examples using getData and setData.

Notebook (*Writing Text*)

Writing notebook text parameters

This section of Notebook relates to inserting text at the current selection in the notebook.

text=*textStr*	Inserts the text at the current selection. Before the text is inserted, Igor converts escape sequences in *textStr* as described in **Escape Sequences in Strings** on page IV-13. Then, it checks for illegal characters. The only character code that is illegal is zero (ASCII NUL character). If it finds an illegal character, Igor generates an error and does not insert the text.
setData=*dataStr*	Inserts the data at the current selection. *dataStr* is either a regular string expression or the result returned by Notebook getData.
zData=*dataStr*	This keyword is used by Igor during the recreation of a notebook subwindow in a control panel. *dataStr* is encoded binary data created by Igor when the recreation macro was generated. It represents the contents of the notebook subwindow in a format private to Igor.
zDataEnd=1	This keyword is used by Igor during the recreation of a notebook subwindow in a control panel. It marks the end of encoded binary data created by Igor when the recreation macro was generated.

NotebookAction

`NotebookAction` `[`**`/W=`*`winName`***`]` ***`keyword = value`*** `[,` ***`keyword = value`*** `…]`

The NotebookAction operation creates or modifies an "action" in a notebook. A notebook action is an object that executes commands when clicked.

See Chapter III-1, **Notebooks**, for general information about notebooks.

NotebookAction returns an error if the notebook is open for read-only. See **Notebook Read/Write Properties** on page III-10 for further information.

Parameters

The parameters are in *keyword* =*value* format. Parameters are automatically limited to legal values before being applied to the notebook.

bgRGB=(*r*, *g*, *b*)	Specifies the action background color. *r*, *g*, and *b* are values from 0 to 65535.
commands=*str*	Specifies the command string to be executed when clicking the action. For multiline commands, add a carriage return (`\r`) between lines.
enableBGRGB=*enable*	
	Uses the background color specified by bgRGB (*enable*=1). Background color is ignored for *enable*=0.
frame=*f*	Specifies the frame enclosing the action. *f*=0: No frame. *f*=1: Single frame (default). *f*=2: Double frame. *f*=3: Triple frame. *f*=4: Shadow frame.
helpText=*helpTextStr*	

Specifies the help string for the action. The text is limited to 255 bytes. On Macintosh, help appears when the cursor is over the action after choosing Help→Show Igor Tips. On Windows, help appears in the status line when the cursor is over the action.

ignoreErrors=*ignore*

Controls whether an error dialog will appear (*ignore*=0) or not (*ignore* is nonzero) if an error occurs while executing the action commands.

linkStyle=*linkStyle* Controls the action title text style. If *linkStyle*=1, the style is the same as a help link (blue underlined). If *linkStyle*=0, the style properties are the same as the preceding text.

name=*name* Specifies the name of the new or modified notebook action. This is a standard Igor name. See **Standard Object Names** on page III-443 for details.

padding={*leftPadding*, *rightPadding*, *topPadding*, *bottomPadding*, *internalPadding*}

Sets the padding in points. *internalPadding* sets the padding between the title and the picture when both elements are present.

picture=*name* Specifies a picture for the action icon. *name* is the name of a picture in the picture gallery (see **Pictures** on page III-448).

If *name* is null (`$""`), it clears the picture parameter.

procPICTName=*name*

Specifies a Proc Picture for the action icon (see **Proc Pictures** on page IV-53). *name* is the name of a Proc Picture or null (`$""`) to clear it. This will be a name like `ProcGlobal#myPictName` or `MyModuleName#myPictName`. If you use a module name, the Proc Picture must be declared static.

If you specify both picture and procPICTName, picture will be used.

quiet=*quiet* Displays action commands in the history area (*quiet*=0), otherwise (*quiet*=1) no commands will be recorded.

scaling={*h*, *v*} Scales the picture in percent horizontally, *h*, and vertically, *v*.

showMode=*mode* Determines if the title or picture are displayed.

mode=1:	Title only.
mode=2:	Picture only.
mode=3:	Picture below title.
mode=4:	Picture above title.
mode=5:	Picture to left of title.
mode=6:	Picture to right of title.

Without a picture specification, the action will use title mode regardless of what you specify.

title=*titleStr* Sets the action title to *titleStr*, which is limited to 255 bytes.

Flags

/W= *winName* Specifies the notebook window of interest.

winName is either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-87 for details on host-child specifications.

If /W is omitted, NotebookAction acts on the top notebook window.

Examples

```
String nb = WinName(0, 16, 1)           // Top visible notebook
NotebookAction name=Action0, title="Beep", commands="Beep"// Create action
NotebookAction name=Action0, enableBGRGB=1, padding={4,4,4,4,4}
```

```
Notebook $nb, findSpecialCharacter={"Action0",1}         // Select action
Notebook $nb, frame=1                                    // Set frame
```

See Also

Chapter III-1, **Notebooks**.

The **Notebook**, **NewNotebook**, and **OpenNotebook** operations; the **SpecialCharacterInfo** and **SpecialCharacterList** functions.

num2char

`num2char(`*`num [, options`*`)`

The num2char function returns a string containing a character.

The *options* parameter was added in Igor Pro 7.00 and defaults to 0.

As of Igor7, Igor represents text internally as UTF-8, a form of Unicode. Previously it represented text as system text encoding. Because of this change, the behavior of num2char is complicated.

Recommended use of num2char in Igor7 or later

If *num* is a Unicode code point, pass 0 for *options* and num2char will return a UTF-8 string containing the character for the Unicode code point represented by *num*.

If you want a string containing a single byte, even though it may not be a valid UTF-8 string, pass 1 for *options* and num2char will return a string containing the single byte whose value is *num*, provided that *num* is between 0 and 255.

Detailed description of num2char in Igor7 or later

If *num* is between 0 and 127, num2char returns a string containing a single byte whose value is *num*. This represents an ASCII character.

If *num* is between 128 and 255 and *options* is 1, num2char returns a string containing a single byte whose value is *num*. This is not valid UTF-8 text, but it is consistent with the behavior of num2char in Igor6.

If *num* is between 128 and 255 and *options* is 0 or omitted, num2char returns the UTF-8 representation of the character for the Unicode code point represented by *num*.

If *num* is greater than 255, num2char returns the UTF-8 representation of the character for the Unicode code point represented by *num* regardless of the value of *options* .

If you provide the *options* parameter, it must be either 0 or 1. Other values may be used for other purposes in the future.

Examples

```
Print num2char(65)           // Prints A
Print num2char(97)           // Prints a
Print num2char(0xF7)         // Prints division sign
Print num2char(0xF7,0)       // Prints division sign
Print num2char(0xF7,1)       // Prints missing character symbol
Print num2char(0x0127)       // Prints small letter h with stroke (h-bar)
Print num2char(0x0127,0)     // Prints small letter h with stroke (h-bar)
Print num2char(0x0127,1)     // Prints small letter h with stroke (h-bar)

// In the case of num2char(0xF7,1),num2char returns a string containing
// a single byte whose value is 0xF7. This is not a valid UTF-8 string.
```

See Also

The **char2num**, **str2num** and **num2str** functions.

Text Encodings on page III-409.

num2istr

`num2istr(`*`num`*`)`

The num2istr function returns a string representing *num* after rounding to the nearest integer.

num2str

`num2str(`*`num`*`)`

The num2str function returns a string representing the number *num*.

Precision is limited to only five decimal places. This can cause unexpected and confusing results. For this reason, we recommend that you use **num2istr** or **sprintf** for better control of the format and precision of the number conversion.

See Also

The **sprintf** operation.

The **str2num**, **char2num** and **num2char** functions.

NumberByKey

`NumberByKey(`*`keyStr, kwListStr`*`[, `*`keySepStr`*` [, `*`listSepStr`*` [, `*`matchCase`*`]]])`

The NumberByKey function returns a numeric value extracted from *kwListStr* based on the specified key contained in *keyStr*. *kwListStr* should contain keyword-value pairs such as "`KEY=value1,KEY2=value2`" or "`Key:value1;KEY2:value2`", depending on the values for *keySepStr* and *listSepStr*.

Use NumberByKey to extract a numeric value from a strings containing "`key1=value1;key2=value2;`" style lists such as those returned by functions like **AxisInfo** or **TraceInfo**.

If the key is not found or if any of the arguments is `""` or if the conversion to a number fails then it returns NaN.

keySepStr, *listSepStr*, and *matchCase* are optional; their defaults are ":", ";", and 0 respectively.

Details

keyStr is limited to 255 bytes.

kwListStr is searched for an instance of the key string bound by *listSepStr* on the left and a *keySepStr* on the right. The text up to the next *listSepStr* is converted to the returned number.

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are usually case-insensitive. Setting the optional *matchCase* parameter to 1 makes the comparisons case sensitive.

In Igor6, only the first byte of *keySepStr* and *listSepStr* was used. In Igor7 and later, all bytes are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *matchCase* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print NumberByKey("AKEY", "AKEY:123;")                        // prints 123
Print NumberByKey("BKEY", "AKEY=123;Bkey=456;", "=")          // prints 456
Print NumberByKey("KEY2", "KEY1=123,KEY2=999,", "=", ",")// prints 999
Print NumberByKey("ckey", "CKEY=123;ckey=456;", "=")          // prints 123
Print NumberByKey("ckey", "CKEY=123;ckey=456;", "=", ";", 1)// prints 456
```

See Also

The **StringByKey**, **RemoveByKey**, **ReplaceNumberByKey**, **ReplaceStringByKey**, **ItemsInList**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

numpnts

`numpnts(`*`waveName`*`)`

The numpnts function returns the total number of data points in the named wave. To find the number of elements in a dimension of a multidimensional wave, use the **DimSize** function.

Do not use numpnts to test if a wave reference is null as this causes a runtime error. Use **WaveExists**.

numtype

`numtype(`*`num`*`)`

The numtype function returns a number which indicates what kind of value *num* contains.

Details

If *num* is a real number, numtype returns a real number whose value is:

0:	If *num* contains a normal number.
1:	If *num* contains +/-INF.
2:	If *num* contains NaN.

If *num* is a complex number, numtype returns a complex number in which the real part is the number type of the real part of *num* and the imaginary part is the number type of the imaginary part of *num*.

NumVarOrDefault

`NumVarOrDefault(`*`pathStr, defVal`*`)`

The NumVarOrDefault function checks to see if the *pathStr* points to a numeric variable. If the numeric variable exists, NumVarOrDefault returns its value. If the numeric variable does not exist, it returns *defVal* instead.

Details

NumVarOrDefault initializes input values of macros so they can remember their state without needing global variables to be defined first. String variables use the corresponding numeric function, **StrVarOrDefault**.

Examples

```
Macro foo(nval,sval)
   Variable nval=NumVarOrDefault("root:Packages:mypack:nvalSav",2)
   String sval=StrVarOrDefault("root:Packages:mypack:svalSav","Hi")

   DFREF dfSav= GetDataFolderDFR()
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S mypack
   Variable/G nvalSav= nval
   String/G svalSav= sval
   SetDataFolder dfSav
End
```

NVAR

`NVAR [/C][/Z]` ***`localName`*** **`[=`** ***`pathToVar`*****`][,`** ***`localName1`*** **`[=`** ***`pathToVar1`*****`]]…`**

NVAR is a declaration that creates a local reference to a global numeric variable accessed in a user-defined function.

The NVAR reference is required when you access a global numeric variable in a function. At compile time, the NVAR statement specifies the local name referencing a global numeric variable. At runtime, it makes the connection between the local name and the actual global variable. For this connection to be made, the global numeric variable must exist when the NVAR statement is executed.

When *localName* is the same as the global numeric variable name and you want to reference a global variable in the current data folder, you can omit *pathToVar*.

pathToVar can be a full literal path (e.g., root:FolderA:var0), a partial literal path (e.g., :FolderA:var0) or $ followed by string variable containing a computed path (see **Converting a String into a Reference Using $** on page IV-57).

You can also use a data folder reference or the /SDFR flag to specify the location of the numeric variable if it is not in the current data folder. See **Data Folder References** on page IV-72 and **The /SDFR Flag** on page IV-74 for details.

If the global variable may not exist at runtime, use the /Z flag and call **NVAR_Exists** before accessing the variable. The /Z flag prevents Igor from flagging a missing global variable as an error and dropping into the Igor debugger. For example:

```
NVAR/Z nv=<pathToPossiblyMissingNumericVariable>
if( NVAR_Exists(nv) )
```

```
   <do something with nv>
endif
```

Note that to create a global numeric variable, you use the **Variable**/G operation.

Flags

/C	Variable is complex.
/Z	Ignores variable reference checking failures.

See Also

NVAR_Exists function.

Accessing Global Variables and Waves on page IV-59.

Converting a String into a Reference Using $ on page IV-57.

NVAR_Exists

`NVAR_Exists(`*`name`*`)`

The NVAR_Exists function returns one if specified NVAR reference is valid or zero if not. It can be used only in user-defined functions.

For example, in a user function you can test if a global numeric variable exists like this:

```
NVAR /Z var1 = gVar1          // /Z prevents debugger from flagging bad NVAR
if (!NVAR_Exists(var1))       // No such global numeric variable?
   Variable/G gVar1 = 0       // Create and initialize it
endif
```

See Also

WaveExists, **SVAR_Exists**, and **Accessing Global Variables and Waves** on page IV-59.

Open

`Open [`*`flags`*`]` ***`refNum`*** **`[as`** ***`fileNameStr`*`]`**

The Open operation can, depending on the flags passed to it:

- Open an existing file to read data from (/R flag without /D).
- Open a to append results to (/A flag without /D).
- Create a new file or overwrite an existing file to write results to (no /D, /R or /A flags).
- Display an Open File dialog (/D/R or /D/A flags with or without /MULT).
- Display a Save File dialog (/D flag without /R or /A).

Parameters

refNum is the name of a numeric variable to receive the file reference number. *refNum* is set by Open if Open actually opens a file for reading or writing (cases 1, 2 and 3). You use *refNum* with the **FReadLine**, **FStatus**, **FGetPos**, **FSetPos**, **FBinWrite**, **FBinRead**, **fprintf**, and **wfprintf** operations to read from or write to the file. When you're finished, use pass *refNum* to the **Close** operation to close the file.

Open does not set the file reference number when the /D flag is used (cases 4 and 5) but you must still supply a *refNum* parameter.

The following discussion of the *pathName* and *fileNameStr* parameters applies when you are attempting to open a file for reading or writing (cases 2, 3, and 5 above).

The targeted file is specified by a combination of the *pathName* parameter and the *fileNameStr* parameter. There are three ways to specify the targeted file:

Method	How To Use It
Symbolic path and simple file name	Use /P=*pathName* and *fileNameStr*, where *pathName* is the name of an Igor symbolic path (see **Symbolic Paths** on page II-21) that points to the folder containing the file and *fileNameStr* is the name of the file.
Symbolic path and partial path	Use /P=*pathName* and *fileNameStrs*, where *pathName* is the name of an Igor symbolic path that points to the folder containing the file and *fileNameStr* is a partial path starting from the folder and leading to the file.
Full path	Use just *fileNameStr*, where *fileNameStr* is a full path to the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

The targeted file is fully specified if *fileNameStr* is a full path or if both *pathName* and *fileNameStr* are present and not empty strings.

The targeted file is not fully specified in any of these cases:

- *as fileNameStr* is omitted
- *fileNameStr* is an empty string
- *fileNameStr* is not a full path and no symbolic path is specified

Opening an Existing File For Reading Only

This covers cases 1 (/R without /D).

If the file is fully-specified but does not exist, an error is generated. If you want to detect and handle the error yourself, use the /Z flag.

If the file is not fully-specified, Open displays an Open File dialog.

If a file is opened, *refNum* is set to the file reference number.

Opening an Existing File For Appending

This covers cases 1 (/R without /D) and 2 (/A without /D).

If the file is fully-specified and exists, it is opened for read/write and the current file position is moved to the end of the file.

If the file is fully-specified but does not exist, the file is created and opened for read/write.

If the file is not fully-specified, Open displays an Open File dialog.

If a file is opened, *refNum* is set to the file reference number.

Opening a File For Write

This covers case 3 (no /R, /A or /D).

If the targeted file exists, it is overwritten.

If the targeted file does not exist and it is fully-specified and targets a valid path, a new file is created.

If the file is fully-specified and targets an invalid path, an error is generated. If you want to detect and handle the error yourself, use the /Z flag.

If the file is not fully-specified, Open displays a Save File dialog.

If a file is opened, *refNum* is set to the file reference number.

Displaying an Open File Dialog To Select a Single File

This covers cases 4 (/D with /R or /A).

Open does not actually open the file but just displays the Open File dialog.

If the user chooses a file in the Open File dialog, the S_fileName output string variable is set to a full path to the file. You can use this in subsequent commands. If the user cancels, S_fileName is set to "".

See the documentation for the /D, /F and /M flags and then read **Displaying an Open File Dialog** on page IV-136 for details.

refNum is left unchanged.

Displaying an Open File Dialog To Select Multiple Files

This covers cases 4 (/D with /R or /A) with the /MULT=1 flag.

Open does not actually open the file but just displays the Open File dialog.

If the user chooses one or more files in the Open File dialog, the S_fileName output string variable is set to a carriage-return-delimited list of full paths to one or more files. You can use this in subsequent commands. If the user cancels, S_fileName is set to "".

See the documentation for the /D, /F, /M and /MULT flags and then read **Displaying a Multi-Selection Open File Dialog** on page IV-137 for details.

refNum is left unchanged.

Displaying a Save File Dialog

This covers cases 5 (/D without /R or /A).

Open does not actually open the file but just displays the Save File dialog.

If the user chooses a file in the Save File dialog, the S_fileName output string variable is set to a full path to the file. You can use this in subsequent commands. If the user cancels, S_fileName is set to "".

See the documentation for the /D, /F and /M flags and then read **Displaying a Save File Dialog** on page IV-138 for details.

refNum is left unchanged.

Flags

/A	Opens an existing file for appending or, if the file does not exist, creates a new file and opens it for appending.
/C=*creatorStr*	Specifies the file creator code. This is meaningful on Macintosh only and is ignored on Windows. For opening an existing file, creator defaults to "????" which means "any creator". For creating a new file, *creatorStr* defaults to "IGR0" which is Igor's creator code.
/D[=*mode*]	Specifies dialog-only mode. /D: A dialog is always displayed. /D=1: Same as /D. /D=2: A dialog is displayed only if *pathName* and *fileNameStr* do not specify a valid file. Use this mode to allow the user to choose a file to be opened by a subsequent operation, such as **LoadWave**. With /D or /D=1, open presents a dialog from which the user can select a file but does not actually open the file. Instead, Open puts the full path to the file into the string variable S_fileName. /D=2 does the same thing except that it skips the dialog if *pathName* and *fileNameStr* specify a valid file. In this case, if pathName and fileNameStr refer to an alias (Macintosh) or shortcut (Windows), the target of the alias or shortcut is returned. If the user clicks the Cancel button, S_fileName is set to an empty string.

	Use Open/D/R to bring up an Open File dialog. See **Displaying an Open File Dialog** on page IV-136 for details.
	Use Open/D/R/MULT=1 to bring up an Open File dialog to select multiple files. See **Displaying a Multi-Selection Open File Dialog** on page IV-137 for details.
	Use Open/D to bring up a Save File dialog. See **Displaying a Save File Dialog** on page IV-138 for details.
	See **Using Open in a Utility Routine** on page IV-139 for an example using /D=2.
	Do not use /Z with /D.
/F=*fileFilterStr*	/F provides control over the file filter menu in the Open File dialog. See **Open File Dialog File Filters** on page IV-137 and **Save File Dialog File Filters** on page IV-139 for details.
/M=*messageStr*	Prompt message text in the dialog used to select the file, if any. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/MULT=m	Use /D/R/MULT=1 to display a multi-selection Open File dialog.
	/D/R/MULT=0 or just /D/R displays a single-selection Open File dialog.
	/MULT=1 is allowed only if /D or /D=1 and /R are specified.
	See **Displaying a Multi-Selection Open File Dialog** on page IV-137 for details.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/R	The file is opened read only.
/T=typeStr	When creating a new file on Macintosh (/A and /R flag omitted), /T sets the Macintosh file type property for the file if it does not already exist. For example, /T="BINA" sets the Macintosh file type to 'BINA'. If /T is omitted the Macintosh file type will be 'TEXT'. Apple has deemphasized Macintosh file types in favor of file name extensions.
	For new code, /F is recommended instead of /T.
	When opening an existing file (/A or /R flag used), /T provides control over the file filter menu in the Open File dialog. See **Open File Dialog File Filters** on page IV-137 for details.
	When creating a new file (/A and /R flag omitted), /T provides control over the file filter menu in the Save File dialog. See **Save File Dialog File Filters** on page IV-139 for details.
/Z[=z]	Prevents aborting of procedure execution if an error occurs, for example if the procedure tries to open a file that does not exist for reading. Use /Z if you want to handle this case in your procedures rather than having execution abort.
	When using /Z, /Z=1, or /Z=2, V_flag is set to 0 if no error occurred or to a nonzero value if an error did occur.
	Do not use /Z with /D.

/Z=0:	Same as no /Z.
/Z=1:	Suppresses normal error reporting. When used with /R, it opens the file if it exists. /Z alone has the same effect as /Z=1.
/Z=2:	Suppresses normal error reporting. When used with /R, it opens the file if it exists or displays a dialog if it does not exist.

Details

When Open returns, if a file was actually opened, the *refNum* parameter will contain a file reference number that you can pass to other operations to read or write data. If the file was not opened because of an error or because the user canceled or because /D was used, *refNum* will be unchanged.

If you use /R (open for read), Open opens an existing file for reading only.

If you use /A, Open opens an existing file for appending. If the file does not exist, it is created and then opened for appending.

If both /R and /A are omitted then Open creates and opens a file. If the specified file does not already exist, Open creates it and opens it for writing. If the file does already exist then Open opens it and sets the current file position to the start of the file. The current file position determines where in the file data will be written. Thus, you will be overwriting existing data in the file.

Warning: If you open an existing file for writing (you do not use /R) then you will overwrite or truncate existing data in the file. To avoid this, open for read (use /R) or open for append (use /A).

Output Variables

The Open operation returns information in the following variables:

`V_flag`	Set only when the /Z flag is used. V_flag is set to zero if the file was opened, to -1 if Open displayed a dialog (because the file was not fully-specified) and the user canceled, and to some nonzero value if an error occurred.
`S_fileName`	Stores the full path to the file that was opened. If /MULT=1 is used, S_fileName is a carriage-return-separated list of full paths to one or more files. If an error occurred or if the user canceled, S_fileName is set to an empty string.

When using /D, the value of V_flag is undefined. Do not use /Z with /D. Use S_fileName to determine if the user selected a file or canceled.

Examples

This example function illustrates using Open to open a text file from which data will be read. The function takes two parameters: an Igor symbolic path name and a file name. If either of these parameters is an empty string, the Open operation will display a dialog allowing the user to choose the file. Otherwise, the Open operation will open the file without displaying a dialog.

```
Function DemoOpen(pathName, fileName)
    String pathName     // Name of symbolic path or "" for dialog.
    String fileName     // File name, partial path, full path or "" for dialog.
    Variable refNum
    String str

    // Open file for read.
    Open/R/Z=2/P=$pathName refNum as fileName

    // Store results from Open in a safe place.
    Variable err = V_flag
    String fullPath = S_fileName

    if (err == -1)
        Print "DemoOpen canceled by user."
        return -1
    endif

    if (err != 0)
        DoAlert 0, "Error in DemoOpen"
        return err
    endif

    Printf "Reading from file \"%s\". First line is:\r", fullPath
    FReadLine refNum, str      // Read first line into string variable
    Print str
    Close refNum
    return 0
End
```

See Also

Symbolic Paths on page II-21.

Close, **FBinRead**, **FBinWrite**, **FReadLine**

FGetPos, **FSetPos**, **FStatus**

fprintf, **wfprintf**

Displaying an Open File Dialog on page IV-136, **Displaying a Multi-Selection Open File Dialog** on page IV-137, **Open File Dialog File Filters** on page IV-137

Displaying a Save File Dialog on page IV-138, **Save File Dialog File Filters** on page IV-139

Using Open in a Utility Routine on page IV-139

The Load File Demo example in "Igor Pro 7 Folder:Examples:Programming".

OpenHelp

`OpenHelp [`*`flags`*`]` ***`fileNameStr`***

The OpenHelp operation opens the specified help file.

The OpenHelp operation was added in Igor Pro 7.00.

Parameters

The help file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If OpenHelp can not determine the location of the file from *fileNameStr* and *pathName*, it returns an error.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/INT[=*interactive*]	Controls whether opening the help file is interactive or not. /INT=1: If the help file being opened needs to be compiled, OpenHelp presents a dialog asking the user whether the file should be compiled. During the compile, a progress dialog is displayed. Any errors are presented to the user in an error dialog. This is the default behavior if /INT is omitted. /INT=0: If the help file being opened needs to be compiled, OpenHelp compiles it without presenting a dialog. Compilation errors are not presented to the user but are reflected in the V_Flag output variable.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing Igor symbolic path.
/V=*visible*	Controls help window visibility. *visible*=0: The help window will be initially hidden. *visible*=1: The help window will be initially visible. This is the default if /V is omitted.
/W=(*left*,*top*,*right*,*bottom*)	Specifies window size and position. Coordinates are in points.
/Z[=*z*]	Controls error reporting. /Z=0: Report errors normally. /Z=0 is the same as omitting /Z altogether. This is the default behavior if /Z is omitted. /Z=1: Suppresses normal error reporting. /Z alone has the same effect as /Z=1. /Z=1 prevents aborting procedure execution if an error occurs, for example if the file does not exist or if there is a compilation error. Use /Z=1 if you want to handle errors in your procedures rather than having execution abort. When using /Z or /Z=1, check V_Flag to see if an error occurred.

Details

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "hd:Folder1:" or "C:\\Folder1\\". See **Symbolic Paths** on page II-21 for details.

If the specified file is already open but not as a help window (for example as a notebook), OpenHelp returns an error.

If the /W or /V flag is used, or both, the window size and position and visibility are set as specified even if the file itself is already open, so long as the file is already opened as a help window.

Output Variables

The OpenHelp operation returns information in the following variables:

`V_Flag`	Set to a non-zero value if an error occurred and to zero if no error occurred.
`V_alreadyOpen`	Set to 1 if the specified help file was already open as a help file or to zero otherwise.

See Also

CloseHelp

OpenNotebook

`OpenNotebook` `[`***`flags`***`] [`***`fileNameStr`***`]`

The OpenNotebook operation opens a file for reading or writing as an Igor notebook.

Unlike the Open operation, OpenNotebook will not create a file if the specified file does not exist. To create a new notebook, use the **NewNotebook** operation.

Parameters

The file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/A	Moves the notebook's selection to the end of the notebook.
/ENCG=*textEncoding*	Specifies the text encoding of the plain text file to be opened as a notebook. This flag was added in Igor Pro 7.00. This is relevant for plain text notebooks only and is ignored for formatted notebooks because they can contain multiple text encodings. See **Plain Text File Text Encodings** on page III-417 and **Formatted Text Notebook File Text Encodings** on page III-421 for details. OpenNotebook uses the text encoding specified by /ENCG and the rules described under **Determining the Text Encoding for a Plain Text File** on page III-417 to determine the source text encoding for conversion to UTF-8. Passing 0 for *textEncoding* acts as if /ENCG were omitted. See **Text Encoding Names and Codes** on page III-434 for a list of accepted values for *textEncoding*.
/K=*k*	Specifies window behavior when the user attempts to close it. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. *k*=3: Hides the window. If you use /K=2 or /K=3, you can still kill the window using the **KillWindow** operation.
/M=*messageStr*	Prompt message text in the dialog used to find the file, if any. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/N=*winName*	Specifies the window name to be assigned to the new notebook. If omitted, it assigns a name like "Notebook0".
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/R	Opens the file as read only.
/T=*typeStr*	Specifies the type or types of files that can be opened.
/V=*visible*	Hides (*visible*= 0) or shows (*visible*= 1; default) the notebook.
/W=(*left*,*top*,*right*,*bottom*)	Specifies window size and position. Coordinates are in points.
/Z	Suppresses error generation. Use this to check if a file exists. If you use /Z, OpenNotebook sets the variable V_flag to 0 if the notebook was opened or to nonzero if there was an error, usually because the specified file does not exist.

Details

The /A (append) flag has no effect other than to move the selection to the end of the notebook after it is opened.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-21 for details.

The /T=*typeStr* flag affects only the dialog that OpenNotebook presents if you do not specify a path and filename. The dialog presents only those files whose type is specified by /T=*typeStr*. There are two file types that are allowed for notebooks: `'TEXT'` which is a plain text file and `'WMT0'` which is a WaveMetrics formatted text file. Therefore, the file type, if you use it, should be either "`TEXT`" or "`WMT0`". If /T=*typeStr* is missing, it defaults to "`TEXTWMT0`". This opens either type of notebook file. On Windows, Igor considers files with ".txt" extensions to be of type TEXT and considers files with ".ifn" to be of type WMT0. See **File Types and Extensions** on page III-404 for details.

See Also

The **Notebook** and **NewNotebook** operations, and Chapter III-1, **Notebooks**.

OpenProc

`OpenProc [`*`flags`*`] [`*`fileNameStr`*`]`

The OpenProc operation opens a file as an Igor procedure file.

Note: This operation is used automatically to open procedure files when you open an Igor experiment. You can invoke OpenProc only from the command line. Do not invoke it from a procedure. To open procedure files from a procedure or from a menu definition, use the Execute/P operation.

Parameters

The file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/A	Moves the procedure window's selection to the end of the window.
/ENCG=*textEncoding*	Specifies the text encoding of the plain text file to be opened as a procedure file. This flag was added in Igor Pro 7.00. OpenProc uses the text encoding specified by /ENCG and the rules described under **Determining the Text Encoding for a Plain Text File** on page III-417 to determine the source text encoding for conversion to UTF-8. Passing 0 for *textEncoding* acts as if /ENCG were omitted. See **Text Encoding Names and Codes** on page III-434 for a list of accepted values for *textEncoding*.
/M=*messageStr*	Prompt message text in the dialog used to find the file, if any. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/R	The file is opened read only.
/T=*typeStr*	Specifies the type or types of files that can be opened.
/V=*visible*	Hides (*visible*= 0) or shows (*visible*= 1; default) the procedure window.
/Z	Suppresses error generation if the specified file does not exist.

Details

The /A (append) flag has no effect other than to move the selection to the end of the procedure file after it is opened.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-21 for details.

OpenProc automatically opens procedure files when you open an Igor experiment. Normally, you will have no use for it. You can not open a procedure file while procedures are executing. Thus, you can't invoke OpenProc from within a procedure. You can only invoke it from the command line or from a user menu definition (actually, you may get away with it in a macro, but it's not recommend).

See Also

Chapter III-13, **Procedure Windows**.

The **Execute** operation.

OperationList

`OperationList(`*`matchStr, separatorStr, optionsStr`*`)`

The OperationList function returns a string containing a list of internal (built-in) or external operation names corresponding to *matchstr*.

Parameters

Only operation names that match *matchStr* string are listed. Use `"*"` to match all names. See **WaveList** for examples.

The first character of *separatorStr* is appended to each operation name as the output string is generated. *separatorStr* is usually ";" for list processing (See **Processing Lists of Waves** on page IV-187 for details).

Use *optionsStr* to further qualify the list of operations. *optionsStr* is a (case-insensitive) string containing one of these values:

`"internal"`	Restricts the list to built-in operations.
`"external"`	Restricts the list to external operations (see **Igor Extensions** on page III-450).

Any other value for *optionsStr* (`"all"` is recommended) will return both internal and external operations.

See Also

The **DisplayProcedure** operation and the **FunctionList**, **MacroList**, **StringFromList**, and **WinList** functions.

Optimize

`Optimize` `[`***`flags`***`]` ***`funcspec, pWave`***

The Optimize operation determines extrema (minima or maxima) of a specified nonlinear function. The function must be defined in the form of an Igor user function.

Use the first form for univariate functions (one dimensional functions; functions taking just one variable). Use the second form with multivariate functions (functions in more than one dimension; functions of more than one variable).

Optimize uses Brent’s method for univariate functions. For multivariate functions you can choose several variations of quasi-Newton methods or simulated annealing.

Flags

/A [=*findMax*]	Finds a maximum (/A=1 or /A) or minimum (/A=0 or no flag).
/D=*nDigits*	Specifies the number of good digits returned (default is 15) by the function being optimized. If you use /X=*xWave* with a single-precision wave, the default is seven. Ignored with simulated annealing (/M={3,0}).
/DSA=*destWave*	Sets a wave to track the current best model only found with simulated annealing (/M={3,0}). *destWave* must have the same number of points as the X vector.
/F=*trustRegion*	Sets the initial trust region when /M={1 or 2, …} with multivariate functions. The value is a scaled step size (see **Multivariate Details**). After the first iteration the trust region is adjusted according to conditions. Ignored with simulated annealing (/M={3,0}).

/H= *highBracket*

/L= *lowBracket*

Find the extrema of a univariate function. *lowBracket* and *highBracket* are X values on either side of the extreme point. An extreme point is found between the bracketing values.

If *lowBracket* and *highBracket* are equal, Optimize adds 1.0 to *highBracket* before looking for an extreme point.

Default values for *lowBracket* and *highBracket* are zero. Thus, if neither *lowBracket* nor *highBracket* is present, this is the same as /L=0/H=1.

Ignored with simulated annealing (/M={3,0}).

/I=*maxIters*

Sets the maximum number of iterations in searching for an extreme point to *maxIters*. Default is 100 for *stepMethod* (/M flag) 0-2, 10000 for *stepMethod* = 3 (simulated annealing).

If you use this form of the /I flag with simulated annealing, *maxItersAtT* is set to *maxIters*/2 and *maxAcceptances* is set to *maxIters*/10.

/I={*maxIters*, *maxItersAtT*, *maxAcceptances*}

Specifies the number of iterations for simulated annealing. The maximum number of iterations is set by *maxIters*, *maxItersAtT* sets the maximum number of iterations at a given temperature in the cooling schedule, and *maxAcceptances* sets the total number of accepted changes in the X vector (whether they increase or decrease the function) at a given temperature.

If you use this form of the flag with any *stepMethod* (/M flag) other than 3, *maxItersAtT* and *maxAcceptances* are ignored.

Defaults for *stepMethod* = 3 are {10000, 5000, 500}.

/M={*stepMethod*, *hessMethod*}

Specifies the method used for selecting the next step (*stepMethod*) and the method for calculating the Hessian (matrix of second derivatives) with multivariate functions.

stepMethod	**Method**
0	Line Search
1	Dogleg
2	More-Hebdon
3	Simulated Annealing

hessMethod	**Method**
0	secant (BFGS)
1	finite differences

Default values are {0,0}. The *hessMethod* variable is ignored if you select *stepMethod* = 3.

/Q

Suppresses printout of results in the history area. Ordinarily, the results of root searches are printed in the history.

/R={*typX1*, *typX2*, …}

/R=*typXWave*

Specifies the expected size of X values with multivariate functions. These values are used to scale X values. If the X values you expect are very different from one, you will get more accurate results if you can give a reasonable estimate. Optimize will use *typXi* to scale X_i to reduce floating-point truncation error.

You must provide the same number of values in either a wave or a list of values as you provide to the /X flag.

Ignored with simulated annealing (/M={3,0}).

/S=*stepMax*

Limits the largest scaled step size allowed with multivariate functions. Optimize will stop if five consecutive steps exceed *stepMax*.

Ignored with simulated annealing (/M={3,0}).

/SSA=*stepWave*	Name of a 3-column wave having number of rows equal to the length of the X vector only when used with simulated annealing (/M={3,0}). *stepWave* sets information about the step size used to generate new X vectors. The step sizes are in terms of normalized X values. The normalization is such that the X_i ranges from -1 to 1 based on the ranges set by the /XSA flag.
	Column zero sets the step size used when creating new trial X vectors. Default is 1.0.
	Column one sets the minimum step size. Default is 0.001.
	Column two sets the maximum step size. Default is 1.0.
/T=*tol*	Sets the stopping criterion with univariate functions. Optimize will attempt to find a minimum within ± *tol*.
	When this form is used with a multivariate function, *gradTol* is set to *tol* and *stepTol* is set to *gradTol*^{2}.
	Ignored with simulated annealing (/M={3,0}).
/T={*gradtol*, *stepTol*}	Sets the stopping criteria for multivariate functions. Iterations stop if a point is found with estimated scaled gradient less than *gradTol*, or if an iteration takes a scaled step shorter than *stepTol*. Default values are {8.53618x10^{-6}, 7.28664x10^{-11}}. These values are (6.022x10^{-16})$^{1/3}$ and (6.022x10^{-16})$^{2/3}$ as suggested by Dennis and Schnabel. 6.022x10^{-16} is the smallest double precision floating point number that, when added to 1, is different from 1.
	Ignored with simulated annealing (/M={3,0}).
/TSA={*InitialTemp*, *CoolingRate*}	
	Used only with simulated annealing (/M={3,0}).
	InitialTemp sets the initial temperature. If *InitialTemp* is set to zero, Optimize calls your function 100 times to estimate the best initial temperature. This is the recommended setting unless your function is very expensive to evaluate (in which case, you may not want to use simulated annealing at all).
	CoolingRate sets the factor by which the temperature is decreased.
/X=*xWave* /X={*x1*, *x2*, …}	Sets the starting point for searching for an extreme point with multivariate functions or with simulated annealing (/M={3,0}). The starting point can be specified with a wave having as many points as the number of independent variables, or you can write out a list of X values in braces. If you are finding extreme points of a univariate function, use /L and /H instead unless you are using the simulated annealing method. If you specify a wave, this wave is also used to receive the result of the extreme point search.
/XSA=*XLimitWave*	Name of a 2-column wave having number of rows equal to the length of the X vector only when used with simulated annealing (/M={3,0}).
	Column zero sets the minimum value allowed for each element of the X vector.
	Column one sets the maximum value allowed for each element of the X vector.
	Default is ±X_i*10 if X_i is nonzero, or ±1 if X_i is zero. While a default is provided, it is highly recommended that you provide an *XLimitWave*.
/Y=*funcSize*	Specifies expected sizes of function values with multivariate functions. If you expect your function will return values very different from one, you should set *funcSize* to the expected size. Optimize will use this value to scale the function results to reduce floating-point truncation error.

Parameters

func specifies the name of your user-defined function that will be optimized.

pwave gives the name of a parameter wave that will be passed to your function as the first parameter. It is not modified by Igor. It is intended for your private use to pass adjustable constants to your function.

Function Format

Finding extreme points of a nonlinear function requires that you realize the function as a Igor user function of a certain form. See **Finding Minima and Maxima of Functions** on page III-295 for detailed examples.

Your function must look like this:

```
Function myFunc(w,x1, x2, …)
   Wave w
   Variable x1, x2

   return f(x1, x2, …)            // an expression …
End
```

A univariate function would have only one X variable.

A multivariate function can use a wave to pass in the X values:

```
Function myFunc(w,xw)
   Wave w
   Wave xw

   return f(xw)                   // an expression …
End
```

Replace "f(…)" with an appropriate numerical expression.

Univariate Details

The method used by Optimize to find extreme points of univariate functions requires that the point be bracketed before starting. If you don't use /L and /H to specify the bracketing X values, the defaults are zero and one. Optimize first attempts to find the requested extreme point using the bracketing values (or the default). If that is unsuccessful, it attempts to bracket an extreme point by expanding the bracketing interval. If a suitable interval is found (the search is by no means perfectly reliable), then the search for an extreme point is made again.

Optimize uses Brent's method for univariate functions, which requires no derivatives. This combines a quadratic extrapolation with checking for wild results. In the case of wild results (points beyond the best current bracketing values) the method reverts to a golden section bisection algorithm. For well-behaved functions, the quadratic extrapolation converges superlinearly. The golden section bisection algorithm converges more slowly but features global convergence, that is, if an extremum is there, it will be found.

The stopping criterion is

$$\left| x + \frac{a+b}{2} \right| + \frac{a-b}{2} \le \frac{2}{3} tol.$$

In this expression, *a* and *b* are the current bracketing values, and *x* is the best estimate of the extreme point within the bracketing interval.

The left side of this expression works out to being simply the distance from the current solution to the boundary of the bracketing interval.

Note: Optimizing a univariate function with the simulated annealing method (/M={3,0}) works like a multivariate function, and this section does not apply. See the sections devoted to simulated annealing.

Multivariate Details

With multivariate functions, Optimize scales certain quantities to reduce floating point truncation error. You enter scaling factors using the /R and /Y flags. The /R flag specifies the expected magnitude of X values; Optimize then uses $X_i/typX_i$ in all calculations. Likewise, /Y specifies the expected magnitude of function values.

This scaling can be important for maintaining accuracy if your X's or Y's are very different from one, and especially if your X's have values spanning orders of magnitude.

The Optimize operation uses a quasi-Newton method with derivatives estimated numerically. The function gradient is calculated using finite differences. For estimation of the Hessian (second derivative matrix) you can use either a secant method (*hessMethod* = 0) or finite differences (*hessMethod* = 1). The finite difference method gives a more accurate estimate and may succeed with difficult functions but requires more function evaluations per iteration. The finite difference method's greater accuracy may reduce the total number of iterations required, so the overall number of function evaluations depends on details of the problem being solved. Usually the secant method requires fewer function evaluations and is preferred for functions that are expensive to evaluate.

Once a Newton step is calculated, there are three choices for the method used to find the best next value-line search along the Newton direction (*stepMethod* = 0), double dogleg (*stepMethod* = 1), or More-Hebdon (*stepMethod* = 2). The best method can be found only by experimentation. See Dennis and Schnabel (cited in **References**) for details.

The /F=*trustRegion*, /S=*stepMax* and /T={…, *stepTol*} all refer to scaled step sizes. That is,

$$stepX_i = \frac{|\Delta x_i|}{\max(|x_i|, typX_i)}.$$

The Optimize operation presumes that an extreme point has been found when either the gradient at the latest point is less than *gradTol* or when the last step taken was smaller than *stepTol*. These criteria both refer to scaled quantities:

$$\max_{1 \le i \le n} \left\{ |g_i| \frac{\max(|x_i|, typX_i)}{\max(|f|, funcSize)} \right\} \le gradTol,$$

or

$$\max_{1 \le i \le n} \left\{ |g_i| \frac{|\Delta x_i|}{\max(|x_i|, typX_i)} \right\} \le stepTol.$$

Simulated Annealing Introduction

The simulated annealing or Metropolis algorithm optimizes a function using a random search of the X vector space. It does not use derivatives to guide the search, making it a good choice if the function to be optimized is in some way poorly behaved. For instance, it is a good method for functions with discontinuities in the function value or in the derivatives.

Simulated annealing also has a good chance of finding a global minimum or maximum of a function having multiple local minima or maxima.

Because simulated annealing uses a random search method, it may require a large number of function evaluations to find a minimum, and it is not guaranteed that it will stop at an actual minimum. For these reasons, it is best to use one of the other methods unless those methods have failed.

The simulated annealing method generates new trial solutions by adding a random vector to the current X vector. The elements of the random vector are set to $stepsize_i$*R_i, where R_i is a random number in the interval (-1, 1). As the solution progresses, the *stepsize* is gradually decreased.

Bad trials, that is, those that change the function value in the wrong direction are accepted with a probability that depends on the simulated temperature. It is this aspect that allows simulated annealing to find a global minimum.

Function values are generated and accepted or rejected for some number of iterations at a given temperature, then the temperature is reduced. The probability of a bad iteration being accepted decreases with decreasing temperature. A too-fast cooling rate can freeze in a bad solution.

Simulated Annealing Details

It is highly recommended that you use the XSA flag to specify *XLimitWave*. This wave sets bounds on the values of the elements of the X vector during the random search. The defaults may be adequate but are totally *ad hoc*. You are better off to specify bounds that make sense to the problem you are solving.

The values of *XLimitWave* in addition to bounding the search space also scale the X vector during computations of probabilities, temperatures, etc. Consequently, the X limits can affect the performance of the algorithm.

A large number of iterations is required to have a good probability of finding a reasonable solution.

It is recommended that you set the initial temperature to zero so that Optimize can estimate a good initial temperature. If you can't afford the 100 function evaluations required, you probably shouldn't be using simulated annealing.

Optimize uses an exponential cooling schedule in which $T_{i+1} = CoolingRate*T_i$ (see the /TSA flag). *CoolingRate* must be in the range 0 to 1. A fast cooling rate (small value of *CoolingRate*) can cause simulated quenching; that is, a bad solution can be frozen in. Very slow cooling will result in slow convergence.

When simulated annealing is selected, the optimization is treated as multivariate even if your function has only a single X input. That is, the output variables and waves are the ones listed under multivariate functions.

Variables and Waves for Output

The Optimize operation reports success or failure via the V_flag variable. A nonzero value is an error code.

Variables for a univariate function:

`V_flag`	0:	Search for an extreme point was successful.
	57:	User abort.
	785:	Function returned NaN.
	786:	Unable to find bracketing values for an extreme point.

If you searched for a minimum:

`V_minloc`	X value at the minimum.
`V_min`	Function value (Y) at the minimum.

If you searched for a maximum:

`V_maxloc`	X value at the maximum.
`V_max`	Function value (Y) at the maximum.

For simulated annealing only:

`V_SANumIncreases`	Number of "bad" iterations accepted.
`V_SANumReductions`	Number of iterations resulting in a better solution.

Variables for a multivariate function:

`V_flag`	0:	Search for an extreme point was successful.
	57:	User abort.
	788:	Iteration limit was exceeded.
	789:	Maximum step size was exceeded in five consecutive iterations.
	790:	The number of points in the typical X size wave specified by /R does not match the number of X values specified by the /X flag
	791:	Gradient nearly zero and no iterations taken. This means the starting point is very nearly a critical point. It could be a solution, or it could be so close to a saddle point or a maximum (when searching for a minimum) that the gradient has no useful information. Try a slightly different starting point.

`V_OptTermCode`	Indicates why Optimize stopped. This may be useful information even if V_flag is zero. Values are:	
	1:	Gradient tolerance was satisfied.
	2:	Step size tolerance was satisfied.

3:	No step was found that was better than the last iteration. This could be because the current step is a solution, or your function may be too nonlinear for Optimize to solve, or your tolerances may be too large (or too small), or finite difference gradients are not sufficiently accurate for this problem.
4:	Iteration limit was exceeded.
5:	Maximum step size was exceeded in five consecutive iterations. This may mean that the maximum step size is too small, or that the function is unbounded in the search direction (that is, goes to -inf if you are searching for a minimum), or that the function approaches the solution asymptotically (function is bounded but doesn't have a well-defined extreme point).
6:	Same as V_flag = 791.

If you searched for a minimum:

`V_min`	Function value (Y) at the minimum.

If you searched for a maximum:

`V_max`	Function value (Y) at the maximum.

Variables for all functions:

`V_OptNumIters`	Number of iterations taken before Optimize terminated.
`V_OptNumFunctionCalls`	Number of times your function was called before Optimize terminated.

Waves for a multivariate function:

`W_extremum`	Solution if you didn't use /X=<*xWave*>. Otherwise the solution is returned in your X wave.
`W_OptGradient`	Estimated gradient of your function at the solution.

See Also

Finding Minima and Maxima of Functions on page III-295 for further details and examples.

References

The Optimize operation uses Brent's method for univariate functions. *Numerical Recipes* has an excellent discussion (see section 10.2) of this method (but we didn't use their code):

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

For multivariate functions Optimize uses code based on Dennis and Schnabel. To truly understand what Optimize does, read their book:

Dennis, J. E., Jr., and Robert B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Methods*, 378 pp., Society for Industrial and Applied Mathematics, Philadelphia, 1996.

Override

```
Override constant objectName = newVal
Override strconstant objectName = newVal
Override Function funcName()
```

The Override keyword redefines a constant, strconstant, or user function. The *objectName* or *funcName* must be the same as the name of the original object or function that is being redefined. The override must be defined before the target object appears in the compile sequence.

See Also

Function Overrides on page IV-98 and **Constants** on page IV-47 for further details.

p

`p`

The p function returns the row number of the current row of the destination wave when used in a wave assignment statement. The row number is the same as the point number for a 1D wave.

Details

Outside of a wave assignment statement p acts like a normal variable. That is, you can assign a value to it and use it in an expression.

See Also

Waveform Arithmetic and Assignments on page II-69.

For other dimensions, the **q**, **r**, and **s** functions.

For scaled dimension indices, the **x**, **y**, **z**, and **t** functions.

p2rect

`p2rect(`*`z`*`)`

The p2rect function returns a complex value in rectangular coordinates derived from the complex value *z* which is assumed to be in polar coordinates (magnitude is stored in the real part and the angle, in radians, in the imaginary part of *z*).

Examples

Assume waveIn and waveOut are complex, then:

```
waveOut = p2rect(waveIn)
```

sets each point of waveOut to the rectangular coordinates based on the magnitude in the real part and the angle (in radians) in the imaginary part of the points in waveIn.

You may get unexpected results if the number of points in waveIn differs from the number of points in waveOut.

See Also

The functions **cmplx**, **conj**, **imag**, **r2polar**, and **real**.

PadString

`PadString(`*`str, finalLength, padValue`*`)`

The PadString function returns a string identical to *str* except that it has been extended to a total length of *finalLength* using bytes of *padValue*. Use zero to create a C-language style string or use 0x20 to pad with spaces (FORTRAN style). This is useful when reading or writing binary files using **FBinRead** and **FBinWrite**.

See Also

The **UnPadString** function.

Panel

`Panel`

Panel is a procedure subtype keyword that identifies a macro as being a control panel recreation macro. It is automatically used when Igor creates a window recreation macro for a control panel. See **Procedure Subtypes** on page IV-193 and **Saving a Window as a Recreation Macro** on page II-42 for details.

PanelResolution

`PanelResolution(`*`wName`*`)`

The PanelResolution function returns the current resolution of the specified panel in pixels per inch. If *wName* is empty, it reutrns the current global setting for panel resolution. If *wName* is the name of a graph window, it returns the resolution for the **ControlBar** area. *wName* can be a subwindow specification.

The PanelResolution function was added in Igor Pro 7.00.

In general, PanelResolution and **ScreenResolution** return the same thing. However, on Windows when the screen resolution is 96 DPI, which is typical for normal-resolution screens, panels can use 72 DPI for compatibility with Igor Pro 6 and earlier.

See Also

Control Panel Resolution on Windows on page III-405, **ScreenResolution**

ParamIsDefault

`ParamIsDefault(`*`pName`*`)`

The ParamIsDefault function determines if an optional user function parameter *pName* was specified during the function call. It returns 1 when *pName* is default (not specified) or it returns 0 when it was specified.

Details

ParamIsDefault works only in the body of a user function and only with optional parameters. The variable *pName* must be valid at compile time; you can not defer lookup to runtime with $.

See Also

Optional Parameters on page IV-33 and **Using Optional Parameters** on page IV-55.

ParseFilePath

`ParseFilePath(`*`mode`*`, `*`pathInStr`*`, `*`separatorStr`*`, `*`whichEnd`*`, `*`whichElement`*`)`

The ParseFilePath function provides the ability to manipulate file paths and to extract sections of file paths.

Parameters

The meaning of the parameters depends on *mode*.

mode	**Information Returned**
0	Returns the element specified by *whichEnd* and *whichElement*. *whichEnd* is 0 to select an element relative to the beginning of *pathInStr*, 1 to select an element relative to the end. *whichElement* is zero-based. Pass ":" if *pathInStr* is a Macintosh HFS path, "\\" if it is a Windows path. See **Path Separators** on page III-401 for details about Macintosh versus Windows paths.
1	Returns the entire *pathInStr*, up to but not including the element specified by *whichEnd* and *whichElement*. *whichEnd* is 0 to select an element relative to the beginning of *pathInStr*, 1 to select an element relative to the end. *whichElement* is zero-based. Pass ":" if *pathInStr* is a Macintosh HFS path, "\\" if it is a Windows path. See **Path Separators** on page III-401 for details about Macintosh versus Windows paths.
2	Returns the entire *pathInStr* with a trailing separator added if it is not already there. This is useful when you have a path to a folder and want to tack on a file name. Pass ":" if *pathInStr* is a Macintosh HFS path, "\\" if it is a Windows path. See **Path Separators** on page III-401 for details about Macintosh versus Windows paths. *whichEnd* and *whichElement* are ignored. Pass 0 for them.
3	Returns the last element of *pathInStr* with the extension, if any, removed. The extension is anything after the last dot in *pathInStr*. *whichEnd* and *whichElement* are ignored. Pass 0 for them.
4	Returns the extension in *pathInStr* or `""` if there is no extension. The extension is anything after the last dot in *pathInStr*. Pass ":" if *pathInStr* is a Macintosh HFS path, "\\" if it is a Windows path. See **Path Separators** on page III-401 for details about Macintosh versus Windows paths. *whichEnd* and *whichElement* are ignored. Pass 0 for them.
5	Returns the entire *pathInStr* but converts it to a format determined by *separatorStr*.

mode	**Information Returned**
	separatorStr = ":" Converts the path to Macintosh HFS style if it is Windows style. Does nothing to a Macintosh HFS path. *separatorStr* = "\\" Converts the path to Windows style if it is Macintosh style. Does nothing to a Windows path. *separatorStr* = "*" Converts the path to the native style of the operating system Igor is running on. Does nothing to a native path. For historical reasons, on Macintosh "native" means colon-separated HFS path, not UNIX path. *separatorStr* = "/" Macintosh-only: Converts the Macintosh-style *pathInStr* input to a Posix (UNIX) path. Unlike the other conversions, the directory or file to which *pathInStr* refers must exist, otherwise "" is returned. To generate a Posix path for a non-existent file, generate the path for the existing folder and append the file name. This always returns "" on Windows. *whichEnd* and *whichElement* are ignored. Pass 0 for them.
6	UNC volume name ("\\Server\Share") if *pathIn* starts with a UNC volume name or "" if not. Pass "*" for *separatorStr*. *whichEnd* and *whichElement* are ignored. Pass 0 for them.
7	UNC server name ("Server" from "\\Server\Share") if *pathIn* starts with a UNC volume name or "" if not. Pass "*" for *separatorStr*. *whichEnd* and *whichElement* are ignored. Pass 0 for them.

mode	**Information Returned**
8	UNC share name ("Share" from "\\Server\Share") if *pathIn* starts with a UNC volume name or "" if not. Pass "*" for *separatorStr*. *whichEnd* and *whichElement* are ignored. Pass 0 for them.
9	Macintosh only. On Windows this mode returns an error. Returns a Posix version of *pathInStr* which must be a full HFS path pointing to an existing volume, directory or file. This is the same as mode 5 except that *separatorStr* must be "*". *whichEnd* and *whichElement* are ignored. Pass 0 for them. You would typically use this mode when you are about to execute a Unix command, which requires Posix paths, via ExecuteScriptText. This mode was created in Igor Pro 7.00 to provide an alternative to the obsolete HFSToPosix function provided by the HSFAndPosix XOP. With HFSToPosix, if the input path referred to a directory, the output always ended with a slash. With ParseFilePath(9), the output will end with a slash only if the input path ends with a colon.
10	Macintosh only. On Windows this mode returns an error. Returns the HFS path corresponding to the Posix path in pathInStr . *pathInStr* must be a full Posix path starting with a slash character. It does not need to point to an existing directory or file. The returned path may or may not refer to an existing volume, folder or file, depending on pathInStr . Pass "*" for *separatorStr*. *whichEnd* and *whichElement* are ignored. Pass 0 for them. You would typically use this mode when you receive a Posix path from a Unix command executed via ExecuteScriptText and you want to use that path in Igor. This mode was created in Igor Pro 7.00 to provide an alternative to the obsolete PosixToHFS function provided by the HSFAndPosix XOP.

Details

When dealing with Windows paths, you need to be aware that Igor treats the backslash character as an escape character. When you want to put a backslash in a literal string, you need to use two backslashes. See **Escape Sequences in Strings** on page IV-13 and **Path Separators** on page III-401 for details.

On Windows two types of file paths are used: drive-letter paths and UNC (“Universal Naming Convention”) paths. For example:

```
// This is a drive-letter path.
C:\Program Files\WaveMetrics\Igor Pro 7 Folder\Igor64.exe

// This is a UNC path.
\\BigServer\SharedApps\Igor Pro 7 Folder\Igor64.exe
```

In this example, ParseFilePath considers the volume name to be `C:` in the first case and `\\BigServer\SharedApps` in the second. The volume name is treated as one element by ParseFilePath, except for modes 7 and 8 which permit you to extract the components of the UNC volume name.

Except for the leading backslashes in a UNC path, ParseFilePath modes 0 and 1 internally strip any leading or trailing separator (as defined by the *separatorStr* parameter) from `pathInStr` before it starts parsing. So if you pass `":Igor Pro 7 Folder:WaveMetrics Procedures:"`, it is the same as if you had passed `"Igor Pro 7 Folder:WaveMetrics Procedures"`.

If there is no element corresponding to *whichElement* and *mode* is 0, ParseFilePath returns `""`.

If there is no element corresponding to *whichElement* and *mode* is 1, ParseFilePath returns the entire *pathInStr*.

Examples

```
String pathIn, pathOut
// Full path
pathIn= "hd:Igor Pro 7 Folder:WaveMetrics Procedures:Waves:Wave Lists.ipf"
// Extract first element.
Print ParseFilePath(0, pathIn, ":", 0, 0)      // Prints "hd"
// Extract second element.
Print ParseFilePath(0, pathIn, ":", 0, 1)   // Prints "Igor Pro 7 Folder"
// Extract last element.
Print ParseFilePath(0, pathIn, ":", 1, 0)   // Prints "Wave Lists.ipf"
// Extract next to last element.
Print ParseFilePath(0, pathIn, ":", 1, 1)      // Prints "Waves"
// Get path to folder containing the file.
// Prints "hd:Igor Pro 7 Folder:WaveMetrics Procedures:Waves:"
Print ParseFilePath(1, pathIn, ":", 1, 0)
// Extract the file name without extension.
Print ParseFilePath(3, pathIn, ":", 0, 0)      // Prints "Wave Lists"
// Extract the extension.
Print ParseFilePath(4, pathIn, ":", 0, 0)      // Prints "ipf"
// Make sure the given path ends with a colon and concatenate file name.
String path = <routine that returns a Macintosh-style path to a folder>
path = ParseFilePath(2, path, ":", 0, 0)
path += "AFile.txt"
```

See Also

Escape Sequences in Strings on page IV-13, **UNC Paths** on page III-401, and **Path Separators** on page III-401 for details. The **RemoveEnding** function.

ParseOperationTemplate

ParseOperationTemplate [***flags***] ***cmdTemplate***

The ParseOperationTemplate operation helps XOP programmers and WaveMetrics programmers write code to implement Igor operations. If you are not an XOP programmer nor a WaveMetrics programmer, it will be of no interest.

ParseOperationTemplate generates starter code for programmers who are creating Igor operations. The starter code is copied to the clipboard, overwriting any previous clipboard contents.

Flags

/C=*c*	If *c* is nonzero, ParseOperationTemplate stores code for your ExecuteOperation and RegisterOperation functions in the clipboard.

c=0:	Do not generate code
c=1:	Generate simplified C code - not recommended
c=2:	Generate C code
c=6:	Generate C++ code

The only difference between /C=6 and /C=2 is that the ExecuteOperation function is declared as extern "C" instead of static. C++ files that use static work fine although extern "C" is correct.

/S=*s*	Stores a definition of your runtime parameter structure in the clipboard if *s* is nonzero.

s=0:	Do not generate the runtime parameter structure
s=1:	Use your mnemonic names - recommended
s=2:	Automatically generate mnemonic names - not recommended

We recommend that you use /S=1 and provide unique mnemonic parameter names in your template. ParseOperationTemplate then uses your parameter names as structure field names.

If you use /S=2, ParseOperationTemplate creates unique field names by concatenating flag or keyword text and your mnemonic names. This is left over from the early days of Operation Handler and is not recommended.

/T	Stores a comment listing your command template in the clipboard.
/TS	Identifies a ThreadSafe operation by adding an extra field to the runtime parameter structure. This is only of use to WaveMetrics programmers.

Parameters

cmdTemplate is the template that describes the syntax for your operation. See the *Igor XOP Toolkit Reference Manual* for details.

Details

ParseOperationTemplate parses your command template, generating structures that embody the syntax of your operation. It then uses these structures to generate code that can serve as a starting point for implementing your operation. The starter code is stored in the clipboard.

For most uses, the recommended flags are:

```
/T/S=1/C=2       // For non-threadsafe operations
/T/S=1/C=2/TS    // For threadsafe operations
```

ParseOperationTemplate sets the following output variable, but only when called from a function or macro:

`V_flag`	0:	*cmdTemplate* was successfully parsed.
	-1:	*cmdTemplate* was not successfully parsed.

If V_flag is nonzero, this indicates that your *cmdTemplate* syntax is incorrect. See the *Igor XOP Toolkit Reference Manual* for details.

Examples

```
Function Test()
   String cmdTemplate
   cmdTemplate = "MyTest"
   cmdTemplate += " /A={number:aNum1,string:aStrH}"
   cmdTemplate += " /B=wave:bWaveH"
   cmdTemplate += " key1={name:k1N1[,wave:k1WaveH,name:k1N2,string[2]:k1StrHArray]}"

   // If your XOP is C instead of C++, use /C=2 instead of /C=6
   TestOperationParser/T/S=1/C=6 cmdTemplate
   Print V_flag, S_value
End
```

See Also

Igor Extensions on page III-450.

PathInfo

`PathInfo` `[`**`/S /SHOW`** `]` ***`pathName`***

The PathInfo operation stores information about the named symbolic path in the following variables:

`V_flag`:	0 if the symbolic path does not exist, 1 if it does exist.
`S_path`:	The full path (e.g., "hd:This:That:").

The path returned is a colon-separated path which can be used on Macintosh or Windows. See **Path Separators** on page III-401 for details.

Flags

/S	Presets the next otherwise undirected open or save file dialog to the given disk folder.
/SHOW	Shows the folder, if it exists, in the Finder (Mac OS X) or Windows Explorer (Windows).

Examples

```
// The following lines perform equivalent actions:
PathInfo/S myPath;Open refNum
Open/P=myPath refNum
```

```
// Show Igor's Preferences folder in the Finder/Windows Explorer.
String fullpath= SpecialDirPath("Preferences",0,0,0)
NewPath/O/Q tempPathName, fullpath
PathInfo/SHOW tempPathName
```

See Also

Symbolic Paths on page II-21.

The **NewPath**, **GetFileFolderInfo**, **ParseFilePath** and **SpecialDirPath** operations.

PathList

PathList(*matchStr*, *separatorStr*, *optionsStr*)

The PathList function returns a string containing a list of symbolic paths selected based on the *matchStr* parameter.

Details

For a path name to appear in the output string, it must match *matchStr*. The first character of *separatorStr* is appended to each path name as the output string is generated.

PathList works like the WaveList function, except that the *optionsStr* parameter is reserved for future use. Pass `""` for it.

Examples

When a new experiment is created there is only one path:

```
Print PathList("*",";","")
```

Prints the following in the history area:

```
  Igor;
```

See Also

The **WaveList** function for an explanation of the *matchStr* and *separatorStr* parameters and for examples. See also **Symbolic Paths** on page II-21 for an explanation of symbolic paths.

PauseForUser

PauseForUser [**/C**] ***mainWindowName*** [**,** ***targetWindowName***]

The PauseForUser operation pauses procedure execution to allow the user to manually interact with a window. For example, you can call PauseForUser from a loop to allow the user to move the cursors on a graph. In this scenario, *targetWindowName* would be the name of the graph and *mainWindowName* would be the name of a control panel containing a message telling the user to adjust the cursors and then click, for example, the Continue button.

If *targetWindowName* is omitted then *mainWindowName* plays the role of target window.

PauseForUser works with graph, table, and panel windows only.

Flags

/C	Tells PauseForUser to return immediately after handling any pending events. See Details.

Details

During execution of PauseForUser, only mouse and keyboard activity directed toward either *mainWindowName* or *targetWindowName* is allowed.

While waiting for user action, PauseForUser disables double-clicks and any contextual menus that can lead to dialogs in order to prevent changes on the command line. It also disables killing windows by clicking the close icon in the title bar unless the window was originally created with the /K=1 flag (kill with no dialog).

If /C is omitted, PauseForUser returns only when the main window has been killed.

If /C is present, PauseForUser handles any pending events, sets V_Flag to the truth the target window still exists, and then returns control to the calling user-defined function. Use PauseForUser/C in a loop if you need to do something while waiting for the user to finish interacting with the target window.

See Also

Pause For User on page IV-140 for examples and further discussion.

PauseUpdate

`PauseUpdate`

The PauseUpdate operation delays the updating of graphs and tables until you invoke a corresponding ResumeUpdate command.

Details

PauseUpdate is useful in a macro that changes a number of things relating to the appearance of a graph. It prevents the graph from being updated after each change. Its effect ends when the macro in which it occurs ends. It also affects updating of tables.

This operation is not allowed from the command line. It is allowed but has no effect in user-defined functions. During execution of a user-defined function, windows update only when you explicitly call the DoUpdate operation.

See Also

The **DelayUpdate**, **DoUpdate**, **ResumeUpdate,** and **Silent** operations.

PCA

`PCA [`*`flags`*`][`*`wave0, wave1,… wave99`*`]`

The PCA operation performs principal component analysis. Input data can be in the form of a list of 1D waves, a single 2D wave, or a string containing a list of 1D waves. The operation can produce multiple output waves depending on the specified flags.

Flags

/ALL	Shortcut for the combination of commonly used flags: /CVAR, /SL, /NF, /IND, /IE, and /RMS.
/COV	Calculates the input wave(s) covariance matrix, which as the input for the remainder of the analysis. The covariance matrix is computed by first creating a matrix copying each input 1D wave into sequential columns and then multiplying that matrix by its transpose.
/CVAR	Computes the cumulative percent variance defined as 100 * sum of first *m* eigenvalues divided by the sum of all eigenvalues. The results are stored in the wave W_CumulativeVAR in the current data folder. See also /VAR.
/IE	Computes the imbedded error. Returns errors in the wave W_IE in the current data folder. The wave is scaled using `SetScale/P x 1,1,"", W_IE`. The imbedded error is a function of the number of factors, the number of rows and columns and the sum of the eigenvectors not included in the significant factors. The behavior of IE determines the number of significant factors.

/IND	Computes the factor indicator function. Note that if you specify /IND the residual standard deviation will also be calculated. Returns results in the wave W_IND in the current data folder. The wave is scaled using `SetScale/P x 1,1,"", W_IND`.
/LEIV	Limits eigenvalues so that the SVD calculation does not require too much memory. The limit is set to the minimum of the number of rows or columns of the input.
/NF	Finds the number of significant factors and stores it in the variable V_npnts. You must use /IND in order to compute the significant factors.
/O	Overwrites input waves.
/Q	Suppresses printing of factors in the history area.
/RSD[=*rsdMode*]	Computes the Residual Standard Deviation (RSD) and returns the RSD in the wave W_RSD in the current data folder. The first element in W_RSD is NaN and all remaining wave elements correspond to the number of significant factors. *rsdMode* =0: Covariance about the origin. *rsdMode* =1: Correlation about the origin.
/RMS	Computes the RMS error. Returns results in the wave W_RMS in the current data folder. The wave is scaled using `SetScale/P x 1,1,"", W_RMS`.
/SCMT	Saves C matrix after the singular value decomposition (SVD) in the wave M_C in the current data folder.
/SCR	Converts the individual wave input into standard scores. Does not work when the input is a single 2D wave. It is an error to convert to standard scores when one or more entries in the waves are NaN or INF. If you use this feature make sure to use the appropriate form of the RSD calculation.
/SDM	Saves a copy of the data matrix at the end of the calculation. This is useful if your input consists of individual waves or if you want to save the computed standard scores. If the input is a 2D matrix, you will get a copy of the input matrix in the wave M_D.
/SEVC	Saves the eigenvalue vector in the wave W_Eigen, which are the raw eigenvalues generated by the SVD, in the current data folder. Normally, if the SVD was applied to a raw data matrix, i.e., not covariance or correlation matrix, you must square each element of the wave to obtain the PCA eigenvalues. Note that this wave has default wave scaling.
/SL	Computes percent significance level and stores it in the wave W_PSL in the current data folder.
/SQEV	Does not square SVD eigenvalues. If you specify /COV there is no need to use this flag. Use only if your input is already a covariance matrix. In this case the results of the SVD are the eigenvalues not their square roots.
/SRMT	Saves R matrix after the SVD in the wave M_R in the current data folder.
/U	Leaves the input waves unchanged only when the input is a 2D wave. Note that covariance calculations will not be made even if the appropriate flag is used.
/VAR	Computes the variance associated with each eigenvalue. The variance is defined as the ratio of the eigenvalue to the sum of all eigenvalues. The results are stored in the wave W_VAR in the current data folder. See also /CVAR above.
/WSTR=*waveListStr*	String containing a list of names for all input waves.
/Z	No error reporting.

Details

The input is either via /WSTR=*waveListStr* or a list of up to 100 1D waves or a single 2D wave following the last flag.

waveListStr is string containing a semicolon-separated list of 1D waves to be used for the data matrix. *waveListStr* can include any legal path to a wave. Liberal names can be quoted or not quoted. It is assumed that all waves are of the same numerical type (either single or double precision) and that all waves have the same number of points.

Regardless of the inputs, the operation expects that the number of rows in the resulting matrix is greater than or equal to the number of columns.

The operation starts by creating the data matrix from the input wave(s). If you provide a list of 1D waves they become the columns of the data matrix. You can choose to use the covariance matrix (/COV) as the data matrix and you can also choose to normalize each column of the data matrix to convert it into standard scores. This involves computing the average and standard deviation of each column and then setting the new values to be:

$$newValue = \frac{oldValue - colAverage}{colStdv}.$$

You can pre-process the input data using **MatrixOp** with the SubtractMean, NormalizeRows, and NormalizeCols functions.

After creating the data matrix the operation computes the singular value decomposition (SVD) of the data matrix. Results of the SVD can be saved or processed further. Save the C and R matrices using /SCMT and /SRMT. These are related to the input data matrix through: $D = R \cdot C$.

The remainder of the operation lets you compute various statistical quantities defined by Malinowski (see References). Use the flags to determine which ones are computed.

The operation generates a number of output waves. All waves are stored in the current data folder.

You can save the input matrix D in the wave M_D, the optional SVD results are stored in the waves M_C that contains the column matrix C, M_R that contains the row matrix R, and W_Eigen that contains the eigenvalues of the data matrix. Note that these can be the eigenvalues or the square of the eigenvalues depending on the input matrix being a covariance matrix or not (see /SQEV).

The optional 1D output waves (W_RSD, W_RMS, W_IE, W_IND, W_PSL) are saved with wave scaling to make it easier to display the wave as a function of the number of factors.

References

Kaiser, H., Computer Program for Varimax Rotation in Factor Analysis, *Educational and Psychological Measurement*, *XIX*, 413-420, 1959.

Malinowski, E.R., *Factor Analysis in Chemistry*, 3rd ed., John Wiley, 2002.

See Also

ICA

pcsr

`pcsr(`*`cursorName`* `[,` *`graphNameStr`*`])`

The pcsr function returns the point number of the point which the specified cursor (A through J) is on in the top (or named) graph. When used with cursors on images or waterfall plots, pcsr returns the row number, and when used with a free cursor, it returns the relative X coordinate.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The pcsr result is not affected by any X axis.

See Also

The **hcsr**, **qcsr**, **vcsr**, **xcsr**, and **zcsr** functions.

Programming With Cursors on page II-249.

Pi

`Pi`

The Pi function returns π (3.141592…).

PICTInfo

`PICTInfo(`*`pictNameStr`*`)`

The PICTInfo function returns a string containing a semicolon-separated list of information about the named picture. If the named picture does not exist, then `""` is returned. Valid picture names can be found in the Pictures dialog.

Details

The string contains six pieces of information, each prefaced by a keyword and colon and terminated with a semicolon.

Keyword	Information Following Keyword
TYPE	One of: “PICT”, “PNG”, “JPEG”, “Enhanced metafile”, “Windows metafile”, “DIB”, “Windows bitmap”, or “Unknown type”.
BYTES	Amount of memory used by the picture.
WIDTH	Width of the picture in pixels.
HEIGHT	Height of the picture in pixels.
PHYSWIDTH	Physical width of the picture in points.
PHYSHEIGHT	Physical height of the picture in points.

Examples

```
Print PICTInfo("PICT_0")
```

will print the following in the history area:

```
TYPE:PICT;BYTES:55734;WIDTH:468;HEIGHT:340;PHYSWIDTH:468;PHYSHEIGHT:340;
```

See Also

The **ImageLoad** operation for loading PICT and other image file types into waves, and the **PICTList** function. The **StringFromList** operation for parsing the information string.

See **Pictures** on page III-448 and **Pictures Dialog** on page III-449 for general information on picture handling.

PICTList

`PICTList(`*`matchStr`*`, `*`separatorStr`*`, `*`optionsStr`*`)`

The PICTList function returns a string containing a list of pictures based on *matchStr* and *optionsStr* parameters. See **Details** for information on listing pictures in graphs, panels, layouts, and the picture gallery.

Details

For a picture name to appear in the output string, it must match *matchStr* and also must fit the requirements of *optionsStr*. The first character of *separatorStr* is appended to each picture name as the output string is generated.

The name of each picture is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all picture names.
"xyz"	Matches picture name xyz only.
"*xyz"	Matches picture names which end with xyz.

"xyz*"	Matches picture names which begin with xyz.
"*xyz*"	Matches picture names which contain xyz.
"abc*xyz"	Matches picture names which begin with abc and end with xyz.

matchStr may begin with the ! character to return windows that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches picture names which do not end with xyz.

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

optionsStr is used to further qualify the picture.

Use `""` accept all pictures in the Pictures Dialog that are permitted by *matchStr*.

Use the WIN: keyword to limit the pictures to the named or target window:

`"WIN:"`	Match all pictures displayed in the top graph, panel, or layout.
`"WIN:`*`windowName`*`"`	Match all pictures displayed in the named graph, panel, or layout window.

Examples

`PICTList("*",";","")`	Returns a list of all pictures in the Pictures Dialog.
`PICTList("*", ";","WIN:")`	Returns a list of all pictures displayed in the top panel, graph, or layout.
`PICTList("*_bkg", ";", "WIN:Layout0")`	
	Returns a list of pictures whose names end in “_bkg” and which are displayed in Layout0.

See Also

The **ImageLoad** operation for loading PICT and other image file types into waves, and the **PICTInfo** function. Also the **StringFromList** function for retrieving items from lists.

See **Pictures** on page III-448 and **Pictures Dialog** on page III-449 for general information on picture handling.

Picture

`Picture` ***`pictureName`***

The Picture keyword introduces an ASCII code picture definition of binary image data.

See Also

Proc Pictures on page IV-53 for further information.

PixelFromAxisVal

`PixelFromAxisVal(`*`graphNameStr, axNameStr, val`*`)`

The PixelFromAxisVal function returns the local graph pixel coordinate corresponding to the axis value in the graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

If the specified axis is not found and if the name is “left” or “bottom” then the first vertical or horizontal axis will be used.

If *graphNameStr* references a subwindow, the returned pixel value is relative to top left corner of base window, not the subwindow.

See Also

The **AxisValFromPixel** and **TraceFromPixel** functions.

PlayMovie

`PlayMovie [`*`flags`*`] [as `*`fileNameStr`*`]`

The PlayMovie operation opens a movie file in a window and plays it.

Parameters

The file to be opened is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

On Windows the file is passed to the operating system to be opened with the default program for the given filename extension and the /W flag is ignored.

Flags

/I	Coordinates are in inches.
/M	Coordinates are in centimeters.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/W=(*left*,*top*,*right*,*bottom*)	Sets the initial coordinates of the movie window (in points unless /I or /M are used before /W).
/Z	No error reporting; an error is indicated by nonzero value of the variable V_flag. If the user clicks the cancel button in the Open File dialog, V_flag is set to -1.

Details

Coordinates are the initial coordinates of the movie window in points unless /I or /M are used before /W. Only the *top* and *left* coordinates are used. The window has the standard width and height for movies.

If either the path or *fileNameStr* is omitted then PlayMovie will bring up a dialog to let you find a movie file. If both are present, PlayMovie opens the file automatically.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-21 for details.

Any movie file can be played, not just movies made by Igor. There is no limit on the number of movie windows opened for playing.

Movie windows are considered transient and are not restored when an experiment is reopened.

See Also

Movies on page IV-230.

The **PlayMovieAction** operation.

PlayMovieAction

`PlayMovieAction` `[/Z]` ***`keyword`*** `[=`***`value`***`][,` ***`keyword`*** `[=`***`value`***`]]`

On Macintosh PlayMovieAction operates on the top movie window, opened via **PlayMovie**, or on a movie file opened via the open keyword (requires Igor Pro 7 or later).

On Windows only movies opened via the open keyword are supported.

If the /Z flag is present, errors are not fatal. V_flag is set to error return regardless.

Parameters

extract	Extracts current frame into an 8-bit RGB image wave named M_MovieFrame. (Can be combined with frame=*f*.)
extract=*e*	Extracts *e* frames into a single multiframe wave, M_MovieChunk. This wave will have 3 planes for RGB and will have *e* chunks.
frame=*f*	Moves to specified movie frame (and stops movie).
getID	Returns top movie ID number in V_Value. Do not use in same call with getTime.
getTime	Reads current movie time into variable V_value (in seconds).
gotoBeginning	Goes to beginning of movie.
gotoEnd	Goes to end of movie.
kill	Kills movie window or closes open movie window.
open=*fullPath*	Opens the specifed movie file to enable frame extraction. No movie window is involved. V_Flag is set to zero if no error occurred and V_Value is set to the file reference number. Prior to Igor Pro 7 the open keyword was supported on Windows only. It now also works on Macintosh. Prior to Igor Pro 7, the ref keyword was required for PlayMovieAction calls after the open keyword. Now it is needed only if multiple files or windows are open. Even then, you can use setFrontMovie to set the active movie window.
ref=*refNum*	The ref keyword is used with all PlayMovieAction commands after using the open keyword to access a movie file. *refNum* must be the file reference number returned in V_Value in the open step. Prior to Igor Pro 7 the ref keyword was supported on Windows only. It now also works on Macintosh.
setFrontMovie= *id*	Sets movie with given *id* as top window or active file. Error if no such window or file (use /Z to suppress errors). Do not use in same call with getID.
start	Starts movie playing. The start keyword works on Macintosh only and only with movie windows, not with files.
step=*s*	Moves by *s* frames into movie (0 is same as 1, negative values move backwards).
stop	Stops movie. The stop keyword works on Macintosh only and only with movie windows, not with files.

Flags

/Z	No error reporting; an error is indicated by nonzero value of the variable V_flag.

Details

Operations are performed in the following order: kill, stop, gotoBeginning, gotoEnd, frame, step, getTime, extract, start. kill overrides all other parameters.

If you want to extract a grayscale image, you can convert the RGB image into grayscale using the ImageTransform command as follows:

```
PlayMovieAction extract
ImageTransform rgb2gray M_MovieFrame
NewImage M_RGB2Gray
```

The open and ref keywords support extracting frames from files with out the need for an open movie window. Prior to Igor Pro 7, this was a Windows only feature. It now also works on Macintosh..

When accessing a file using the open keyword, none of the keywords related to movie windows or playing a movie are supported.

When you are finished extracting frames, use the kill keyword to close the file.

To get a full path for use with the open keyword, use the **PathInfo** or **Open** /D/R commands.

Examples

These commands show to determine the number of frames in a simple movie:

```
PlayMovieAction open = <full path to movie file>
PlayMovieAction stop,gotoEnd,getTime
Variable tend= V_value
PlayMovieAction step=-1,getTime
Print "frames= ",tend/(tend-V_value)
PlayMovieAction kill
```

See Also

Movies on page IV-230.

The **PlayMovie** operation.

PlaySnd

PlaySnd [***flags***] ***fileNameStr***

Note: PlaySnd is obsolete. Use **PlaySound** instead.

Available only on the Macintosh.

The PlaySnd operation plays a sound from the file's data fork, or from an `'snd '` resource.

Parameters

The file containing the sound is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/I=*resourceIndex*	Specifies the `'snd '` resource to load by resource index, starting from 1.
/M=*promptStr*	Specifies a prompt if PlaySnd needs to put up a dialog to find the file.
/N=*resNameStr*	Specifies the resource to load by resource name.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/Q	Quiet: suppresses the insertion of `'snd '` info into the history area.
/R=*resourceID*	Specifies the `'snd '` resource to load by resource ID.
/Z	Does not play the sound, just checks for its existence.

Details

If none of /I, /N or /R are specified, PlaySnd tries to play a sound stored in the data fork of the file. If the file dialog is used, only files of type `'sfil'` are shown.

If any of /I, /N or /R are specified, PlaySnd tries to play a sound from an `'snd '` resource. Most programs store sounds in `'snd '` resources. If the file dialog is used, files of all types are shown.

If /P=*pathName* is omitted, then *fileNameStr* can take on three special values:

"Clipboard"	Loads data from Clipboard.
"System"	Loads data from System file.
"Igor"	Loads data from Igor Pro application.

If you specify /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-21 for details.

There are no sounds in the Igor Pro application file.

If the file is not fully specified and *fileNameStr* is not one of these special values, then PlaySnd presents a dialog from which you can select a file. "Fully specified" means that Igor can determine the name of the file (from the *fileNameStr* parameter) and the folder containing the file (from the /P=*pathName* flag or from the *fileNameStr* parameter).

PlaySnd sets the variable V_flag to 1 if the sound exists and fits in available memory or to 0 otherwise.

If the sound exists, PlaySnd also sets the string variable S_Info to:

`"SOURCE:`*`sourceName`*`;RESOURCENAME:`*`resourceName`*`;RESOURCEID:`*`resourceID`*`"`

If the sound is not a resource then *resourceName* is `""` and *resourceID* is 0. *sourceName* will be the name of the file that was loaded or "Clipboard", "System" or "Igor".

Examples

```
PlaySnd/I=1/P=mySnds/Z "Wild Eep"
If (V_flag)             // Any 'snd ' in the "Wild Eep" file?
   Print S_info         // Yes, print resource number, etc.
Endif
```

This prints the following into the history area:

```
SOURCE:resource fork;RESOURCENAME:Wild Eep;RESOURCEID:8;
```

PlaySound

`PlaySound [/A[=`*`a`*`] /C]` ***`soundWave`***

`PlaySound /A[=`*`a`*`] {`*`soundWave1, soundWave2`*` [, `*`soundWaveN…`*`]}`

The PlaySound operation plays the audio samples in the named wave. The various sound output parameters — number of samples, sample rate, number of channels, and number of bits of resolution — are determined by the corresponding parameters of the wave.

Flags

/A[=*a*] Plays sounds asynchronously so that sounds will continue to play after the command itself has executed.

/A=0:	Same as no /A flag.
/A=1:	Plays sounds asynchronously; same as /A.
/A=2:	Stop playing any current sound before starting this one.
/A=3:	Return with user abort error if output buffers are full (rather than waiting.) Use `GetRTError(1)` to detect and clear the error condition.

/C Obsolete - do not use.

On Windows /C causes sound wave data greater than 16-bits to be converted to 16-bit integer. Such data should range from -32768 to +32767.

On Macintosh /C is ignored.

Details

The wave's time per point, as determined by its X scaling, must be a valid sampling rate. A value of 1/44100 (CD standard) is typical.

Sound waves should be 16 bit integers with a range of -32768 to +32767. On Macintosh as of Igor version 6.11, 32-bit floating point data with a range of -1 to +1 can also be used. For backward compatibility, 8-bit integer data with a range of -128 to +127 is also supported.

With the /A flag, the sound plays asynchronously (i.e., the command returns before the sound is finished). If another command is issued before the sound is finished then the new command will wait until the last sound finishes. A PlaySound without the /A flag can play on top of the current sound. The transition between sounds should be seamless on Macintosh but may be slightly delayed on Windows.

It is OK to kill a sound wave immediately after PlaySound returns even if the /A flag is used.

To play a stereo sound, provide a 2 column wave with the left channel in column 0. Actually, the software will attempt to play as many channels as there are columns in the wave. You can also use multiple1D waves with the /A flag. To use this method, enclose the list of 1D waves in braces

Note: The SoundInput operations provide matching sound recording capabilities. See the **SoundInStatus** operation.

Examples

Under Windows, support for sound is somewhat idiosyncratic so these sound examples may not work correctly with your particular hardware configuration.

```
Make/B/O/N=1000 sineSound                  // 8 bit samples
SetScale/P x,0,1e-4,sineSound              // Set sample rate to 10Khz
sineSound= 100*sin(2*Pi*1000*x)            // Create 1Khz sinewave tone
PlaySound sineSound
```

The following example will create a rising pitch in the left channel and a falling pitch in the right channel:

```
Make/W/O/N=(20000,2) stereoSineSound       // 16 bit data
SetScale/P x,0,1e-4,stereoSineSound        // Set sample rate to 10Khz
stereoSineSound= 20000*sin(2*Pi*(1000 + (1-2*q)*150*x)*x)
PlaySound/A stereoSineSound                // 16 bit, asynchronous
```

Multichannel sounds as in the previous example but from multiple 1D waves:

```
Make/W/O/N=20000 stereoSineSoundL,stereoSineSoundR    // 16 bit data
SetScale/P x,0,1e-4,stereoSineSoundL,stereoSineSoundR// Set sample rate to 10Khz
stereoSineSoundL= 20000*sin(2*Pi*(1000 + 150*x)*x)// rising pitch in left
stereoSineSoundR= 20000*sin(2*Pi*(1000 - 150*x)*x)// falling in right
PlaySound/A {stereoSineSoundL,stereoSineSoundR}   // two 1D waves
```

See Also

SoundLoadWave, **SoundSaveWave**

pnt2x

`pnt2x(`*`waveName, pointNum`*`)`

The pnt2x function returns the X value of the named wave at the point *pointNum*. The point number is truncated to an integer before use.

For higher dimensions, use **IndexToScale**.

Details

The result is derived from the wave's X scaling, not any X axis of a graph it may be displayed in.

If you would like to convert a fractional point number to an X value you can use:
`leftx(`*`waveName`*`)+deltax(`*`waveName`*`)*`*`pointNum`*.

See Also

DimDelta, **DimOffset**, **x2pnt**, **IndexToScale**

Waveform Model of Data on page II-57 and **Changing Dimension and Data Scaling** on page II-63 for an explanation of waves and dimension scaling.

Point

The Point structure is used as a substructure usually to store the location of the mouse on the screen.

```
Structure Point
   Int16 v
   Int16 h
EndStructure
```

PointF

The PointF structure is the same as Point but with floating point fields.

```
Structure Point
   float v
   float h
EndStructure
```

poissonNoise

`poissonNoise(`*`num`*`)`

The poissonNoise function returns a pseudo-random value from the Poisson distribution whose probability distribution function is

$$f(x;\lambda)=\frac{e^{-\lambda}\lambda^{x}}{x!},\qquad \begin{array}{c}\lambda>0\\ x=0,1,2...\end{array}$$

with mean and variance equal to *numl* (= λ).

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable “random” numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-344.

Chapter III-12, **Statistics** for a function and operation overview.

poly

`poly(`*`coefsWaveName, x1`*`)`

The poly function returns the value of a polynomial function at x = *x1*.

coefsWaveName is a wave that contains the polynomial coefficients. The number of points in the wave determines the number of terms in the polynomial.

Examples

To fill `wave0` with 100 points containing the polynomial $1 + 2*x + 3*x^2 + 4*x^3$ evaluated over the range from x = -1 to x= 1 (and graph it):

```
Make coefs = {1, 2, 3, 4}        // f(x) = 1 + 2*x + 3*x^2 + 4*x^3
Make/N=100/O wave0; SetScale/I x, -1, 1, wave0; Display wave0
wave0 = poly(coefs, x)
```

poly2D

`poly2D(`*`coefsWaveName, x1, y1`*`)`

The poly2D function returns the value of a 2D polynomial function at x = *x1*, y = *y1*.

coefsWaveName is a wave that contains the polynomial coefficients. The number of points in the wave determines the number of terms in the polynomial and therefore the polynomial degree.

Details

The coefficients wave contains polynomial coefficients for low degree terms first. All coefficients for terms of a given degree must be present, even if they are zero. Among coefficients for a given degree, those for terms having higher powers of X are first. Thus, poly2D returns, for a coefficient wave cw:

f(x,y) = cw[0] + cw[1]*x + cw[2]*y + cw[3]*x^2 + cw[4]*x*y + cw[5]*y^2 + …

A 2D polynomial of degree N has (N+1)(N+2)/2 terms.

Examples

To fill `wave0` with 400 points (20 by 20) containing the polynomial 1 + 2*x + 2.5*y + 3*x^2 + 3.5*xy + 4*y^2 evaluated over the range x = (-1, 1) and y = (-1, 1) and make a contour plot of it:

```
Make/O coefs = {1, 2, 2.5, 3, 3.5, 4}
Make/N=(20,20)/O wave0
SetScale/I x, -1, 1, wave0
SetScale/I y, -1, 1, wave0
wave0 = poly2D(coefs, x, y)
Display; AppendMatrixContour wave0
```

The polynomial is second degree, so the first command above made the wave coefs with six elements because (2+1)(2+2)/2 = 6.

To fill `wave0` with 100 points containing the polynomial 1 + 2*x + 3*y+ 4*x^2 + 4*y^2 + 5*x^3 + 6*y^3 (note the lack of cross terms) evaluated over the range x = (-1, 1) and y = (-1, 1) (the contour plot already made should update with the new data). The first zero eliminates the second-order cross term x*y and the second and third zeros eliminate the third-order cross terms x^2*y and x*y^2:

```
Make/O coefs = {1, 2, 3, 4, 0, 4, 5, 0, 0, 6}
wave0 = poly2D(coefs, x, y)
```

PolygonArea

`PolygonArea(`*`xWave, yWave`*`)`

The PolygonArea function returns the area of a simple, closed, convex or nonconvex planar polygon described by consecutive vertices in *xWave* and *yWave*.

A simple polygon has no internal “holes” and its boundary curve does not intersect itself. Both *xWave* and `yWave` must be 1D, real, numerical waves of the same dimensions. The minimum number of vertices is 3. The function uses the shoelace algorithm to compute the area (see theorem 1.3.3 in the reference below). If there is any error in the input, the function returns NaN.

Example

```
Function estimatePi(num)
   Variable num

   Make/O/N=(num+1) xxx,yyy
   xxx=sin(2*pi*x/num)
   yyy=cos(2*pi*x/num)

   printf "Relative Error=%g\r",(pi-PolygonArea(xxx,yyy))/pi
End
```

See also

The **areaXY** and **faverageXY** functions.

References

O’Rourke, Joseph, *Computational Geometry in C*, 2nd ed., Cambridge University Press, New York, 1998.

popup

`popup `*`menuList`*

The popup keyword is used with Prompt statements in Functions and Macros. It indicates that you want a pop-up menu instead of the normal text entry item in a DoPrompt simple input dialog (or a Macro’s missing parameter dialog (archaic)). *menuList* is a string expression containing a list of items, separated by semicolons, that are to appear in the pop-up menu.

Pop-up menus accept both numeric and string parameters. For numeric parameters, the number of the item selected is placed in the variable. Numbering starts from one. For string parameters, the selected item’s text is placed in the string variable.

Pop-up items support all of the special characters available for user-defined menu definitions (see **Special Characters in Menu Item Strings** on page IV-125) with the exception that items in pop-up menus are limited to 50 bytes, keyboard shortcuts are not supported, and special characters must be enabled.

See Also

Prompt, **DoPrompt**, and **Pop-Up Menus in Simple Dialogs** on page IV-133.

See **WaveList**, **TraceNameList**, **ContourNameList**, **ImageNameList**, **FontList**, **MacroList**, **FunctionList**, **StringList**, and **VariableList** for functions useful in generating lists of Igor objects.

Chapter III-14, **Controls and Control Panels** for details about control panels and controls.

PopupContextualMenu

`PopupContextualMenu [ /C=(`*`xpix, ypix`*`) /N /ASYN[=`*`func`*`] ]` ***`popupStr`***

The PopupContextualMenu operation displays a pop-up menu.

The menu appears at the current mouse position or at the location specified by the /C flag.

The content of the menu is specified by *popupStr* as a semicolon-separated list of items or, if you include the /N flag, by a user-defined menu definition referred to by the name contained in *popupStr*.

If you omit the /ASYN flag, the menu is tracked and the operation does not return until the user makes a selection or cancels the menu by clicking outside of its window.

If you include /ASYN, the menu is displayed and the operation returns immediately. When the user makes a selection, then the result is sent to the specified function or to the user-defined menu's execution text. You can use /ASYN to allow a background task to continue while a contextual menu is popped up. /ASYN requires Igor Pro 7.00 or later.

Parameters

If *popupStr* specifies the pop-up menu’s items (/N is not specified), then *popupStr* is a semicolon-separated list of items such as “yes;no;maybe;”, or a string expression that returns such a list, such as **TraceNameList**.

The menu items can be formatted and checkmarked, like user-defined menus can. See **Special Characters in Menu Item Strings** on page IV-125.

If /N is specified, *popupStr* must be the name of a user-defined menu that also has the popupcontextualmenu keyword. See Example 3.

Flags

/ASYN	When used with /N: The user-defined menu is displayed and operation returns immediately. The result of menu selection is handled by the user-defined menu's execution text. See **User-Defined Menus** on page IV-117.

/ASYN=*func*	When used without /N: The user-defined menu is displayed and operation returns immediately. The result of menu selection is handled by calling the named function, which must have the following format:
	`Function func(popupStr, selectedText, menuItemNum)`
	`   String popupStr`
	`   String selectedText`
	`   Variable menuItemNum`

/C=(*xpix*, *ypix*)	Sets the coordinates of the menu's top left corner. Units are in pixels relative to the top-most window or the window specified by /W, like the MOUSEX and MOUSEY values passed to a window hook. See the window hook example, below and **SetWindow**. If /C is not specified, the menu's top left corner appears at the current mouse position.
/N	Indicates that *popupStr* contains the name of a menu definition instead of containing a list of menu items.
/W=*winName*	The /C coordinates are relative to the top/left corner of the named window or subwindow. If you omit /W, /C uses the top-most window having focus. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy. /W was added in Igor Pro 7.00.

Details

If you omit /N and /ASYN, PopupContextualMenu sets the following variables:

`V_flag=0`	User cancelled the menu without selecting an item, or there was an error such as an empty *popupStr*.
`V_flag=`>= 1	1 if the first menu item was selected, 2 for the second, etc.
`S_selection`	`""` if the user cancelled or error, else the text of the selected menu item.

If you include /N and omit /ASYN, PopupContextualMenu sets the following variables in a manner similar to **GetLastUserMenuInfo**:

V_kind The kind of menu that was selected:

V_kind	Menu Kind
0	Normal text menu item, including **Optional Menu Items** (see page IV-122) and **Multiple Menu Items** (see page IV-122).
3	"*FONT*"
6	"*LINESTYLEPOP*"
7	"*PATTERNPOP*"
8	"*MARKERPOP*"
9	"*CHARACTER*"
10	"*COLORPOP*"
13	"*COLORTABLEPOP*"

See **Specialized Menu Item Definitions** on page IV-123 for details about these special user-defined menus.

V_flag -1 if the user didn't select any item, otherwise V_flag returns a value which depends on the kind of menu the item was selected from:

V_kind	V_flag Meaning
0	Text menu item number (the first menu item is number 1).
3	Font menu item number (use S_selection, instead).
6	Line style number (0 is solid line)
7	Pattern number (1 is the first selection, a SW-NE light diagonal).
8	Marker number (1 is the first selection, the X marker).
9	Character as an integer, = char2num(S_selection). Use S_selection instead.
10	Color menu item (use V_Red, V_Green, V_Blue, and V_Alpha instead).
13	Color table list menu item (use S_selection instead).

S_selection The menu item text, depending on the kind of menu it was selected from:

V_kind	S_selection Meaning
0	Text menu item text.
3	Font name or "default".
6	Name of the line style menu or submenu.
7	Name of the pattern menu or submenu.
8	Name of the marker menu or submenu.
9	Character as string.
10	Name of the color menu or submenu.
13	Color table name.

In the case of **Specialized Menu Item Definitions** (see page IV-123), S_selection will be the title of the menu or submenu, etc.

V_Red, V_Green, V_Blue, V_Alpha

If a user-defined color menu ("*COLORPOP*" menu item) was selected then these values hold the red, green, and blue values of the chosen color. The values range from 0 to 65535.

Will be 0 if the last user-defined menu selection was not a color menu selection.

If you include /N and /ASYN, PopupContextualMenu sets the following variables:

`V_flag=0`	There was an error such as an empty *popupStr* or *popupStr* did not name a compiled user-defined menu.
`V_flag=-1`	No error. The named user menu was valid and no item was selected yet.
`S_selection`	`""`

If you include /N and omit /ASYN, PopupContextualMenu sets the following variables:

`V_flag=0`	There was an error such as an empty *popupStr*.
`V_flag=-1`	No error. *popupStr* was valid and no item was selected yet.
`S_selection`	`""`

Examples

Example 1 - *popupStr* contains a list of menu items

```
// Menu formatting example
String checked= "\\M0:!" + num2char(18) + ":"  // checkmark code
String items= "first;\M1-;"+checked+"third;"   // 2nd is divider, 3rd is checked
PopupContextualMenu items
switch( V_Flag )
    case 1:
        // do something because first item was chosen
        break;
    case 3:
        // do something because first item was chosen
        break;
endswitch
```

Example 2 - *popupStr* contains a list of menu items

```
// Window hook example
SetWindow kwTopWin hook=TableHook, hookevents=1      // mouse down events
Function TableHook(infoStr)
    String infoStr

    Variable handledEvent=0
    String event= StringByKey("EVENT",infoStr)
    strswitch(event)
        case "mousedown":
            Variable isContextualMenu= NumberByKey("MODIFIERS",infoStr) & 0x10
            if( isContextualMenu )
                Variable xpix= NumberByKey("MOUSEX",infoStr)
                Variable ypix= NumberByKey("MOUSEY",infoStr)
                PopupContextualMenu/C=(xpix,ypix) "yes;no;maybe;"
                strswitch(S_selection)
                    case "yes":
                        // do something because "yes" was chosen
                        break
                    case "no":
                        break
                    case "maybe":
                        // do something because "maybe" was chosen
                        break
            endswitch
            handledEvent=1
        endif
    endswitch
    return handledEvent
End
```

Example 3 - popupStr contains the name of a user-defined menu

```
// User-defined contextual menu example

// dynamic menu (to keep WaveList items updated), otherwise not required.
// contextualmenu keyword is required, and implies /Q for all menu items.
//
// NOTE: Actions here are accomplished by the menu definition's
// execution text, such as DoSomethingWithColor.
// See Example 4 for another approach.
//
Menu "ForContext", contextualmenu, dynamic
    "Hello", Beep
    Submenu "Color"
        "*COLORPOP*", DoSomethingWithColor()
    End
    Submenu "Waves"
        WaveList("*",";",""), /Q, DoSomethingWithWave()
    End
End

Function DoSomethingWithColor()
    GetLastUserMenuInfo
    Print V_Red, V_Green, V_Blue, V_Alpha
End

Function DoSomethingWithWave()
    GetLastUserMenuInfo
    WAVE w = $S_value
    Print "User selected "+GetWavesDataFolder(w,2)
End

// Use this code in a function or macro:
PopupContextualMenu/N "ForContext"
if( V_flag < 0 )
    Print "User did not select anything"
endif
```

Example 4 - popupStr contains the name of a user-defined menu

```
// User-defined contextual menu example

Menu "JustColorPop", contextualmenu
    "*COLORPOP*(65535,0,0)", ;// initially red, empty execution text
End

// Use this code in a function or macro
PopupContextualMenu/C=(xpix, ypix)/N "JustColorPop"
if( V_flag < 0 )
    Print "User did not select anything"
else
    Print V_Red, V_Green, V_Blue, V_Alpha
endif
```

Example 5 - popupStr contains a list of menu items, asynchronous popup result

```
Function YourFunction()
    // Use this code in a function or macro:
    PopupContextualMenu/ASYN=Callback "first;second;third;"
    (YourFunction continues...)
End

// Routine called when/if popup menu item is selected. selectedItem=1 is the first item.
Function Callback(String list, String selectedText, Variable selectedItem)
```

```
    Print "Callback: ", list, selectedText, selectedItem
End
```

Example 6 - popupStr contains the name of a user-defined menu, asynchronous popup result

```
Function YourFunction()
    PopupContextualMenu/ASYN/N "ForContext"
    (YourFunction continues...)
End
// Selection result is handled in "ForContext" menu's execution texts, as in Example 4
```

See Also

Creating a Contextual Menu on page IV-149, **User-Defined Menus** on page IV-117.

Special Characters in Menu Item Strings on page IV-125 and Chapter III-14, **Controls and Control Panels**, for details about control panels and controls.

The **SetWindow** and **PopupMenu** operations.

PopupMenu

`PopupMenu [/Z] ctrlName [keyword = value [, keyword = value …]]`

The PopupMenu operation creates or modifies a pop-up menu control in the target or named window.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the PopupMenu control to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **Button** and **DefaultGUIControls** for more appearance details.

bodyWidth=*width* Specifies an explicit size for the body (nontitle) portion of a PopupMenu control. By default (bodyWidth=0), the body portion autosizes depending on the current text. If you supply a bodyWidth>0, then the body is fixed at the size you specify regardless of the body text. This makes it easier to keep a set of controls right aligned when experiments are transferred between Macintosh and Windows, or when the default font is changed.

disable=*d* Sets user editability of the control.

d=0:	Normal.
d=1:	Hide.
d=2:	Draw in gray state; disable control action.

fColor=(*r*,*g*,*b*) Sets the initial color of the title. *r*, *g*, and *b* range from 0 to 65535. fColor defaults to black (0,0,0). To further change the color of the title text, use escape sequences as described for title=*titleStr*.

focusRing=*fr* Enables or disables the drawing of a rectangle indicating keyboard focus:

fr=0:	Focus rectangle will not be drawn.
fr=1:	Focus rectangle will be drawn (default).

On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.

font="*fontName*" Sets the font used for the pop-up title, e.g., `font="Helvetica"`.

fsize=*s*	Sets the font size for the pop-up title.
fstyle=*fs*	*fs* is a bitwise parameter with each bit controlling one aspect of the font style as follows: Bit 0: Bold Bit 1: Italic Bit 2: Underline Bit 4: Strikethrough See **Setting Bit Parameters** on page IV-12 for details about bit settings.
help={*helpStr*}	Sets the help for the control. The help text is limited to a total of 255 bytes. You can insert a line break by putting "\r" in a quoted string.
mode=*m*	Specifies the pop-up title location. *m*=0: Title is in pop-up menu. *m*=1: Title is to the left of pop-up menu, the chosen menu item appears in the pop-up menu, and menu item number *m* is initially selected.
noproc	Specifies that no procedure is to execute when choosing in the pop-up menu.
popColor=(*r*,*g*,*b*)	Specifies the color initially chosen in the color pop-up palette. *r*, *g*, and *b* are integers from 0 to 65535. See the **Colors, Color Tables, Line Styles, Markers, and Patterns** section.
popmatch=*matchStr*	Sets mode to the enabled menu item that matches *matchStr*. *matchStr* may be a "wildcard" expression. See **StringMatch**. If no item is matched, mode is unchanged.
popvalue=*valueStr*	Sets the string displayed by the menu when first created, if mode is not zero. See **Popvalue Keyword** section.
pos={*left*,*top*}	Sets the position of the pop-up menu in pixels.
pos+={*dx*,*dy*}	Offsets the position of the pop-up in pixels.
proc=*procName*	Specifies the procedure to execute when the pop-up menu is clicked. See ***Pop-up Menu Action Procedure*** below.
rename=*newName*	Gives pop-up menu a new name.
size={*width*,*height*}	Sets pop-up menu size in pixels.
title=*titleStr*	Sets title of pop-up menu to the specified string expression. Defaults to "" (no title). Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.
userdata(*UDName*)=*UDStr*	Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create. You can retrieve the data using **GetUserData**.
userdata(*UDName*)+=*UDStr*	Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.
value=*itemListSpec*	Specifies the pop-up menu's items. *itemListSpec* can take several forms as described below under ***Setting The Popup Menu Items***.

win=*winName*	Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Flags

/Z	No error reporting.

Details

The target window, or the window named with the win=*winName* keyword, must be a graph or panel.

Pop-up Menu Action Procedure

The action procedure for a pop-up menu control takes a predefined WMPopupAction structure as a parameter to the function:

```
Function PopupMenuAction(PU_Struct) : PopupMenuControl
   STRUCT WMPopupAction &PU_Struct
   ...
   return 0
End
```

The ": PopupMenuControl" designation tells Igor to include this procedure in the list of available popup menu action procedures in the PopupMenu Control dialog used to create a popup menu.

See **WMPopupAction** for details on the WMPopupAction structure.

Although the return value is not currently used, action procedures should always return zero.

You may see an old format pop-up menu action procedure in old code:

```
Function PopupMenuAction (ctrlName,popNum,popStr) : PopupMenuControl
   String ctrlName
   Variable popNum     // which item is currently selected (1-based)
   String popStr       // contents of current popup item as string
   ...
   return 0
End
```

This old format should not be used in new code.

Setting The Popup Menu Items

This section discusses popup menus containing lists of text items. The next section discusses popup menus for choosing colors, line styles, markers and patterns.

The items in the popup menu are determined by the *itemListSpec* parameter used with the value keyword. *itemListSpec* can take several different forms from simple to complex.

No matter what the form, Igor winds up storing an expression that returns a string in the popup menu's internal structure. This expression may be a literal string ("Red;Green;Blue;"), a call to a built-in or user-defined function that returns a string, or the path to a global string variable. Igor evaluates this expression when the popup menu is first created and again each time the user clicks on the menu. You can see the string expression for a given popup menu using the PopupMenu Control dialog.

The right form for *itemListSpec* depends on your application. Here is a guide to choosing the right form with the simpler forms first.

A literal string expression

Use this if you know the items you want in your popup menu when you write the PopupMenu call. For example:

```
Function PopupDemo1()   // Literal string
   NewPanel
   PopupMenu popup0, value="Red;Green;Blue;"
End
```

This method is limited to 1000 bytes of menu item text.

A function call

Use this if you need to compute the popup menu item list when the user clicks the popup menu. The function must return a string containing a semicolon-separated list of menu items. This example creates a popup menu which displays the name of each wave in the current data folder at the time the menu is clicked:

```
Function PopupDemo2()   // Built-in function
   NewPanel
   PopupMenu popup0, value=WaveList("*", ";", "")
End
```

You can also use a user-defined function. This example shows how to list waves from other than the current data folder:

```
Function/S MyPopupWaveList()
   DFREF saveDF

   // Create some waves for demo purposes
   saveDF = GetDataFolderDFR()
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S PopupMenuDemo
   Make/O demo0, demo1, demo2
   SetDataFolder saveDF

   saveDF = GetDataFolderDFR()
   SetDataFolder root:Packages:PopupMenuDemo
   String list = WaveList("*", ";", "")
   SetDataFolder saveDF

   return list
End

Function PopupDemo3()   // User-defined function
   NewPanel
   PopupMenu popup0, value=MyPopupWaveList()
End
```

# followed by a local string variable specifying items

Use this when the popup menu item list is not known when you write the code but you can compute it at runtime. For example:

```
Function PopupDemo4()   // Local string variable specifying items
   NewPanel
   String quote = "\""
   String list
   if (CmpStr(IgorInfo(2),"Windows") == 0)
      list = quote + "Windows XP; Windows VISTA;" + quote
   else
      list = quote + "Mac OS X 10.4;Mac OS X 10.5;" + quote
   endif
   PopupMenu popup0, value=#list
End
```

The strange-looking use of the quote string variable is necessary because the parameter passed to the value=# keyword is evaluated once when the PopupMenu command executes and the result of that evaluation is evaluated again when the PopupMenu is created or clicked. The result of the first evaluation must be a legal string expression.

This method is limited to 1000 bytes of menu item text.

# followed by a local string variable specifying a function

Use this when you need to compute the popup menu item list at click time and you need to select the function which computes the list when the popup menu is created. For example:

```
Function/S WindowsItemList()
   String list
   list = "Windows XP; Windows VISTA;"
   return list
End

Function/S MacItemList()
   String list
   list = "Mac OS X 10.4;Mac OS X 10.5;"
   return list
End

Function PopupDemo5()   // Local string variable specifying function
   String listFunc
   if (CmpStr(IgorInfo(2),"Windows") == 0)
      listFunc = "WindowsItemList()"
   else
      listFunc = "MacItemList()"
   endif
   NewPanel
   PopupMenu popup0, value=#listFunc
End
```

This form is useful when you create a control panel in an independent module. Since the control panel runs in the global name space, you must specify the independent module name in the invocation of the function that provides the popup menu items. For example:

```
// Calling a non-static function in an independent module from #included code
#pragma IndependentModuleName=IM
. . .
String listFunc= GetIndependentModuleName()+"#PublicFunctionInIndepMod()"
PopupMenu popup0, value=#listFunc

// Calling a static function in an independent module from #included code
#pragma IndependentModuleName=IM
#pragma ModuleName=ModName
. . .
String listFunc= GetIndependentModuleName()+"#ModName#StaticFunctionInIndepMod()"
PopupMenu popup0, value=#listFunc
```

We use GetIndependentModuleName rather than hard-coding the name of the independent module so that the code will continue to work if the name of the independent module is changed. Also, because this code does not depend on the specific name of the independent module, it can be added to an independent module via a #included procedure file.

Also see **GetIndependentModuleName** and **Independent Modules and Popup Menus** on page IV-227.

# followed by a quoted literal path to a global string variable

Use this if you want to compute the popup menu item list before it is clicked, not each time it is clicked. This would be advantageous if it takes a long time to compute the item list, and the list changes only at well-defined times when you can set the global string variable.

The global string variable must exist when the PopupMenu command executes and when the menu is clicked. In this example, the gPopupMenuItems global string variable is created and initialized when the popup menu is created but can be changed to a different value later before the menu is clicked:

```
Function PopupDemo6()   // Global string variable containing list
   NewDataFolder/O root:Packages
   NewDataFolder/O root:Packages:PopupMenuDemo
   String/G root:Packages:PopupMenuDemo:gPopupMenuItems = "Red;Green;Blue;"

   NewPanel
   PopupMenu popup0 ,value=#"root:Packages:PopupMenuDemo:gPopupMenuItems"
End
```

# followed by a local string variable containing a path to a global string variable

Use this when the popup menu item list contents will be stored in a global string variable whose location is not known until the popup menu is created. For example:

```
Function PopupDemo7()   // Local string containing path to global string
   String graphName = WinName(0, 1, 1)// Name of top graph
   if (strlen(graphName) == 0)
      Print "There are no graphs."
      return -1
   endif

   NewDataFolder/O root:Packages
   NewDataFolder/O root:Packages:PopupMenuDemo

   // Create data folder for graph
   NewDataFolder/O root:Packages:PopupMenuDemo:$(graphName)
```

```
    String list = "Red;Green;Blue;"
    String/G root:Packages:PopupMenuDemo:$(graphName):gPopupMenuItems = list

    NewPanel

    String path         // Local string containing path to global string
    path = "root:Packages:PopupMenuDemo:" + graphName + ":gPopupMenuItems"
    PopupMenu popup0, value=#path

    return 0
End
```

Colors, Color Tables, Line Styles, Markers, and Patterns

You can create PopupMenu controls for color, color tables, line style (dash modes), markers, and patterns. To do so, simply specify the *itemListSpec* parameter to the value keyword as one of `"*COLORPOP*"`, `"*COLORTABLEPOP*"`, `"*COLORTABLEPOPNONAMES*"`, `"*LINESTYLEPOP*"`, `"*MARKERPOP*"`, or `"*PATTERNPOP*"`. In these modes the body of the control will contain a color box, a color table (gradient), a line style sample, a marker, or a pattern sample.

For these special pop-up menus, `mode=0` ("Title in Box" checked) is not used.

For a line style pop-up menu, the mode value is the line style number plus one. Thus line style 0 (a solid line) is `mode=1`.

For a marker pop-up, the mode value is the marker number plus one, and marker 0 (the + marker) is `mode=1`.

For a pattern pop-up, the mode value is the **SetDrawEnv** fillPat number minus 4, so mode=1 corresponds to `fillpat=5`, the SW-NE lines fill pattern shown above.

For a color table pop-up, the mode value is the CTabList() index plus 1, so mode=1 corresponds to the first item in the list returned by **CTabList**, which is "Grays":

```
ControlInfo $ctrlName                            // Sets V_Value
Print StringFromList(V_Value-1,CTabList())       // Prints "Grays"
```

ControlInfo also returns the color table name in S_Value.

To set the pop-up to a given color table name, you can use code like this:

```
Variable m = 1 + WhichListItem(ctabName, CTabList())
PopupMenu $ctrlName mode=m
```

For color pop-up menus, you set the current value using the `popColor=(r,g,b)` keyword. On output (via the popStr parameter of your action procedure or via the S_value output from **ControlInfo**) the color is encoded as "(*r*,*g*,*b*)" where *r*, *g*, and *b* are numbers. To get these numerical values, you can extract them from the string using the MyRGBstrToRGB function below or use ControlInfo which sets V_Red, V_Green, V_Blue and V_Alpha.

The following example demonstrates the line style and color pop-up menus. To run the example, copy the following code to the procedure window of a new experiment and then run the panel macro.

```
Window Panel0() : Panel
    PauseUpdate; Silent 1           // building window …
    NewPanel /W=(150,50,400,182)
    PopupMenu popup0,pos={74,31},size={96,20},proc=ColorPopMenuProc,title="colors"
    PopupMenu popup0,mode=1,popColor= (0,65535,65535),value= "*COLORPOP*"
    PopupMenu popup1,pos={9,68},size={221,20},proc=LStylePopMenuProc
    PopupMenu popup1,title="line styles",mode=1,value= "*LINESTYLEPOP*"
EndMacro

Function ColorPopMenuProc(ctrlName,popNum,popStr) : PopupMenuControl
    String ctrlName
    Variable popNum
    String popStr

    Variable r,g,b
    MyRGBstrToRGB(popStr,r,g,b)       // One way to get r, g, b
    print popStr," gives: ",r,g,b

    ControlInfo $ctrlName             // Another way: Sets V_Red,V_Green,V_Blue,V_Alpha
    Printf "ControlInfo returned (%d,%d,%d,%d)\r", V_Red, V_Green, V_Blue, V_Alpha

    return 0
End
```

```
// Take (r,g,b) string and extract out numeric r,g,b values
Function MyRGBstrToRGB(rgbStr,r,g,b)
    String rgbStr
    Variable &r, &g, &b

    r= str2num(rgbStr[1,inf])
    variable spos= strsearch(rgbStr,",",0)
    g= str2num(rgbStr[spos+1,inf])
    spos= strsearch(rgbStr,",",spos+1)
    b= str2num(rgbStr[spos+1,inf])
    return 1
End

Function LStylePopMenuProc(ctrlName,popNum,popStr) : PopupMenuControl
    String ctrlName
    Variable popNum
    String popStr

    print "style:",popNum-1

    return 0
End
```

Popvalue Keyword

There are times when the displayed value cannot be determined and saved such that it can be displayed when the pop-up menu is recreated. For instance, because window recreation macros are evaluated in the root folder, a pop-up menu of waves may not contain the correct list when a panel is recreated. That is, the intention may be to have the menu show a particular wave from a data folder other than root. When the panel recreation macro runs, the function that lists waves will list waves in the root data folder. The desired selection may be wrong or nonexistent.

Similarly, a pop-up menu of fonts may need to display a particular font upon recreation on a different computer having a different list of fonts. The mode=*m* keyword probably won't pick the correct font from the new list.

The solution to these problems is to save the correct selection with the popvalue=*valueStr* keyword. The list function will not be executed when the menu is first created. If the menu is popped, the list function will be evaluated, and the correct list will be displayed then.

It is a good idea to set the mode=*m* keyword to the correct number, if it is known. That way, when the menu is popped the correct item is chosen.

Normally you can let Igor redraw the pop-up menu when it redraws the graph or control panel containing it. However, there are situations in which you may want to force the pop-up menu to be redrawn. This can be done using the **ControlUpdate** operation.

See Also

The **ControlInfo** operation for information about the control. The **ControlUpdate**, **WaveList**, and **TraceNameList** operations. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data. **Special Characters in Menu Item Strings** on page IV-125.

PopupMenuControl

`PopupMenuControl`

PopupMenuControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined pop-up menu control. See **Procedure Subtypes** on page IV-193 for details. See **PopupMenu** for details on creating a popup menu control.

PossiblyQuoteName

`PossiblyQuoteName(`*`nameStr`*`)`

The PossiblyQuoteName function returns the input name string if it conforms to the rules of standard wave or Data Folder names. If it does not, then the name is returned in single quotes. This is used when generating a command string that you will pass to the Execute command. You might get the input name string from a function such as **NameOfWave** or **CsrXWave**.

Examples

```
Print PossiblyQuoteName("wave0")        // prints wave0
Print PossiblyQuoteName("wave 0")       // prints 'wave 0'
```

Details

See **Programming with Liberal Names** on page IV-157 for an example.

Preferences

`Preferences [/Q] [`*`newPrefsState`*`]`

The Preferences operation sets or displays the state of user preferences.

User preferences affect the creation of *new* graphs, panels, tables, layouts, notebooks, procedure windows, and the command window. They also affect the appearance of waves appended to graphs and tables, and objects appended to layouts.

Parameters

If *newPrefsState* is present, it sets the state of user preferences as follows:

newPrefsState=0:	Preferences off (use factory defaults).
newPrefsState=1:	Preferences on.

If *newPrefsState* is omitted, the state of user preferences is printed in the history area.

Flags

/Q	Disables printing to the history.

Details

The Preferences operation sets the variable V_flag to the state of user preferences that were in effect *before* the Preferences command executed: 1 for on, 0 for off.

You can also set the state of Preferences with the Misc menu.

Under most circumstances we want procedures to be independent of preferences so that a particular procedure will do the same thing regardless of the state of preferences. To achieve this, preferences are automatically off when you initiate procedure execution. When execution is complete, the state of preferences is restored to what it was before.

If you want preferences to be in effect during procedure execution, you must turn it on with the Preferences operation.

If the preferences setting is changed by a procedure, the effect of the call is propagated down the calling chain. If a macro changes the preferences setting, that change is undone when the macro returns. If a function changes the preferences setting, the change persists after the function returns. However, even with a function, the changed preferences state does not persist when Igor regains control.

Examples

```
Function Test()
   Variable oldPrefState
   Preferences 1; oldPrefState=V_flag      // remember prefs setting
   Make wave0=x
   Display wave0                 // Display uses preferences
   Preferences oldPrefState      // put prefs back, like a macro would
End
```

See Also

Chapter III-18, **Preferences**.

PrimeFactors

`PrimeFactors [/Q]` ***`inNumber`***

PrimeFactors calculates the prime factors of *inNumber*. By default factors are printed in the history and are also stored in the wave W_PrimeFactors in the current data folder.

Flags

/Q	Suppresses printing of factors in the history area.

Details

The largest number that this operation can handle is 2^{32}-1.

Print

`Print [`***`flags`***`]`***` expression `***`[, `***`expression`***`]…`

The Print operation prints the evaluated expressions in the history area.

Parameters

An expression can be a wave, a numeric expression (e.g., `3*π/4`), a string expression (e.g., `"Today is " + date()`), or a individual structure element or an entire structure variable.

Flags

/C	Evaluates all numeric expressions as complex.
/D	Prints a greater number of digits.
/F	Prints numeric wave data (1D and 2D waves only) using "nice," easily readable formatting.
/LEN=*len*	Sets the string break length to *len* number of bytes. The default is 200 and *len* is clipped to between 200 and 1000.
/S	Obsolete. Numeric results are printed with a moderate number of digits whether you use /S or not. To print more digits, use /D.
/SR	Prints a wave subrange for expressions that start as "*`waveName`*`[`". Without /SR, such an expression is taken as the start of a numeric expression such as `wave[3]-wave[2]`. (You can still use *wave*[*pnt*] but only if it does not start the numeric expression.) Wave subrange printing is not done with /F. You can specify a single row or column using [*r*] syntax. For example, to print column 4 of a matrix, use: `Print mymat[][4]`

Details

Numeric expressions are always evaluated in double precision. The /D flag just controls the number of digits displayed.

Print determines if an expression is real, complex, or string from the first symbol in the expression. Usually this works fine, but occasionally Print guesses wrong and you may have to rearrange your expression. For example:

```
Print 1+cmplx(1,2)
```

will give an error because the first symbol, "1", is real but the expression should be complex. Changing this to

```
Print cmplx(1,2)+1
```

will work.

Printing numeric or string expressions involving structure elements must not start with the structure element. Instead an appropriate numeric or string literal must appear first so that Igor can determine what kind of expression to compile. For example rather than

```
Print astruct.astring + "hello"
```

use

```
Print "" + astruct.astring + "hello"
```

Print breaks long strings into multiple lines. If there are no natural breaks (carriage returns or semicolons) within a default length, then it breaks the string arbitrarily.

The default line length is 200 bytes. You can override this using the /LEN flag. The maximum number of bytes that can be printed on a line in the history area is 1000.

When printing waves, you can use either formatted (specified by /F) or unformatted (default) methods. Unformatted output is in an executable syntax for each printed line: `wave={}`.

Note: Executing lines printed from floating point waves will not exactly reproduce the source data due to round-off or insufficient digits in the printed output.

Printing formatted wave data gives easily (human) readable output, and works best for small 1D and 2D waves. If the data are too large or in an unsupported format (3D or greater, or the wave is text), then the output will be unformatted. Formatting is done using spaces, so the output will look best in a fixed-width font.

Printed wave data, both formatted and unformatted, are limited to no more than 100 lines of output. When the line limit is exceeded a warning message will be printed at the end of the truncated output. For text waves, output is limited to 50 bytes of each string element, and there is no warning when a string is truncated.

See Also

The **printf** operation.

The **PrintGraphs**, **PrintTable**, **PrintLayout** and**PrintNotebook** operations.

printf

`printf` ***`formatStr`*** `[,` ***`parameter`*** **`[,`** ***`parameter`***`]…]`

The printf operation prints formatted output to the history area.

Parameters

formatStr is a string which specifies the formatting of the output.

The type of the parameter, string or numeric, must agree with the corresponding conversion specification in *formatStr*, or else the results will be indeterminate.

The printf parameters can be numeric or string expressions. Numeric and string structure fields are allowed except that complex structure fields and non-numeric (e.g., WAVE, FUNCREF) structure fields are not allowed.

Details

The *formatStr* contains literal text and conversion specifications.

A conversion specification starts with the % character and ends with a conversion character (for example, g, e, f, d, or s as illustrated below). In between the % and the conversion character you may include one or more flag characters, a field width specifier, and a precision specifier. The first % corresponds to the first parameter, the second % corresponds to the second parameter, etc. If *formatStr* contains no % characters, no parameters are expected.

Here are some simple examples. `numVar` is a numeric variable and `strVar` is a string variable.

```
printf "The answer is: %g\r", numVar
printf "Created wave %s\r", strVar
printf "Created wave %s, %d points\r", strVar, numVar
```

%g is a general-purpose format (floating point or scientific notation) that represents the value of numVar. %d is an integer format that represents the value of `numVar`. %s specifies that the corresponding parameter (strVar) is a string.

The "\r" in these examples appends a carriage return to the end of the printed text.

Here is a complex example using all of these elements of a conversion specification:

```
printf "%+015.4f\r", 1e6*PI
```

This prints:

```
The answer is: +003141592.6536
```

"+" is a flag character that tells printf to put a + or - sign in front of the number.

"015" is a field width specifier that tells printf to print the number in a field of at least 15 bytes, padded with leading zeros. Using "15" instead of "015" would cause printf to pad with spaces before the + sign instead of zeros after it.

".4" is a precision specifier that tells printf to print four digits after the decimal point.

"f" tells printf to use a floating point format.

The most common conversions characters are “f” for floating point, “g” for general, “d” for decimal, and “s” for string. They are interpreted as for the printf() function in the C programming language.

The escape codes \t and \r represent the tab and return characters respectively. See **Escape Sequences in Strings** on page IV-13 for more information.

The supported flag characters and their meanings are as follows:

-	Left align the result in the field.
+	Put a plus or minus sign before the number.
<space>	Put a space before a positive number.
#	Specifies alternate form for e, f, g, and x formats.

The meaning of the precision specifier depends on the numeric format (%g, %e, %f, %d, etc.) being used:

e, E, f	Precision specifies number of digits after decimal point.
g, G	Precision specifies maximum number of significant digits.
d, o, u, x, X	Precision specifies minimum number of digits.

You can replace both the field width and precision specifiers with an asterisk. This gets the field width or precision specifier from a parameter. For example:

```
printf "%*.*f\r" 4, 3, 1e6*PI
```

means that the field width is 4 and the precision is 3. You could use numeric expressions instead of the literal numbers to control the field width and precision algorithmically.

Here is a complete list of the conversion characters supported by printf:

f — Converts a numeric parameter as [-]ddd.ddd, where the number of digits after the decimal point is determined by the precision specifier and defaults to 6. If the # flag is present, a decimal point will be used even if there are no digits to the right of it.

This conversion character uses the “round-to-half-even” rule, also known as “banker’s rounding”. When the truncated digits are exactly 0.5000..., the quantity is rounded to an even number. For example:

```
Printf "%.0f\r", 15.5          // Prints 16 (rounded up to even)
Printf "%.0f\r", 16.5          // Prints 16 (rounded down to even)
```

e, E — Converts a numeric parameter as [-]d.ddde+/-dd, where the number of digits after the decimal point is determined by the precision specifier and defaults to 6. If you use “E” instead of “e” then printf uses a capital “E” in the number. If the # flag is present, a decimal point will be used even if there are no digits to the right of it.

g, G — Converts a numeric parameter using “f” or “e” style conversion depending on the magnitude of the number. “e” is used if the exponent is less than -4 or greater than the precision. “G” uses “f” or “E” style conversion. If the # flag is present, a decimal point will be used even if there are no digits to the right of it and trailing zeros will not be removed.

d, o, u — Converts a numeric parameter as a signed decimal integer, unsigned octal integer or unsigned decimal integer. The precision defaults to one and specifies the minimum number of digits to print.

These conversion characters use the “round-away-from-zero” rule, like Igor’s **round** function. For example:

```
Printf "%d\r", 15.5            // Prints 16 (rounded away from zero)
Printf "%d\r", 16.5            // Prints 17 (rounded away from zero)
```

Unlike printf, sprintf and fprintf, wfprintf truncates rather than rounding.

x, X	Converts a numeric parameter as an unsigned hexadecimal integer, rounding floating point values. Also supports integer data up to 64 bits. Unlike printf, sprintf and fprintf, wfprintf truncates rather than rounding. The "x" style uses lower case for the hexadecimal numerals "abcdef" where the "X" style uses upper case. The precision defaults to one and specifies the minimum number of digits to print. If the # flag is present, the string "0x" or "0X" is prepended to the number if it is not zero.
s	Converts a string parameter which can contain no more than 1000 bytes. If a precision is specified, it sets the maximum number of bytes from the string parameter to be printed.
b	WaveMetrics extension. Converts a numeric parameter to binary.
c	Converts a numeric parameter to a single character.
%	Prints a % sign. No parameter is used.
%W	WaveMetrics extension. See description below.

Igor also supports a non-C, WaveMetrics extension to the conversion characters recognized by printf. This conversion specification starts with "%W". It is followed by a flag digit and a format character. For example,

```
printf "%W0Ps", 12.345E-6
```

prints 12.345000µs. In this example, the "%W0" introduces the WaveMetrics conversion specification. The "0" (zero) following the "W" is the flag digit. The "P" that follows is the format specifier character, which prints the number using a prefix, in this case, "µ".

There is only one WaveMetrics format specifier character, "P", which prints using a prefix such as µ, m, k, or M. It recognizes two flag-digits, "0" or "1". Option "0" prints with no space between the numeric part and the prefix character while flag "1" prints with 1 space. Numbers greater than tera or less than femto print using a power of ten notation. Here are a few examples:

```
printf "%.2W0PHz", 12.342E6       // prints 12.34MHz
printf "%.2W1PHz", 12.342E6       // prints 12.34 MHz
printf "%.0W0Ps", 12.342E-6       // prints 12µs
printf "%.0W1Ps", 12.342E-9       // prints 12 ns
```

See Also

The **sprintf**, **fprintf,** and **wfprintf** operations; **Creating Formatted Text** on page IV-244 and **Escape Sequences in Strings** on page IV-13.

PrintGraphs

PrintGraphs [***flags***] ***graphSpec*** [**,** ***graphSpec***]…

The PrintGraphs operation prints one or more graphs.

PrintGraphs prints one or more graphs on a single page from the command line or from a procedure. The graphs can be overlaid or positioned any way you want.

Parameters

The *graphSpec* specifies the name of a graph to print, the position of the graph on the page and some other options.

Flags

/C=*num*	Renders graphs in black and white (*num*=0) or in color (*num*=1; default).
/D	Disables high resolution printing. This flag is of use only on Macintosh. It has no effect on Windows.
/G=*grout*	Specifies grout, the spacing between objects, for tiling in prevailing units.
/I	Coordinates are in inches.
/M	Coordinates are in centimeters.

/R	Coordinates are in percent of page size (see **Examples**).
/PD[=*d*]	Displays print dialog. This allows the user to use Print Preview or to print to a file. If present the /PD flag must be the first flag. *d*=0: Default. Prints without displaying the Print dialog. *d*=1: Displays the Print dialog. /PD is equivalent to /PD=1. *d*=2: Displays the Print Preview dialog. Requires Igor Pro 7.00 or later.
/S	Stacks graphs.
/T	Tiles graphs.

Details

Graph coordinates are in inches (/I) or centimeters (/M) relative to the top left corner of the physical page. If none of these options is present, coordinates are assumed to be in points.

The form of a *graphSpec* is:

graphName `[(`***left, top, right, bottom***`)] [`**/F=*f***`] [`**/T**`]`

Here are some examples:

```
// Take size and position from window size and position.
PrintGraphs Graph0, Graph1

// Specify size and position explicitly.
PrintGraphs/I Graph0(1, 1, 6, 5)/F=1, Graph1(1, 6, 6, 10)/F=1
```

If the coordinates are missing and the /T or /S flags are present before *graphSpec* then the graphs are tiled or stacked. If the coordinates are missing but no /T or /S flags are present then the graph is sized and positioned based on its position on the desktop.

Finally there are these *graphSpec* options, which appear after the graph name:

/F=*f*	Specifies a frame around the graph. *f*=0: No frame (default). *f*=1: Single frame. *f*=2: Double frame. *f*=3: Triple frame. *f*=4: Shadow frame.
/T	Graph is transparent. This allows special effects when graphs are overlaid. For this to be effective, the graph and its contents must also be transparent. Graphs are transparent only if their backgrounds are white. Annotations have their own transparent/opaque settings. PICTs may have been created transparent or opaque; an opaque PICT cannot be made transparent.

Examples

You can put an entire *graphSpec* into a string variable and use the string variable in its place. In this case the name of the string variable must be preceded by the $ character. This is handy for printing from a procedure and also keeps the PrintGraphs command down to a reasonable number of characters. For example:

```
String spec0, spec1, spec2
spec0 = "Graph0(1, 1, 6, 5)/F=1"
spec1 = "Graph1(1, 6, 6, 10)/F=1"
spec2 = ""                        // PrintGraphs will ignore spec2.
PrintGraphs/I $spec0, $spec1, $spec2
```

If you use a string for a *graphSpec* and that string contains no characters then PrintGraphs will ignore that *graphSpec*.

See Also

The **PrintSettings**, **PrintTable**, **PrintLayout** and **PrintNotebook** operations.

PrintLayout

`PrintLayout [/C=`*`num`*` /D] `*`winName`*

The PrintLayout operation prints the named page layout window.

Parameters

winName is the window name of the page layout to print.

Flags

/C=*num*	Renders graphs, tables, and annotations in black-and-white (*num*=0) or in color (*num*=1; default). It has no effect on pictures, which are colored independently.
/D	Prints the layout at the default resolution of the output device. Otherwise it is printed at the highest resolution. This flag is of use only on Macintosh. It has no effect on Windows.

Details

Normally page layouts are printed at the highest available resolution of the output device (printer, plotter, or whatever). On Macintosh, it may not work properly at high resolution with some unusual output devices. If this happens, you can try using the /D flag to see if it works properly at the default resolution.

See Also

The **PrintSettings**, **PrintGraphs**, **PrintTable** and **PrintNotebook** operations.

PrintNotebook

`PrintNotebook [`*`flags`*`] `*`notebookName`*

The PrintNotebook operation prints the named notebook window.

Parameters

notebookName is either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-87 for details on host-child specifications.

Flags

/B=*hiResMethod* — *Macintosh only*; this flag has no effect on Windows.

hiResMethod=1:	Print HiRes PICTs using high resolution bitmaps.
hiResMethod=0:	Don’t print HiRes PICTs using high resolution bitmaps.
hiResMethod=-1:	Print using the default method. Prints HiRes PICTs using high resolution bitmaps and is the same as method 1.

/P=(*startPage*,*endPage*) — Specifies a page range to print. 1 is the first page.

/S=*selection* — Controls what is printed.

selection=0:	Print entire notebook (default).
selection=1:	Print selection only.

Details

If no /B flag is given, the default method of handling HiRes PICTs is used (/B=1). Printing of HiRes PICTs is not well supported on the Macintosh, so by default it prints them using temporary high resolution bitmaps. If a future version of the Mac OS improves in this respect, we will change the default method to print directly.

See Also

Chapter III-1, **Notebooks**.

The **PrintSettings**, **PrintGraphs**, **PrintTable** and **PrintLayout** operations.

PrintSettings

`PrintSettings [/I /M /W=`*`winName`*`] [copySource=`*`source`*`, orientation=`*`o`*`,`
`margins={`*`left,top,right,bottom`*`}, scale=`*`s`*`, colorMode=`*`m`*`, getPrinterList,`
`getPrinter, setPrinter=`*`printerNameStr`*`, getPageSettings, getPageDimensions]`

The PrintSettings operation gets or sets parameters associated with printing, such as a list of available printers or page setup information for a particular window.

An exception is the graphMode and graphSize keyword pair which affect printing of all graphs. This pair was added in Igor Pro 7.00.

Prior to Igor Pro 7.00, PrintSettings applied to a page layout affected the size and orientation of the layout page. In Igor Pro 7.00 and later, the size and orientation of the layout page are independent of print settings. See **Page Layout Page Sizes** on page II-391 for details.

When getting or setting page setup information, PrintSettings acts on a particular window called the destination window. The destination window is the top graph, table, page layout, or notebook window or the window specified by the /W flag.

PrintSettings can not act on page setup records associated with the command window, procedure windows, help windows, control panel, XOP windows, or any type of window other than graphs, tables, page layouts, and notebooks.

The PrintSettings operation services the keywords in the order shown above, not in the order in which they appear in the command. Thus, for example, the getPageSettings and getPageDimensions keywords report the settings after all other keywords are executed.

Flags

/I	Measurements are in inches. If both /I and /M are omitted, measurements are in points.
/M	Measurements are in centimeters. If both /I and /M are omitted, measurements are in points.
/W=*winName*	Acts on the page setup record of the graph, table, page layout, or notebook window identified by *winName*. If *winName* is omitted or if *winName* is `""`, then it used the page setup for the top window.

Keywords

colorMode=*m*	Sets the color mode for the page setup to monochrome (*m*=0) or to color (*m*=1). This keyword does nothing on Macintosh because it is not supported by Mac OS X.
copySource=*source*	Copies page setup settings from the specified source to the destination window. *source* can be the name of a graph, table, page layout, or notebook window or it can be one of the following special keywords: Default_Settings: Sets the page setup record to the default for the associated printer as specified by the printer driver. Factory_Settings: Sets the page setup record to the WaveMetrics factory default. This is the page setup you get when creating a new window with user preferences turned off. Preferred_Settings: Sets the page setup record to the user preferred page setup. This is the page setup you get when creating a new window with user preferences turned on. Because there is only one page setup for all graphs and one page setup for all tables, this has no effect when the destination window is a graph or table. It does work for layouts and notebooks.
getPageDimensions	Returns page dimensions via the string variable S_value, which contains keyword-value pairs that can be extracted using **NumberByKey** and **StringByKey**. See **Details** for keyword-value pair descriptions.

getPageSettings	Returns page setup settings in the string variable S_value, which contains keyword-value pairs that can be extracted using **NumberByKey** and **StringByKey**. See **Details** for keyword-value pair descriptions.
getPrinter	Returns the name of the selected printer for the destination window in the string variable S_value. On Macintosh the returned value will be `""` if the setPrinter keyword was never used on the destination window. This means that the window will use the operating system's "current printer".
getPrinterList	Returns a semicolon-separated list of printer names in the string variable S_value.

Mac OS X:	Returns a list of printers added through Print Center.
Windows:	Returns the names of any local printers and names of network printers to which the user has made previous connections.

graphMode=*g* Sets the printing mode for graphs:

1:	Fill page
2:	Same size
3:	Same aspect ratio
4:	Custom size as set by graphSize keyword
5:	Same size or shrink to fit page (default)

The graphMode keyword was added in Igor Pro 7.00.

graphSize={*left*, *top*, *width*, *height*}

Sets the custom graph size used when graphMode is 4. Parameters are in points unless /I or /M is used.

Invoking the graphSize keyword automatically sets the graphMode to 4.

left and *top* are clipped so that they are no smaller than the minimum allowed by the printer driver. *width* and *height* are not clipped.

This setting is not saved and is set to a default value when Igor starts.

The graphSize keyword was added in Igor Pro 7.00.

margins={*left*, *top*, *right*, *bottom*}

Sets the page margins. Dimensions are in points unless /I or /M is used.

The margins are clipped so that they are no smaller than the minimum allowed by the printer driver and no larger than one-half the size of the paper.

The terms *left*, *top*, *right*, and *bottom* refer to the sides of the page after possible rotation for landscape orientation.

Passing zero for all four margins sets the margins to the minimum margin allowed by the printer.

On Macintosh only, passing -1 for all four margins sets the margins to whatever minimum margin is allowed by the printer, even if the printer is changed later. This is how Igor Pro behaved on Macintosh prior to the creation of the PrintSettings operation, when the minimum printer margins were always used.

orientation=*o*	Sets the paper orientation to portrait (*o*=0) or to landscape (*o*=nonzero).
scale=*s*	In Igor Pro 7 and later, the scale keyword returns an "unimplemented" error, unless *s*=100, because it is currently not supported. Let us know if this feature is important to you. Though *s*=100 does not generate an error, it does nothing. You can still set the scaling manually using the Page Setup dialog.

setPrinter=*printerNameStr*

Sets the selected printer for the destination window.

SetPrinter attempts to preserve orientation, margins, scale, and color mode but other settings may revert to the default state.

printerNameStr is a name as returned by the getPrinterList keyword and may not be identical to the name displayed in various dialogs. For example, on Mac OS X, the printer name “DESKJET 840C” is returned by getPrinterList as “DESKJET_840C”. The latter is the “Queue Name” displayed by the Mac OS X Print Center or Printer Setup Utility programs.

If *printerNameStr* is `""`, the printer for the destination window is set to the default state. This means different things depending on the operating system:

Mac OS X:	The destination window will use the operating system’s “current printer”, as if the setPrinter keyword had never been used.
Windows:	The destination window will use the system default printer.

If you receive an error when using setPrinter, use the getPrinterList keyword to verify that the printer name you are using is correct. Verify that the printer is connected and turned on.

Windows printer names are sometimes UNC names of the form “\\Server\Printer”. You must double-up backslashes when using a UNC name in a literal string. See **UNC Paths** on page III-401 for details.

Details

All graphs in the current experiment share a single page setup record so if you change the page setup for one graph, you change it for all graphs.

All tables in the current experiment share a single page setup record.

Each page layout window has its own page setup record.

Each notebook window has its own page setup record.

The keyword-value pairs for the getPageSettings keyword are as follows:

Keyword	Information Following Keyword
ORIENTATION:	0 if the page is in portrait orientation, 1 if it is in landscape orientation.
MARGINS:	The left, top, right, and bottom margins in points, separated by commas.
SCALE:	The page scaling expressed in percent. 50 means that the graphics are drawn at 50% of their normal size.
COLORMODE:	0 for black&white, 1 for color. This is not supported on Macintosh and always returns 1.

The keyword-value pairs for the getPageDimensions keyword are as follows:

Keyword	Information Following Keyword
PAPER:	The left, top, right, and bottom coordinates of the paper in points, separated by commas. The top and left are negative numbers so that the page can start at (0,0).
PAGE:	The left, top, right, and bottom coordinates of the page in points, separated by commas. The term page refers to the part of the paper inside the margins. The top/left corner of the page is always at (0, 0).
PRINTAREA:	The left, top, right, and bottom coordinates of the page in points, separated by commas. The print area is the part of the paper on which printing can occur, as determined by the printer. This is equal to the paper inset by the minimum supported margins. The top and left are negative numbers so that the page can start at (0,0).

Examples

For an example using the PrintSettings operation, see the PrintSettings Tests example experiment file in the "Igor Pro 7 Folder:Examples:Testing" folder.

Here are some simple examples showing how you can use the PrintSettings operation.

```
Function GetOrientation(name)     // Returns 0 (portrait) or 1 (landscape)
   String name            // Name of graph, table, layout or notebook

   PrintSettings/W=$name getPageSettings
   Variable orientation = NumberByKey("ORIENTATION", S_value)
   return orientation
End

Function SetOrientationToLandscape(name)
   String name            // Name of graph, table, layout or notebook

   PrintSettings/W=$name orientation=1
End

Function/S GetPrinterList()
   PrintSettings getPrinterList
   return S_value
End

Function SetPrinter(destWinName, printerName)
   String destWinName, printerName

   PrintSettings/W=$destWinName setPrinter=printerName
   return 0
End
```

See Also

The **PrintGraphs**, **PrintTable**, **PrintLayout** and **PrintNotebook** operations.

PrintTable

`PrintTable [/P=(`*`startPage`*`,`*`endPage`*`) /S=`*`selection`*`]` ***`winName`***

The PrintTable operation prints the named table window.

Parameters

winName is the window name of the table to print.

Flags

/P=(*startPage*,*endPage*)	Specifies a page range to print. 1 is the first page. If /P is omitted all pages are printed unless /S is used.
/S=*selection*	Controls what is printed. *selection*=0: Print entire table (default). *selection*=1: Print selection only.

See Also

Chapter II-11, **Tables**.

The **PrintSettings**, **PrintGraphs**, **PrintLayout** and **PrintNotebook** operations.

Proc

`Proc` ***`macroName`*** **`(`**`[`***`parameters`***`]`**`)`** `[:`***`macro type`***`]`

The Proc keyword introduces a macro that does not appear in any menu. Otherwise, it works the same as **Macro**. See **Macro Syntax** on page IV-110 for further information.

ProcedureText

`ProcedureText(`*`macroOrFunctionNameStr`*`[,`*`linesOfContext`*`[,`*`procedureWinTitleStr`*`]])`

The ProcedureText function returns a string containing the text of the named macro or function as it exists in some procedure file, optionally with additional lines that are before and after to provide context or to collect documenting comments.

Alternatively, all of the text in the specified procedure window can be returned.

Parameters

macroOrFunctionNameStr identifies the macro or function. It may be just the name of a global (nonstatic) procedure, or it may include a module name, such as `"myModule#myFunction"` to specify the static function myFunction in a procedure window that contains a `#pragma ModuleName=myModule` statement.

If *macroOrFunctionNameStr* is set to "", and *procedureWinTitleStr* specifies the title of a single procedure window, then all of the text in the procedure window is returned.

linesOfContext optionally specifies the number of lines around the function to include in the returned string. The default is 0 (no additional contextual lines of text are returned). This parameter is ignored if *macroOrFunctionNameStr* is "" and *procedureWinTitleStr* specifies the title of a single procedure window.

Setting *linesOfContext* to a positive number returns that many lines before the procedure and after the procedure. Blank lines are not omitted.

Setting *linesOfContext* to -1 returns lines before the procedure that are not part of the preceding macro or function. Usually these lines are comment lines describing the named procedure. Blank lines are omitted.

Setting *linesOfContext* to -n, where n>1, returns at most n lines before the procedure that are not part of the preceding macro or function. Blank lines are not omitted in this case. n can be -inf, which acts the same as -1 but includes blank lines.

The optional *procedureWinTitleStr* can be the title of a procedure window (such as `"Procedure"` or `"File Name Utilities.ipf"`). The text of the named macro or function in the specified procedure window is returned.

You can use *procedureWinTitleStr* to select one of several static functions with identical names among different procedure windows, even if they do not use a `#pragma moduleName=myModule` statement.

Advanced Parameters

If `SetIgorOption IndependentModuleDev=1`, *procedureWinTitleStr* can also be a title followed by a space and, in brackets, an independent module name. In such cases ProcedureText retrieves function text from the specified procedure window and independent module. (See **Independent Modules** on page IV-224 for independent module details.)

For example, in a procedure file containing:

```
#pragma IndependentModule=myIM
#include <Axis Utilities>
```

A call to ProcedureText like this:

```
String text=ProcedureText("HVAxisList",0,"Axis Utilities.ipf [myIM]")
```

will return the text of the `HVAxisList` function located in the Axis Utilities.ipf procedure window, which is normally a hidden part of the `myIM` independent module.

You can see procedure window titles in this format in the Windows→Procedure Windows menu when `SetIgorOption IndependentModuleDev=1` and when an experiment contains procedure windows that comprise an independent module, as does `#include <New Polar Graphs>`.

procedureWinTitleStr can also be just an independent module name in brackets to retrieve function text from *any* procedure window that belongs to the named independent module:

```
String text=ProcedureText("HVAxisList",0,"[myIM]")
```

See Also

Regular Modules on page IV-222 and **Independent Modules** on page IV-224.

The **WinRecreation** and **FunctionList** functions.

ProcGlobal

ProcGlobal#*procPictureName*

The ProcGlobal keyword is used with Proc Pictures to avoid possible naming conflicts with any other global pictures in the experiment. When you add a picture to an experiment using the Pictures dialog, such a picture is global in scope and may potentially have the same name as a Proc Picture. When a Proc Picture is global (and only then), you should use the ProcGlobal keyword to make sure that the Proc Picture is used with your code and to avoid confusion with pictures in the Pictures dialog.

See Also

See **Proc Pictures** on page IV-53 for details. **Pictures Dialog** on page III-449.

Project

Project [/C={*long*,*lat*}/M=*method* /P={*p1*,*p2*,…}] *longitudeWave*, *latitudeWave*

The Project operation calculates projections of XY data, which most often are longitude and latitude waves of geographic coordinates. The output waves are W_XProjection and W_YProjection. Longitude and Latitude are in degrees.

Parameters

longitudeWave is the name of the wave supplying the longitude or equivalent coordinates. *latitudeWave* is the name of the wave supplying the latitude or equivalent coordinates.

Flags

/C={*long*,*lat*}	Specifies longitude and latitude center of projection. By default *long*=0 and *lat*=90.
/M=*method*	Indicates the type of projection. *method* can be one of the following: 0: Orthographic (default). 1: Stereographic. 2: Gnomonic. 3: General perspective. 4: Lambert equal area. 5: Equidistant. 6: Mercator. 7: Transverse Mercator. 8: Albers Equal Area conic.
/P={*p1*,*p2*,…}	One or more parameters required by a particular projection. See the following sections for parameters required by the various projections.

Gnomonic

Here there is one extra parameter that defines the boundaries based on the angle. The specific expression for the limit is that cos(c) in Eq. (5-3) of Snyder is greater than the specified parameter:

`/P={cos(c)}`

The actual transformation uses Eqs. (22-4) and (22-5) of Snyder with k' given by (22-3).

General Perspective

Here there is one extra parameter that defines the boundaries based on the angle. The specific expression for the limit is that cos(c) in Eq. (5-3) of Snyder is greater than the specified parameter.

The actual transformation uses Eqs. (22-4) and (22-5) with k' given by (22-3). Here we specify the height H is units of sphere radius. The tilt of the plane is specified by omega and gamma following the notation of Snyder page 175.

The parameters actually specified by the command are:

/P={*H*, *omega*, *gamma*, *deltax*, *deltay* }

H is the height (in radii) above the surface of the earth, *gamma* is the azimuth east of north of the Y axis, and *omega* is the tilt angle or the angle between the projection plane and the tangent plane. The x output will be limited to ± *deltax* and the y output will be limited to the range ± *deltay*.

Mercator

This projection requires the following parameters:

`/P={`*minLong*`,`*maxLong*`,`*minLat*`,`*maxLat*`}`

If /P is not specified, the default is {0,360,-90,90}

Note that this projection flips the sign of y when cos(longitude-long_0) changes sign. If you are plotting a continuous path in which consecutive points exhibit the sign change, you should add a NaN entry in the wave so that the path does not wrap.

Albers Equal Area Conic

This projection requires:

/P={*minLong*, *maxLong*, *minLat*, *maxLat*, *Phi1*, *Phi2*}

Phi1 and *Phi2* are the specification of the two standard parallels, the other four parameters determine the boundary of the map area for display.

References

Snyder, John P., *Map Projections—A Working Manual*, U.S.G.S. Professional Paper 1395, U.S. Government Printing Office, Washington D.C., 1987, reprinted 1989, 1994, 1997 with corrections.

See Also

"Transforming Data into a Common Spatial Reference" in the "IgorGIS Help" file.

Prompt

`Prompt` ***`variableName`*****`,`** ***`titleStr`*** `[`**`, popup,`** ***`menuListStr`***`]`

The Prompt command is used in functions for the simple input dialog and in macros for the missing parameter dialog. Prompt supplies text to describe *variableName* to the user, and optionally provides a pop-up menu of choices for the value of *variableName*.

Parameters

variableName is the name of a macro input parameter or function variable.

titleStr is a string or string expression containing the text to present in the dialog to describe what *variableName* is. *titleStr* is limited to 255 bytes.

The optional keyword `popup` is used to provide a pop-up list of choices for the values of *variableName*. If `popup` is used, then *menuListStr* is required.

menuListStr is a string or string expression that contains a semicolon-separated list of choices for the value of *variableName*. If *variableName* is a string, choosing from this list will set the string to the selection. If it is a numeric variable, then it is set to the item number of the selection (if the first item is selected, the numeric variable is set to 1, etc.).

Details

In macros, there must be a blank line after the set of input parameter declarations and prompt statements and there must not be any blank lines within the set.

In user-defined functions, Prompt may be used anywhere within the body of the function, but must precede any DoPrompt that uses the Prompt variable.

menuListStr may be continued on succeeding lines only in macros, as long as no comment is appended to the Prompt line. The additional lines should start with a semicolon, and are appended to the *menuListStr*s on preceding lines.

See Also

For use in user-defined functions, see **The Simple Input Dialog** on page IV-132.

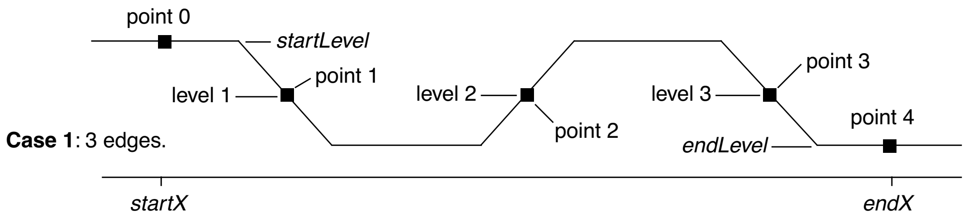
For use in macros, see **The Missing Parameter Dialog** on page IV-113.

For use in functions and macros, see the **DoPrompt** and **popup** keywords.

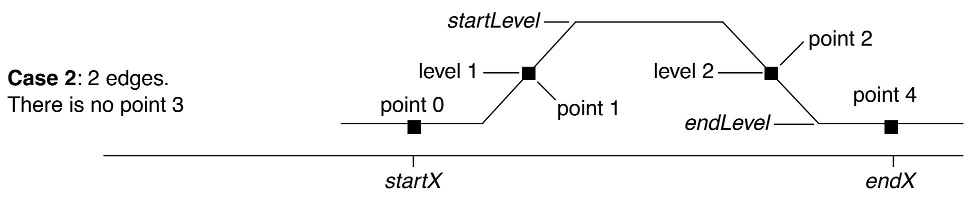
PulseStats

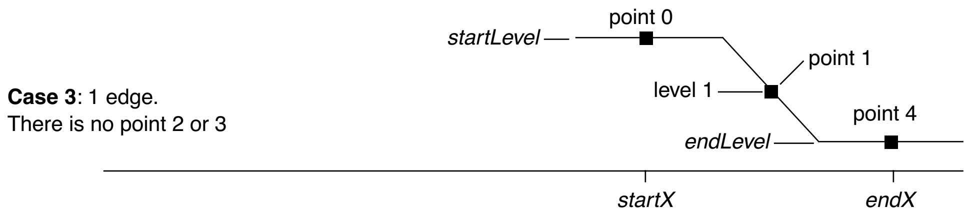
`PulseStats [`*`flags`*`]` ***`waveName`***

The PulseStats operation produces simple statistics on a region of the named wave that is expected to contain three edges as shown below. If more than three edges exist, PulseStats works on the first three edges it finds.



PulseStats handles other cases in which there are only one or two edges.





Flags

/A=*n*	Determines *startLevel* and *endLevel* automatically by averaging *n* points centered at *startX* and *endX*. This does not work in case 2, which requires that you use the /L flag. Default is /A=1.
/B=*box*	Sets box size for sliding average. This should be an odd number. If /B=*box* is omitted or *box* equals 1, no averaging is done.
/F=*f*	Specifies levels 1, 2, and 3 as a fraction of (*endLevel*-*startLevel*): `level1 = level2 = level3 = f*(endLevel-startLevel) + startLevel` *f* must be between 0 and 1. The default value is 0.5 which sets the levels to midway between the base levels.
/L=(*startLevel*, *endLevel*)	Sets *startLevel* and *endLevel* explicitly.
/M=*dx*	Sets minimum edge width. Once an edge is found, the search for the next edge starts *dx* units beyond the found edge. Default *dx* is 0.
/P	Output edge locations (see **Details**) are set in terms of point number. If /P is omitted, edge locations are set in terms of X values.
/Q	Prevents results from being printed in history and prevents error if edge is not found.
/R=(*startX*,*endX*)	Specifies an X range of the wave to search. You may exchange *startX* and *endX* to reverse the search direction.
/R=[*startP*,*endP*]	Specifies a point range of the wave to search. You may exchange *startP* and *endP* to reverse the search direction. If you specify the range as /R=[*startP*] then the end of the range is taken as the end of the wave. If /R is omitted, the entire wave is searched.

/T=*dx*	Forces search in two directions for a possibly more accurate result. *dx* controls where the second search starts.

Details

The /B=box, /T=dx, /P and /Q flags behave the same as for the **FindLevel** operation.

PulseStats considers a region of the input wave between two X locations, called *startX* and *endX*. *startX* and *endX* are set by the /R=(*startX*,*endX*) flag. If this flag is missing, *startX* and *endX* default to the start and end of the entire wave.

The *startLevel* and *endLevel* values define the base levels of the pulse. You can explicitly set these levels with the /L=(*startLevel*, *endLevel*) flag or you can let PulseStats find the base levels for you by using the /A=*n* flag. With this flag, PulseStats determines *startLevel* and *endLevel* by averaging *n* points centered at *startX* and at *endX*. In case 2, you must use /L=(*startLevel*, *endLevel*) since *startLevel* is not at point 0.

Given *startLevel* and *endLevel* and an *f* value (which you can set with the /F=*f* flag), PulseStats computes level1, level2 and level3 which are always equal. With the default *f* value of 0.5, level1 is midway between *startLevel* and *endLevel*.

With these levels defined, PulseStats searches the wave from *startX* to *endX* looking for one, two or three level crossings. PulseStats sets the following variables:

`V_flag`	0: All three level crossings were found. 1: One or two level crossings were found. 2: No level crossings were found.
`V_PulseLoc1`	X location where level1 was found.
`V_PulseLoc2`	X location where level2 was found.
`V_PulseLoc3`	X location where level3 was found.
`V_PulseLvl0`	*startLevel* value.
`V_PulseLvl123`	Level1 value that is the same as level2 and level3.
`V_PulseLvl4`	*endLevel* value.
`V_PulseAmp4_0`	Pulse amplitude (*endLevel* - *startLevel*).
`V_PulseWidth2_1`	Left pulse width (x distance between point 2 and point 1).
`V_PulseWidth3_2`	Right pulse width (x distance between point 3 and point 2).
`V_PulseWidth3_1`	Pulse period (x distance between point 3 and point 1).
`V_PulsePolarity`	Trend of the edge at point 1 (-1 if decreasing, +1 if increasing).

X locations and distances are in terms of the X scaling of the source wave, unless you use the /P flag in which case they are in terms of point number.

If any level crossings are missing then PulseStats sets the associated variables to NaN (Not a Number). If one crossing is missing, variables depending on point 3 are set to NaN. If two crossings are missing, variables depending on points 2 and 3 are set to NaN. If all crossings are missing, variables depending on points 1, 2, and 3 are set to NaN. You can use the numtype function to test a variable to see if it is NaN.

The PulseStats operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

See Also

The **FindLevel** operation about the /B=*box*, /T=*dx*, /P and /Q flags, **EdgeStats** and the **numtype** function.

PutScrapText

`PutScrapText` ***`textStr`***

The PutScrapText operation places *textStr* on the Clipboard (aka “scrap”). This text will be used when the user subsequently chooses Paste from the Edit menu.

Details

All contents of the Clipboard (including pictures) are cleared before the text is placed there.

Examples

Put two lines of text into the Clipboard:

```
String text = "This is the first line.\rAnd this is the second."
PutScrapText text
```

Empty the Clipboard:

```
PutScrapText ""
```

See Also

The **GetScrapText** function and the **SavePICT** operation.

pwd

`pwd`

The pwd operation prints the full path of the current data folder to the history area. It is equivalent to Print GetDataFolder(1).

pwd is named after the UNIX "print working directory" command.

See Also

GetDataFolder, **cd**, **Dir**, **Data Folders** on page II-99

q

`q`

The q function returns the current column index of the destination wave when used in a multidimensional wave assignment statement. The corresponding scaled column index is available as the **y** function.

Details

Unlike **p**, outside of a wave assignment statement, q does not act like a normal variable.

See Also

Waveform Arithmetic and Assignments on page II-69.

For other dimensions, the **p**, **r**, and **s** functions.

For scaled dimension indices, the **x**, **y**, **z** and **t** functions.

qcsr

`qcsr(`*`cursorName`*` [, `*`graphNameStr`*`])`

The qcsr function can be used with cursors on images or waterfall plots to return the column number. It can also be used with free cursors to return the relative Y coordinate.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

See Also

The **hcsr**, **pcsr**, **vcsr**, **xcsr**, and **zcsr** functions.

Programming With Cursors on page II-249.

Quit

`Quit [/N/Y]`

The Quit operation quits Igor Pro.

Flags

/N	Quits without saving changes and without dialog.
/Y	Saves current experiment before quitting without putting up dialog unless current experiment is "Untitled".

r

`r`

The r function returns the current layer index of the destination wave when used in a multidimensional wave assignment statement. The corresponding scaled layer index is available as the **z** function.

Details

Unlike **p**, outside of a wave assignment statement, r does not act like a normal variable.

See Also

Waveform Arithmetic and Assignments on page II-69. For other dimensions, the **p**, **q**, **s**, and **t** functions. For scaled dimension indices, the **x**, **y**, **z**, and **t** functions.

r2polar

`r2polar(`*`z`*`)`

The r2polar function returns a complex value in polar coordinates derived from the complex value *z*, which is assumed to be in rectangular coordinates. The magnitude is stored in the real part and the angle (in radians) is stored in the imaginary part of the returned complex value.

Examples

Assume waveIn and waveOut are complex.

```
waveOut= r2polar(waveIn)
```

sets each point of waveOut to the polar coordinates derived from the real and imaginary parts of waveIn.

You may get unexpected results if the number of points in waveIn differs from the number of points in waveOut.

See Also

The functions **cmplx**, **conj**, **imag**, **p2rect**, and **real**.

RatioFromNumber

`RatioFromNumber [`*`flags`*`]` ***`num`***

The RatioFromNumber operation computes two integers whose ratio is equal to *num* ± *maxError (*/MERR flag). The ratio is returned in V_numerator and V_denominator.

Parameters

num is the number to approximate by V_numerator/V_denominator.

Flags

/MERR=*maxError*	Specifies the maximum tolerable error. The computed ratio differs from *num* by no more than *maxError* (default value is *num**1e-6). *maxError* must be a value between 0 and *num*. See Details about setting *maxError* to 0.
/MITS = *maxIterations*	Keeps returned values small by specifying a small number for *maxIterations*. *maxIterations* must be a value between 1 and 32767 (default is 100).

/V[=*v*]	Prints output variables to history. *v*=1: Prints variables (same as /V). *v*=0: Nothing printed (same as no /V).

Details

The ratio is computed by continued fraction expansion and recurrence relations for the convergents and checking num - (V_numerator/V_denominator) against *maxError*.

Setting *maxError* = 0 computes a maximally accurate ratio. The returned values can be surprisingly large:

```
RatioFromNumber/V/MERR=0 (1/1666)
  V_numerator= 4398046511104; V_denominator= 7.3271454874993e+15;
  ratio= 0.00060024009603842; V_difference= 0;
```

Using the default /MERR returns the expected 1 and 1666. The difference is attributable to floating-point roundoff errors.

The ratio is computed by continued fraction expansion and recurrence relations for the convergents and checking *num* - (V_numerator/V_denominator) against /MERR.

Output Variables

RatioFromNumber sets the following output variables:

`V_difference`	V_numerator/V_denominator - *num* (positive if the approximation is too big).
`V_flag`	0: V_difference less than or equal to /MERR. 1: V_difference greater than /MERR.
`V_numerator`, `V_denominator`	Values for the numerator and denominator. The ratio of V_numerator/V_denominator approximates *num*.
`V_iterations`	The number of iterations actually used.

Examples

```
RatioFromNumber/V pi
  V_numerator= 355; V_denominator= 113; ratio= 3.141592920354;
  V_difference= 2.6676418940497e-07; V_iterations= 3;
```

```
RatioFromNumber/V/MITS=2 pi
  V_numerator= 22; V_denominator= 7; ratio= 3.1428571428571;
  V_difference= 0.0012644892673497; V_iterations= 1;
```

See Also

The **gcd** and **trunc** functions.

Rect

The Rect structure is used as a substructure usually to store the coordinates of a window or control.

```
Structure Rect
   Int16 top
   Int16 left
   Int16 bottom
   Int16 right
EndStructure
```

RectF

The RectF structure is the same as Rect but with floating point fields.

```
Structure RectF
   float top
   float left
   float bottom
   float right
EndStructure
```

ReadVariables

`ReadVariables`

The ReadVariables operation reads variables into an experiment.

ReadVariables is used automatically when you open an experiment. You need not invoke it.

real

`real(`***`z`***`)`

The real function returns the real component of the complex value *z*.

See Also

The functions **cmplx**, **conj**, **imag**, **p2rect**, and **r2polar**.

Redimension

`Redimension [`***`flags`***`]` ***`waveName`*** `[,` ***`waveName`***`]…`

The Redimension operation remakes the named waves, preserving their contents as much as possible.

Flags

/B	Converts waves to 8-bit signed integer or unsigned integer if /U is present.
/C	Converts real waves to complex.
/D	Converts single precision waves to double precision.
/E=*e*	Controls the redimension mode: *e*=0: No special action (default). *e*=1: Force reshape without converting or moving data. *e*=2: Perform endian swap. See **FBinRead** for a discussion of endian byte ordering.
/I	Converts waves to 32-bit signed integer or unsigned integer if /U is present.
/L	Converts waves to 64-bit signed integer or unsigned integer if /U is present. Requires Igor Pro 7.00 or later.
/N=*n*	*n* is the new number of points each wave will have. Multidimensional waves are converted to 1 dimension. If n =-1, the wave is converted to a 1-dimensional wave with the original number of rows.
/N=(*n1*, *n2*, *n3*, *n4*)	*n1*, *n2*, *n3*, *n4* specify the number of rows, columns, layers, and chunks each wave will have. Trailing zeros can be omitted (e.g., /N=(*n1*, *n2*, 0, 0) can be abbreviated as /N=(*n1*, *n2*)). If any dimension size is to remain unchanged, pass -1 for that dimension.
/R	Converts complex waves to real by discarding the imaginary part.
/S	Converts double precision waves to single precision.
/U	Converts integer waves to unsigned.
/W	Converts waves to 16-bit integer (unsigned integer if /U is present).
/Y=*type*	Specifies wave data type. See details below.

Wave Data Types

As a replacement for the above number type flags you can use /Y=*numType* to set the number type as an integer code. See the **WaveType** function for code values. Do not use /Y in combination with other type flags. This technique cannot be used to change the number type without changing the real/complex setting.

Details

The waves must already exist. New points in waves that are extended are zeroed.

In general, Redimension does not move data from one dimension to another. For instance, if you have a 6x6 matrix wave, and you would like it to be 3x12, the rows have been shortened and the data for the last three rows is lost.

As a special case, if converting to or from a 1D wave, Redimension will leave the data in place while changing the dimensionality of the wave. For example, you can use Redimension to convert a 36-element 1D wave into a 6x6 matrix in which the elements in the first column (column 0) are the first 6 elements of the 1D wave, the elements of the second column are the next 6, etc. When redimensioning from a 1D wave, columns are filled first, then layers, followed by chunks.

Examples

Reshaping a 1D wave having 4 elements to make a 2x2 matrix:

```
Make/N=4 vector=x
Redimension/N=(2,2) vector
```

See Also

Make, **DeletePoints**, **InsertPoints**, **Concatenate**, **SplitWave**

Remove

`Remove`

When interpreting a command, Igor treats the Remove operation as **RemoveFromGraph**, **RemoveFromTable**, or **See Also**, depending on the target window. This does not work when executing a user-defined function. Therefore, we recommend that you use **RemoveFromGraph**, **RemoveFromTable**, or **RemoveLayoutObjects** rather than Remove.

RemoveByKey

`RemoveByKey(`*`keyStr, kwListStr`*` [, `*`keySepStr`*` [, `*`listSepStr`*` [, `*`matchCase`*`]]])`

The RemoveByKey function returns *kwListStr* after removing the keyword-value pair specified by *keyStr*. *kwListStr* should contain keyword-value pairs such as `"KEY=value1,KEY2=value2"` or `"Key:value1;KEY2:value2"`, depending on the values for *keySepStr* and *listSepStr*.

Use RemoveByKey to remove information from a string containing a `"key1:value1;key2:value2;"` or `"key1=value1,key2=value2,"` style list such as those returned by functions like **AxisInfo** or **TraceInfo**.

If *keyStr* is not found then *kwListStr* is returned unchanged.

keySepStr, *listSepStr,* and *matchCase* are optional; their defaults are ":", ";", and 0 respectively.

Details

keyStr is limited to 255 bytes.

kwListStr is searched for an instance of the key string bound by *listSepStr* on the left and a *keySepStr* on the right. The key, the *keySepStr*, and the text up to and including the next *listSepStr* (if any) are removed from the returned string.

If the resulting string contains only *listSepStr* characters, then an empty string (`""`) is returned.

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are usually case-insensitive. Setting the optional *matchCase* parameter to 1 makes the comparisons case sensitive.

In Igor6, only the first byte of *keySepStr* and *listSepStr* was used. In Igor7 and later, all bytes are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *matchCase* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print RemoveByKey("AKEY", "AKEY:123;BKEY:val")        // prints "BKEY:val"
Print RemoveByKey("AKEY", "akey=1;BK=b;", "=")        // prints "BK=b;"
Print RemoveByKey("AKEY", "AKEY=1,BK=b,", "=", ",")   // prints "BK=b,"
Print RemoveByKey("ckey","CKEY:1;BKEY:2")             // prints "BKEY:2"
Print RemoveByKey("ckey","CKEY:1;BKEY:2",":",";",1)   // prints "CKEY:1;BKEY:2"
```

See Also

The **NumberByKey**, **StringByKey**, **ReplaceNumberByKey**, **ReplaceStringByKey**, **ItemsInList**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

RemoveContour

`RemoveContour [/W=`*`winName`*`]` ***`contourInstanceName`*** **`[,`** ***`contourInstanceName`*** **`]…`**

The RemoveContour operation removes the traces, and releases memory associated with the contour plot of *contourInstanceName* in the target or named graph.

Parameters

contourInstanceName is usually simply the name of a wave. More precisely, *contourInstanceName* is a wave name, optionally followed by the # character and an instance number to identify which contour plot of a given wave is to be removed.

Flags

/W=*winName*	Removes contours from the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

If the axes used by the contour plot are no longer in use, they will also be removed.

An contour instance name in a string can be used with the $ operator to specify *imageInstance*.

Examples

```
Display;AppendMatrixContour zw       //new graph, contour of zw matrix
AppendMatrixContour zw               //two contours of zw
RemoveContour zw#1                   //remove the second contour
```

See Also

The **AppendMatrixContour** and **AppendXYZContour** operations.

RemoveEnding

`RemoveEnding(`*`str`*` [, `*`endingStr`*`])`

The RemoveEnding function removes one character from the end of *str*, or it removes the *endingStr* from the end of *str*.

endingStr is optional. If missing, one character is removed from the end of *str*.

Details

endingStr is compared to the end of *str* using a case insensitive comparison (such as cmpstr uses). If the end of *str* does not match *endingStr*, the unaltered *str* is returned.

Examples

```
Print RemoveEnding("123")                      // prints "12"
Print RemoveEnding("no semi" , ";")            // prints "no semi"
Print RemoveEnding("trailing semi;" , ";")     // prints "trailing semi"
Print RemoveEnding("file.txt" , ".TXT")        // prints "file"
```

See Also

The **cmpstr** and **ParseFilePath** functions.

RemoveFromGizmo

RemoveFromGizmo [***flags***]

The RemoveFromGizmo operation removes the specified object from the specified list and optionally performs an update.

Documentation for the RemoveFromGizmo operation is available in the Igor online help files only. In Igor, execute:

```
DisplayHelpTopic "RemoveFromGizmo"
```

RemoveFromGraph

RemoveFromGraph [/W=*winName*/Z] *traceName* [, *traceName*]…

The RemoveFromGraph operation removes the specified wave traces from the target or named graph. A trace is a representation of the data in a wave, usually connected line segments.

Parameters

traceName is usually just the name of a wave.

More generally, *traceName* is a wave name, optionally followed by the # character and an instance number - for example, wave0#1. See **Instance Notation** on page IV-19 for details.

Flags

/W=*winName*	Removes traces from the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/Z	Suppresses errors if specified trace or image is not on the graph.

Details

Up to 100 *traceName*s may be specified, subject to the 1000 byte command length limit.

If the axes used by the given trace are not in use after removing the trace, they will also be removed.

A string containing a trace name can be used with the $ operator to specify *traceName*.

Specifying $"#0" for *traceName* removes the first trace in the graph. $"#1" removes the second trace in the graph, and so on. $"" is equivalent to $"#0".

Note that removing all the contour traces from a contour plot is not the same as removing the contour plot itself. Use the **RemoveContour** operation.

Examples

The command:

```
Display myWave,myWave;Modify mode(myWave#1)=6
```

appends two instances of myWave to the graph.The first/backmost instance of myWave is instance 0, and its trace name is just myWave as a synonym for myWave#0. The second or frontmost instance of myWave is myWave#1 and it is displayed with the cityscape mode.

To remove the second instance from the graph requires the command:

```
RemoveFromGraph myWave#1
```

or

```
String MyTraceName="myWave#1"
RemoveFromGraph $MyTraceName
```

See Also

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

RemoveFromLayout

`RemoveFromLayout` ***`objectSpec`*** `[,` ***`objectSpec`***`]`…

Deprecated — use **RemoveLayoutObjects**.

The RemoveFromLayout operation removes the specified objects from the top layout.

Parameters

objectSpec is either an object name (e.g., Graph0) or an *objectName* with an instance (e.g., Graph0#1). An instance is needed only if the same object appears in the layout more than one time. Graph0 is equivalent to Graph0#0 and Graph0#1 refers to the second instance of Graph0 in the layout.

See Also

The **RemoveLayoutObjects** operation.

RemoveFromList

`RemoveFromList(`*`itemOrListStr, listStr`* `[,` ***`listSepStr`*** `[,` ***`matchCase`***`]])`

The RemoveFromList function returns *listStr* after removing the item or items specified by *itemOrListStr*. *listStr* should contain items separated by the *listSepStr* character, such as "abc;def;".

If *itemOrListStr* contains multiple items, they should be separated by the *listSepStr* character, too.

Use RemoveFromList to remove item(s) from a string containing a list of items separated by a single character, such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

If all items in *itemOrListStr* are not found or if any of the arguments is `""` then *listStr* is returned unchanged (unless *listStr* contains only list separators, in which case an empty string is returned).

listSepStr and *matchCase* are optional; their defaults are ";" and 1 respectively.

Details

itemStr may have any length.

listStr is searched for an instance of the item string(s) bound by *listSepStr* on the left and right. All instances of the item(s) and any trailing *listSepStr* (if any) are removed from the returned string.

If the resulting string contains only *listSepStr* characters, then an empty string (`""`) is returned.

listStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *listSepStr* are case-sensitive. Searches for *items* in *itemOrListStr* are usually case-sensitive. Setting the optional *matchCase* parameter to 0 makes the comparisons case insensitive.

In Igor6, only the first byte of *listSepStr* was used. In Igor7 and later, all bytes are used.

If *matchCase* is specified, then *listSepStr* must also be specified.

Examples

```
Print RemoveFromList("wave1", "wave0;wave1;")         // prints "wave0;"
Print RemoveFromList("wave1", ";wave1;;;")            // prints ""
Print RemoveFromList("KEY=joy", "AX=3,KEY=joy", ",")  // prints "AX=3,"
Print RemoveFromList("fred", "fred\twilma", "\t")     // prints "wilma"
Print RemoveFromList("fred;barney","fred;wilma;barney")// prints "wilma;"
Print "X"+RemoveFromList("",";;;;")+"Y"               // prints "XY"
Print RemoveFromList("FRED", "fred;wilma")        // prints "fred;wilma"
Print RemoveFromList("FRED", "fred;wilma", ";", 0)    // prints "wilma"
```

See Also

The **FindListItem**, **FunctionList**, **ItemsInList**, **RemoveByKey**, **RemoveListItem**, **StringFromList**, **StringList**, **TraceNameList**, **UpperStr**, **VariableList**, and **WaveList** functions.

RemoveFromTable

`RemoveFromTable` `[`**`/W=`*`winName`***`]` ***`columnSpec`*** `[,` ***`columnSpec`***`]`…

The RemoveFromTable operation removes the specified columns from the top table.

Parameters

*columnSpec*s are the same as for the **Edit** operation; usually they are just the names of waves.

Flags

/W=*winName*	Removes columns from the named table window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

See Also

Edit about *columnSpec*s, and **AppendToTable**.

RemoveImage

`RemoveImage [/W=winName/Z] imageInstance [, imageInstance]…`

The RemoveImage operation removes the given image from the target or named graph.

Parameters

imageInstance is usually simply the name of a wave. More precisely, *imageInstance* is a wave name, optionally followed by the # character and an instance number to identify which image of a given wave is to be removed.

Flags

/W=*winName*	Removes an image from the named graph window or subwindow. When omitted, action will affect the active window or subwindow. Must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/Z	Suppresses errors if specified image is not on the graph.

Details

If the axes used by the given image are not in use after removing the image, they will also be removed.

An image name in a string can be used with the $ operator to specify *imageInstance*.

See Also

The **AppendImage** operation.

RemoveLayoutObjects

`RemoveLayoutObjects [/PAGE=page/W=winName/Z] objectSpec [, objectSpec]`

The RemoveLayoutObjects operation removes the specified object or objects from the top page layout, or from the layout specified by the /W flag. It targets the active page or the page specified by the /PAGE flag.

Unlike the RemoveFromLayout operation, RemoveLayoutObjects can be used in user-defined functions. Therefore, RemoveLayoutObjects should be used in new programming.

Parameters

objectSpec is either an object name (e.g., Graph0) or an *objectName* with an instance (e.g., Graph0#1). An instance is needed only if the same object appears in the layout more than one time. Graph0 is equivalent to Graph0#0 and Graph0#1 refers to the second instance of Graph0 in the layout.

Flags

/PAGE=*page*	Removes the object from the specified page. Page numbers start from 1. To target the active page, omit /PAGE or use *page*=0. The /PAGE flag was added in Igor Pro 7.00.
/W=*winName*	*winName* is the name of the page layout window from which the object is to be removed. If /W is omitted or if *winName* is `$""`, the top page layout is used.
/Z	Does not report errors if the specified layout object does not exist.

See Also

NewLayout, **AppendLayoutObject**, **ModifyLayout**, **LayoutPageAction**

RemoveListItem

`RemoveListItem(`*`index, listStr`*` [, `*`listSepStr`*` [, `*`offset`*`]])`

The RemoveListItem function returns *listStr* after removing the item specified by the list index *index*.

RemoveListItem removes an item from a string containing a list of items separated by a separator, such as strings returned by functions like **TraceNameList** and **AnnotationList**.

Parameters

index is the zero-based index of the list item that you want to remove.

listStr contains a series of text items separated by *listSepStr*. The trailing separator is optional though recommended.

listSepStr is optional. If omitted it defaults to ";". Prior to Igor Pro 7, only the first byte of *listSepStr* was used. Now all bytes are used.

offset is optional and requires Igor Pro 7 or later. If omitted it defaults to 0. The search begins offset bytes into *listStr*. When iterating through lists containing large numbers of items, using the *offset* parameter provides dramatically faster execution. For an example using the offset parameter, see **StringFromList**.

Details

RemoveListItem differs from **RemoveFromList** in that it specifies the item to be removed by index and removes only that item, while RemoveFromList specifies the item to be removed by value, and removes all matching items.

If *index* less than 0 or greater than ItemsInList(*listStr*) - 1, or if *listSepStr* is `""` then *listStr* is returned unchanged (unless *listStr* contains only list separators, in which case an empty string is returned).

If the resulting string contains only *listSepStr* characters, then an empty string (`""`) is returned.

Examples

```
Print RemoveListItem(1, "wave0;wave1;w2;")        // Prints "wave0;w2;"
```

See Also

The **AddListItem**, **FindListItem**, **FunctionList**, **ItemsInList**, **RemoveByKey**, **RemoveFromList**, **StringFromList**, **StringList**, **TraceNameList**, **VariableList**, **WaveList**, and **WhichListItem** functions.

RemovePath

`RemovePath [/A/Z]` ***`pathName`***

The RemovePath operation removes a path from the list of symbolic paths. RemovePath is an old name for the new **KillPath** operation, which we recommend you use instead.

Rename

`Rename` ***`oldName, newName`***

The Rename operation renames waves, strings, or numeric variables from *oldName* to *newName*.

Parameters

oldName may be a simple object name or a data folder path and name. *newName* must be a simple object name.

Details

You can not rename an object using a name that already exists. The following will result in an error:

```
Make wave0, wave1
// Rename wave0 and overwrite wave1.
Rename wave0, wave1           // This will not work.
```

However, you can achieve the desired effect as follows:

```
Make wave0, wave1
Duplicate/O wave0, wave1; KillWaves wave0
```

See Also

The **Duplicate** operation.

RenameDataFolder

RenameDataFolder sourceDataFolderSpec, newName

The RenameDataFolder operation changes the name of the source data folder to the new name.

sourceDataFolderSpec can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

newName is just the new name for the data folder, without any path.

Details

RenameDataFolder generates an error if the new name is already in use as a data folder contained within the source data folder.

Examples

```
RenameDataFolder root:foo,foo2       // Change name of foo to foo2
```

See Also

Chapter II-8, **Data Folders**.

RenamePath

RenamePath oldName, newName

The RenamePath operation renames an existing symbolic path from *oldName* to *newName*.

See Also

Symbolic Paths on page II-21

RenamePICT

RenamePICT oldName, newName

The RenamePICT operation renames an existing picture to from *oldName* to *newName*.

See Also

Pictures on page III-448.

RenameWindow

RenameWindow oldName, newName

The RenameWindow operation renames an existing window or subwindow from *oldName* to *newName*.

Parameters

oldName is the name of an existing window or subwindow.

When identifying a subwindow with *oldName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

See Also

The **DoWindow** operation.

ReorderImages

`ReorderImages` `[`***`/W=winName`***`]` ***`anchorImage`***`,` **`{`*****`imageA`*****`,`** ***`imageB`*****`, …}`**

The ReorderImages operation changes the ordering of graph images to that specified in the braces.

Flags

/W=*winName*	Reorders images in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

Igor keeps a list of images in a graph and draws the images in the listed order. The first image drawn is consequently at the bottom. All other images are drawn on top of it. The last image is the top one; no other image obscures it.

ReorderImages works by removing the images in the braces from the list and then reinserting them at the location specified by *anchorImage*. If *anchorImage* is not in the braces, the images in braces are placed before *anchorImage*.

If the list of images is A, B, C, D, E, F, G and you execute the command

```
ReorderImages F, {B,C}
```

images B and C are placed just before F: A, D, E, **B**, **C**, **F**, G.

The result of

```
ReorderImages E, {D,E,C}
```

is to reorder C, D and E and put them where E was. Starting from the initial ordering this gives A, B, **D**, **E**, **C**, F, G.

ReorderImages generates an error if the same trace is in the list twice.

In Igor7 or later, *anchorImage* can be _front_ or _back_. To move A to the front, you can write:

```
ReorderImage _front_, {A}
```

See Also

The **ReorderTraces** operation.

ReorderTraces

`ReorderTraces` `[`***`/W=winName`***`]` ***`anchorTrace`***`,` **`{`*****`traceA`*****`,`** ***`traceB`*****`, …}`**

The ReorderTraces operation changes the ordering of graph traces to that specified in the braces.

Flags

/W=*winName*	Reorders traces in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

Igor keeps a list of traces in a graph and draws the traces in the listed order. The first trace drawn is consequently at the bottom. All other traces are drawn on top of it. The last trace is the top one; no other trace obscures it.

ReorderTraces works by removing the traces in the braces from the list and then reinserting them at the location specified by *anchorTrace*. If *anchorTrace* is not in the braces, the traces in braces are placed before *anchorTrace*.

If the list of traces is A, B, C, D, E, F, G and you execute the command

```
ReorderTraces F, {B,C}
```

traces B and C are placed just before F: A, D, E, **B**, **C**, **F**, G.

The result of

```
ReorderTraces E, {D,E,C}
```

is to reorder C, D and E and put them where E was. Starting from the initial ordering results in A, B, **D**, **E**, **C**, F, G.

ReorderTraces generates an error if the same trace is in the list twice.

In Igor7 or later, *anchorImage* can be _front_ or _back_. To move A to the front, you can write:

```
ReorderTraces _front_, {A}
```

See Also

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

The **ReorderImages** operation.

ReplaceNumberByKey

ReplaceNumberByKey(*keyStr*, *kwListStr*, *newNum* [, ***keySepStr***
[, ***listSepStr*** [, ***case***]]]**)**

The ReplaceNumberByKey function returns *kwListStr* after replacing the numeric value of the keyword-value pair specified by *keyStr*. *kwListStr* should contain keyword-value pairs such as `"KEY=value1,KEY2=value2"` or `"Key:value1;KEY2:value2"`, depending on the values for *keySepStr* and *listSepStr*.

Use ReplaceNumberByKey to add or modify numeric information in a string containing a `"key1:value1;key2:value2;"` style list such as those returned by functions like **AxisInfo** or **TraceInfo**.

If *keyStr* is not found in *kwListStr*, then the key and the value are appended to the end of the returned string.

keySepStr, *listSepStr,* and *case* are optional; their defaults are ":", ";", and 0 respectively.

Details

The actual string appended is:

[*listSepStr*] *keyStr keySepStr newNum listSepStr*

The optional leading list separator *listSepStr* is added only if *kwListStr* does not already end with a list separator.

keyStr is limited to 255 bytes.

kwListStr is searched for an instance of the key string bound by *listSepStr* on the left and a *keySepStr* on the right. The text up to the next ";" is replaced by *newNum* after conversion to text using the %.15g format (see **printf** for format conversion specifications).

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are usually case-insensitive. Setting the optional *case* parameter to 0 makes the comparisons case sensitive.

In Igor6, only the first byte of *keySepStr* and *listSepStr* was used. In Igor7 and later, all bytes are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *case* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print ReplaceNumberByKey("K1", "K1:7;", 4)              // prints "K1:4;"
Print ReplaceNumberByKey("k2", "K2=8;", 5, "=")         // prints "K2=5;"
Print ReplaceNumberByKey("K3", "K3:9,", 6, ":", ",")    // prints "K3:6,"
Print ReplaceNumberByKey("k3", "K0:9", 6, ":", ",")     // prints "K0:9,k3:6,"
Print ReplaceNumberByKey("k3", "K3:9,", 6, ":", ",")    // prints "K3:6,"
Print ReplaceNumberByKey("k3", "K3:9,", 6, ":", ",", 1) // prints "K3:9,k3:6,"
```

See Also

The **ReplaceStringByKey**, **NumberByKey**, **StringByKey**, **RemoveByKey**, **ItemsInList**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

ReplaceString

`ReplaceString(`*`replaceThisStr, inStr, withThisStr`*` [, `*`caseSense`*` [, `*`maxReplace`*`]])`

The ReplaceString function returns *inStr* after replacing any instance of *replaceThisStr* with *withThisStr*.

The comparison of *replaceThisStr* to the contents of *inStr* is case-insensitive. Setting the optional *caseSense* parameter to nonzero makes the comparison case-sensitive.

Usually all instances of *replaceThisStr* are replaced. Setting the optional *maxReplace* parameter limits the replacements to that number.

Details

If *replaceThisStr* is not found, *inStr* is returned unchanged.

If *maxReplace* is less than 1, then no replacements are made. Setting `maxReplace = Inf` is the same as omitting it.

Examples

```
Print ReplaceString("hello", "say hello", "goodbye")// prints "say goodbye"
Print ReplaceString("\r\n", "line1\r\nline2", "") // prints "line1line2"
Print ReplaceString("A", "an Ack-Ack", "a", 1)    // prints "an ack-ack"
Print ReplaceString("A", "an Ack-Ack", "a", 1, 1) // prints "an ack-Ack"
Print ReplaceString("", "input", "whatever")   // prints "input" (no change)
```

See Also

The **ReplaceStringByKey**, **cmpstr**, **StringMatch**, and **strsearch** functions.

ReplaceStringByKey

**`ReplaceStringByKey(`*`keyStr, kwListStr, newTextStr`*` [, `*`keySepStr`*`
    [, `*`listSepStr`*` [, `*`matchCase`*`]]])`**

The ReplaceStringByKey function returns *kwListStr* after replacing the text value of the keyword-value pair specified by *keyStr*. *kwListStr* should contain keyword-value pairs such as `"KEY=value1,KEY2=value2"` or `"Key:value1;KEY2:value2"`, depending on the values for *keySepStr* and *listSepStr*.

Use ReplaceStringByKey to add or modify text information in a string containing a `"key1:value1;key2:value2;"` style list such as those returned by functions like **AxisInfo** or **TraceInfo**.

If *keyStr* is not found in *kwListStr*, then the key and the value are appended to the end of the returned string.

keySepStr, *listSepStr,* and *matchCase* are optional; their defaults are ":", ";", and 0 respectively.

Details

The actual string appended is:

[*listSepStr*] *keyStr keySepStr newTextStr listSepStr*

The optional leading list separator *listSepStr* is added only if *kwListStr* does not already end with a list separator.

keyStr is limited to 255 bytes.

kwListStr is searched for an instance of the key string bound by a ";" on the left and a ":" on the right. The text up to the next ";" is replaced by *newTextStr*.

If *newTextStr* is `""`, any existing value is deleted, but the key, the key separator, and the list separator are retained. To remove a keyword-value pair, use the **RemoveByKey** function.

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are case-insensitive. Setting the optional *matchCase* parameter to 1 makes the comparisons case-sensitive.

In Igor6, only the first byte of *keySepStr* and *listSepStr* was used. In Igor7 and later, all bytes are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *matchCase* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print ReplaceStringByKey("KY", "KY:a;KZ:c", "b")      // prints "KY:b;KZ:c"
Print ReplaceStringByKey("KY", "ky=a;", "b", "=")     // prints "ky=b;"
Print ReplaceStringByKey("KY", "KY:a,", "b", ":", ",")// prints "KY:b,"
Print ReplaceStringByKey("ky", "ZZ:a,", "b", ":", ",")// prints "ZZ:a,ky:b,"
Print ReplaceStringByKey("kz", "KZ:a,", "b", ":", ",")// prints "KZ:b,"
Print ReplaceStringByKey("kz", "KZ:a,", "b", ":", ",", 1)// prints "KZ:a,kz:b,"
```

See Also

The **ReplaceString**, **ReplaceNumberByKey**, **NumberByKey**, **StringByKey**, **ItemsInList**, **RemoveByKey**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

ReplaceText

`ReplaceText [/W=`*`winName`*`/N=`*`name`*`]` ***`textStr`***

The ReplaceText operation replaces the text in the most recently created or changed annotation or in the annotation specified by /W=*winName* and/N=*name*.

Parameters

textStr can contain escape codes to set the font, size, style, color and other properties. See **Annotation Escape Codes** on page III-53 for details.

If the annotation is a color scale, this command replaces the text of the color scale's main axis label.

Flags

/N=*name*	Replaces the text of the named tag or textbox.
/W=*winName*	Replaces text in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

See Also

Tag, **TextBox**, **ColorScale**, **Legend**, **AppendText**, **Annotation Escape Codes** on page III-53.

ReplaceWave

`ReplaceWave [/W=`*`winName`*`] allinCDF`
`ReplaceWave [/X/W=`*`winName`*`] trace=`*`traceName, waveName`*
`ReplaceWave [/X/Y/W=`*`winName`*`] image=`*`imageName, waveName`*
`ReplaceWave [/X/Y/W=`*`winName`*`] contour=`*`contourName, waveName`*

The ReplaceWave operation replaces waves displayed in a graph with other waves. The waves to be replaced, and the replacement waves are chosen by the flags, the keyword and the wave names on the command line.

Flags

/W=*winName*	Replaces the wave in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/X	Replaces the wave defining the X data spacing.
/Y	Replaces the wave defining the Y data spacing.

Keywords

allinCDF	Searches the current data folder for waves with the same names as waves used in the graph. If found and if the waves are of the correct type, they replace the existing waves. Thus, if you have several data folders with identically-named waves containing data from different experimental runs, you can browse through the runs by moving from one data folder to another, using `ReplaceWave allinCDF` to update the graph.

contour=*contourName*	Replaces the wave supplying the Z data for *contourName*. If /X or /Y is used, replaces the wave used to set the X or Y data spacing (if the Z data are in a matrix) or the wave used to supply the X or Y positions if XYZ triplets were specified with three separate waves.
image=*imageName*	Replaces the wave supplying the Z data for *imageName*. If /X or /Y is used, replaces the wave used to set the X or Y data spacing.
trace=*traceName*	Replaces the wave associated with *traceName*. With the /X flag, *waveName* will replace the X wave associated with *traceName*, otherwise it will replace the Y wave. Note that *traceName* is derived from the Y wave name; if you created a graph using `Display jack vs sam`, you would use `ReplaceWave/X trace=jack,newsam` to replace the X wave.

Details

Waves are replaced in the graph specified by /W=*winName* otherwise waves are replaced in the top graph.

Updating a contour plot in response to replacing a wave can be time-consuming. If you must replace more than one wave, put all the commands separated by semicolons on a single line. In a macro, use **DelayUpdate** to prevent updates between command lines.

When using the allinCDF keyword, ReplaceWave cannot find waves buried in dynamic annotation text (for instance, using the \{} syntax in an annotation). ReplaceWave will not replace waves used for error bars, either.

Subsets of data, including individual rows or columns from a matrix, may be specified using **Subrange Display Syntax** on page II-250.

Examples

Make XY plot, then replace the waves:

```
Make fred=x, sam=log(x)
Display fred vs sam
Make fred2=2*x, sam2=ln(x)
ReplaceWave/X trace=fred, sam2
ReplaceWave trace=fred, fred2           // trace is now named fred2
```

Make contour plot with XYZ triplet waves, then replace the waves. Note the DelayUpdate commands after the first two ReplaceWave commands:

```
Make/N=100 junkx, junky, junkz          // Waves for XYZ triplets
junkx=trunc(x/10)                       // X wave for XYZ triplets
junky=mod(x,10)                         // Y wave for XYZ triplets
junkz=sin(junkx[p])*cos(junky[p])       // Z wave for XYZ triplets
Display; AppendXYZContour junkz vs {junkx, junky}  // Make contour plot
Make/O/N=150 junkx2, junky2, junkz2     // Make replacement waves
junkx2=trunc(x/15)
junky2=mod(x,15)
junkz2=sin(junkx2[p])*cos(junky2[p])
ReplaceWave/X contour=junkz,junkx2; DelayUpdate
ReplaceWave/Y contour=junkz,junky2; DelayUpdate
ReplaceWave contour=junkz,junkz2
```

This example is suitable for copying all the lines and pasting into the command line, or for use in a macro. If you are typing on the command line, you would want to put the ReplaceWave commands all on one line:

```
ReplaceWave/X contour=junkz,junkx2; ReplaceWave/Y contour=…
```

See Also

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

Resample

`Resample [`*`flags`*`] `*`waveName`*` [, `*`waveName`*`]…`

The Resample operation resamples *waveName* by interpolating or up-sampling (set by /UP=*upSample*), lowpass filtering, and decimating or down-sampling (set by /DOWN=*downSample*).

Lowpass filtering is specified with /N and /WINF or with /COEF=*coefsWaveName*.

The sampling frequency (1/DimDelta) of a resampled output wave *waveName* is changed by the ratio of *downSample*/*upSample*. For example, if *upSample*=4 and *downSample*=3, then the final sampling rate is 4/3 of the original value.

Straight interpolation can be accomplished by setting *upSample* to the interpolation factor and *downSample*=1, in which case the sample rate is multiplied by *upSample*. Deltax(*waveName*) will be proportionally smaller.

For decimation only, set *upSample*=1 and *downSample* to the decimation factor. The sample rate is divided by *downSample*, and deltax(*waveName*) will be proportionally larger.

Use **RatioFromNumber** to choose appropriate values for *upSample* and *downSample*, or use /SAME=*sWaveName* or /RATE=*sampRate*. See **Resampling Rates Example** for details.

When using /COEF=*coefsWaveName*, the filter coefficients should implement a low-pass filter appropriate for the *upSample* and *downSample* values or aliasing (filtering errors) will result. See **Advanced Externally-Supplied Low Pass Filter Example** for details.

Resampling Rates Flags

The *upSample* and *downSample* values define how much interpolation and decimation to perform. They can be set directly with /UP and /DOWN or indirectly with /SAME or /RATE

/DOWN=*downSample*

Down-samples or decimates the filtered result by this integer factor after up-sampling and lowpass filtering. The default is 1 (no down-sampling).

For example, /DOWN=3 places only every third value in the output wave.

Down-sampling divides the sampling rate of the filtered data by a factor of *downSample*. The DimDelta(*waveName*, *dim*) value is multiplied by the same factor.

/RATE=*sampRate*

Converts the output *waveName* to the specified sampling rate frequency (normally Hz).

The necessary *upSample* and *downSample* values for each *waveName* are computed internally as if you had executed:

```
RatioFromNumber (deltax(waveName)*sampRate)
upSample = V_numerator
downSample = V_denominator
```

/RATE returns V_numerator and V_denominator set to these automatically-determined values for the last *waveName*.

/SAME=*sWaveName*

Converts the output *waveName* to the same sampling rate as *sWaveName*, 1/DimDelta(*sWaveName*, *dim*). The necessary *upSample* and *downSample* values are computed internally as if you had executed:

```
Variable dd = DimDelta(waveName,dim)
RatioFromNumber dd/DimDelta(sWaveName,dim)
upSample = V_numerator
downSample = V_denominator
```

/SAME returns V_numerator and V_denominator set to these automatically determined values for the last *waveName*.

/UP=*upSample*

Up-samples or interpolates the input by this integer factor. The default is 1 (no up-sampling).

For example, /UP=4 inserts three extra points between each input point (producing 4 times as many values) before the lowpass filtering and down-sampling occurs.

Up-sampling multiplies the sampling rate of the input data by a factor of *upSample*, though no additional signal information is created. The DimDelta(*waveName*, *dim*) value is divided by the same factor.

Internal Sinc Reconstruction Filter Flags

If /COEF=*coefsWaveName* is not specified, Resample computes a windowed sinc filter from /N, /DOWN, /UP, and /WINF flag values.

If /COEF=*coefsWaveName* is specified, then *coefsWaveName* supplies the filter, and /N and /WINF are ignored. See **Externally-Supplied Low Pass Filter Flags**.

/COEF
: Replaces the first *waveName* with coefficients generated by *downSample*, *upSample*, *numReconstructionSamples*, and *windowKind*, a windowed sinc impulse response.

When resampling multiple *waveName*s with different filters (because /RATE or /SAME were specified and the multiple *waveName*s had different sampling rates), the filter used to resample the last *waveName* is returned.

/N=*numReconstructionSamples*

Specifies the number of input values used to created the up-sampled values (default is 21).

The value of *numReconstructionSamples* must be odd.

The size of the computed filter is (*numReconstructionSamples*-1) * *upSample* + 1.

Bigger is better: 15 is usually on the low side for yielding reasonably accurate results, and although 101 will nearly always give very good results, it will be slow.

Use /COEF to output the impulse response, and the FFT to display the frequency response of the interpolator:

```
Make/O coefs
Variable numReconstructionSamples= 51, upSample= 5
Resample/COEF/N=(numReconstructionSamples)/UP=(upSample) coefs
Variable evenNum= 2*floor((numpnts(coefs)+1)/2)
FFT/OUT=3/PAD={evenNum}/DEST=coefs_FFT coefs
Display coefs_FFT
```

Bigger is also slower: the filtering is computed in the time-domain, and execution time is linearly related to

upSample/*downSample* * *numReconstructionSamples*.

/WINF=*windowKind*

Applies the window, *windowKind*, to the computed filter coefficients. If /WINF is omitted, the Hanning window is used. For no coefficient windowing, use /WINF=None, though this is discouraged.

Windows alter the frequency response of the filter in obvious and subtle ways, enhancing the stop-band rejection or steepening the transition region between passed and rejected frequencies. They matter less when *numReconstructionSamples* is large.

Choices for *windowKind* are:

Bartlett, Blackman367, Blackman361, Blackman492, Blackman474, Cos1, Cos2, Cos3, Cos4, Hamming, Hanning, KaiserBessel20, KaiserBessel25, KaiserBessel30, Parzen, Poisson2, Poisson3, Poisson4, and Riemann.

See **FFT** for window equations and details.

Externally-Supplied Low Pass Filter Flags

/COEF =*coefsWaveName*

Identifies the wave, *coefsWaveName*, containing filter coefficients that implement a low-pass filter with a cutoff frequency of the lesser of 0.5/*upSample* and 0.5/*downSample*, where 0.5 corresponds to the Nyquist frequency of the up-sampled data.

For example, if *upSample*=2, then the filter must contain the classic “half-band” filter, which stops the higher half of the frequencies and passes the lower half. If *upSample*=10, then the filter must pass only the lowest 1/10th of the frequencies.

For *downSample* > *upSample*, the low-pass filter's cutoff frequency must be 0.5/*downSample*. This prevents the decimation from introducing aliasing to the resampled data.

To avoid shifting the output with respect to the input, *coefsWaveName* must have an odd length with the "center" coefficient in the middle of the wave.

The length of *coefsWaveName* must be 1+*upSample***n*, where *n* is any even integer.

Note: Instead of using /N=*numReconstructionSamples* with /COEF=*coefsWaveName*, *numReconstructionSamples* is computed from *upSample* and the number of points in *coefsWaveName*:

numReconstructionSamples=1+(numpnts(*coefsWaveName*)-1)/*upSample*.

Coefficients are usually symmetrical about the middle point, but this is not enforced.

coefsWaveName must not be a destination *waveName*.

coefsWaveName must be single- or double-precision numeric and one-dimensional.

Data Range Flags

/DIM=*d*	Specifies the wave dimension to resample. For *d* =0, 1, …, resampling is along rows, columns, etc. The default is /DIM=0, which resamples each individual column (each one a channel, say left and right) in a multidimensional *waveName* where each row comprises all sound samples at a particular time. To resample in multiple dimensions, execute the command once for each dimension. For example, use /DIM=0 followed by another command with /DIM=1 to resample a two-dimensional wave in each direction.
E=*endEffect*	Determines how to handle the ends of the resampled wave(s) (w) when fabricating missing neighbor values.

endEffect=0:	Bounce method. Uses w[*i*] in place of the missing w[-*i*] and w[*n*-*i*] in place of the missing w[*n*+*i*].
endEffect=1:	Wrap method. Uses w[*n*-*i*] in place of the missing w[-*i*] and vice versa.
endEffect=2:	Zero method (default). Uses 0 for any missing value.
endEffect=3:	Repeat method. Uses w[0] in place of the missing w[-*i*] and w[*n*] in place of the missing w[*n*+*i*].

Parameters

waveName can be a wave with any number of dimensions. Only one dimension is resampled. Use multiple Resample calls to resample across multiple dimensions.

Without /DIM, resampling is done along the row (first) dimension. Each column is resampled as if it were a separate one-dimensional row. This allows multichannel audio to be resampled to another frequency.

If /DIM=1, then resampling proceeds across all the columns of each row.

If /COEF is specified without *coefsWaveName*, then the first *waveName* is overwritten by the filter coefficients instead of being resampled.

Details

The filtering convolution is performed in the time-domain. That is, the FFT is not employed to filter the data. For this reason the coefficients length (/N or the length of *coefsWaveName*) should be small in comparison to the resampled waves.

Resample assumes that the middle point of *coefsWaveName* corresponds to the delay=0 point. The "middle" point number = trunc(numpnts(*coefsWaveName* -1)/2). *coefsWaveName* usually contains the two-sided impulse response of a filter, an odd number of points, and implements a low-pass filter whose cutoff frequency is the lesser of 0.5/*upSample* and 0.5/*downSample* (0.5 corresponds to the Nyquist frequency = 1/2 sampling frequency).

When /COEF creates a coefficients wave it sets the X scale deltax to 1 and alters the leftx value so that the zero-phase (center) coefficient is located at x=0.

Simple Examples

Interpolation by factor of 4, default filter:

```
Resample/UP=4 data
```

Decimation by factor of 3, default filter:

```
Resample/DOWN=3 data
```

Match sampling rates, default filter:

```
Resample/SAME=dataAtDesiredRate dataAtWrongRate1, dataAtWrongRate2,...
```

Resample waves to 10 KHz sampling rate:

```
Resample/RATE=10e3 dataAtWrongRate1, dataAtWrongRate2,...
```

Interpolate an image by a factor of 2:

```
Resample/UP=2 image              // default is /DIM=0, resample rows
Resample/UP=2/DIM=1 image        // resample across columns
```

Note: Interpolating by a factor of two does not produce an image with twice as many rows and columns. The new number of rows = (original rows-1)**upSample* +1, and a similar computation applies to columns.

Resampling Rates Example

Suppose we have an audio wave sampled at 44,100 Hz and we wish to resample it to a higher 192,000 Hz frequency.

We can use /RATE= 192000 and let Resample determine the correct values (provided *waveName* has its X scaling set properly to reflect sampling at 44100 Hz), but let's compute *upSample* and *downSample* ourselves.

Because the sampling rate = 1/deltax(*wave*), we can recast the /SAME formula to RatioFromNumber (*desiredSamplingRate*/*currentSamplingRate*):

```
•RatioFromNumber/V (192000 / 44100)
  V_numerator= 640; V_denominator= 147;
  ratio= 4.3537414965986;
  V_difference= 0;
```

Then *upSample*=640 and *downSample*=147.

The 44100 Hz input data will be interpolated by 640 to 28,224,000 Hz.

The result is low-pass filtered with a "cutoff frequency" of 1/640th of the interpolated Nyquist frequency = (28224000/2)/640 = 22,050 Hz, the same as the input signal's original Nyquist frequency.

The result will be decimated by 147 to 192,000 Hz, which is the desired output sampling frequency.

Note: If *downSample* had been greater than *upSample*, then the low-pass filter's cutoff frequency would have been 1/*downSample*th of the interpolated Nyquist frequency = (28224000/2)/*downSample*. This prevents the decimation from introducing aliasing to the resampled data.

```
Resample/UP=640/DOWN=147 sound          // convert 44.1 KHz to 192 KHz
```

Advanced Externally-Supplied Low Pass Filter Example

You can generate an appropriate filter by executing commands like these:

```
// Compute a filter for after the input is upsampled
// to restore the frequency content to the original range.
Variable fc = min(0.5 / upSample, 0.5 * upSample / downSample)
// Transition width, small widths need big n
Variable tw= fc/10
// Set end of pass band
Variable f1= fc-tw/2
// Set start of stop band
Variable f2= fc+tw/2
// Use bigger values of n to make the filter smoother
```

```
Variable nReconstruct= 31
Variable n= (nReconstruct-1)*upSample+1        // odd = no phase shift
// Create a wave to hold the coefficients; it gets resized to n
Make/O/N=0 coefsWaveName
FilterFIR/COEF/LO={f1,f2,n} coefsWaveName
```

However, FilterFIR does not create windowed sinc lowpass filters that have the endearing property that the original input values are unaltered in the filtered output, though only if *upSample* > *downSample*. This is called a "Nyquist filter" or "Kth-band filter" in the literature.

If *upSample* > *downSample*, you can enforce the Nyquist criterion by "zeroizing" the designed filter by setting every *upSample*th value to 0 except the center one.

```
// coefsWaveName length must be 1+upSample*n, where n is any even integer
Function Zeroize(w, upSample)
    Wave w                              // coefsWaveName
    Variable upSample                   // upSample value

    Variable n= DimSize(w,0)
    Variable centerP= floor((n-1)/2)          // if n=101, centerP= 50
    Variable i
    for (i=0; i<n; i+=upSample)
        if( i != centerP )
            w[i] = 0
        endif
    endfor
End
```

Resample zeroizes the internally-generated low pass filter when *upSample* > *downSample*.

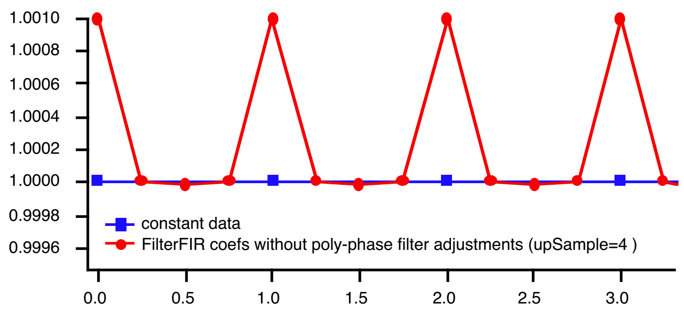
Additionally, the FilterFIR command generates a low-pass filter whose gain needs to be multiplied by *upsample*:

*coefsWaveName *= upSample*

When designing an externally supplied filter, you should also consider the filter's "polyphase" nature; *coefsWaveName* is actually a set of *upSample* interleaved filters, each with its own response. It makes sense to adjust these filters to produce consistent responses. If you don't, the results will contain ringing with a period of *upSample/downSample*. This is most apparent when *downSample* is 1.

Using the filter we've designed so far with *upSample*=4, here's the output of a constant-input wave:

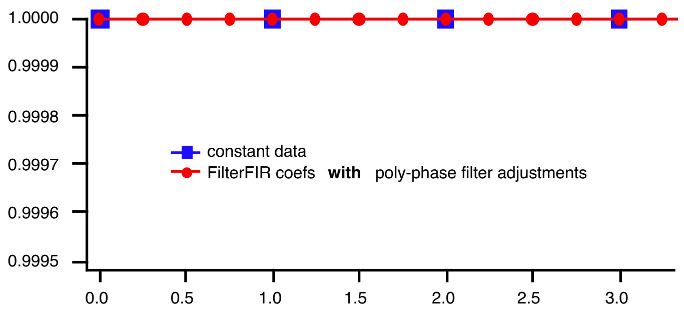
```
Make/O constantData= 1
Resample/COEF=coefsWaveName/UP=4 constantData
```



The graph shows that the filter response at 0 Hz for the first of 4 filters is 1.0010, the second and fourth filter's responses are very close to 1.0, and the third filter's response at 0 Hz is a little less than 1.0.

These variations can be eliminated by normalizing the sum of each polyphase filter to 1.0:

```
Function PolyphaseNormalize(w, upSample)
    Wave w                                  // coefsWaveName
    Variable upSample                       // upSample value

    Variable n= DimSize(w,0)
    Variable filt
    // for each filter (0..upSample-1)
    for (filt=0; filt<upSample; filt+=1)
        Variable total=0
        Variable pt
        // compute total for this filter
        for (pt=filt; pt<n; pt+=upSample)
            total += w[pt]
        endfor
        // divide by total to normalize total to 1
        for (pt=filt; pt<n; pt+=upSample)
            w[pt] /= total
        endfor
    endfor
End
```



Now the filter is ready to be used to filter data:

```
Resample/COEF=coefsWaveName/UP=(upSample)/DOWN=(downSample) dataWave
```

You can see that designing an externally-supplied lowpass filter is much more complicated than using the internal sinc reconstruction filter, which does all this zeroizing, scaling, and polyphase normalization for you.

References

Mintzer, F., On half-band, third-band, and Nth band FIR filters and their design, *IEEE Trans. on Acoust., Speech, Signal Process.*, *ASSP-30*, 734-738, 1982.

See Also

RatioFromNumber, **FilterFIR**, **interp**, **Interp2D**, **Interpolate2**, **ImageInterpolate**, **Loess**

ResumeUpdate

`ResumeUpdate`

The ResumeUpdate operation cancels the corresponding **PauseUpdate**.

This operation is of use in macros. It is not allowed from the command line. It is allowed but has no effect in user-defined functions. During execution of a user-defined function, windows update only when you explicitly call the DoUpdate operation.

See Also

The **DelayUpdate**, **DoUpdate**, and **PauseUpdate** operations.

return

`return` `[`***`expression`***`]`

The return flow control keyword immediately stops execution of the current procedure. If called by another procedure, it returns *expression* and control to the calling procedure.

Functions can return only a single value directly to the calling procedure with a return statement. The return value must be compatible with the function type. A function may contain any number of return statements; only the first one encountered during procedure execution is evaluated.

A macro has no return value, so return simply quits the macro.

See Also

The Return Statement on page IV-35.

Reverse

`Reverse` `[`***`type flags`***`][`**`/DIM=`*`d`*` /P`**`]` ***`waveA`*** `[`**`/D = `*`destWaveA`***`][, `***`waveB`*** `[`**`/D = `*`destWaveB`***`][, …]]`

The Reverse operation reverses data in a wave in a specified dimension. It does not accept text waves.

Flags

/DIM = *d*	Specifies the wave dimension to reverse. *d*=-1: Treats entire wave as 1D (default). For *d*=0, 1, …, operates along rows, columns, etc.
/P	Suppresses adjustment of dimension scaling. Without /P the scaled dimension value of reversed points remains the same.

Type Flags *(used only in functions)*

Reverse also can use various type flags in user functions to specify the type of destination wave reference variables. These type flags do not need to be used except when it is needed to match another wave reference variable of the same name or to identify what kind of expression to compile for a wave assignment. See **WAVE Reference Types** on page IV-67 and **WAVE Reference Type Flags** on page IV-68 for a complete list of type flags and further details.

Wave Parameters

Note: ***All*** wave parameters must follow *wave* in the command. All wave parameter flags and type flags must appear immediately after the operation name (Reverse).

/D=*destWave*	Specifies the name of the wave to hold the reversed data. It creates *destWave* if it does not already exist or overwrites it if it exists.

Details

If the optional /D = *destWave* flag is omitted, then the wave is reversed in place.

See Also

Sorting on page III-126, **Sort**, **SortColumns**

RGBColor

The RGBColor structure is used as a substructure usually to store various color settings.

```
Structure RGBColor
   UInt16 red
   UInt16 green
   UInt16 blue
EndStructure
```

RGBAColor

The RGBAColor structure is the same as RGBColor but with an alpha field to represent translucency.

```
Structure RGBAColor
   UInt16 red
   UInt16 green
   UInt16 blue
   UInt16 alpha
EndStructure
```

rightx

`rightx(`*`waveName`*`)`

The rightx function returns the X value corresponding to point N of the named 1D wave of length N.

Details

Note that the point numbers in a wave run from 0 to N-1 so there is no point with this X value. To get the X value of the last point in a wave (point N-1), use the following:

```
pnt2x(waveName,numpnts(waveName)-1)        // N = numpnts(waveName)
```

which is more accurate than:

```
rightx(waveName) - deltax(waveName)
```

The rightx function is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details. The equivalent information for any dimension can be calculated this way:

IndexN = DimSize(*wave*, *dim*)*DimDelta(*wave*, *dim*) + DimOffset(*wave*, *dim*)

Here *IndexN* is the value of the scaled dimension index corresponding to element N of the dimension *dim* in a wave named *wave* that has N elements in that dimension.

See Also

The **deltax** and **leftx** functions, also the **pnt2x** and **numpnts** functions.

For an explanation of waves and dimension scaling, see **Changing Dimension and Data Scaling** on page II-63.

For multidimensional waves, see **DimDelta**, **DimOffset**, and **DimSize**.

root

`root[:dataFolderName[:dataFolderName[:…]]][:objectName]`

Igor's data folder hierarchy starts with the root folder as its basis. The root data folder always exists and it contains all other objects (waves, variables, strings, and data folders). By default, the root data folder is the current data folder in a new experiment. In commands, root is used as part of a path specifying the location of a data object in the folder hierarchy.

See Also

Chapter II-8, **Data Folders**.

Rotate

`Rotate rotPoints, waveName [, waveName]…`

The Rotate operation rotates the Y values of waves in wavelist by *rotPoints* points.

Parameters

If *rotPoints* is positive then values are rotated from the start of the wave toward the end and *rotPoints* values from the end of a wave wrap around to the start of the wave.

If *rotPoints* is negative then values are rotated from the end of the wave toward the start and *rotPoints* values from the start of a wave wrap around to the end of the wave.

Details

The X scaling of the named waves is changed so that the X values for the Y values remains the same except for the points that wrap around.

The Rotate operation is not multidimensional aware. To rotate rows or columns of 2D waves, see the rotateRows, rotateCols, rotateLayers and rotateChunks keywords for **MatrixOp** and the rotateRows and rotateCols keywords for **ImageTransform**.

For general information about multidimensional analysis, see **Analysis on Multidimensional Waves** on page II-86.

See Also

The shift parameter of the **WaveTransform** operation.

round

`round(num)`

The round function returns the integer value closest to *num*.

The rounding method is "away from zero".

See Also

The **ceil**, **floor**, and **trunc** functions.

rtGlobals

`#pragma rtGlobals = 0, 1, 2, or 3`

`#pragma rtglobals=<n>` is a compiler directive that controls compiler and runtime behaviors for the procedure file in which it appears.

This statement must be flush against the left edge of the procedure file with no indentation. It is usually placed at the top of the file.

`#pragma rtglobals=0` turns off runtime creation of globals. This is obsolete.

`#pragma rtglobals=1` is a directive that turns on runtime lookup of globals. This is the default behavior if #pragma rtGlobals is omitted from a given procedure file.

`#pragma rtGlobals=2` turns off compatibility mode. This is mostly obsolete. See **Legacy Code Issues** on page IV-104 for details.

`#pragma rtglobals=3` turns on runtime lookup of globals, strict wave reference mode and wave index bounds checking.

rtGlobals=3 is recommended.

See **The rtGlobals Pragma** on page IV-48 for a detailed explanation of rtGlobals.

s

`s`

The s function returns the current chunk index of the destination wave when used in a multidimensional wave assignment statement. The corresponding scaled chunk index is available as the **t** function.

Details

Unlike **p**, outside of a wave assignment statement, s does not act like a normal variable.

See Also

Waveform Arithmetic and Assignments on page II-69.

For other dimensions, the **p**, **r**, and **q** functions.

For scaled dimension indices, the **x**, **y**, **z** and **t** functions.

Save

`Save [`*`flags`*`]` ***`waveList`*** **`[as`** ***`fileNameStr`*** **`]`**

The Save operation saves the named waves to disk as text (/F, /G or /J) or as Igor binary.

Parameters

waveList is either a list of wave names or, if the /B flag is present, a string list of references to waves. For example, the following commands are equivalent, assuming that the waves in question are in the root data folder and root is the current data folder:

```
Save/J wave0,wave1 as "Test.dat"
Save/J root:wave0,root:wave1 as "Test.dat"
Save/J/B "wave0;wave1;" as "Test.dat"
Save/J/B "root:wave0;root:wave1;" as "Test.dat"
String list="root:wave0;root:wave1;"; Save/J/B list as "Test.dat"
```

The form using the /B flag and a string containing a list of references to waves saves a very large number of waves using one command. This is not possible using a list of wave names because of the 1000 byte command line length limit. When using this form, the string must contain semicolon-separated wave names or data folder paths leading to waves. Liberal names in the string may be quoted or unquoted.

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If it cannot determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/A[=*a*]	Appends to the file rather than overwriting it (with /T, /G or /J). *a*=0: Does not append. *a*=1: Appends to the file with a blank line before the appended data (same as /A only). *a*=2: Appends to the file with no blank line before the appended data.
/B	The *waveList* parameter is a string containing a list of references to waves instead of a literal list of waves.
/C	Saves a copy of the wave when saving as Igor binary.
/DLIM=*delimStr*	Specifies the string to use as a column delimiter. This flag affects general text saves (/G) and delimited text saves (/J) only. *delimStr* defaults to tab. It can consist of multiple characters. If you choose a delimiter that also appears in the data you are saving, for example if you choose to save text waves containing commas using comma as the separator, the resulting file is likely to be misinterpreted by any software loading it. /DLIM was added in Igor Pro 7.00.
/DSYM=*dsStr*	Specifies a string containing the character to use as the decimal symbol for all numbers (default is a period). If *dsStr* is empty (`""`), then the decimal symbol is as defined in system preferences as of when Igor was launched.
/E=*useEscapeCodes*	Determines whether to use escape sequences for special characters. /E=1: Converts carriage-return, linefeed, tab, and backslash characters to escape sequences when writing general or delimited text files (default; same as no /E). /E=0: No escape sequences used in general or delimited text files. When saving text waves containing backslashes (such as Windows paths) in a file intended for another program, you probably should use /E=0.
/F	Writes delimited and general text files with numeric formatting as it appears in the top table. Has no effect if there is no top table or if the wave being saved does not appear in the top table. **Note**: The text written to the file is exactly as displayed in the table. Set the table to display as many digits of precision as you want in the file. **Note**: Fractional and out-of-range floating point wave data can not be formatted as octal or hex. See **Octal Numeric Formats** on page II-192 and **Hexadecimal Numeric Formats** on page II-192 for details. **Note**: When saving a multi-column wave (1D complex wave or multi-dimensional wave), all columns of the wave are saved using the table format for the first table column from the wave.
/G	Saves waves in general text format.
/H	"Adopts" the waves specified by *waveList*. "Adopt" means that any connection between the waves and external files is severed. The waves become part of the current experiment. When the experiment is next saved, the waves are saved in the experiment file (for an packed experiment) or in the experiment folder (for an unpacked experiment).

	When you use the /H flag, all other flags and the *fileNameStr* parameter are ignored. The wave is not actually saved but rather is marked for saving as part of the current experiment.
	You would normally do this to make an experiment more self-contained which makes it easier to send to other people. See **Sharing Versus Copying Igor Binary Files** on page II-137 and the **LoadWave** /H flag.
/I	Presents a dialog from which you can specify file name and folder.
/J	Saves waves in delimited text format.
	The delimiter defaults to tab unless you specify another delimiter using /DLIM.
/M=*termStr*	Specifies the terminator character or characters to use at the end of each line of text. The default is `/M="\r"`, which uses a carriage return character. This is the Macintosh convention. To use the Windows convention, carriage return plus linefeed, specify `/M="\r\n"`. To use the Unix convention, just a linefeed, specify `/M="\n"`.
/O	Overwrites file if it exists already.
/P=*pathName*	Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.
/T	Saves waves in Igor Text format.

/U={*writeRowLabels*, *rowPositionAction*, *writeColLabels*, *colPositionAction*}

These parameters affect the saving of a matrix (2D wave) to a delimited text (/J) or general text (/G) file. They are accepted no matter what the save type is but are ignored when they don't apply.

If *writeRowLabels* is nonzero, Save writes the row labels of the matrix as the first column of data in the file.

rowPositionAction has one of the following values:

0:	Don't write a row position column.
1:	Writes a row position column based on the row scaling of the matrix wave.
2:	Writes a row position column based on the contents of the row position wave for the matrix. The row position wave is an optional 1D wave whose name is the same as the matrix wave but with the prefix "RP_".

writeColumnLabels and *columnPositionAction* have analogous meanings. The prefix used for the column position wave is "CP_".

See Chapter II-9, **Importing and Exporting Data**, for further details.

/W	Saves wave names (with /G or /J).

Details

The Save operation saves only the named waves; it does not save the entire experiment.

Waves saved in Igor binary format are saved one wave per file. If you are saving more than one wave, you must not specify a *fileNameStr*. Save will give each file a name which consists of the wave name concatenated with ".ibw".

When you save a wave as Igor binary, unless you use the /C flag to save a copy, the current experiment subsequently references the file to which the wave was saved. See **References to Files and Folders** on page II-22 for details.

In a general text file (/G), waves with different numbers of points are saved in different groups. Waves with different precisions and number types are saved in same group if they have the same number of points.

In a delimited text file (/J), all waves are saved in one group whether or not they have the same number of points.

If you save multiple 2D waves, the blocks of data are written one after the other.

If you save 3D waves, the data for each wave is written as a contiguous block having as many columns as there are columns in the wave, and R*L rows, where R is the number of rows in the multidimensional wave and L is the number of layers. All rows for layer 0 are saved followed by all rows for layer 1, and so on.

If you save 4D waves, the data for each wave is written as a contiguous block having R*L*C rows, where R is the number of rows, L is the number of layers and C is the number of chunks. Igor writes all data for chunk 0 followed by all data for chunk 1, and so on.

The Save operation will always present a save dialog if you try to save to an existing file without using the overwrite flag.

Here are some details about saving an Igor binary file.

If you omit the path or the file name, the Save operation will normally present a save dialog. However, if the wave has already been saved to a stand-alone file and if you use the overwrite flag, it will save the wave to the same file without a dialog. Also, if the wave has never been saved and the current experiment is an unpacked experiment, it will save to the home folder without a dialog.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like “`hd:Folder1:`” or “`C:\\Folder1\\`”. See **Symbolic Paths** on page II-21 for details.

Examples

This function uses the string list of references to waves to save some or all of the waves in the current data folder:

```
Function SaveWavesFromCurrentDF(matchStr)
   String matchStr                  // As for the WaveList function.

   String list
   list = WaveList(matchStr, ";", "")
   Save/O/J/W/I/B list
End
```

For example, to save all of the waves in the current data folder, execute:

```
SaveWavesFromCurrentDF("*")
```

To save those waves in the current data folder whose name starts with “wave”, execute:

```
SaveWavesFromCurrentDF("wave*")
```

This function saves all of the waves used in a particular graph:

```
Function SaveWavesFromGraph(graphName)      // Saves all waves in graph.
   String graphName                         // "" for top graph.

   String list, traceList, traceName
   Variable index = 0
   list = ""
   traceList = TraceNameList(graphName, ";", 1)
   do
      traceName = StringFromList(index, traceList, ";")
      if (strlen(traceName) == 0)
         break
      endif
      Wave w = TraceNameToWaveRef(graphName, traceName)
      list += GetWavesDataFolder(w,2) + ";"
      index += 1
   while(1)

   if (strlen(list) > 0)
      Save/O/J/W/I/B list
   endif
End
```

See Also

Exporting Data on page II-156.

SaveData

`SaveData [`*`flags`*`] `*`fileOrFolderNameStr`*

The SaveData operation writes data from the current data folder of the current experiment to a packed experiment file on disk or to a file system folder. “Data” means Igor waves, numeric and string variables, and data folders containing them. The data is written as a packed experiment file or as unpacked Igor binary files in a file-system folder.

Warning: If you make a mistake using SaveData, it is possible to overwrite critical data, even entire folders containing critical data. It is your responsibility to make sure that any file or folder that you can not afford to lose is backed up. If you provide procedures for use by other people, you should warn them as well.

SaveData provides a way to save data for archival storage or unload data from memory during a lengthy process like data acquisition. The file or files that SaveData writes are disassociated from the current experiment.

Use SaveData to save experiment data using Igor procedures. To save experiment data interactively, use the Save Copy button in the Data Browser (Data menu).

Parameters

fileOrFolderNameStr specifies the packed experiment file (if /D is omitted) or the file system folder (if /D is present) in which the data is to be saved. The documentation below refers to this file or folder as the “target”.

If you use a full or partial path for *fileOrFolderNameStr*, see **Path Separators** on page III-401 for details on forming the path.

If *fileOrFolderNameStr* is omitted or is empty (`""`), SaveData displays a dialog from which you can select the target. You also get a dialog if the target is not fully specified by *fileOrFolderNameStr* or the /P=*pathName* flag.

Flags

/D [=*d*]	Writes to a file-system folder (a directory). If omitted, SaveData writes to an Igor packed experiment file.

d=1:	If the target folder already exists, the new data is “mixed-in” with the data already there (same as /D).

d=2:	If the target folder already exists, **it is completely deleted before the writing of data starts**.

If in doubt, use /D=1. See **Details** below.

/I	Presents a dialog in which you can interactively choose the target.

/J=*objectNamesStr*	Saves only the objects named in the semicolon-separated list of object names. See **Details** below.

/L=*saveFlags*	Controls what kind of data objects are saved with a bit for each data type:

saveFlags	**Bit Number**	**Saves this Type of Object**
1	0	Waves
2	1	Numeric variables
4	2	String variables

To save multiple data types, sum the values shown in the *saveFlags* column. For example, /L=1 saves waves only, /L=2 saves numeric variables only and /L=3 saves both waves and numeric variables.

If /L is not specified, all of these object types are saved. This is equivalent to /L=7. All other bits are reserved and must be set to zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/M=*modDateTime*	Saves waves modified on or after the specified modification date/time. Waves modified before *modDateTime* will not be saved. Applies to waves only (not variables or strings). *modDateTime* is in standard Igor time format — seconds since 1/1/1904. If *modDateTime* is zero, all waves will be saved, as if there were no /M flag at all.
/O	Overwrites existing files or folders on disk. Warning: If you use the /O flag and if the target already exists, it will be overwritten without any warning. If you use /O with /D=2, **you will completely overwrite the target folder and all of its contents, including subfolders**. Do not use /O with /D unless you are absolutely sure you know what your doing.
/P=*pathName*	Specifies the folder in which to save the specified file or folder. *pathName* is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\`". See **Symbolic Paths** on page II-21 for details. When used with the /D flag, if /P=*pathName* is present and *fileOrFolderNameStr* is ":", the target is the directory specified by /P=*pathName*.
/Q	Suppresses normal messages in the history area.
/R	Recursively saves subdata folders.
/T [=*topLevelName*]	Creates an enclosing data folder in the target with the specified name, *topLevelName*, and writes the data to the new data folder. If just /T is specified, it creates an enclosing data folder in the target using the name of the data folder being saved. However, if the data folder being saved is the root data folder, the name Data is used instead of root. In packed experiment files and unpacked experiment folders, the root data folder is implicit. If /T is omitted, the contents of the current data folder are saved with no enclosing data folder.

Details

If /J=*objectNamesStr* is used, then only the objects named in *objectNamesStr* are saved. For example, specifying `/J="wave0;wave1;"` will save only the two named waves, ignoring any other data in all data folders.

The list of object names used with /J must be semicolon-separated. A semicolon after the last object name in the list is optional. The object names must not be quoted even if they are liberal. The list is limited to 1000 characters.

Using `/J=""` acts like no /J at all.

The /M=*modDateTime* flag can be used in data acquisition projects to save only those waves modified since the previous save. For example, assume that we have a global variable in the root data folder named gLastWaveSaveDateTime. Then this function will write out only those waves modified since the previous save:

```
Function SaveModifiedWaves(savePath)
    String savePath     // Symbolic path pointing to output directory
    NVAR lastSave = root:gLastWaveSaveDateTime
    SaveData/O/P=$savePath/D=1/L=1/M=(lastSave) ":"
    lastSave = datetime
End
```

Because the datetime function and the wave modification date have a coarse resolution (one second), this function may sometimes save the same wave twice.

The /M flag makes sense only in conjunction with the /D=1 flag because /D=1 is the only way to mix-in new data with existing data.

Writing to a Packed Experiment File

When writing to a packed file, SaveData creates a standard packed Igor experiment file which you can open as an experiment, browse using the Data Browser, or access using the **LoadData** operation.

If you do not use the /O (overwrite) flag and the packed file already exists on disk, SaveData will present a dialog to confirm which file you want to write to. If you use the /O flag, SaveData will overwrite without presenting a dialog. When writing a packed file, SaveData always completely overwrites the preexisting packed file.

Appending to a packed experiment file is not supported because dealing with the possibility of name conflicts (e.g., two waves with the same name in the same data folder in the packed experiment file) would be technically difficult, very slow and errors would result in corrupted files.

Writing to a File-System Folder

When saving to a folder on disk, SaveData writes wave files, variables files, and subfolders. This resembles the experiment folder of an unpacked experiment, but it does not contain other unpacked experiment files, such as history or procedures. You can browse the folder using the Data Browser or access it using the **LoadData** operation.

If the target directory does not exist, SaveData creates it.

If you do not use the /O (overwrite) flag and the target folder already exists on disk, SaveData will present a dialog to confirm that you want to write to it. SaveData checks for the existence of the top file system folder only. For example, if you write data to hd:Data:Run1, SaveData will display a dialog if hd:Data:Run1 exists. But SaveData will not display a dialog for any folders inside hd:Data:Run1.

If you use the /O flag, SaveData will write without presenting a dialog.

When writing to a directory, SaveData can operate in one of two modes. If you use /D=1 or just /D, SaveData operates in "mix-in" mode. If you use /D=2, SaveData operates in "delete" mode.

Warning: If the target directory exists and delete mode is used, SaveData deletes the target directory and all of its contents. Then SaveData creates the target directory and writes the data to it. This is a complete overwrite operation.

If the target directory exists and mix-in mode is used, SaveData does not do any explicit deletion. It writes data to the target directory and any subdirectories. Conflicting files in any directory are overwritten but other files are left intact.

To prevent you from inadvertently deleting an entire volume, SaveData will not permit you to target the root directory of any volume. You must target a subdirectory.

The /J flag will not work as expected when writing numeric and string variables in mix-in mode. Instead of mixing-in the specified variables, SaveData will overwrite all variables already in the target. This is because all numeric and string variables in a particular data folder are stored in a single file-system folder (named "variables"), so it is not possible to mix-in. Since waves are written one-to-a-file, /J will work as expected for waves.

When SaveData writes a wave to a file-system folder, the file name for the wave is the same as the wave name, with the extension ".ibw" added. This is true even if the wave in the experiment was loaded from a file with a different name.

Outputs

SaveData sets the variable V_flag to zero if the operation succeeded or to nonzero if it failed. The main use for this is to determine if the user clicked Cancel during an interactive save. This would occur if you use the /I flag or if you omit /O and the target already exists. V_flag will also be nonzero if an error occurs during the save.

SaveData sets the string variable S_path to the full file system path to the file or folder that was written. S_path uses Macintosh path syntax (e.g., "hd:FolderA:FolderB:"), even on Windows. When saving unpacked, S_path includes a trailing colon.

Examples

Write the contents of the current data folder and all subdata folders to a packed experiment file:

```
Function SaveDataInPackedFile(pathName, fileName)
    String pathName             // Name of symbolic path
    String fileName             // Name of packed file to be written

    SaveData/R/P=$pathName fileName
End
```

Write the contents of the current data folder and all subdata folders to an unpacked file-system folder:

```
Function SaveDataInUnpackedFolder(pathName, folderName)
```

```
   String pathName               // Name of symbolic path
   String folderName             // Name of file-system folder

   SaveData/D=1/R/P=$pathName folderName
End
```

Copy the contents of an unpacked file-system folder to a packed experiment file:

```
Function TransferUnpackedToPacked(path1, folderName, path2, fileName)
   String path1           // Points to parent of unpacked folder
   String folderName      // Name of folder containing unpacked data
   String path2           // Points to folder where file is to be written
   String fileName        // Name of packed file to be written

   DFREF savedDF = GetDataFolderDFR()

   NewDataFolder/O/S :TempTransfer

   // Load all data from the unpacked folder.
   LoadData/D/Q/R/P=$path1 folderName

   // Save all data to the packed file.
   SaveData/R/P=$path2 fileName

   KillDataFolder :              // Kill TempTransfer

   SetDataFolder savedDF
End
```

See Also

The **LoadData** and **SaveGraphCopy** operations; the **SpecialDirPath** function. **Saving Package Preferences** on page IV-237; **Exporting Data** on page II-156; **The Data Browser** on page II-106.

SaveExperiment

SaveExperiment [**flags**] [**as** ***fileName***]

The SaveExperiment operation saves the current experiment.

Warning: SaveExperiment overwrites any previously-existing file named fileName.

Parameters

The optional *fileName* string contains the name of the experiment to be saved. *fileName* can be the currently open experiment, in which case it overwrites the experiment file.

If *fileName* and *pathName* are omitted and the experiment is Untitled, you will need to locate where the experiment file will be saved interactively via a dialog.

If you use a full or partial path for *pathName*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/C	Saves an experiment copy (valid only when *fileName* or *pathName* is provided or both if experiment is Untitled).
/F={*format*, *unpackedExpFolderNameStr*, *unpackedExpFolderMode*}	Specifies the experiment file format. See ***Experiment File Format*** below for details.
/P=*pathName*	Specifies folder in which to save the experiment. *pathName* is the name of an existing symbolic path.

Details

SaveExperiment acts like the Save menu command in the File menu. If the experiment is associated with an already saved file, then SaveExperiment with no parameters will simply save the current experiment. If the experiment resides only in memory and has not yet been saved, then a dialog will be presented unless the path and file name are specified.

If you use a full path in the name you will not need the /P flag. If instead you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like “`hd:Folder1:`” or “`C:\\Folder1\\`”. See **Symbolic Paths** on page II-21 for details.

Experiment File Format

For background information on experiment file formats, see **Experiments** on page II-17.

The /F flag provides control of the file format of a previously-unsaved experiment independent of the user's preferences as set in the Experiment Settings section of the Miscellaneous Settings dialog. It also allows you to save a previously-saved experiment using a different experiment file format.

If you just want to save the current experiment in its current format, you don't need to use /F.

If you use /F, you must fully-specify the location of the experiment file through the /P flag and the *fileName* parameter or through *fileName* alone if it contains a full path.

The format parameter controls the experiment file format used by SaveExperiment:

format =-1:	Default format.
format =0:	Unpacked experiment file
format =1:	Packed experiment file

If /F is omitted or if *format* is -1 then the experiment is saved in its current format or, if it was never saved to disk, in the format specified in the Experiment Settings section of the Miscellaneous Settings dialog.

If *format* = 0, the experiment is saved in unpacked experiment file format. *fileName* must end with ".uxp" or ".uxt".

If *format* = 1, the experiment is saved in packed experiment file format. *fileName* must end with ".pxp" or ".pxt".

Unpacked Experiment Folder

The unpacked experiment folder is the folder in which wave files, the history file, the variables file, and other experiment files are stored for an unpacked experiment. See **Saving as an Unpacked Experiment File** on page II-17 for details.

The /F *unpackedExpFolderNameStr* parameter specifies the name of the experiment folder for an unpacked experiment. It contains a folder name, not a full or partial path. It is ignored unless saving in unpacked experiment format.

The unpacked experiment folder is created in the same directory as the experiment file.

If /F=0 is used and *unpackedExpFolderNameStr* is "" then the experiment folder name is the same as the experiment file name with the extension removed and a space and "Folder" added.

If the specified unpacked experiment folder already exists and is the current experiment's unpacked experiment folder, it is reused. "Reuse" means that SaveExperiment saves files in the unpacked experiment folder, possibly overwriting files already in it, but does not delete any files or folders already in it.

The *unpackedExpFolderMode* parameter controls what happens if the folder to be used as the unpacked experiment folder already exists and is not the current experiment's unpacked experiment folder:

unpackedExpFolderMode=0 :	SaveExperiment returns an error.
unpackedExpFolderMode=1 :	SaveExperiment displays a dialog asking the user if it is OK to reuse the folder. If the user answers yes, the operation proceeds. Otherwise, it returns an error.
unpackedExpFolderMode=2 :	SaveExperiment reuses the folder without asking the user.

Warning: If you pass 2 for *unpackedExpFolderMode*, files and folders in the unpacked experiment folder may be overwritten without the user's express permission.

SaveGraphCopy

`SaveGraphCopy [`*`flags`*`] [as `*`fileNameStr`*`]`

The SaveGraphCopy operation saves a graph and its waves in an Igor packed experiment file.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/I	Presents a dialog from which you can specify file name and folder.
/O	Overwrites file if it exists already.
/P=*pathName*	Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.
/W= *winName*	*winName* is the name of the graph to be saved. If /W is omitted or if *winName* is `""`, the top graph is saved.
/Z	Errors are not fatal and error dialogs are suppressed. See Details.

Details

The main uses for saving as a packed experiment are to save an archival copy of data or to prepare to merge data from multiple experiments (see **Merging Experiments** on page II-19). The resulting experiment file preserves the data folder hierarchy of the waves displayed in the graph starting from the “top” data folder, which is the data folder that encloses all waves displayed in the graph. The top data folder becomes the root data folder of the resulting experiment file. Only the graph, its waves, dashed line settings, and any pictures used in the graph are saved in the packed experiment file, not procedures, variables, strings or any other objects in the experiment.

SaveGraphCopy does not work well with graphs containing controls. First, the controls may depend on waves, variables or FIFOs (for chart controls) that SaveGraphCopy will not save. Second, controls typically rely on procedures which are not saved by SaveGraphCopy.

SaveGraphCopy does not know about dependencies. If a graph contains a wave, wave0, that is dependent on another wave, wave1 which is not in the graph, SaveGraphCopy will save wave0 but not wave1. When the saved experiment is open, there will be a broken dependency.

SaveGraphCopy sets the variable V_flag to 0 if the operation completes normally, to -1 if the user cancels, or to another nonzero value that indicates that an error occurred. If you want to detect the user canceling an interactive save, use the /Z flag and check V_flag after calling SaveGraphCopy.

The **SaveData** operation also has the ability to save data from a graph to a packed experiment file. SaveData is more complex but a bit more flexible than SaveGraphCopy.

Examples

This function saves all graphs in the experiment to individual packed experiment files.

```
Function SaveAllGraphsToPackedFiles(pathName)
    String pathName         // Name of an Igor symbolic path.

    String graphName
    Variable index

    index = 0
    do
        graphName = WinName(index, 1)
        if (strlen(graphName) == 0)
            break
        endif

        String fileName
        sprintf fileName, "%s.pxp", graphName

        SaveGraphCopy/P=$pathName/W=$graphName as fileName

        index += 1
    while(1)
End
```

See Also

SaveTableCopy and **SaveData** operations; **Merging Experiments** on page II-19.

SaveNotebook

`SaveNotebook [`***`flags`***`]`***`notebookName`***` [as `***`fileNameStr`***`]`

The SaveNotebook operation saves the named notebook.

Parameters

notebookName is either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-87 for details on host-child specifications.

If *notebookName* is an host-child specification, /S must be used and *saveType* must be 3 or higher.

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/ENCG=*textEncoding*

Specifies text encoding in which the notebook is to be saved.

This flag was added in Igor Pro 7.00.

This is relevant for plain text notebooks only and is ignored for formatted notebooks because they can contain multiple text encodings. See **Plain Text File Text Encodings** on page III-417 and **Formatted Text Notebook File Text Encodings** on page III-421 for details.

If omitted, the file is saved in its original text encoding. Normally you should omit /ENCG. Use it only if you have some reason to change the file's text encoding.

Passing 0 for *textEncoding* acts as if /ENCG were omitted.

See **Text Encoding Names and Codes** on page III-434 for a list of accepted values for *textEncoding*.

This flag does not affect HTML export. Use /H instead.

/H={*encodingName*, *writeParagraphProperties*, *writeCharacterProperties*, *PNGOrJPEG*, *quality*, *bitDepth*}

Controls the creation of an HTML file.

encodingName specifies the HTML file text encoding. The recommended value is "UTF-8".

writeParagraphProperties determines what paragraph properties SaveNotebook will write to the HTML file. This is a bitwise parameter with the bits defined as follows:

Bit 0: Write paragraph alignment.
Bit 1: Write first indent.
Bit 2: Write minimum line spacing.
Bit 3: Write space-before and space-after paragraph.

All other bits are reserved for future use and should be set to zero.

writeCharacterProperties determines what character properties SaveNotebook will write to the HTML file. This is a bitwise parameter with the bits defined as follows:

Bit 0: Write font families.
Bit 1: Write font sizes.
Bit 2: Write font styles.
Bit 3: Write text colors.
Bit 4: Write text vertical offsets.

All other bits are reserved for future use and should be set to zero.

If you set bit 2, SaveNotebook exports only the bold, underline, and italic styles because other character styles are not supported by HTML.

PNGOrJPEG determines whether SaveNotebook will write picture files as PNG or JPEG:

0: PNG (default).
1: JPEG.
2: JPEG.

In Igor7 and later, there is no difference between *PNGOrJPEG*=1 and *PNGOrJPEG*=2.

See **Details** for more on HTML picture files.

quality specifies the degree of compression or image quality when writing pictures as JPEG files. Legal values are in the range 0.0 to 1.0.

In Igor7 or later, the quality used is 0.9 regardless of what you pass for this parameter.

bitDepth specifies the color depth when writing pictures as JPEG files. Legal values are legal: 1, 8, 16, 24, and 32.

In Igor7 or later, the bit depth used is 32 regardless of what you pass for this parameter.

/I	Saves interactively. A dialog is displayed.
/M=*messageStr*	Specifies prompt message used in save dialog. But see **Prompt Does Not Work on Macintosh** on page IV-137.
/O	Overwrites existing file without asking permission.
/P=*pathName*	Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.
/S=*saveType*	Controls the type of save.

saveType=1:	Normal save (default).
saveType=2:	Save-as.
saveType=3:	Save-a-copy.
saveType=4:	Export as RTF (Rich Text Format).
saveType=5:	Export as HTML (Hypertext Markup Language).
saveType=6:	Export as plain text.
saveType=7:	Export as formatted notebook.

Details

Interactive (/I) means that Igor displays the Save, Save As, or Save a Copy dialog.

The save will be interactive under the following conditions:

- You include the /I flag and the *saveType* is 2, 3, 4, 5, 6, or 7.
- *saveType* is 2, 3, 4, 5, 6, or 7 and you do not specify the path or filename.

If the *saveType* is normal and the notebook has previously been saved to a file then the /I flag, the path and file name that you specify, if any, are ignored and the notebook is saved to its associated file without user intervention.

The full path to the saved file is stored in the string S_path. If the save was unsuccessful, S_path will be `""`.

If you use /P=*pathName*, note that it is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-21 for details.

Exporting as RTF

For background information on writing RTF files, see **Import and Export Via Rich Text Format Files** on page III-20.

Exporting as HTML

For background information on writing HTML files, see **Exporting a Notebook as HTML** on page III-21.

You can pass "UTF-8" or "UTF-2" for the *encodingName* parameter. In virtually all cases, you should use "UTF-8".

When creating an HTML file, SaveNotebook can write pictures using the PNG or JPEG graphics formats. PNG is recommended because it is lossless.

See Also

Chapter III-1, **Notebooks**.

Setting Bit Parameters on page IV-12 for further details about bit settings.

SavePackagePreferences

`SavePackagePreferences [/FLSH=`*`flush`*` /KILL /P=`*`pathName`*`]` ***`packageName`*`,`**
   `prefsFileName`*`, `*`recordID`*`, `*`prefsStruct`

The SavePackagePreferences operation saves preference data in the specified structure so that it can be accessed later via the **LoadPackagePreferences** operation.

Note: The package preferences structure must not use fields of type Variable, String, WAVE, NVAR, SVAR or FUNCREF because these fields refer to data that may not exist when LoadPackagePreferences is called.

The structure can use fields of type char, uchar, int16, uint16, int32, uint32, int64, uint64, float and double as well as fixed-size arrays of these types and substructures with fields of these types.

The data is stored in memory and by default flushed to disk when the current experiment is saved or closed and when Igor quits.

If the /P flag is present then the location on disk of the preference file is determined by *pathName* and *prefsFileName*. However in the usual case the /P flag will be omitted and the preference file is located in a file named *prefsFileName* in a directory named *packageName* in the Packages directory in Igor's preferences directory.

Note: You must choose a very distinctive name for *packageName* as this is the only thing preventing collisions between your package and someone else's package.

See **Saving Package Preferences** on page IV-237 for background information and examples.

Parameters

packageName is the name of your package of Igor procedures. It is limited to 31 bytes and must be a legal name for a directory on disk. This name must be very distinctive as this is the only thing preventing collisions between your package and someone else's package.

prefsFileName is the name of a preference file to be saved by SavePackagePreferences. It should include an extension, typically ".bin".

prefsStruct is the structure containing the data to be saved in the preference file on disk.

recordID is a unique positive integer that you assign to each record that you store in the preferences file. If you store more than one structure in the file, you would use distinct *recordID*s to identify which structure you want to save. In the simple case you will store just one structure in the preference file and you can use 0 (or any positive integer of your choice) as the *recordID*.

Flags

/FLSH=*flush*	Controls when the data is actually written to the preference file: *flush*=0: The data will be flushed to disk when the current experiment is saved, reverted or closed or when Igor quits. This is the default behavior used when /FLSH is omitted and is recommended for most purposes. *flush*=1: The data is flushed to disk immediately.
/KILL	Instead of saving prefsStruct under the specified record ID, that record is deleted from the package's preference if it exists. If it does not exist, nothing is done and no error is returned.
/P=*pathName*	Specifies the directory in which to save the file specified by *prefsFileName*. *pathName* is the name of an existing symbolic path. See **Symbolic Paths** on page II-21 for details. `/P=$<empty string variable>` acts as if the /P flag were omitted.

Details

SavePackagePreferences sets the following output variables:

`V_flag`	Set to 0 if preferences were successfully saved or to a nonzero error code if they were not saved. The latter case is unlikely and would indicate some kind of corruption such as if Igor's preferences directory were deleted.
`V_structSize`	Set to the size in bytes of prefsStruct. This may be useful in handling structure version changes.

Example

See the example under **Saving Package Preferences in a Special-Format Binary File** on page IV-237.

See Also

LoadPackagePreferences.

SavePICT

`SavePICT [`*`flags`*`] [as `*`fileNameStr`*`]`

The SavePICT operation creates a picture file representing the top graph, table or layout. The picture file can be opened by many word processing, drawing, and page layout programs.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

If you omit *fileNameStr* but include /P=*pathName*, SavePICT writes the file using a default file name. The default file name is the window name followed by an extension, such as ".png", ".emf" or ".svg", that depends on the graphic format being exported.

If you specify the file name as "`Clipboard`", and do *not* specify a /P=*pathName*, Igor copies the picture to the Clipboard, rather than to a file. EPS is a file-only format and can not be stored in the clipboard.

If you specify the file name as "`_string_`" the output will be saved into a string variable named S_Value, which is used with the **ListBox** binary bitmap display mode.

If you use the special name _PictGallery_ with the /P flag, then the picture will be stored in Igor's picture gallery (see **Pictures** on page III-448) with the name you provide via *fileNameStr*. This feature was added in support of making movies using the /PICT flag with **NewMovie**.

Flags

/B=*dpi* Controls image resolution in dots-per-inch (*dpi*). The legal values for *dpi* are *n**72 where *n* can be from 1 to 8. The actual image *dpi* is not used. Igor calculates *n* from your value of *dpi* and then multiplies *n* by your computer's screen resolution. This is because bitmap images that are not an integer multiple of the screen resolution look quite bad.

Also see the /RES flag.

/C=*c* Specifies color mode.

c=0:	Black and white.
c=1:	RGB color (default).
c=2:	CMYK color (EPS and native TIFF only).

/D=*d* Obsolete in Igor Pro 7 or later.

/E=*e* Sets graphics format used when exporting a graphic. See **Details** for formats. See also Chapter III-5, **Exporting Graphics (Macintosh)**, or Chapter III-6, **Exporting Graphics (Windows)**, for a description of these modes and when to use them.

/EF= *e* Sets font embedding.

e=0:	No font embedding. Not honored in Igor Pro 7 or later.
e=1:	Embed nonstandard fonts.
e=2:	Embed all fonts.

/I Specifies that /W coordinates are inches.

/M Specifies that /W coordinates are centimeters.

/N=*winSpec* /N is antiquated but still supported. Use /WIN instead.

/O Overwrites file if it exists.

/P=*pathName* Saves file into a folder specified by *pathName*, which is the name of an existing symbolic path.

/PGR=(*firstPage*, *lastPage*)

Controls which pages in a multi-page layout are saved.

firstPage and lastPage are one-based page numbers. All pages from *firstPage* to *lastPage* are saved if the file format supports it.

The special value 0 refers to the current page and -1 refers to the last page in the layout.

Currently only the PDF formats support saving multiple pages. Other file formats save only *firstPage* and ignore the value of *lastPage*.

/PGR was added in Igor Pro 7.00.

/PICT=*pict* Saves specified named picture rather than the target window. Native format of the picture is used and all format flags are ignored.

/PLL=*p* Specifies Postscript language level when used in conjunction with EPS export.

p=1:	For very old Postscript printers.
p=2:	For all other uses (default).

/Q=*q* Sets quality factor (0.0 is lowest, 1.0 is highest). Default is dependent on individual format. Used only by lossy formats such as JPEG.

/R=*resID*	Obsolete in Igor Pro 7 or later.
/RES=*dpi*	Controls the resolution of image formats in dots-per-inch. Unlike the similar /B flag, the value for /RES is the actual output resolution and is useful when your publisher demands a specific resolution.
/S	Suppresses the preview that is normally included with an EPS file. Obsolete in Igor Pro 7 or later.
/SNAP=*s*	Saves a snapshot (screen dump) of a graph or panel window. *s*=1: Include all controls in capture. *s*=2: Capture only window data content. Snapshot mode is available only for graphs and panels and only for bitmap export formats PNG, JPEG, and TIFF at screen resolution. When using /W to specify the size of a graph, the capture is sized to fit within the specified rectangle while maintaining the window aspect ratio. Coordinates used with /W are in pixels.
/T=*t*	Obsolete QuickTime export type. Not supported in Igor Pro 7 or later.
/TRAN[=1 *or* 0]	Makes white background areas transparent using an RGBA type PNG when used with native PNG export of graphs or page layouts.
/W=(*left*,*top*,*right*,*bottom*)	Specifies the size of the picture when exporting a graph. If /W is omitted, it uses the graph window size. When exporting a page layout, specifies the part of the page to export. Only objects that fall completely within the specified area are exported. If /W is omitted, the area of the layout containing objects is exported. When exporting a page layout in Igor Pro 7.00 or later, you can specify /W=(0,0,0,0) to use the full page size. Coordinates for /W are in points unless /I or /M are specified before /W.
/WIN=*winSpec*	Saves the named window or subwindow. *winSpec* can be just a window name, or a window name following by a "#" character and the name of the subwindow, as in `/WIN=Panel0#G0`.
/Z	Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SavePICT sets the variable V_flag to 0 if the operation succeeds or to a nonzero error code if it fails.

If you specify a path using the /P=*pathName* flag, then Igor saves the file in the folder identified by the path. Note that *pathName* is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\\`". See **Symbolic Paths** on page II-21 for details. Otherwise, with no path specified, Igor presents a standard save dialog to let you specify where the file is to be saved.

Graphics formats, specified via /E, are as follows:

/E Value	Macintosh File Format	Windows File Format
-9	SVG file.	SVG file.
-8	PDF file.	PDF file.
-7	TIFF file. Lossless but larger file than PNG; best for text, graph traces, and simple images with sharp edges. The default resolution is 72 dpi. You can specify the resolution with the /B or /RES flag. Cross-platform compatible.	
-6	JPEG file. Lossy compression; best used for grayscale and color images with smooth tones. The /Q flag specifies compression quality and the /B or /RES flag sets the resolution. Cross-platform compatible.	

/E Value	Macintosh File Format	Windows File Format
-5	PNG (Portable Network Graphics) file. Lossless compression; best for text, graph traces, and simple images with sharp edges. The default resolution is 72 dpi. Specify the resolution with /B or /RES. Cross-platform compatible.	
-4	High resolution bitmap PICT file. Default resolution is 288 dpi. Specify the resolution with /B or /RES.	Device-independent bitmap file (DIB). Default resolution is 4x screen resolution. Specify the resolution with /B or /RES.
-3	Encapsulated PostScript (EPS) file. Use /S to suppress the screen preview if exporting to Latex.	Encapsulated PostScript (EPS) file. Use /S to suppress the screen preview if exporting to Latex.
-2	Quartz PDF.	High-resolution Enhanced Metafile (EMF).
-1	Quartz PDF (was PostScript PICT).	Obsolete (was PostScript-enhanced metafile).
0	Quartz PDF (was PostScript PICT with QuickDraw text).	Obsolete (was PostScript-enhanced metafile).
1	Low resolution Quartz PDF at 1x normal size.	High-resolution Enhanced Metafile (EMF).
2	Low resolution Quartz PDF at 2x normal size.	High-resolution Enhanced Metafile (EMF).
4	Low resolution Quartz PDF at 4x normal size.	High-resolution Enhanced Metafile (EMF).
8	Low resolution Quartz PDF at 8x normal size.	High-resolution Enhanced Metafile (EMF).

The low resolution PDF formats on Macintosh are probably not useful and are just placeholders for compatibility with old procedures.

See Also

The **ImageSave** operation for saving waves as PICTs and other image file formats. The **LoadPICT** operation.

See Chapter III-5, **Exporting Graphics (Macintosh)**, or Chapter III-6, **Exporting Graphics (Windows)**, for a description of the /E modes.

SaveTableCopy

`SaveTableCopy [`*`flags`*`][as `*`fileNameStr`*`]`

The SaveTableCopy operation saves a copy of the data displayed in a table on disk. The saved file can be an Igor packed experiment file, a tab-delimited text file, or a comma-separated values text file.

When saving as text, by default the data format matches the format shown in the table. This causes trunctation if the underlying data has more precision than shown in the table. If you specify /F=1, SaveTableCopy uses as many digits as needed to represent the data with full precision.

The point column is never saved.

To save data as text with full precision, use the **Save** operation.

When saving 3D and 4D waves as text, only the visible layer is saved. To save the entirety of a 3D or 4D wave, use the **Save** operation.

Parameters

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If Igor can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/A=*a*	Appends data to the file rather than overwriting. *a*=0: Does not append. *a*=1: Appends to the file with a blank line before the appended data. *a*=2: Appends to the file with no blank line before the appended data. /A applies when saving text files and is ignored when saving packed experiment files. If the file does not exist, a new file is created and /A has no effect.
/F=*f*	Controls the precision of saved numeric data. *f*=0: Numeric data is written exactly as shown in the table. This may cause truncation. This is the default behavior if /F is omitted. *f*=1: Numeric data is written with as many digits as needed to represent the data with full precision. The /F flag was added in Igor Pro 7.00
/I	Presents a dialog from which you can specify file name and folder.
/M=*termStr*	Specifies the terminator character or characters to use at the end of each line of text. The default is `/M="\r"` on Macintosh and `/M="\r\n"` on Windows; it is used when /M is omitted. To use the Unix convention, just a linefeed, specify `/M="\n"`.
/N=*n*	Specifies whether to use column names, titles, or dimension labels. *n* is a bitwise parameter with the bits defined as follows: Bit 0: Include column names or titles. The column title is included if it is not empty. If it is empty, the column name is included. Bit 1: Include horizontal dimension labels if they are showing in the table. The default setting for *n* is 1. All other bits are reserved and must be zero.
/O	Overwrites file if it exists already.
/P=*pathName*	Specifies the folder to store the file in. *pathName* is the name of an existing symbolic path.
/S=*s*	Saves all of the data in the table (*s*=0; default) or the selection only (*s*=1). /S applies when saving text files and is ignored when saving packed experiment files.
/T=*saveType*	Specifies the file format of the saved table. *saveType*=0: Packed experiment file. *saveType*=1: Tab-delimited text file. *saveType*=2: Comma-separated values text file. *saveType*=3: Space-delimited values text file.
/W= *winName*	*winName* is the name of the table to be saved. If /W is omitted or if *winName* is `""`, the top table is saved.
/Z	Errors are not fatal and error dialogs are suppressed. See Details.

Details

The main uses for saving a table as a packed experiment are to save an archival copy of data or to prepare to merge data from multiple experiments (see **Merging Experiments** on page II-19). The resulting experiment file preserves the data folder hierarchy of the waves displayed in the table starting from the "top" data folder, which is the data folder that encloses all waves displayed in the table. The top data folder becomes the root data folder of the resulting experiment file. Only the table and its waves are saved in the packed experiment file, not variables or strings or any other objects in the experiment.

SaveTableCopy does not know about dependencies. If a table contains a wave, wave0, that is dependent on another wave, wave1 which is not in the table, SaveTableCopy will save wave0 but not wave1. When the saved experiment is open, there will be a broken dependency.

The main use for saving as a tab or comma-delimited text file is for exporting data to another program.

When calling SaveTableCopy from a procedure, you should call DoUpdate before calling SaveTable copy. This insures that the table is up-to-date if your procedure has redimensioned or otherwise changed the number of points in the waves in the table.

SaveTableCopy sets the variable V_flag to 0 if the operation completes normally, to -1 if the user cancels, or to another nonzero value that indicates that an error occurred. If you want to detect the user canceling an interactive save, use the /Z flag and check V_flag after calling SaveTableCopy.

The **SaveData** operation also has the ability to save a table to a packed experiment file. SaveData is more complex but a bit more flexible than SaveTableCopy.

Examples

This function saves all tables to a single tab-delimited text file.

```
Function SaveAllTablesToTextFile(pathName, fileName)
   String pathName         // Name of an Igor symbolic path.
   String fileName

   String tableName
   Variable index

   index = 0
   do
      tableName = WinName(index, 2)
      if (strlen(tableName) == 0)
         break
      endif

      SaveTableCopy/P=$pathName/W=$tableName/T=1/A=1 as fileName

      index += 1
   while(1)
End
```

See Also

SaveGraphCopy and **SaveData** operations; **Merging Experiments** on page II-19.

sawtooth

`sawtooth(`*`num`*`)`

The sawtooth function returns ((*num* +n2π) mod 2π)/2π where n is used to correct if *num* is negative. Sawtooth is used to create arbitrary periodic waveforms like sine and cosine.

Examples

```
wave1 = sawtooth(x)
```

creates a sawtooth in wave1 whose Y values range from 0 to 1 as its X values go through 2π units.

```
wave1 = exp(sawtooth(x))
```

creates a series of exponentials in wave1 of amplitude exp(1) and period 2π.

You can also use sawtooth to create periodic repetitions of a given part of a wave:

```
wave1 = wave2(sawtooth(x))
```

creates a periodic repetition of wave2 in wave1 given the correct X scaling for the waves.

ScaleToIndex

`ScaleToIndex(`*`wave`*`, `*`coordValue`*`, `*`dim`*`)`

The ScaleToIndex function returns the number of the element in the requested dimension whose scaled index value is closest to *coordValue*.

The ScaleToIndex function was added in Igor Pro 7.00.

Parameters

dim is a dimension number: 0 for rows, 1 for columns, 2 for layers, 3 for chunks.

coordValue is a scaled index in that dimension.

Details

The ScaleToIndex function returns the value of the expression:

```
round((coordValue - DimOffset(wave,dim)) / DimDelta(wave,dim))
```

With *dim*=0, ScaleToIndex is equivalent to using x2pnt.

See Also

IndexToScale, **x2pnt**, **DimDelta**, **DimOffset**

Waveform Model of Data on page II-57 for an explanation of wave scaling.

ScreenResolution

`ScreenResolution`

The ScreenResolution function returns the logical resolution of your video display screen in dots per inch (dpi). On Macintosh this is always 72. On Windows it is usually 96 (small fonts) or 120 (large fonts).

Examples

```
// 72 is the number of points in an inch which is constant.
Variable pixels = numPoints * (ScreenResolution/72)   // Convert points to pixels
Variable points = numPixels * (72/ScreenResolution)   // Convert pixels to points
```

See Also

PanelResolution

sec

`sec(`*`angle`*`)`

The sec function returns the secant of *angle* which is in radians:

$$\sec(x) = \frac{1}{\cos(x)}.$$

In complex expressions, *angle* is complex, and sec(*angle*) returns a complex value.

See Also

sin, **cos**, **tan**, **csc**, **cot**

sech

`sech(`*`x`*`)`

The sech function returns the hyperbolic secant of *x*.

$$\operatorname{csch}(x) = \frac{1}{\cosh(x)} = \frac{2}{e^{x} + e^{-x}}.$$

In complex expressions, *x* is complex, and sech(*x*) returns a complex value.

See Also

cosh, **tanh**, **coth**, **csch**

Secs2Date

`Secs2Date(`*`seconds`*`, `*`format`*` [, `*`sep`*`])`

The Secs2Date function returns a string containing a date.

With *format* values 0, 1, and 2, the formatting of dates depends on operating system settings entered in the Language & Region control panel (*Macintosh*) or the Region control panel (*Windows*).

If *format* is -1, the format is independent of operating system settings. The fixed-length format is "*day* /*month* /*year* (*dayOfWeekNum*)", where *dayOfWeekNum* is 1 for Sunday, 2 for Monday… and 7 for Saturday.

If format is -2, the format is YYYY-MM-DD.

The optional sep parameter affects format -2 only. If sep is omitted, the separator character is "-". Otherwise, sep specifies the separator character.

Parameters

seconds is the number of seconds from 1/1/1904 to the date to be returned.

format is a number between -2 and 2 which specifies how the date is to be constructed.

Examples

```
Print Secs2Date(DateTime,-2)      // 1993-03-14
Print Secs2Date(DateTime,-2,"/")  // 1993/03/14
Print Secs2Date(DateTime,-1)      // 15/03/1993 (2)
Print Secs2Date(DateTime,0)       // 3/15/93 (depends on system settings)
Print Secs2Date(DateTime,1)       // Monday, March 15, 1993 (depends on system settings)
Print Secs2Date(DateTime,2)       // Mon, Mar 15, 1993 (depends on system settings)
```

See Also

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-78.

The **date**, **date2secs** and **DateTime** functions.

Secs2Time

`Secs2Time(`*`seconds`*`, `*`format`*`, [`*`fracDigits`*`])`

The Secs2Time function returns a string containing a time.

Parameters

seconds is the number of seconds from 1/1/1904 to the time to be returned.

format is a number between 0 and 5 that specifies how the time is to be constructed. It is interpreted as follows:

0:	Normal time, no seconds.
1:	Normal time, with seconds.
2:	Military time, no seconds.
3:	Military time, with seconds and optional fractional seconds.
4:	Elapsed time, no seconds.
5:	Elapsed time, with seconds and optional fractional seconds.

“Normal” formats (0 and 1) follow the preferred formatting of the short time format as set in the International control panel (*Macintosh*) or in the Regional and Language Options control panel (*Windows*).

“Military” means that the hour is a number from 0 to 23. Hours greater than 23 are wrapped.

“Elapsed” means that the hour is a number from -9999 to 9999. The result for hours outside that range is undefined.

The *fracDigits* parameter is optional and specifies the number of digits of fractional seconds. The default value is 0. The *fracDigits* parameter is ignored for format=0, 1, 2,and 4.

Examples

```
Print Secs2Time(DateTime,0)              // prints 1:07 PM
Print Secs2Time(DateTime,1)              // prints 1:07:28 PM
Print Secs2Time(DateTime,2)              // prints 13:07
Print Secs2Time(DateTime,3)              // prints 13:07:29
Print Secs2Time(30*60*60+45*60+55,4)     // Prints 30:45
Print Secs2Time(30*60*60+45*60+55,5)     // Prints 30:45:55
```

See Also

For a discussion of how Igor represents dates, see **Date/Time Waves** on page II-78.

The **Secs2Date**, **date**, **date2secs** and **DateTime** functions. Also, **Operators** on page IV-5 for ?: details.

SelectNumber

`SelectNumber(`*`whichOne`*`, `*`val1`*`, `*`val2`*` [, `*`val3`*`])`

The SelectNumber function returns one of *val1*, *val2*, or (optionally) *val3* based on the value of *whichOne*.

SelectNumber(*whichOne*, *val1*, *val2*) returns *val1* if *whichOne* is zero, else it returns *val2*.

SelectNumber(*whichOne*, *val1*, *val2*, *val3*) returns *val1* if *whichOne* is negative, *val2* if *whichOne* is zero, or *val3* if *whichOne* is positive.

Details

SelectNumber works with complex (or real)*val1*, *val2*, and *val3* when the result is assigned to a complex wave or variable. (Print expects a real result, see the “causes error” example, below).

If *whichOne* is NaN, then NaN is returned.

whichOne must always be a real value.

Unlike the ? : conditional operator, SelectNumber always evaluates all of the numeric expression parameters *val1*, *val2*, …

SelectNumber works in a macro, whereas the conditional operator does not.

Examples

```
Print SelectNumber(0,1,2)                        // prints 1
Print SelectNumber(0,1,2,3)                      // prints 2
wv=SelectNumber(numtype(wv[p])==2,wv[p],0)       // replace NaNs with zeros

// chooses among complex values
Variable/C cx= SelectNumber(negZeroPos,cmplx(-1,-1),0,cmplx(1,1))

// causes error because Print expects a real value (not complex)
Print SelectNumber(negZeroPos,cmplx(-1,-1),0,cmplx(1,1))

// The real function expects a complex result
Print real(SelectNumber(negZeroPos,cmplx(-1,-1),0,cmplx(1,1)))
```

See Also

The **SelectString** and **limit** functions, and **Waveform Arithmetic and Assignments** on page II-69. Also, **Operators** on page IV-5 for details about the ?: operator.

SelectString

`SelectString(`*`whichOne`*`, `*`str1`*`, `*`str2`*` [, `*`str3`*`])`

The SelectString function returns one of *str1*, *str2*, or (optionally) *str3* based on the value of *whichOne*.

SelectString(*whichOne*, *str1*, *str2*) returns *str1* if *whichOne* is zero, else it returns *str2*.

SelectString(*whichOne*, *str1*, *str2*, *str3*) returns *str1* if *whichOne* is negative, *str2* if *whichOne* is zero, or *str3* if *whichOne* is positive.

Details

If *whichOne* is NaN, then `""` is returned.

whichOne must always be a real value.

Unlike the ? : conditional operator, SelectString always evaluates all of the string expression parameters *str1*, *str2*, …

SelectString works in a macro, whereas the conditional operator does not.

Examples

```
Print SelectString(0,"hello","there")                // prints "hello"
Print SelectString(1,"hello","there")                // prints "there"
Print SelectString(-3,"hello","there","jack")        // prints "hello"
Print SelectString(0,"hello","there","jack")         // prints "there"
Print SelectString(100,"hello","there","jack")       // prints "jack"
```

See Also

The **SelectNumber** function and **String Expressions** on page IV-12. Also, **Operators** on page IV-5 for details about the ?: operator.

SetActiveSubwindow

`SetActiveSubwindow` ***`subWinSpec`***

The SetActiveSubwindow operation specifies the subwindow that is to be activated. This operation is mainly for use by recreation macros.

Parameters

subWinSpec specifies an existing subwindow. See **Subwindow Syntax** on page III-87 for details on subwindow specifications.

Use `_endfloat_` for *subWinSpec* to make a newly-created floating panel not be the default target.

See Also

GetWindow with the activeSW keyword.

SetAxis

`SetAxis` `[`***`flags`***`]` ***`axisName`*** `[`**`,`** ***`num1`*****`,`** ***`num2`***`]`

The SetAxis operation sets the extent (or “range”) of the named axis.

Parameters

axisName is usually “left”, “right”, “top” or “bottom”, but it can also be the name of a free axis, such as “vertCrossing”.

If *axisName* is a vertical axis such as “left” or “right” then *num1* sets the bottom end of the axis and *num2* sets the top end of the axis.

If *axisName* is a horizontal axis such as “top” or “bottom” then *num1* sets the left end of the axis and *num2* sets the right end of the axis.

You can flip the graph by reversing *num1* and *num2* (or by using /A/R). This is particularly useful for images, because Igor plots an image inverted.

If you pass * (asterisk) for *num1* and/or *num2* then the corresponding end of the axis will be autoscaled.

Flags

/A[=*a*]	Autoscale axis (when used, *num1*, *num2* should be omitted).	
	a=0:	No autoscale. Same as no /A flag.
	a=1:	Normal autoscale. Same as /A.
	a=2:	Autoscale Y axis to a subset of the data defined by the current X axis range.
/E=*z*	Sets the treatment of zero when the axis is in autoscale mode.	
	z=0:	Normal mode where zero is not treated special.
	z=1:	Forces the smaller end of the axis to be set to zero (autoscale from zero).
	z=2:	Axis is symmetric about zero.
	z=3:	If the data is unipolar (all positive or all negative), this behaves like /E=1 (autoscale from zero). If the data is bipolar, it behaves like /E=0 (normal autoscaling).
/N=*n*	Sets the algorithm for axis autoscaling.	
	n=0:	Normal mode; sets the axis limits equal to the data limits.
	n=1:	Picks nice values for the axis limits.
	n=2:	Picks nice values; also ensures that the data is inset from the axis ends.
/R	Reverses the autoscaled axis (smaller values at the left for horizontal axes, at the top for vertical axes) when used with /A. Although it only has an effect for autoscale, it can be used with nonautoscale version of SetAxis so that the next time the Axis Range tab is used the “reverse axis” checkbox will already be set.	

/W=*winName*	Sets axes in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/Z	No error reporting if named axis doesn't exist in a style macro.

SetBackground

`SetBackground` ***`numericExpression`***

The SetBackground operation sets *numericExpression* as the current unnamed background task.

SetBackground works only with the unnamed background task. New code should used named background tasks instead. See **Background Tasks** on page IV-298 for details.

The background task runs while Igor is not busy with other things. Normally, there won't be a background task. The most common use for the background task is to monitor or drive a continuous data acquisition process.

Parameters

numericExpression is a single precision numeric expression that Igor executes when it isn't doing anything else.

Details

numericExpression is expected to return one of three numeric values:

0:	Background task executed normally.
1:	Background task wants to stop.
2:	Background task encountered error and wants to stop.

Usually the expression will be a call to a user-defined numeric function or external function to drive or monitor data acquisition. The expression should be designed to execute very quickly and it should not present a dialog to the user nor should it create or destroy windows. Generally, it should do nothing more than store data into waves or variables. You can use Igor's dependency mechanism to perform more extensive tasks.

SetBackground designates the background task but you must use CtrlBackground to start it. You can also use KillBackground to stop it. You can not call SetBackground from the background function itself.

See Also

The **BackgroundInfo**, **CtrlBackground**, **CtrlNamedBackground**, **KillBackground**, and **SetProcessSleep** operations, and **Background Tasks** on page IV-298.

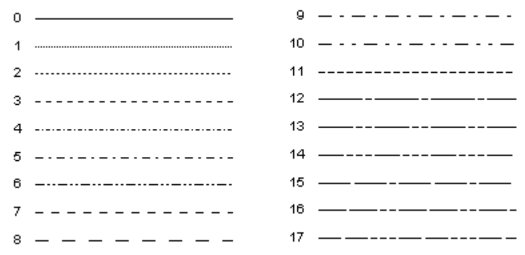
SetDashPattern

`SetDashPattern` ***`dashNumber`*****`, {`*****`d1`*****`, `*****`s1`*****` [, `*****`d2`*****`, `*****`s2`*****`]…}`**

The SetDashPattern operation defines a dashed-line pattern for a user-defined dashed line. These dashed lines are used by the drawing tools and the Modify Waves Appearance dialog, and are elsewhere referred to as "line styles".

Parameters

dashNumber specifies which dash pattern is to be set. It must be between 1 and 17. Dash pattern 0 is reserved for a solid line.



{*d1,s1* [,*d2,s2*]…} defines the dash pattern. The dash pattern consists of 1 to 8 “dash,skip” pairs. Each pair consists of the number of drawn points followed by the number of skipped points.

d1 specifies the number of drawn points and *s1* specifies the number of skipped points in the first “dash,skip” pair. *d2* and *s2* specify the number of drawn and skipped points in the second pair and so on. Each draw or skip value must be between 1 and 127.

Details

SetDashPattern updates *all* graphs, panels and layouts so that any dashed lines will be updated with the new pattern. If you repeatedly call SetDashPattern from within a macro, you should precede the commands with the PauseUpdate operation to prevent multiple updates (which would be slow).

Dashed lines may also be redefined by the Dashed Lines dialog which you can choose from the Misc menu.

The dashed line patterns are saved as part of the experiment. When a new experiment is opened, the preferred dash patterns are restored.

Some programs and printer drivers do not properly render dashed lines with many “dash,skip” pairs.

Examples

```
Make test; Display test
SetDashPattern 17, {20,3,15,8}       // sets last dashed line pattern
ModifyGraph lstyle(test)=17          // apply pattern to trace
```

See Also

PauseUpdate and **ResumeUpdate** operations, and **Dashed Lines** on page III-440.

SetDataFolder

`SetDataFolder` ***`dataFolderSpec`***

The SetDataFolder operation sets the current data folder to the specified data folder.

Parameters

dataFolderSpec can be a simple name (MyDataFolder), a path (root:MyDataFolder) or a string expression containing a name or path. It can also be a data folder reference created by the **DFREF** keyword or returned by **GetDataFolderDFR**.

If *dataFolderSpec* is a path it can be a partial path relative to the current data folder (:MyDataFolder) or an absolute path starting from root (root:MyDataFolder).

Examples

```
SetDataFolder foo              // Sets CDF to foo in the current data folder
SetDataFolder :bar:foo         // Sets CDF to foo in bar current data folder
SetDataFolder root:foo         // Sets CDF to foo in the root data folder
DFREF savedDF= GetDataFolderDFR()    // Remember current data folder
NewDataFolder/O/S root:MyDataFolder  // Set CDF to a new data folder
Variable/G newVariable=1             // Do work in the new data folder
SetDataFolder savedDF                // Restore current data folder
```

See Also

Chapter II-8, **Data Folders** and **Data Folder References** on page IV-72.

SetDimLabel

`SetDimLabel` ***`dimNumber, dimIndex, label, wavelist`***

The SetDimLabel operation sets the dimension label or dimension element label to the specified label.

Parameters

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers and 3 for chunks.

If *dimIndex* is -1, it sets the label for the entire dimension. For dimIndex ≥ 0, it sets the dimension label for that element of the dimension.

label is a name (e.g., time), not a string (e.g., "time").

label is limited to 31 bytes.

Details

Dimension labels can contain up to 31 bytes and may contain spaces and other normally-illegal characters if you surround the name in single quotes or if you use the $ operator to convert a string expression to a name.

Dimension labels have the same characteristics as object names. See **Object Names** on page III-443 for a discussion of object names in general.

See Also

GetDimLabel, **FindDimLabel**

Dimension Labels on page II-85 and **Example: Wave Assignment and Indexing Using Labels** on page II-75 for further usage details and examples.

SetDrawEnv

`SetDrawEnv [/W=`*`winName`*`]` ***`keyword`*** **`[=`*`value`*`][,`** ***`keyword`*** **`[=`*`value`*`]]…`**

The SetDrawEnv operation sets properties of the drawing environment.

If one or more draw objects are selected in the top window then the SetDrawEnv command will apply only to those objects.

If no objects are selected *and* if the keyword save *is not* used *then* the command applies only to the next object drawn.

If no objects are selected *and* if the keyword “save” *is* used *then* the command sets the environment for all following objects.

Each draw layer has its own draw environment settings.

Parameters

SetDrawEnv can accept multiple *keyword*=*value* parameters on one line.

In the following descriptions, (*r*, *g*, *b*) specifies a color. *r*, *g*, and *b* are each a number from 0 to 65535. (0, 0, 0) specifies black. (65535, 65535, 65535) specifies white.

Also note that the abs and rel values for the coordinate keywords “xcoord” and “ycoord” are the literal strings “abs” and “rel”; they are not substitute names for numbers, names, or strings.

arrow=*arr*	Specifies the arrow head position on lines. *arr*=0: No arrowhead (default). *arr*=1: Arrowhead at end. *arr*=2: Arrowhead at start. *arr*=3: Arrowhead at start and end.
arrowfat=*afat*	Sets ratio of arrowhead width to length (default is 0.5).
arrowlen=*alen*	Sets length of arrowhead in points (default is 10).
arrowSharp=*s*	Specifies the continuously variable barb sharpness between -1.0 and 1. 0. *s*=1: No barb; lines only. *s*=0: Blunt (default). *s*=-1: Diamond.
arrowframe=*f*	Specifies the stroke outline thickness of the arrow in points (default is *f*=0 for solid fill).
astyle=s	Specifies which side of the line has barbs relative to a right-facing arrow. *s*=0: None. *s*=1: Top. *s*=2: Bottom. *s*=3: Both (default).
dash=*dsh*	*dsh* is a dash pattern number between 0 and 17 (see **SetDashPattern** for patterns). 0 (solid line) is the default.

fillbgc=(*r*, *g*, *b*)	Specifies fill background color. Default is the window’s background color.
fillfgc=(*r*, *g*, *b*)	Specifies fill foreground color. The default is white.
fillpat=*fpatt*	Specifies fill pattern density. *fpatt*=-1: Erase to background color. *fpatt*=0: No fill. *fpatt*=1: 100% (solid pattern, default). *fpatt*=2: 75% gray. *fpatt*=3: 50% gray. *fpatt*=4: 25% gray.
fname="*fontName*"	Sets font name, default is the default font or the graph font.
fsize=*size*	Sets text size, default is 12 points.
fstyle=*fs*	*fs* is a bitwise parameter with each bit controlling one aspect of the font style as follows: Bit 0: Bold Bit 1: Italic Bit 2: Underline Bit 4: Strikethrough See **Setting Bit Parameters** on page IV-12 for details about bit settings.
gedit= *flag*	Supplies optional edit flag for a group of objects. Use with gstart. *flag*=0: Select entire group, moveable (default). *flag*=1: Individual components editable as if not grouped. Allows objects to be grouped by name but still be editable.
gname= *name*	Supplies optional *name* for an object group. Use with gstart.
gstart	Marks the start of a group of objects.
gstop	Marks the end of a group of objects.
gradient=<*parameters*>	Controls color gradients for drawing element fills. See **Gradient Fills** on page III-441 for details.
gradientExtra=<*parameters*>	Controls color gradient details for drawing element fills. See **Gradient Fills** on page III-441 for details.
linebgc=(*r*, *g*, *b*)	Sets the line background color. Default is window’s background color.
linefgc=(*r*, *g*, *b*)	Sets the line foreground color, default is black.
linepat=*patt*	Specifies the line pattern/density. *patt*=1: 100% (solid pattern, default). *patt*=2: 75% gray. *patt*=3: 50% gray. *patt*=4: 25% gray.
linethick=*thick*	*thick* is a line thickness ≥ 0, default is 1 point.
origin= *x0*,*d0*	Moves coordinate system origin to *x0*,*d0*. Unlike translate, rotate, and scale, this survives a change in coordinate system and is most useful that way. See **Coordinate Transformation**.

pop	Pops a draw environment from the stack. Pops should always match pushes.
push	Pushes the current draw environment onto a stack (limited to 10).
rotate= *deg*	Rotates coordinate system by *deg* degrees. Only makes sense if X and Y coordinate systems are the same. See **Coordinate Transformation**.
rounding=*rnd*	Radius for rounded rectangles in points, default is 10.
rsabout	Redefines coordinate system rotation or scaling to occur at the translation point instead of the current origin. To use, combine rotate or scale with translate and rsabout parameters.
save	Stores the current drawing environment as the default environment.
scale= *sx,sy*	Scales coordinate system by *sx* and *sy*. Affects only coordinates — not line thickness or arrow head sizes. See **Coordinate Transformation**.
textrgb=(*r*, *g*, *b*)	Sets text color, default is black.
textrot=*rot*	Text rotation in degrees. *rot* is a value from -360 to 360. 0 is normal (default) horizontal left-to-right text, 90 is vertical bottom-to-top text, etc.
textxjust=*xj*	Sets horizontal text alignment. *xj*=0: Left aligned text (default). *xj*=1: Center aligned text. *xj*=2: Right aligned text.
textyjust=*yj*	Sets vertical text alignment. *yj*=0: Bottom aligned text (default). *yj*=1: Middle aligned text. *yj*=2: Top aligned text.
translate= *dx,dy*	Shifts coordinate system by *dx* and *dy*. Units are in the current coordinate system. See **Coordinate Transformation**.
xcoord=abs	X coordinates are absolute window coordinates (default for all windows *except graphs* where the default is xcoord=prel). The unit of measurement is pixels if the window is a panel, otherwise they are points. The left edge of the window (or of the printable area in a layout) is at x=0.
xcoord=rel	X coordinates are relative window coordinates. x=0 is at the left edge of the window; x=1 is at the right edge.
xcoord=prel	X coordinates are relative plot rectangle coordinates (graphs only). x=0 is at the left edge of the rectangle; x=1 is at the right edge of the rectangle. This coordinate system ideal for objects that should maintain their size and location relative to the axes, and *is the default for graphs*.
xcoord=*axisName*	X coordinates are in terms of the named axis (graphs only).
ycoord=abs	Y coordinates are absolute window coordinates (default for all windows *except graphs* where the default is ycoord=prel). The unit of measurement is pixels if the window is a panel, otherwise they are points. The top edge of the window (or the of the printable area in a layout) is at y=0.
ycoord=rel	Y coordinates are relative window coordinates. y=0 is at the top edge of the window; y=1 is at the bottom edge.

ycoord=prel	Y coordinates are relative plot rectangle coordinates (graphs only). y=0 is at the top edge of the rectangle; y=1 is at the bottom edge of the rectangle. This coordinate system ideal for objects that should maintain their size and location relative to the axes, and *is the default for graphs*.
ycoord=*axisName*	Y coordinates are in terms of the named axis (graphs only).

Flags

/W=*winName*	Sets the named window or subwindow for drawing. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Coordinate Transformation

The execution order for the translate, rotate, scale, and origin parameters is important. Translation followed by rotation is different than rotation followed by translation. When using multiple keywords in one SetDrawEnv operation, the order in which they are applied is origin, translate, rotate followed by scale regardless of the command order (with the exception of the rsabout parameter). Before using origin with the save keyword, you should use push to save the current draw environment and then use pop after drawing objects using the new origin.

Examples

Following is a simple example of arrow markers:

```
NewPanel
SetDrawEnv arrow= 1,arrowlen= 30,save
SetDrawEnv arrowsharp= 0.3
DrawLine 61,67,177,31
SetDrawEnv arrowsharp= 1
DrawLine 65,95,181,59
SetDrawEnv astyle= 1
DrawLine 69,123,185,87
SetDrawEnv arrowframe= 1
DrawLine 73,151,189,115
```

You can position objects in one coordinate system and then draw them in another with the origin keyword. In the following coordinate transformation example, we position arrows in axis units but size them in absolute units.

```
Make/O jack=sin(x/8)
Display jack
SetDrawEnv xcoord=bottom,ycoord=left,save
SetDrawEnv push
SetDrawEnv origin=50,0
SetDrawEnv xcoord=abs,ycoord=abs,arrow=1,arrowlen=20,arrowsharp=0.2,save
DrawLine 0,0,50,0          // arrow 50 points long pointing to the right
DrawLine 0,0,0,50          // arrow 50 points long pointing down
// now let's move over, rotate a bit and draw the same arrows:
SetDrawEnv translate=100,0
SetDrawEnv rotate=30,save
DrawLine 0,0,50,0
DrawLine 0,0,0,50
SetDrawEnv pop
```

Now try zooming in on the graph. You will see that the first pair of arrows always starts at 50 on the bottom axis and 0 on the left axis whereas the second pair is 100 points to the right of the first.

See Also

Chapter III-3, **Drawing**, and **DrawAction**.

SetDrawLayer

`SetDrawLayer` `[`**`/K/W=`*`winName`***`]`* **`layerName`***

The SetDrawLayer operation makes all future drawing operations use the named layer.

Parameters

Valid *layerName*s for graphs:

ProgBack	UserBack	ProgAxes	UserAxes	ProgFront	UserFront	Overlay

Valid *layerName*s for page layouts:

ProgBack	UserBack	ProgFront	UserFront	Overlay

Valid *layerName*s for control panels:

ProgBack	UserBack	ProgFront	UserFront	Overlay

There are really only three layers for control panels. ProgFront is treated as an alias for ProgBack and UserFront is treated as an alias for UserBack.

Flags

/K	Kills (erases) the given layer.
/W=*winName*	Sets the named window or subwindow for drawing. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The Overlay layer is drawn above all else. It is not included when printing or exporting graphics and is provided for programmers who wish to add user-interface drawing elements without disturbing graphics drawing elements. Overlay was added in Igor Pro 7.00.

The back-to-front order of the layers is shown by the layer pop-up menu obtained by clicking the Layer icon in the drawing palette: . A checkmark indicates the current layer. Non-drawing layers are indicated with gray text.

See Also

Drawing Layers on page III-68 and the **DrawAction** operation.

SetEnvironmentVariable

`SetEnvironmentVariable(`*`varName, varValue`*`)`

The SetEnvironmentVariable function creates an environment variable in Igor's process and sets its value to *varValue*. If a variable named *varName* already exists, its value is set to *varValue*.

The function returns 0 if it succeeds or a nonzero value if it fails.

The SetEnvironmentVariable function was added in Igor Pro 7.00.

Parameters

varName	The name of an environment variable which does not need to actually exist. It must not be an empty string and may not contain an equals sign (=).
varValue	The new contents for the variable. On Windows, if *varValue* is an empty string, the variable is removed. On other platforms, the variable is always set to *varValue*.

Details

The environment of Igor's process is composed of a set of key=value pairs that are known as environment variables. Any child process created by calling ExecuteScriptText inherits the environment variables of Igor's process.

SetEnvironmentVariable changes the environment variables present in Igor's process and any future process created by ExecuteScriptText but does not affect any other processes already created.

On Windows, environment variable names are case-insensitive. On other platforms, they are case-sensitive.

Examples

```
Variable result
result = SetEnvironmentVariable("SOME_VARIABLE", "15")
result = SetEnvironmentVariable("SOME_OTHER_VARIABLE", "string value")
```

See Also

GetEnvironmentVariable, **UnsetEnvironmentVariable**

SetFileFolderInfo

`SetFileFolderInfo [`*`flags`*`] [`*`fileOrFolderNameStr`*`]`

The SetFileFolderInfo operation changes the properties of a file or folder.

Parameters

fileOrFolderNameStr specifies the file or folder to be changed.

If you use a full or partial path for *fileOrFolderNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Folder paths should not end with single Path Separators. See the **MoveFolder Details** section.

If Igor can not determine the location of the file or folder from *fileOrFolderNameStr* and /P=*pathName*, it displays a dialog allowing you to specify the file to be deleted. Use /D to select a folder in this event, otherwise Igor prompts your for a file.

Flags

At least one of the seven following flags is required, or nothing is actually accomplished:

/CDAT=*cdate*	Specifies the number of seconds since midnight January 1, 1904 when the file or folder was first created.
/INV[=*inv*]	Sets the visibility of a file. *inv*=0: File is visible. *inv*=1: Default; file is invisible (*Macintosh*) or Hidden (*Windows*).
/MDAT=*mDate*	Specifies the number of seconds since midnight January 1, 1904 when the file or folder was modified most recently.
/RO[=*ro*]	Sets the read/write state of a file or folder. *ro*=0: File or folder is writable. *ro*=1: File or folder is locked (default).

On Macintosh, locking the file or folder is equivalent to setting the locked property manually using the Get Info window in the Finder.

On Windows, locking the file or folder is equivalent to setting the read-only property manually using the Properties window in Windows Explorer.

If *fileOrFolderNameStr* refers to a file (not a folder), SetFileFolderInfo updates the file properties to reflect values given with the following keywords:

/CRE8=*creatorStr*	Sets the four-character creator code string, such as 'IGR0' (Igor Pro creator code). Ignored on Windows, where files have no “creator code”; instead file extensions are “registered” or “owned” by one, and only one, application. You cannot change that ownership from Igor Pro.
/FTYP=*fTypeStr*	Sets the four-character file type code, such as 'TEXT' or 'IGsU' (packed experiment). Ignored on Windows. Use **MoveFile** to change the file extension.
/STA[=*st*]	Specifies whether the file is a stationery file or not. *st*=1: Stationery file (default). *st*=0: Normal file. Ignored on Windows. Use **MoveFile** to change the file extension.

Optional Flags

/D	Uses the Select Folder dialog rather than Open File dialog when *pathName* and *fileOrFolderNameStr* do not specify an existing file or folder.
/P=*pathName*	Specifies the folder to look in for the file. *pathName* is the name of an existing symbolic path.
/R[=*r*]	Recursively applies change(s) to all files or folders in the folder specified by /P=*pathName* or *fileOrFolderNameStr*, and the folder itself: *r*=0: No recursion. Same as no /R. *r*=1: Recursively apply changes to files. *r*=2: Recursively apply changes to folders, including the folder specified by *pathName* or *fileOrFolderNameStr*. *r*=3: Recursively apply changes to both files and folders (default). /R requires /D and a folder specification.
/Z[=z]	Prevents procedure execution from aborting if SetFileFolderInfo tries to set information about a file or folder that does not exist. Use /Z if you want to handle this case in your procedures rather than having execution abort. /Z=0: Same as no /Z at all. /Z=1: Used for setting information for a file or folder only if it exists. /Z alone has the same effect as /Z=1. /Z=2: Used for setting information for a file or folder if it exists and displaying a dialog if it does not exist.

Variables

SetFileFolderInfo returns information about the file or folder in the following variables:

`V_flag`	0:	File or folder was found.
	-1:	User cancelled the Open File dialog.
	>0:	An error occurred, such as the specified file or folder does not exist.
`S_path`		File system path of the selected file or folder.

Examples

Change the file creator code; no complaint if it doesn't exist:

```
SetFileFolderInfo/Z /CRE8="CWIE", "Macintosh HD:folder:afile.txt"
```

Set the file modification date:

```
Variable mDate= Date2Secs(2000,12,25) + hrs*3600+mins*60+secs
SetFileFolderInfo/P=myPath/MDAT=(mDate), "afile.txt"
```

Remove read-only property from a folder and everything within it:

```
SetFileFolderInfo/P=myPath/D/R/RO=0
```

See Also

The **GetFileFolderInfo**, **MoveFile**, and **FStatus** operations. The **IndexedFile**, **date2secs**, and **ParseFilePath** functions.

SetFormula

`SetFormula` ***`waveName, expressionStr`***

`SetFormula` ***`variableName, expressionStr`***

The SetFormula operation binds the named wave, numeric or string variable to the expression or, if the expression is `""`, unbinds it from any previous expression. In user functions, SetFormula must be used to create dependencies.

Parameters

expressionStr is a string containing a numeric or string expression, depending on the type of the bound object.

Pass an empty string (`""`) for *expressionStr* to clear any previous dependency expression associated with the wave or variable.

Details

The dependent object (the wave or variable) will depend on the objects referenced in the string expression. The expression will be reevaluated any time an object referred to in the expression is modified.

Besides being set from a string expression this differs from just typing:

`name` `:=` *`expression`*

in that syntax errors in *expressionStr* are not reported and are not fatal. You end up with a dependency assignment that is marked as needing to be recompiled. The recompilation will be attempted every time an object is created or when the procedure window is recompiled.

Use the Object Status dialog in the Misc menu to check up on dependent objects.

Examples

This command makes the variable `v_sally` dependent on the user-defined function `anotherFunction`, waves `wave_fred` and `wave_sue`, and the system variable `K2`:

```
SetFormula v_sally, "anotherFunction(wave_fred[1]) + wave_sue[0] + K2"
```

This is equivalent to:

```
v_sally := anotherFunction(wave_fred[1]) + wave_sue[0] + K2
```

except that no error will be generated for the SetFormula if, for instance, wave_fred does not exist.

A string variable dependency can be created by a command such as:

```
SetFormula myStringVar, "note(wave_joe)"
```

observe that *expressionStr* is a string *containing* a string expression, and that:

```
SetFormula myStringVar,note(wave_joe)
```

is not the same thing. In this case the note of wave_joe would contain the expression that myStringVar would depend on! Also, wave_joe would have to exist for Igor to understand the statement.

See Also

Chapter IV-9, **Dependencies**, and the **GetFormula** function.

SetIgorHook

`SetIgorHook [/K/L] [`*`hookType`*` = [`*`procName`*`]]`

The SetIgorHook operation tells Igor to call a user-defined "hook" function at the following times:

- After procedures have been successfully compiled (**AfterCompiledHook**)
- After a file is opened (**AfterFileOpenHook**)
- After the MDI frame window is resized on Windows (**AfterMDIFrameSizedHook**)
- After a window is created (**AfterWindowCreatedHook**)
- Before the debugger is opened (**BeforeDebuggerOpensHook**)
- Before an experiment is saved (**BeforeExperimentSaveHook**)
- Before a file is opened (**BeforeFileOpenHook**)
- Before a new experiment is opened (**IgorBeforeNewHook**)
- Before Igor quits (**IgorBeforeQuitHook**)
- When a menu item is selected (**IgorMenuHook**)
- During Igor's quit processing (**IgorQuitHook**)
- When Igor starts or a new experiment is created (**IgorStartOrNewHook**)

The term "hook" is used as in the phrase "to hook into", meaning to intercept or to attach.

Hook functions are typically used by a sophisticated procedure package to make sure that the package's private data is consistent.

In addition to using SetIgorHook, you can designate hook functions using fixed function names (see **User-Defined Hook Functions** on page IV-264). The advantage of using SetIgorHook over fixed hook names is that you don't have to worry about name conflicts.

You can designate hook functions for specific windows using window hooks (see **SetWindow** on page V-739).

Flags

/K	Removes *procName* from the list of functions called for the *hookType* events. If *procName* is not specified all *hookType* functions are removed. If *hookType* is not specified all functions are removed for all *hookType* events, returning Igor to the pre-SetIgorHook state.
/L	Executes *procName* last. Without /L, a newly added hook function runs before previously registered hook functions. A function that has been previously registered with SetIgorHook can be moved from being called first to being called last by calling SetIgorHook again with /L. To move a function from being called last to being called first requires removing the hook function with /K and then calling SetIgorHook without /L.

Parameters

hookType	Specifies one of the fixed-name hook function names: **AfterCompiledHook** **AfterFileOpenHook** **AfterMDIFrameSizedHook** **AfterWindowCreatedHook** **BeforeDebuggerOpensHook** **BeforeExperimentSaveHook** **BeforeFileOpenHook** **IgorBeforeNewHook** **IgorBeforeQuitHook**

IgorMenuHook

IgorQuitHook

IgorStartOrNewHook

See the note below about these *hookType* names.

hookType is required except with /K.

procName	Names the user-defined hook function that is called for the *hookType* event.

Details

The parameters and return type of the user-defined function *procName* varies depending on the *hookType* it is registered for.

For example, a function registered for the AfterFileOpenHook type must have the same parameters and return type as the shown for the **AfterFileOpenHook** on page IV-266.

The *procName* function is called *after* any window-specific hook for these *hookType*s, and the *procName* function is called *before* any other hook functions previously registered by calling SetIgorHook *unless the /L flag is given*, in which case it still runs after window-specific hook functions, but also *after* all other previously registered hook functions.

The *procName* function should return a nonzero value (1 is typical) to prevent later functions from being called. Returning 0 allows successive functions to be called.

SetIgorHook does not work at Igor start or new experiment time, so SetIgorHook IgorStartOrNewHook is disallowed. Define a global or static fixed-name **IgorStartOrNewHook** function (see page IV-275).

The saved Igor experiment file remembers the SetIgorHooks that are in effect when the experiment is saved:

Hook Function Interactions

After all the SetIgorHook functions registered for *hookType* have run (and all have returned 0), any static fixed-name hook functions are called and then the (only) fixed-name user-defined hook function, if any, is called. As an example, when a menu event occurs, Igor handles the event by calling routines in this order:

1. The top window's hook function as set by **SetWindow**
2. Any SetIgorHook-registered hook functions
3. Any static fixed-named IgorMenuHook functions (in any independent module)
4. The one-and-only non-static fixed-named IgorMenuHook function (in only the ProcGlobal independent module)

1. SetWindow event	**2. SetIgorHook *hookType***	**3. User-defined Hook Function(s)**
(called first)	**(called second)**	**(called last)**
enableMenu	IgorMenuHook	IgorMenuHook
menu	IgorMenuHook	IgorMenuHook

Note: Although you can technically use one of the fixed-name functions, as described in **User-Defined Hook Functions** on page IV-264, for *procName*, the result would be that the function will be called twice: once as a registered named hook function and once as the fixed-named hook function. That is, don't use SetIgorHook this way:

```
SetIgorHook AfterFileOpenHook=AfterFileOpenHook // NO
```

Variables

SetIgorHook returns information in the following variables:

`S_info`	Semicolon-separated list of all current hook functions associated with *hookType*, listed in the order in which they are called. S_info includes the full independent module paths (e.g.,"ProcGlobal#MyMenuHook;MyIM#MyModule#MyMenuHook;").

Examples

This hook function invokes the Export Graphics menu item when Command-C (*Macintosh*) or Ctrl+C (*Windows*) is selected for a graph, preventing the usual Copy.

```
SetIgorHook IgorMenuHook=CopyIsExportHook
```

```
Function CopyIsExportHook(isSelection,menuName,itemName,itemNo,win,wType)
    Variable isSelection
    String menuName,itemName
    Variable itemNo
    String win
    Variable wType

    Variable handledIt= 0
    if( isSelection && wType==1 ) // menu was selected, window is graph
        if( Cmpstr(menuName,"Edit")==0 && CmpStr(itemName,"Copy")==0 )
            DoIgorMenu "Edit", "Export Graphics"        // dialog instead
            handledIt= 1            // don't call other IgorMenuHook functions.
        endif
    endif
    return handledIt
End
```

To unregister CopyIsExportHook as a hook procedure:

```
SetIgorHook/K IgorMenuHook=CopyIsExportHook // unregister CopyIsExportHook
```

To discover which functions are associated with a *hookType*, use a command such as:

```
SetIgorHook IgorMenuHook   // inquire about names registered for IgorMenuHook
Print S_info               // list of functions
```

To remove (or “unregister”) named hooks:

```
SetIgorHook/K                  // removes all hook functions for all hookTypes
SetIgorHook/K IgorMenuHook     // removes all IgorMenuHook functions
SetIgorHook/K IgorMenuHook=CopyIsExportHook// removes only this hook function
```

See Also

The **SetWindow** operation and **User-Defined Hook Functions** on page IV-264.

Independent Modules on page IV-224.

SetIgorMenuMode

SetIgorMenuMode MenuNameStr, MenuItemStr, Action

The SetIgorMenuMode operation allows an Igor programmer to disable or enable Igor’s built-in menus and menu items. This is useful for building applications that will be used by end-users who shouldn’t have access to all Igor’s extensive and confusing functionality.

Parameters

MenuNameStr	The name of an Igor menu, like “File”, “Graph”, or “Load Waves”.
MenuItemStr	The text of an Igor menu item, like “Copy” (in the Edit menu) or “New Graph” (in the Windows menu). For menu items in submenus, such as the “Load Waves” submenu in the “Data” menu, *MenuItemStr* is the name of the submenu.
Action	One of DisableItem, EnableItem, DisableAllItems, or EnableAllItems. DisableItem and EnableItem disable or enable just the single item named by *MenuNameStr* and *MenuItemStr*. If *MenuItemStr* is `""`, then the menu itself is disabled. DisableAllItems and EnableAllItems disable and enable all the items in the menu named by *MenuNameStr*.

Details

All menu names and menu item text are in English. This ensures that code developed for a localized version of Igor will run on all versions. Note that no trailing “...” is used in *MenuItemStr*.

The SetIgorMenuModeProc.ipf procedure file includes procedures and commands that disable or enable every menu and item possible. It is in your Igor Pro 7 folder, in WaveMetrics Procedures:Utilities. It is not intended to be used as-is. You should make a copy and edit the copy to include just the parts you need.

The text of some items in the File menu changes depending on the type of the active window. In these cases you must pass generic text as the *MenuItemStr* parameter. Use "Save Window", "Save Window As", "Save Window Copy", "Adopt Window" and "Revert Window" instead of "Save Notebook" or "Save Procedure", etc. Use "Page Setup" instead of "Page Setup For All Graphs", etc. Use "Print" instead of "Print Graph", etc.

The Edit→Insert File menu item was previously named Insert Text. For compatibility reasons, you can specify either "Insert File" or "Insert Text" as MenuItemStr to modify this item.

See Also

The **DoIgorMenu** operation.

SetIgorOption

`SetIgorOption` `[`***`mainKeyword,`***`]` ***`keyword= value`***

`SetIgorOption` `[`***`mainKeyword,`***`]` ***`keyword= ?`***

The SetIgorOption operation makes unusual and temporary changes to Igor Pro's behavior. This operation is not compilable and you will need to use the **Execute** operation to use it in a user function. The details of the syntax depend on the application and are documented where the alternate behaviors are described. In most cases the current value of a setting can be read using the *keyword*=? syntax. Simple numeric options are stored in V_flag and color options are stored in V_Red, V_Green, V_Blue, and V_Alpha The settings last for the life of the Igor session.

See Also

Syntax Coloring on page III-359 for some usage examples; **SetIgorOption IndependentModuleDev=1** on page IV-225; **Conditional Compilation** on page IV-100; **MarkPerfTestTime** operation.

SetMarquee

`SetMarquee` `[`**`/W=`*****`winName`***`]` ***`left, top, right, bottom`***

The SetMarquee operation creates a marquee on the target graph or layout window or the specified window or subwindow.

The *left*, *top*, *right*, *bottom* coordinates are the same as those returned by the GetMarquee operation (screen units measured in points).

If the coordinates are all 0, the marquee, if it exists, is killed.

The optional axis modes supported by GetMarquee are not supported by SetMarquee.

Flags

/W=*winName*	Specifies the named window or subwindow. When omitted, action will affect the active window or subwindow. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

See Also

The **GetMarquee** operation.

SetProcessSleep

`SetProcessSleep` ***`sleepTicks`***

The SetProcessSleep operation is obsolete and does nothing as of Igor Pro 7.00. It is documented here in case you come across it in old Igor procedure code. Do not use it in new code.

The SetProcessSleep operation determines how much time Igor will give to background tasks or other Macintosh applications executing in the background. This operation does nothing on Windows.

Parameters

sleepTicks is the amount of time given to background tasks in sixtieths of a second. *sleepTicks* values between 0 and 60 are valid.

Details

Igor starts up with sleepTicks = 1. Use 0 to give Igor maximum time, use a larger number to give other applications more time.

Background tasks are used mainly by data acquisition programs.

See Also

Background Tasks on page IV-298 and the **SetBackground** operation.

SetRandomSeed

`SetRandomSeed` ***`seed`***

The SetRandomSeed operation seeds the random number generator used for the **enoise** and **gnoise** functions. Use SetRandomSeed if you need "random" numbers that are reproducible. If you don't use SetRandomSeed, the random number generator is initialized using the system clock when Igor starts. This almost guarantees that you will never get the same sequence twice unless you use SetRandomSeed.

Flags

/BETR[=better]	If better is absent or non-zero, a better method is used for seeding the Mersenne Twister random number generator.

Parameters

seed should be a number in the interval (0, 1]. For any given *seed*, enoise or gnoise or any of the other random-number generator functions generates a particular sequence of pseudorandom numbers. Calling SetRandomSeed with the same seed restarts and repeats the sequence.

Details

Internally *seed* is scaled to a 32-bit unsigned integer. Consequently, the number of different values for the internally-scaled seed is less than the resolution of the double-precision numbers in the (0, 1] range.

You should use /BETR unless you need consistency with older versions of Igor. /BETR was introduced in Igor Pro 6.20.

The Mersenne Twister random number generator is used for most of Igor's noise functions (and optionally for enoise and gnoise), and internally for operations that need random sequences, such as **StatsSample** or **StatsResample**.

Without the /BETR flag, SetRandomSeed maps *seed* to an internal 16-bit integer for seeding the Mersenne Twister random number generator. With /BETR it maps to an internal 32-bit integer seed. So using /BETR reduces the chance that two values of *seed* will map to the same internal integer seed.

See Also

The **enoise** and **gnoise** functions. **Noise Functions** on page III-344.

SetScale

`SetScale [/I/P]` ***`dim, num1, num2`*** `[,` ***`unitsStr`***`]`***`, waveName`*** `[,` ***`waveName`***`]…`

`SetScale` ***`d, num1, num2`*** `[,` ***`unitsStr`***`],` ***`waveName`*** `[,` ***`waveName`***`]…`

The SetScale operation sets the dimension scaling or the data full scale for the named waves.

Parameters

The first parameter *dim* must be one of the following:

Character	Signifies
d	Data full scale.
t	Scaling of the chunks dimension (t scaling).
x	Scaling of the rows dimension (x scaling).
y	Scaling of the columns dimension (y scaling).
z	Scaling of the layers dimension (z scaling).

If setting the scaling of any dimension (*x, y, z,* or *t*), *num1* is the starting index value — the scaled index for the first point in the dimension. The meaning of *num2* changes depending on the /I and /P flags. If you use /P, then *num2* is the delta value — the difference in the scaled index from one point to the next. If you use /I, *num2* is the "ending value" — the index value for the last element in the dimension. If you use neither flag, *num2* is the "right value" — the index value that the element *after the last element in the dimension* would have.

These three methods are just three different ways to specify the two scaling values, the starting value and the delta value, that are stored for each dimension of each wave.

If setting the data full scale (*d*), then *num1* is the nominal minimum and *num2* is the nominal maximum data value for the waves. The data full scale values are not used. They serve only to document the minimum and maximum values the waves are expected to attain. No flags are used when setting the data full scale.

The *unitsStr* parameter is a string that identifies the natural units for the x, y, z, t, or data values of the named waves. Igor will use this to automatically label graph axes. This string must be one to 49 bytes such as "m" for meters, "g" for grams or "s" for seconds. If the waves have no natural units you can pass `""` for this parameter.

Setting *unitsStr* to "dat" (case-sensitive) tells Igor that the wave is a date/time wave containing data in Igor date/time format (seconds since midnight on January 1, 1904). Date/time waves must be double-precision.

Flags

At most one flag is allowed, and then only if dimension scaling (not data full scale) is being set:

/I	Inclusive scaling. *num2* is the ending index — the index value for the very last element in the dimension.
/P	Per-point scaling. *num2* is the delta index value — the difference in scaled index value from one element to the next.

Details

SetScale will not allow the delta scaling value to be zero. If you execute a SetScale command with a delta value of zero, it will set the delta value to 1.0.

If you do not use the /P flag, SetScale converts *num1* and *num2* into a starting index value and a delta index value. If you call SetScale on a dimension with fewer than two elements, it does this conversion as if the dimension had two elements.

Prior to Igor Pro 3.0, Igor supported only 1D waves. "SetScale x" was used to set the scaling for the rows dimensions and "SetScale y" was used to set the data full scale. With the addition of multidimensional waves, "SetScale y" is now used to set the scaling of the columns dimension and "SetScale d" is used to set the data full scale. For backward compatibility, "SetScale y" on a 1D wave sets the data full scale.

When setting the dimension scaling of a numeric wave, you can omit the *unitsStr* parameter. Igor will set the wave's scaling but not change its units. However, when setting the dimension scaling of a text wave, you must supply a *unitsStr* parameter (use `""` if the wave has no units). If you don't, Igor will think that the text wave is the start of a string expression and will attempt to treat it as the *unitsStr*.

See Also

See Also

CopyScales, **DimDelta**, **DimOffset**, **DimSize**, **WaveUnits**

For an explanation of waves and dimension scaling, see **Changing Dimension and Data Scaling** on page II-63.

For further discussion of how Igor represents dates, see **Date/Time Waves** on page II-78.

SetVariable

`SetVariable [/Z]` ***`ctrlName`*** `[`***`keyword = value`*** `[,` ***`keyword = value`*** `…]]`

The SetVariable operation creates or modifies a SetVariable control in the target window.

A SetVariable control sets the value of a global numeric or string variable or a point in a wave when you type or click in the control. A SetVariable can also hold its own value without the need for a global or wave.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the SetVariable control to be created or changed.

The following keyword=value parameters are supported:

activate — Activates the control and selects the text that sets the value. Use **ControlUpdate** to deactivate the control and deselect the text.

appearance={*kind* [, *platform*]}

Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings.

kind can be one of `default`, `native`, or `os9`.

platform can be one of `Mac`, `Win`, or `All`.

See **Button** and **DefaultGUIControls** for more appearance details.

bodyWidth=*width* — Specifies an explicit size for the body (nontitle) portion of a SetVariable control. By default (bodyWidth=0), the body portion is the amount left over from the specified control width after providing space for the current text of the title portion. If the font, font size or text of the title changes, then the body portion may grow or shrink. If you supply a bodyWidth>0, then the body is fixed at the size you specify regardless of the body text. This makes it easier to keep a set of controls right aligned when experiments are transferred between Macintosh and Windows, or when the default font is changed.

disable=*d* — Sets user editability of the control.

d=0:	Normal.
d=1:	Hide.
d=2:	No user input.

fColor=(*r*,*g*,*b*) — Sets the initial color of the title. *r*, *g*, and *b* range from 0 to 65535. fColor defaults to black (0,0,0). To further change the color of the title text, use escape sequences as described for title=*titleStr*.

focusRing=*fr* — Enables or disables the drawing of a rectangle indicating keyboard focus:

fr=0:	Focus rectangle will not be drawn.
fr=1:	Focus rectangle will be drawn (default).

On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.

font="*fontName*" — Sets the font used to display the value of the variable, e.g., `font="Helvetica"`.

format=*formatStr* — Sets the numeric format of the displayed value, e.g., format="%g". Not used with string variables. Never use leading text or the "%W" formats, because Igor reads the value back without interpreting the units. For a description of *formatStr*, see the **printf** operation.

frame=*f* — Sets the frame for the value readout.

f=0:	Value unframed.
f=1:	Value framed (default).

fsize=*s* — Sets the size of the type used to display the variable’s value.

fstyle=*fs*	*fs* is a bitwise parameter with each bit controlling one aspect of the font style as follows: Bit 0: Bold Bit 1: Italic Bit 2: Underline Bit 4: Strikethrough See **Setting Bit Parameters** on page IV-12 for details about bit settings.
help={*helpStr*}	Sets the help for the control. The help text is limited to a total of 255 bytes. You can insert a line break by putting "\r" in a quoted string.
labelBack=(*r*,*g*,*b*) or 0	Specifies the background fill color for labels. *r*, *g*, and *b* are integers from 0 to 65535. The default is 0, which uses the window's background color.
limits={*low*,*high*,*inc*}	Sets the limits of the allowable values (*low* and *high*) for the variable. *inc* sets the amount by which the variable is incremented if you click the control's up/down arrows. This applies to numeric variables, not to string variables. If *inc* is zero then the up/down arrows will not be drawn.
live=*l*	Determines when the readout is updated. *l*=0: Update only after variable changes (default). *l*=1: Update as variable changes.
noedit=*val*	noedit=1 prevents the user from clicking (or tabbing into) a SetVariable control to directly edit its value. This is useful when you want to make a string read-only or when you want to restrict a numeric setting to those available only via the control's up or down arrow buttons. noedit=0 reactivates user editing. noedit=2 is deprecated as of Igor Pro 6.34 but still supported. It allows the use of formatting escape codes described under **Annotation Escape Codes**. Use styledText=1, instead.
noproc	No procedure is to execute when the control's value is changed.
pos={*left*,*top*}	Sets the position of the control in pixels.
pos+={*dx*,*dy*}	Offsets the position of the control in pixels.
proc=*procName*	Sets the procedure to execute when the control's value is changed.
rename=*newName*	Gives control a new name.
styledText=val	styledText=1 allows the use of formatting escape codes described under **Annotation Escape Codes** on page III-53. This works for string SetVariable controls only, not for numeric controls. For example: `SetVariable sv0 value=_STR:"\\JC\\K(65535,0,0)Centered Red Text"` styledText=0 treats escape codes as plain text. The styledText keyword was added in Igor Pro 6.34. For compatibility with earlier versions of Igor, the combination of noedit=1 and styledText=1 is recorded as noedit=2 in recreation macros.
size={*width*,*height*}	Sets width of control in pixels. *height* is ignored.

title=*titleStr*	Sets the title of the control to the specified string expression. The title is displayed to the left of the control. If *titleStr* is empty (`""`), the name of the controlled variable is displayed as the title. Use `title=" "` (put a space within the quotation marks) to create a “blank” title. Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.
userdata(*UDName*)=*UDStr*	Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.
userdata(*UDName*)+=*UDStr*	Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.
value=*varOrWaveName*	Sets the numeric or string variable or wave element to be controlled. If *varOrWaveName* references a wave, the point is specified using standard bracket notation with either a numeric point number or a row label, for example: `value=awave[4]` or `value=awave[%alabel]`. You may also use a 2D, 3D, or 4D wave and specify a column, layer, and chunk index or dimension label in addition to the row index. You can have the control store the value internally rather than in a global variable. In place of varName, use _STR:str or _NUM:num. For example: `NewPanel; SetVariable sv1,value=_NUM:123`
valueColor=(*r*,*g*,*b*)	Sets the color of the value text. *r*, *g*, and *b* range from 0 to 65535. valueColor defaults to black (0,0,0).
valueBackColor=(*r*,*g*,*b*)	Sets the background color under the value text. *r*, *g*, and *b* range from 0 to 65535.
valueBackColor=0	Sets the background color under the value text to the default color, the standard document background color used on the current operating system, which is usually white.
win=*winName*	Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Flags

/Z	No error reporting.

Details

The target window must be a graph or panel.

SetVariable Action Procedure

The action procedure for a SetVariable control takes a predefined WMSetVariableAction structure as a parameter to the function:

```
Function ActionProcName(SV_Struct) : SetVariableControl
   STRUCT WMSetVariableAction &SV_Struct
   …
   return 0
End
```

The “`: SetVariableControl`” designation tells Igor to include this procedure in the Procedure pop-up menu in the SetVariable Control dialog.

See **WMSetVariableAction** for details on the WMSetVariableAction structure.

Although the return value is not currently used, action procedures should always return zero.

You may see an old format SetVariable action procedure in old code:

```
Function procName(ctrlName,varNum,varStr,varName) : SetVariableControl
   String ctrlName
   Variable varNum      // value of variable as number
   String varStr        // value of variable as string
   String varName       // name of variable
   …
   return 0
End
```

This old format should not be used in new code.

Examples

Executing the commands:

```
Variable/G globalVar=99
SetVariable setvar0 size={120,20}
SetVariable setvar0 font="Helvetica", value=globalVar
```

creates a SetVariable control that displays the value of globalVar.

See Also

The **printf** operation for an explanation of *formatStr*, and **SetVariable** on page III-370.

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

SetVariableControl

`SetVariableControl`

SetVariableControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined SetVariable control. See **Procedure Subtypes** on page IV-193 for details. See **SetVariable** for details on creating a SetVariable control.

SetWaveLock

`SetWaveLock` ***`lockVal, waveList`***

The SetWaveLock operation locks a wave or waves and protects them from modification. Such protection is not absolute, but it should prevent most common attempts to change or kill a wave.

Parameters

lockVal can be 0, to unlock, or 1, to lock the wave(s).

waveList is a list of waves or it can be `allinCDF` to act on all waves in the current data folder.

See Also

WaveInfo to check if a wave is locked.

SetWaveTextEncoding

`SetWaveTextEncoding [`*`flags`*`]` ***`newTextEncoding, elements,`*** **`[`*`wave, wave`*`, ...]`**

The SetWaveTextEncoding operation changes the text encoding of the specified waves and/or the text encoding of all waves in the specified data folder.

Wave text encodings are mostly an issue in dealing with pre-Igor Pro 7 experiments containing non-ASCII text. Most users will have no need to worry about or change them. You should not use this operation unless you have a thorough understanding of text encoding issues or are instructed to use it by someone who has a thorough understanding.

See **Wave Text Encodings** on page III-422 for essential background information.

SetWaveTextEncoding can work on a list of specific waves or on all of the waves in a data folder (/DF flag). When working on a data folder, it can work on just the data folder itself or recursively on sub-data folders as well.

If /CONV is present, SetTextWaveEncoding actually converts the text to a different text encoding. You would use this, for example, to convert text stored as Shift JIS (Japanese non-Unicode) into UTF-8 (Unicode).

If /CONV is omitted, SetTextWaveEncoding merely causes Igor to reinterpret text. You would do this to tell Igor Pro 7 what text encoding is used for a wave created by Igor Pro 6 if Igor Pro 7 gets it wrong.

Conversion does not change the characters that make up text - it merely changes the numeric codes used to represent those characters. Reinterpretation does not change the numeric codes but does change the characters by changing the interpretation of the numeric codes.

The SetWaveTextEncoding operation was added in Igor Pro 6.30. However, prior to Igor Pro 7, it permits reinterpretation only, not conversion. In some cases it may be necessary to fix text encoding issues in Igor Pro 6.3x before opening an experiment in Igor Pro 7.

Parameters

newTextEncoding specifies the text encoding to set the wave element to. See **Text Encoding Names and Codes** on page III-434 for a list of codes.

newTextEncoding can be the special value 255 which marks a text wave's content as really containing binary data, not text. See **Text Waves Containing Binary Data** on page III-424 below for details.

elements is a bitwise parameter that specifies one or more elements of a wave, as follows:

Bit	**Value**	**Meaning**
0	1	Wave name
1	2	Wave units
2	4	Wave note
3	8	Wave dimension labels
4	16	Text wave content

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

wave, *wave*, ... is a list of the targeted waves. The list is optional and typically should be omitted if you use the /DF flag. However, if you specify a data folder via /DF and you also list specific waves, SetWaveTextEncoding works on all waves in the data folder as well as the specifically listed waves.

Flags

/BINA={*markAsBinary*, *diagnosticsFlags*}

If *markAsBinary* is 1, SetWaveTextEncoding marks the content of any text wave as binary if the data contains control characters (codes less than 32) other than carriage return (13), linefeed (10), and tab (9).

If *markAsBinary* is 0, SetWaveTextEncoding acts as if /BINA were omitted.

diagnosticsFlags is optional and defaults to 1. If you omit it you can also omit the braces (/BINA=1).

diagnosticsFlags is a bitwise parameter defined as follows:

Bit 0:	Emit diagnostic message for each wave marked as binary.

All other bits are reserved for future use.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The /BINA=1 flag works with with the content of text waves only. It skips non-text waves. It is also independent of the *elements* parameter. That is, it marks the text wave content and only the text wave content as binary even if *elements* is something other than 16.

If you pass 255 for *newTextEncoding* when using the /BINA flag, this tells SetWaveTextEncoding to stop processing after marking binary waves. No further conversion or reinterpretation is done.

See **Text Waves Containing Binary Data** on page III-424 for further discussion.

/CONV={*errorMode* [, *defaultTextEncoding*, *diagnosticsFlags*]}

Causes the text data to be converted to the specified text encoding. If /CONV is present, SetTextWaveEncoding actually converts the text. If it is omitted, SetTextWaveEncoding merely causes Igor to reinterpret it.

defaultTextEncoding and *diagnosticsFlags* are optional and are further discussed below. If you omit them you can also omit the braces (/CONV=1).

If /CONV is specified and the original text encoding is binary (255) then SetWaveTextEncoding does nothing, because all conversions involving binary are NOPs.

If /CONV is specified and *newTextEncoding* is binary (255) then no conversion is done, because all conversions involving binary are NOPs, but the wave is marked as binary. Thus this amounts to the same thing as reinterpreting the wave's text data as binary.

If /CONV is specified, *defaultTextEncoding* is omitted or is -1, and the original text encoding is unknown (0) then SetWaveTextEncoding does nothing. That is, it does no reinterpretation or conversion.

errorMode determines how SetWaveTextEncoding behaves if the conversion can not be done because the text can not be mapped to the specified text encoding. This will occur if the text contains characters that can not be represented in the specified text encoding or if Igor's notion of the original text encoding is wrong. In the latter case, call SetWaveTextEncoding without /CONV to correct Igor's interpretation of the text and then call SetWaveTextEncoding with /CONV to do the conversion.

errorMode takes one of these values:

1:	Generate error. SetWaveTextEncoding returns an error to Igor.
2:	Use a substitute character for any unmappable characters. The substitute character for most text encodings is either control-Z or a question mark.
3:	Skip unmappable input characters. Any unmappable characters will be missing in the output.
4:	Use escape sequences representing any unmappable characters or invalid source text. If the source text is valid in the source text encoding but can not be represented in the destination text encoding, unmappable characters are replaced with \uXXXX where XXXX specifies the UTF-16 code point of the unmappable character in hexadecimal. If the conversion can not be done because the source text is not valid in the source text encoding, invalid bytes are replaced with \xXX where XX specifies the value of the invalid byte in

defaultTextEncoding is optional. If it is present, not -1, and if the wave text element's original encoding is unknown (0), then the wave text element is treated as if it were the specified *defaultTextEncoding*. This allows you to convert the text of Igor Pro 6 waves that are set to unknown when you know that they are really some other text encoding. For example, if you know a wave's text data text encoding is Shift JIS, you can convert it to UTF-8 in one step, like this:

```
SetWaveTextEncoding /CONV={1,4} 1, 16, textWave0
```

Without the *defaultTextEncoding*, you would have to do two steps - the first to tell Igor what the real text encoding is and the second to do the conversion:

```
// Text encoding is Shift JIS
SetWaveTextEncoding 4, 16, textWave0

// Convert to UTF-8
SetWaveTextEncoding /CONV=1 1, 16, textWave0
```

Passing -1 for *defaultTextEncoding* acts the same as omitting it.

diagnosticsFlags is an optional bitwise parameter defined as follows:

Bit 0:	Emit diagnostic message if text conversion succeeds.
Bit 1:	Emit diagnostic message if text conversion fails.
Bit 2:	Emit diagnostic message if text conversion is skipped.
Bit 3:	Emit summary diagnostic message.

All other bits are reserved for future use.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

diagnosticsFlags defaults to 2 (bits 1 set) if the /DF flag is not present and to 10 (bits 1 and 3 set) if the /DF flag is present.

Text conversion may be skipped because the wave element is marked as unknown text encoding, because it is marked as binary, because of the /ONLY or /SKIP flags, or because *newTextEncoding* is the same as the wave element's original text encoding.

/DF={*dfr*, *recurse*, *excludedDFR*}

dfr is a reference to a data folder. SetWaveTextEncoding operate on all waves in the specified data folder. If *dfr* is null ($"") SetWaveTextEncoding acts as if /DF was omitted.

If *recurse* is 1, SetWaveTextEncoding works recursively on all sub-data folders. Otherwise it affects only the data folder referenced by *dfr*.

excludedDFR is an optional reference to a data folder to be skipped by SetWaveTextEncoding. For example, this command sets the text encoding of text wave data for all waves in all data folders except for root:Packages and its sub-data folders:

```
SetWaveTextEncoding /DF={root:,1,root:Packages} 1, 16
```

If *excludedDFR* is null ($"") SetWaveTextEncoding acts as if *excludedDFR* was omitted and no data folders are excluded.

/ONLY=*targetedTextEncoding*

SetWaveTextEncoding changes only wave elements that are currently set to *targetedTextEncoding*. For example, this command converts the content of all waves that are currently set to unknown text encoding (0) to UTF-8 (1), treating the waves originally marked as unknown as Shift JIS (4) waves:

```
SetWaveTextEncoding /DF={root:,1} /ONLY=0 /CONV={1,4} 1, 16
```

Passing -1 for *targetedTextEncoding* acts as if you omitted /ONLY altogether.

/SKIP=*skipTextEncoding*

SetWaveTextEncoding skips all wave elements that are currently set to *skipTextEncoding*. For example, this command converts the content of all waves to UTF-8 (1) except those that are set to Japanese (4):

```
SetWaveTextEncoding /DF={root:,1} /SKIP=4 /CONV=1 1, 16
```

As explained under /CONV, binary (255) wave elements are always skipped and unknown (0) wave elements are skipped if the /CONV defaultTextEncoding parameter is omitted.

Passing -1 for *skipTextEncoding* acts as if you omitted /SKIP altogether.

/TYPE=*type* SetWaveTextEncoding changes only waves of the specified type.

type is:

1:	Text waves only
2:	Non-text waves only
3:	All waves (default)

/Z[=z]	Prevents procedure execution from aborting if SetWaveTextEncoding generates an error. Use /Z or the equivalent, /Z=1, if you want to handle errors in your procedures rather than having execution abort. /Z does not suppress invalid parameter errors. It suppresses only errors in doing text encoding reinterpretation or conversion.

Details

Because SetWaveTextEncoding is intended for use by experts or by users instructed by experts, it does not respect the lock state of waves. That is, it will change waves even if they are locked using SetWaveLock.

For background information on wave text encodings, see **Wave Text Encodings** on page III-422.

A wave's text wave content text encoding can also be set to the special value 255. This marks a text wave as really containing binary data, not text. See **Text Waves Containing Binary Data** on page III-424 for details.

Using SetWaveTextEncoding

One of the main uses for SetWaveTextEncoding is to set the encoding settings to some value other than 0 (unknown) so that Igor's idea of the text encoding used for the items accurately reflects the actual text encoding. We call this "reinterpreting" the item. It does not change the numeric codes representing the text but rather just changes the setting that controls Igor's idea of how the text is encoded. This does change the meaning of the underlying numeric codes. In other words, it changes the characters represented by the text.

You would do reinterpretation if you load an Igor6 experiment and Igor7 interprets the waves using the wrong text encoding. For example, if you load an experiment that you know uses Shift JIS encoding but Igor interprets it as Windows-1252, you get garbage for Japanese text. Reinterpreting it as Shift JIS fixes this. An example is provided below.

The other use for SetWaveTextEncoding is to actually change the numeric codes representing the text - i.e., to convert the content to a different text encoding. For example, if you have text waves from Igor Pro 6 that are encoded in Japanese (Shift JIS), you may want to convert the text to UTF-8 (a form of Unicode) so that you can combine Japanese and non-Japanese characters or use other features that require Unicode. This also applies to western text containing non-ASCII characters encoded as MacRoman or Windows-1252. Converting changes the underlying numeric codes but does not change the characters represented by the text.

The main use for converting text is to convert Igor Pro 6 waves from whatever encoding they use, which typically will be MacRoman, Windows-1252, or Shift JIS, to UTF-8 (a form of Unicode) which is a more modern representation but is not backward compatible with Igor Pro 6.

If the /CONV flag is omitted SetWaveTextEncoding does reinterpretation. If the /CONV flag is present then SetWaveTextEncoding does text conversion except if the original text encoding is unknown (0) or binary (255) in which case it does nothing.

If a wave has mistakenly been marked as containing binary, use SetTextWaveEncoding without /CONV to set it to the correct text encoding.

If a wave's text encoding is set to unknown (0) but you know that it really contains text in some specific encoding, you can use SetTextWaveEncoding without /CONV to set it to the correct text encoding and then use SetTextWaveEncoding with /CONV to convert it to the desired final text encoding. Alternately you can combine these two steps by using /CONV and providing a value for the optional *defaultTextEncoding* parameter.

Output Variables

The SetWaveTextEncoding operation returns information in the following variables:

`V_numConversionsSucceeded`	Set only when the /CONV flag is used. Zero otherwise. V_numConversionsSucceeded is set to the number of successful text conversions.
`V_numConversionsFailed`	Set only when the /CONV flag is used. Zero otherwise. V_numConversionsFailed is set to the number of unsuccessful text conversions.

`V_numConversionsSkipped`	Set only when the /CONV flag is used. Zero otherwise. V_numConversionsSkipped is set to the number of skipped text conversions. Text conversion may be skipped because the wave element is marked as unknown text encoding, because it is marked as binary, because of the /ONLY or /SKIP flags or because newTextEncoding is the same as the wave element's original text encoding. V_numConversionsSkipped does not count waves skipped because of the /TYPE flag.

Examples

```
// Keep in mind that, if /CONV is present, SetWaveTextEncoding does nothing
// for wave text elements currently set to binary (255).
// Also, if /CONV is present, SetWaveTextEncoding does nothing
// for wave text elements currently set to unknown (0) if the /CONV
// optional defaultTextEncoding parameter is omitted.
// In the following examples 1 means UTF-8, 4 means Shift JIS, and 16
// means text wave content.

// Reinterpret specific text waves as Shift JIS if they are currently set to unknown
SetWaveTextEncoding /ONLY=0 4, 16, textWave0, textWave1

// Convert specific waves' content to UTF-8
SetWaveTextEncoding /CONV=1 1, 16, textWave0, textWave1

// Reinterpret all text waves as Shift JIS if they are currently set to unknown
SetWaveTextEncoding /DF={root:,1} /ONLY=0 4, 16

// Convert all waves' content to UTF-8
SetWaveTextEncoding /DF={root:,1} /CONV=1 1, 16

// Convert all text waves' content from Shift JIS to UTF-8
// if it is currently set to unknown
SetWaveTextEncoding /DF={root:,1} /ONLY=0 /TYPE=1 /CONV={1,4} 1, 16

// Same as before but exclude the root:Packages data folder
SetWaveTextEncoding /DF={root:,1,root:Packages} /ONLY=0 /TYPE=1 /CONV={1,4} 1, 16

// Mark a text wave as really containing binary data
SetWaveTextEncoding 255, 16, textWaveContainingBinaryData

// Convert Chinese, Japanese and Korean text wave data to UTF-8.
// This example illustrates how to do a conversion based on criteria
// that require inspecting each wave.
Function ConvertCJKToUTF8(dfr, recurse)
    DFREF dfr
    Variable recurse

    Variable index = 0
    do
        Wave/Z w = WaveRefIndexedDFR(dfr, index)
        if (!WaveExists(w))
            break
        endif
        if (WaveType(w) == 0)   // Text wave?
            Variable currentEncoding = WaveTextEncoding(w,5)// Wave content current
    encoding
            switch(currentEncoding)
                case 4:         // Japanese (Shift JIS)
                case 5:         // Traditional Chinese (Big5)
                case 6:         // Simplified Chinese (ISO-2022-CN)
                case 7:         // Macintosh Korean (EUC-KR)
                case 8:         // Windows Korean (Windows-949)
                    SetWaveTextEncoding /CONV=1 1, 16, w
                    break
            endswitch
        endif
        index += 1
    while(1)
    if (recurse)
        Variable numChildDataFolders = CountObjectsDFR(dfr, 4)
        Variable i
        for(i=0; i<numChildDataFolders; i+=1)
            String childDFName = GetIndexedObjNameDFR(dfr, 4, i)
            DFREF childDFR = dfr:$childDFName
            ConvertCJKToUTF8(childDFR, 1)
        endfor
    endif

    SetDataFolder saveDFR
End
```

See Also

Text Encodings on page III-409, **Wave Text Encodings** on page III-422, **Text Encoding Names and Codes** on page III-434, **Text Waves Containing Binary Data** on page III-424

WaveTextEncoding, **ConvertTextEncoding**, **ConvertGlobalStringTextEncoding**

SetWindow

`SetWindow` ***`winName`*** **`[, `*`keyword`*` = `*`value`*`]…`**

The SetWindow operation sets the window note and user data for the named window or subwindow. SetWindow can also set hook functions for a base window or exterior subwindow (interior subwindows not supported).

Parameters

winName can be a window or subwindow name. It can also be the keyword `kwTopWin` to specify the topmost graph, panel, layout, table, or notebook window.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

activeChildFrame=*f*	Determines if the frame indicating the active subwindow in the named window or subwindow is drawn.
	f=-1: Default. Recursively check ancestors of this window. If one is found with a value of either 0 or 1, use that value. If the topmost window has a value of -1, then the frame is drawn.
	f=1: If this subwindow or one of its descendants becomes active, then the frame is drawn.
	f=0: If this subwindow or one of its descendants becomes active, then the frame is not drawn.
graphicsTech=*t*	Sets the graphics technology used to draw the window. This flag may be useful in rare cases to work around graphics limitations. See **Graphics Technology** (see page III-445) for background information.
	t=0: Default: The graphics technology specified in Miscellaneous Settings dialog, Miscellaneous category. This is Qt Graphics by default.
	t=1: Native: Core Graphics on Macintosh, GDI+ on Windows.
	t=2: Old: Core Graphics on Macintosh, GDI on Windows.
hide=*h*	Hides or unhides widows or subwindows.
	h=0: Unhides a subwindow or base window.
	h=1: Hides a subwindow or base window.
	h=2: Unhides without restoring minimized windows (*Windows* only).
	When unhiding subwindows, you should combine with needUpdate=1 if conditions require the subwindow to be redrawn since the window was hidden.

hook=*procName* Sets the window hook function that Igor will call when certain events happen. Use `SetWindow hook=$""` to specify no hook function.

See **Unnamed Window Hook Functions** on page IV-286 for further details.

hook(*hName*)=*procName*

Defines a named window hook *hName* and sets the function that Igor will call when certain events happen. *hName* can be any legal name. Named hooks are called before any unnamed hooks.

Use `$""` for *procName* to specify no hook.

See **Named Window Hook Functions** on page IV-277 for further details.

To hook a subwindow, see **Window Hooks and Subwindows** on page IV-277.

hookcursor=*number* Sets the mouse cursor. This keyword is antiquated. See **Setting the Mouse Cursor** on page IV-282 for the preferred technique.

hookevents=*flags* Bitfield of flags to enable certain events for the unnamed hook function:

Bit 0:	Mouse button clicks.
Bit 1:	Mouse moved events.
Bit 2:	Cursor moved events.

To set bit 0 and bit 1 (mouse clicks and mouse moved), use $2^0+2^1 = 1+2 = 3$ for *flags*. Use 7 to also enable cursor moved events. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

This keyword applies to the unnamed hook function only. It does not affect named hook functions which always receive all events.

markerHook= {*hookFuncName*, *start*, *end*}

Specifies a user function and marker number range for custom markers. The marker range can be any positive integers less than 1000 and can overlap built-in marker numbers. See **Custom Marker Hook Functions** on page IV-289 for details.

Use $"" for hookFuncName to specify no hook.

needUpdate= *n* Marks a window as needing an update (*n*=1) or takes no action (*n*=0).

note=*noteStr* Sets the window note to *noteStr*, replacing any existing note.

note+=*noteStr* Appends *noteStr* to current contents of the window note.

sizeLimit= {*minWidth*, *minHeight*, *maxWidth*, *maxHeight*}

Imposes limits on a window's size when resized with the mouse or by calling **MoveWindow**. The units of the limits are the same as those returned by **GetWindow** wsize.

The sizeLimit keyword was added in Igor Pro 7.00.

To allow the window width to grow essentially without bound, pass INF for maxWidth.

To allow the window height to grow essentially without bound, pass INF for maxHeight.

If *minWidth* > *maxWidth* or *minHeight* > *maxHeight*, the maximum dimensions are set to the minimum, effectively fixing the size of the window in that dimension. For control panels, it is better to use `ModifyPanel fixedSize=1`.

Combining these limits with ModifyGraph width and height modes leads to unexpected results and is discouraged.

If you first use sizeLimit and then execute `ModifyPanel fixedSize=1` on the same window, the fixedSize command takes precedence. If you execute `SetWindow sizeLimit` on a control panel that has `fixedSize=1`, Igor generates an error.

Igor includes a `SetWindow sizeLimit` command in a window recreation macro if necessary. Igor6 does not support `SetWindow sizeLimit` so this causes an error when recreating the window in Igor6. If you never set the `sizeLimit` property for a window, or if the minimum dimensions are very small and maximum dimensions are INF, then no SetWindow command is generated.

userdata=*UDStr*
userdata(*UDName*)=*UDStr*

Sets the window or subwindow user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata+=*UDStr*
userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current window or subwindow user data. Use the optional (*UDName*) to append to the named user data.

Details

For details on named window hooks, see **Window Hook Functions** on page IV-276.

Unnamed window hook functions are supported for backward compatibility only. New code should use named window hook functions. For details on unnamed window hooks, see **Unnamed Window Hook Functions** on page IV-286.

For details on marker hooks, see **Custom Marker Hook Functions** on page IV-289.

See Also

The **GetWindow, SetIgorHook**, and **SetIgorMenuMode** operations and **AxisValFromPixel**, **NumberByKey**, **PopupContextualMenu**, and **TraceFromPixel** functions. The **GetUserData** operation for retrieving named user data.

ShowIgorMenus

`ShowIgorMenus [`*`MenuNameStr`*` [, `*`MenuNameStr`*`] …`

The ShowIgorMenus operation shows the named built-in menus or, if none are explicitly named, shows all built-in menus in the menu bar.

User-defined menus attached to built-in menus are also affected by this operation.

Parameters

MenuNameStr	The name of an Igor menu, like “File”, “Data”, or “Graph”.

Details

See **HideIgorMenus** for details.

See Also

Chapter IV-5, **User-Defined Menus**.

The **HideIgorMenus**, **DoIgorMenu**, and **SetIgorMenuMode** operations.

ShowInfo

`ShowInfo [/CP=num /W=winName]`

The ShowInfo operation puts an information panel on the target or named graph. The information panel contains cursors and readouts of values associated with waves in the graph.

Flags

/CP=*num*	Selects a cursor pair to display in the info panel. *num*=0: Selects cursor A and cursor B. *num*=1: Selects cursor C and cursor D. *num*=2: Selects cursor E and cursor F. *num*=3: Selects cursor G and cursor H. *num*=4: Selects cursor I and cursor J.
/CP={*n1*,*n2*,...}	Allows you to select multiple cursor pairs to be displayed in the info panel. The numbers n1, n2, etc., are the same as the single-pair version of this flag. This form of /CP was added in Igor Pro 7.00.
/W=*winName*	Displays info panel in the named window.

See Also

Info Panel and Cursors on page II-248.

The **HideInfo** operation.

Programming With Cursors on page II-249.

ShowTools

`ShowTools [/A/W=winName][toolName]`

The ShowTools operation puts a tool palette for drawing along the left hand side of the target or named graph or control panel, and optionally activates the named tool.

Flags

/A	Sizes window automatically to make extra room for the tool palette. This preserves the proportion and size of the actual graph area.
/W=*winName*	Shows tool palette in the named window. This must be the first flag specified when used in a Proc or Macro or on the command line.

Parameters

If you specify a *toolName* (which can be one of: normal, arrow, text, line, rect, rrect, oval, or poly) the named tool is activated. Specifying the “normal” tool has the same effect as issuing the GraphNormal command for a graph that has the drawing tools selected.

Details

The activated tool is not highlighted until the top graph or control panel becomes the topmost (activated) window. Use `DoWindow/F` to bring a window to the top (or “front”).

See Also

The **DoWindow**, **GraphNormal**, **GraphWaveDraw**, **GraphWaveEdit**, and **HideTools** operations.

SinIntegral

`SinIntegral(z)`

The SinIntegral(z) function returns the sine integral of z.

If z is real, a real value is returned. If z is complex then a complex value is returned.

The SinIntegral function was added in Igor Pro 7.00.

Details

The sine integral is defined by

$$Si(z)=\int_0^z \frac{\sin(t)}{t}dt.$$

IGOR computes the SinIntegral using the expression:

$$Si(z)=z\,{}_1F_2\left(\frac{1}{2};\frac{3}{2},\frac{3}{2};-\frac{z^2}{4}\right).$$

References

Abramowitz, M., and I.A. Stegun, "Handbook of Mathematical Functions", Dover, New York, 1972. Chapter 5.

See Also

CosIntegral, **ExpIntegralE1**, **hyperGPFQ**

sign

`sign(`*`num`*`)`

The sign function returns -1 if *num* is negative or 1 if it is not negative.

Silent

`Silent` ***`num`***

The Silent operation is largely obsolete. Only very specalized uses remain and most users can ignore this operation.

Prior to Igor Pro 7, Silent was used to enable or disable the display of macro commands in the command line as they were executed. It was also used to enable compatibility modes for very old experiments.

Parameters

If *num* is 2, commands issued by AppleEvents or ActiveX Automation are not shown in the history are of the command window. Use 3 to re-enable.

If *num* is 100, 101 or 102, all procedures are recompiled. For 102, the time to recompile is displayed in the history.

sin

`sin(`*`angle`*`)`

The sin function returns the sine of *angle* which is in radians.

In complex expressions, *angle* is complex, and sin(*angle*) returns a complex value:

$$\sin(x+iy)=\sin(x)\cosh(y)+i\cos(x)\sinh(y).$$

See Also

asin, **cos**, **tan**, **sec**, **csc**, **cot**

sinc

`sinc(`*`num`*`)`

The sinc function returns sin(*num*)/*num*. The sinc function returns 1.0 when *num* is zero. *num* must be real.

sinh

`sinh(`*`num`*`)`

The sinh function returns the hyperbolic sine of *num*:

$$\sinh(x) = \frac{e^x - e^{-x}}{2}.$$

In complex expressions, *num* is complex, and sinh(*num*) returns a complex value.

See Also

cosh, **tanh**, **coth**

Sleep

`Sleep [`*`flags`*`] `*`timeSpec`*

The Sleep operation puts Igor to sleep for a while. After the while is up, Igor continues execution.

You could use Sleep, for example, to give an instrument time to perform an action or to allow a user to admire a graph before proceeding.

More advanced programmers may prefer to use a background task as an alternative. See **Background Tasks** on page IV-298.

Parameters

The format of *timeSpec* depends on which flags, if any, are present.

If no flags are present, then *timeSpec* is in *hh*:*mm*:*ss* format and specifies the number of elapsed hours, minutes and seconds to sleep.

Flags

/A	*timeSpec* is an absolute time in 24 hour format (e.g., 16:00:00).
/A/W	Wait until tomorrow if absolute time has passed.
/B	Stop sleeping if the user clicks the mouse button.
/C=*cursor*	Controls what kind of cursor to display during sleep. *cursor*=-1: No cursor change. *cursor*=0: Hour glass (default). *cursor*=1: Arrow. *cursor*=2: “Click”. *cursor*=3: Spinning beachball. *cursor*=4: Watch with spinning hands. *cursor*=5: Jacob’s ladder. *cursor*=6: Displays a progress dialog instead of changing the cursor. Other: Watch. *cursor* values 3 through 6 require Igor Pro 7.00 or later.
/M=*message*	If you use /C=6, the progress dialog displays *message* above the progress bar. By default the message reads "Sleeping".
/Q	Continue executing the procedure containing the Sleep operation even if the **User Abort Key Combinations** were pressed.
/S	*timeSpec* is a numeric expression in seconds.

/T	*timeSpec* is a numeric expression in ticks (about 1/60 of a second).

Details

The Sleep operation does *not* let the user choose menus, move cursors, run procedures, draw in graphs, or do any other interactive task.

Normally *timeSpec* specifies an amount of elapsed time. If the /A flag is present, then *timeSpec* is an absolute time when sleep is to end. If the specified absolute time has already passed, no sleep occurs unless you also use /W, which makes it wait until tomorrow.

If you specify time in hh:mm:ss format, you can also specify the time indirectly through a string variable. See the examples.

You can end sleep by pressing the **User Abort Key Combinations**. Normally when you do this, it aborts any procedure that is running. However, if you use the /Q flag, the procedure continues running normally.

Examples

These examples assume the current time is 4 PM:

```
Sleep 00:01:30              // sleeps for 1 minute, 30 seconds
Sleep/A 23:30:00            // sleeps until 11:30 PM
Sleep/A 03:00:00            // doesn't sleep at all because time is past
Sleep/A/W 03:00:00          // sleeps until 3 AM tomorrow
String str1= "03:00:00"     // put wakeup call time in string
Sleep/A/W $str1             // sleeps until 3 AM tomorrow
Sleep/B/C=2/S/Q 60          // sleep 60 seconds, or until user clicks,
                            //  and keep going (don't abort)
```

The following function creates a graph and then periodically updates the displayed data. By default, it pauses for a number of seconds specified by the interval parameter. The use of /B allows the user to make the function proceed to the next data set without delay.

Because the /Q flag is omitted, pressing the **User Abort Key Combinations** or pressing Igor's Abort button terminates the function instead of merely aborting the current Sleep call.

```
Function AnimatedGraph(Variable interval)
    Make/N=200/O junk
    SetScale/I x 0, 2*pi, junk
    junk=sin(x)
    Display junk
    DoUpdate

    Variable i
    for (i = 0; i < 10; i++)
        Sleep/S/C=2/B interval
        junk = sin(x*(i+2))
        DoUpdate
    endfor
End
```

Slider

Slider [**/Z**] ***controlName*** [***key*** [**=** ***value***]][**,** ***key*** [**=** ***value***]]…

The Slider operation creates or modifies a Slider control in the target window.

A Slider control sets or displays a single numeric value. The user can adjust the value by dragging a thumb along the length of the Slider.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the Slider control to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}

	Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings. *kind* can be one of `default`, `native`, or `os9`. *platform* can be one of `Mac`, `Win`, or `All`. **Note**: The Slider control reverts to `os9` appearance on Macintosh if thumbColor isn't the default blue (0,0,65535). See **Button** and **DefaultGUIControls** for more appearance details.
disable=*d*	Sets user editability of the control. *d*=0: Normal. *d*=1: Hide. *d*=2: Draw in gray state; disable control action.
fColor=(*r*,*g*,*b*)	Sets the color of the tick marks. *r*, *g*, and *b* range from 0 to 65535. fColor defaults to black (0,0,0).
focusRing=*fr*	Enables or disables the drawing of a rectangle indicating keyboard focus: *fr*=0: Focus rectangle will not be drawn. *fr*=1: Focus rectangle will be drawn (default). On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.
font="*fontName* "	Sets the font used to display the tick labels, e.g., `font="Helvetica"`.
fsize=*s*	Sets the size of the type for tick mark labels.
help={*helpStr*}	Sets the help for the control. The help text is limited to a total of 255 bytes. You can insert a line break by putting "\r" in a quoted string.
limits= {*low*,*high*,*inc*}	*low* sets left or bottom value, *high* sets right or top value. Use *inc*=0 for continuous or use desired increment between stops.
live=*l*	Controls updating of readout. *l*=0: Update only after mouse is released. *l*=1: Update as slider moves (default).
noproc	Specifies that no procedure is to execute when the control's value is changed.
pos={*left,top*}	Sets the position of the slider in pixels.
pos+={*dx*,*dy*}	Offsets the position of the slider in pixels.
proc=*procName*	Specifies the procedure to execute when the control's thumb is moved by the user.
rename=*newName*	Gives control a new name.
side=*s*	Controls slider thumb. *s*=0: Thumb is blunt. *s*=1: Thumb points right or down (default). *s*=2: Thumb points up or left.
size={*width,height*}	Sets width or height of control in pixels. *height* is ignored if vert=0 and *width* is ignored if vert=1.
thumbColor=(*r*,*g*,*b*)	Sets dominant foreground color of thumb. *r*, *g*, and *b* are integers from 0 to 65535. Only the hue and saturation are used. Therefore (0,1000,0) is the same tint of green as (0,10000,0).

ticks=*t*	Controls slider ticks. *t*=0: No ticks. *t*=1: Number of ticks is calculated from limits (no ticks drawn if calculated value is less than 2 or greater than 100). Default value. *t*>1: *t* is the number of ticks distributed between the start and stop position. Ticks are labeled using the same automatic algorithm used for graph axes. Use negative tick values to force ticks to not be labeled. Ticks are shown on the side specified by the side keyword and are not drawn if side=0.
tkLblRot= *deg*	Rotates tick labels. *deg* is a value between -360 and 360.
userdata(*UDName*)=*UDStr*	Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.
userdata(*UDName*)+=*UDStr*	Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.
userTicks={*tvWave*,*tlblWave*}	User-defined tick positions and labels. *tvWave* contains the tick positions, and text wave *tlblWave* contains the labels. See ModifyGraph userticks for more info. Overrides normal ticking specified by ticks keyword.
value=*v*	*v* is the new value for the Slider.
valueColor=(*r*,*g*,*b*)	Sets the color of the tick labels. *r*, *g*, and *b* range from 0 to 65535. valueColor defaults to black (0,0,0).
variable= *var*	Sets the variable (*var*) that the slider will update. It is not necessary to connect a Slider to a variable — you can get a Slider's value using the **ControlInfo** operation.
vert=*v*	Set vertical (*v* =1; default) or horizontal (*v* =0) orientation of the slider.
win=*winName*	Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Flags

/Z	No error reporting.

Details

The target window must be a graph or panel.

If you use negative ticks to suppress automatic labeling, you can label tick marks using drawing tools (panels only).

Slider Action Procedure

The action procedure for a Slider control takes a predefined `WMSliderAction` structure as a parameter to the function:

```
Function ActionProcName(S_Struct) : SliderControl
   STRUCT WMSliderAction &S_Struct
   …
   return 0
End
```

The "`: SliderControl`" designation tells Igor to include this procedure in the Procedure pop-up menu in the Slider Control dialog.

See **WMSliderAction** for details on the WMSliderAction structure.

Although the return value is not currently used, action procedures should always return zero.

You may see an old format Slider action procedure in old code:

```
Function MySliderProc(name, value, event) : SliderControl
   String name             // name of this slider control
   Variable value          // value of slider
   Variable event          // bit field:bit 0:value set; 1:mouse down,
                           //  2:mouse up, 3:mouse moved

   return 0                // other return values reserved
End
```

This old format should not be used in new code.

Examples

```
Function SliderExample()
   NewPanel /W=(150,50,501,285)
   Variable/G var1
   Execute "ModifyPanel cbRGB=(56797,56797,56797)"
   SetVariable setvar0,pos={141,18},size={122,17},limits={-Inf,Inf,1},value=var1
   Slider foo,pos={26,31},size={62,143},limits={-5,10,1},variable=var1
   Slider foo2,pos={173,161},size={150,53}
   Slider foo2,limits={-5,10,1},variable=var1,vert=0,thumbColor=(0,1000,0)
   Slider foo3,pos={80,31},size={62,143}
   Slider foo3,limits={-5,10,1},variable=var1,side=2,thumbColor=(1000,1000,0)
   Slider foo4,pos={173,59},size={150,13}
   Slider foo4,limits={-5,10,1},variable=var1,side=0,vert=0
   Slider foo4,thumbColor=(1000,1000,1000)
   Slider foo5,pos={173,90},size={150,53}
   Slider foo5,limits={-5,10,1},variable= var1,side=2,vert=0
   Slider foo5,ticks=5,thumbColor=(500,1000,1000)
End
```

See Also

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

SliderControl

`SliderControl`

SliderControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined slider control. See **Procedure Subtypes** on page IV-193 for details. See **Slider** for details on creating a slider control.

Slow

`Slow` ***`ticks`***

The Slow operation is obsolete. Prior to Igor Pro 7 it slowed down execution of macros for debugging purposes. It now does nothing.

Smooth

`Smooth [`*`flags`*`] `*`num, waveName`*` [, `*`waveName`*`…]`

The Smooth operation smooths the named waves using binomial (Gaussian) smoothing, boxcar (sliding average) smoothing, Savitzky-Golay (polynomial) smoothing, or running-median filtering.

Parameters

num is the number of smoothing operations to be applied for binomial smoothing or the integer number of points in the smoothing window for boxcar, Savitzky-Golay, and running-median smoothing.

Each *waveName* is smoothed in-place, overwriting the values with the smoothed result. *waveName* may be a floating point or integer wave.

If *waveName* complex, the real and imaginary parts are smoothed independently.

If *waveName* contains NaNs, the results are undefined. (The **Loess** and **Interpolate2** operations can fill in NaNs).

Flags

/B [=*b*]	Invokes boxcar smoothing algorithm. If given, *b* specifies the number of passes to use when smoothing the data with smoothing factor *num* (box width). The number of passes can be any value between 1 and 32767.
/DIM=*d*	Specifies the wave dimension to smooth. *d*=-1: Treats entire wave as 1D (default). For *d*=0, 1,…, operates along rows, columns, etc.
/E=*endEffect*	Determines how to handle the ends of the wave (w) when fabricating missing neighbor values. *endEffect*=0: Bounce method (default). Uses w[*i*] in place of the missing w[-*i*] and w[*n*-*i*] in place of the missing w[*n*+*i*]. *endEffect*=1: Wrap method. Uses w[*n*-*i*] in place of the missing w[-*i*] and vice versa. *endEffect*=2: Zero method. Uses 0 for any missing value. *endEffect*=3: Repeat method. Uses w[0] in place of the missing w[-*i*] and w[*n*] in place of the missing w[*n*+*i*].
/EVEN [=*evenAllowed*]	Specifies the smoothing increment for boxcar smoothing (/B). Values are: 0: Increments even values of *num* to the next odd value. Default when /EVEN omitted. 1: Uses even values of *num* for boxcar smoothing despite the half-sample shifting this introduces in the smoothed output (prior to version 6, this shift was prevented). Same as/EVEN alone.
/F [=*f*]	Selects the boxcar or multipass binomial smoothing method: *f*=0: Slow, but accurate, method (default). *f*=1: Fast method. Same as /F alone.
/M=*threshold*	Invokes running-median smoothing and specifies an absolute numeric threshold used to optionally replace "outliers". Points that differ from the central median by an amount exceeding *threshold* are replaced, either with the *replacement* value specified by /R, or otherwise with the median value. Special *threshold* values are: 0: Replace all values with running-median values or the *replacement* value. (NaN): Replace only NaN input values with running-median values or the *replacement* value. The smoothing factor *num* is the number of points in the smoothing window used to compute each median.
/MPCT=*percentile*	Used with /M to compute a smoothed value that is a different percentile than the median. /M must be present if /MPCT is used. *percentile* is a value from 0 to 100.

Roughly speaking, the smoothed value returned is the smallest value in the smoothing window that is greater than the smallest *percentile* % of the values. See "Median and Percentile Smoothing Details", below.

percentile=0:	The smoothed value is the minimum value in the smoothing window.
percentile=50:	The smoothed value is the median of the values in the smoothing window. This is the default if /MPCT is omitted.
percentile=100:	The smoothed value is the maximum value in the smoothing window.

/R=*replacement*	Specifies the value that replaces input values that exceed the central median by *threshold* (requires /M). *replacement* can be any value (including NaN or ±Inf if *waveName* is floating point).
/S=*sgOrder*	Invokes Savitzky-Golay smoothing algorithm and specifies the smoothing order. *sgOrder* must be either 2 or 4.

Binomial Smoothing Details

For binomial smoothing, use no flags (other than /DIM, /E, and /F) and a *num* value from 1 to 32767.

The binomial smooth algorithm automatically switches to a nearly equivalent, but *much* faster, multipass box smooth at smooth factor of 50. The original algorithm can be used when you set this global variable:

```
Variable/G root:V_doOrigBinomSmooth=1
```

To get the pre-Igor Pro 6 behavior you also need to add the /F flag.

The /F (fast boxcar smoothing) algorithm creates small errors when the data has a large offset. For some data sets you may want to subtract the mean of the data before smoothing and add it back in afterwards.

The binomial smoothing algorithm does not detect and ignore NaNs in the input data.

Boxcar Smoothing Details

For boxcar smoothing, use the /B flag and a *num* value from 1 to 32767.

For *num* < 2, no smoothing is done.

If *num* is even and /EVEN is not specified, *num* is incremented to the next (odd) integer.

If *num* is even and /EVEN is specified, each smoothed output is formed from one more previous value than future values.

The boxcar smoothing algorithm detects and ignores NaNs in the input data. If *num* is less than the number of NaNs near the output point, then the result is NaN. Otherwise the average of the non-NaN neighboring points is used to compute the smoothed result.

Savitzky-Golay Smoothing Details

For Savitzky-Golay smoothing, use the /S flag and an odd *num* value from 5 to 25. An even value for *num* returns an error. If *sgOrder*=4, then *num*= 5 gives no smoothing at all so *num* should be at least 7.

The Savitzky-Golay smoothing algorithm does not detect and ignore NaNs in the input data.

Median and Percentile Smoothing Details

For running-median smoothing, use the /M flag and a *num* value from 1 to 32767. When *num* is 1, no smoothing is done.

If *num* is even, the median is the average of the two middle values.

For example, the median of 6 values around data[i] is the median of data[i-3], data[i-2], data[i-1], data[i], data[i+1], and data[i+2], and if these values were already sorted, the median would be the average of data[i-1] and data[i].

Use /M=0 to replace all values with the median over the smoothing window or use `/M=`*`threshold`*`/R=(NaN)` to replace outliers with NaNs.

Use `/M=(NaN)` to replace only NaN input values with the running-median values or the *replacement* value.

The running-median smoothing algorithm detects and ignores NaNs in the input data. If *num* is less than the number of NaNs near the output point, then the result is NaN. Otherwise the median of the non-NaN neighboring points is used to compute the smoothed result.

The running-median is a special case of running-percentile, with *percentile*=50.

The /M and /MPCT algorithm uses an interpolated rank to compute the value of percentiles other than 0 and 100.

Using Example 1 from <http://cnx.org/content/m10805/latest/> ("A Third Definition"), the 25th percentile (/MPCT=25) of the 8 values:

```
    Make/O sortedData={3,5,7,8,9,11,13,15}// Already sorted, rank 1 to 8
```

The first step is to compute the rank (R) of the 25th percentile. This is done using the following formula: R= (*percentile*/100)*(*num*+1), where *percentile* is 25 and *num* is 8, so here R = 2.25.

If R were an integer, the Pth percentile would be the number with rank R; if R were 2 the result would be the 2nd value = 5.

Since R is not an integer, we compute the Pth percentile by interpolation as follows:

1. Define IR as the integer portion of R (the number to the left of the decimal point). For this example, IR=2.
2. Define FR as the fractional portion of R. For this example, FR=0.25
3. Find the values with Rank IR and with Rank IR+1. For this example, this means the values with Rank 2 and the score with Rank 3. The values are 5 and 7.
4. Interpolate by multiplying the difference between the values by FR and add the result to the lower values. For these data, this is 0.25(7-5)+5=5.5

Therefore, the 25th percentile is 5.5:

```
    Smooth/M=0/MPCT=(percentile) 8, sortedData  // 8-point smoothing window
    Print sortedData[3] // prints 5.5, the 25th percentile of all 8 values
```

Smoothing Window and End Effects Details

These smoothing algorithms compute the output value for a given point using each point's neighbors. Except for running-median smoothing, each algorithm combines neighboring points before and after the point being smoothed. At the start or end of a wave some points will not have enough neighbors so some method for fabricating neighbor values must be implemented. The /E flag specifies the method.

The running-median filter, however, ignores /E. At each end of the data fewer values are included in the median calculation, so that values "beyond" the end of data are not needed.

The first output value is the median of `wave[0, floor((`*`num`*`-1)/2)]`. For example, if *num* = 7, then the first output value is the median of wave[0], wave[1], wave[2], and wave[3]. Because that is an even number of points, the median is the average of the two middle values. Continuing the example, if the values were 3, 1, 7, and 5, the two middle values are 3 and 5. The computed median would be (3+5)/2=4.

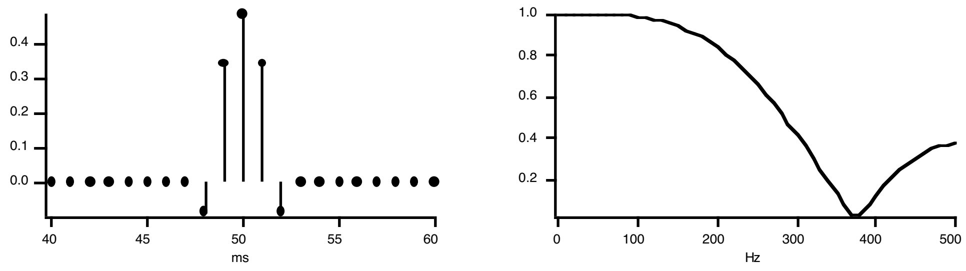
Examples

Box smoothing example:

```
Make/N=100 wv; Display wv
wv=gnoise(1)
Smooth/B/E=3 3,wv    // output[p] = average of wv[p-1], wv[p] and wv[p+1]
                     // /E=3 causes wv[0] = (w[0]+w[0]+w[1])/3
                     // and wv[n-1] = (w[n-2]+w[n-1]+w[n-1])/3
```

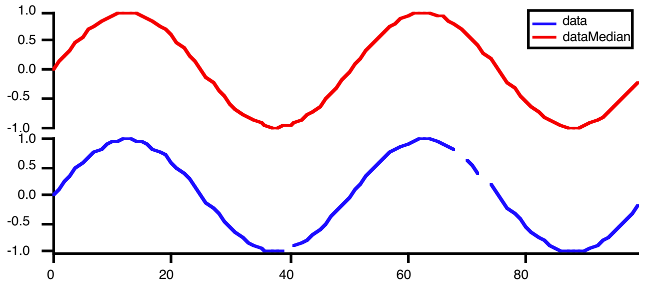
Demonstrate the impulse response of Savitzky-Golay Smoothing:

```
Make/O/N=100 wv
wv= p==50    // 1 at center of wave, 0 elsewhere; an impulse
SetScale/P x, 0, 1/1000, "s", wv           // 1000 Hz sampling rate
Smooth/S=2 5,wv
Display wv
ModifyGraph mode=8,marker=19
FFT/MAG/DEST=fftMag wv
Display fftMag
```



Replace NaN with median:

```
Make/O/N=100 data= enoise(1)>.9 ? NaN : sin(x/8)      // signal with NaNs
Duplicate/O data, dataMedian
Smooth/M=(NaN) 5, dataMedian      // replace (only) NaNs with 5-point median
```



Binomial Smoothing References

Marchand, P., and L. Marmet, *Revues of Scientific Instrumentation 54*, 1034, 1983.

Savitzky-Golay Smoothing References

Savitzky, A., and M.J.E. Golay, *Analytical Chemistry*, *36*, 1627-1639, 1964.

Steiner, J., Y. Termonia, and J. Deltour, *Analytical Chemistry*, *44*, 1906-1909, 1972.

Madden, H., *Analytical Chemistry*, *50*, 1386-1386, 1978.

Percentile References

<http://en.wikipedia.org/wiki/Percentile>

<http://cnx.org/content/m10805/latest/>

See Also

See the **Loess**, **MatrixConvolve**, and **MatrixFilter** operations for true 2D smoothing.

FilterFIR, **FilterIIR**, **Loess**, **Interpolate2**

Also see the “Smooth Operation Responses” example experiment.

SmoothCustom

SmoothCustom [**/E=*endEffect***] ***coefsWaveName*, *waveName*** [, ***waveName***]…

Note: SmoothCustom is obsolete. Use the **FilterFIR** operation instead. For multidimensional data use the **MatrixConvolve** or **MatrixFilter** operations.

The SmoothCustom operation smooths waves by convolving them with *coefsWaveName*.

Parameters

coefsWaveName must be single or double floating point, must not be one of the destination *waveName*s, must not be complex.

waveName is a numeric destination wave that is overwritten by the convolution of itself and *coefsWaveName*.

Flags

/E=*endEffect*	End effect method, a value between 0 and 3. See the **Smooth** operation for a description of the /E flag.

Details

The convolution is in the time domain. That is, the FFT is not employed. For this reason the length of *coefsWaveName* should be small or small in comparison to the destination waves.

SmoothCustom presumes that the middle point of *coefsWaveName* corresponds to the delay = 0 point. The "middle" point number = trunc(numpnts(*coefsWaveName*-1)/2). *coefsWaveName* usually contains the two-sided impulse response of a filter, and contains an odd number of points. This is the type of wave created by FilterFIR.

SmoothCustom ignores the X scaling of all the waves.

The SmoothCustom operation is not multidimensional aware. See **Analysis on Multidimensional Waves** on page II-86 for details.

Sort

`Sort [ /A /DIML /C /R ]` ***`sortKeyWaves, sortedWaveName`*** `[,` ***`sortedWaveName`***`]…`

The Sort operation sorts the *sortedWaveName*s by rearranging their Y values to put the data values of *sortKeyWaves* in order.

Parameters

sortKeyWaves is either the name of a single wave, to use a single sort key, or the name of multiple waves in braces, to use multiple sort keys.

All waves must be of the same length.

The *sortKeyWaves* must not be complex.

Flags

/A[=*a*]	Alphanumeric sort. When *sortKeyWaves* includes text waves, the normal sorting places "wave1" and "wave10" before "wave9". The optional *a* parameter requires Igor Pro 7.00 or later. Use /A or /A=1 to sort the number portion numerically, so that "wave9" is sorted before "wave10". Use /A=2 to ignore + and - characters in the text so that "Text-09" sorts before "Text-10".
/C	Case-sensitive sort. When *sortKeyWaves* includes text waves, the sort is case-insensitive unless you use the /C flag to make it case-sensitive.
/DIML	Moves the dimension labels with the values (keeps any row dimension label with the row's value).
/LOC	Performs a locale-aware sort. When *sortKeyWaves* includes text waves, the text encoding of the text waves' data is taken into account and sorting is done according to the sorting conventions of the current system locale. This flag is ignored if the text waves' data encoding is unknown, binary, Symbol, or Dingbats. This flag cannot be used with the /A flag. See Details for more information. The /LOC flag was added in Igor Pro 7.00.
/R	Reversed sort; sort from largest to smallest.

Details

sortKeyWaves are not actually sorted unless they also appear in the list of destination waves.

The sort algorithm does not maintain the relative position of items with the same key value.

When the /LOC flag is used, the bytes stored in the text wave at each point are converted into a Unicode string using the text encoding of the text wave data. These Unicode strings are then compared using OS specific text comparison routines based on the locale set in the operating system. This means that the order of sorted items may differ when the same sort is done with the same data under different operating systems or different system locales.

When /LOC is omitted the sort is done on the raw text without regard to the waves' text encoding.

Examples

```
Sort/R myWave,myWave       // sorts myWave in decreasing order
Sort xWave,xWave,yWave     // sorts x wave in increasing order,
                           // corresponding yWave values follow.
Make/O/T myWave={"1st","2nd","3rd","4th"}
Make/O key1={2,1,1,1}      // places 2nd, 3rd, 4th before 1st.
Make/O key2={0,1,3,2}      // arranges 2nd, 3rd, 4th as 2nd, 4th, 3rd.
Sort {key1,key2},myWave    // sorts myWave in increasing order by key1.
                           // For equal key1 values, sorted by key2.
                           // Result is myWave={"2nd","4th","3rd","1st"}
Make/O/T tw={"w1","w10","w9","w-2.1"}
Sort/A tw,tw     // sorts tw in increasing number-aware order:
                 // Result is tw={"w-2.1","w1","w9","w10"}
```

See Also

Sorting on page III-126

MakeIndex, **IndexSort**, **Reverse**, **SortColumns**, **SortList**

FindDuplicates

SortColumns

SortColumns [*flags*] keyWaves={*waveList*}, sortWaves={*waveList*}

The SortColumns operation rearranges data in columns of the *sortWaves* using the data movements that would sort the values of the *keyWaves* if they were sorted.

The SortColumns operation was added in Igor Pro 7.00.

Parameters

keyWaves is a lists of 1 or more wave references in braces separated by commas. The first listed wave is the primary sort key, the second is the secondary sort key, and so on. The *keyWaves* list can contain a maximum of 10 waves. The key waves can be either text or real numeric waves. Complex waves, wave reference waves and data folder reference waves can not be used as key waves.

sortWaves is a lists of one or more wave references in braces separated by commas. The *sortWaves* list can contain a maximum of 100 waves.

Flags

/A	Alphanumeric sort. When *keyWaves* includes text waves, or the /KNDX flag is used and the first wave in the *sortWaves* list is a text wave, the normal sorting places "wave1" and "wave10" before "wave9". Use /A to sort the number portion numerically, so that "wave9" is sorted before "wave10". /A cannot be used with the /LOC flag.
/C	Case-sensitive sort. When *keyWaves* includes text waves, or the /KNDX flag is used and the first wave in the *sortWaves* list is a text wave, the sort is case-insensitive unless you use the /C flag to make it case-sensitive.
/DIML	Moves the row dimension labels with the data values. Column dimension labels remain unchanged.
/KNDX={*c0, c1, ... c9*}	
	Specifies up to 10 columns of the first wave in the *sortWaves* list to use as the sort keys. This flag and the *keyWaves* keyword are mutually exclusive. If this flag is used then the first wave in the *sortWaves* list must be either a real numeric or text wave.

/LOC	Locale aware sort. When *keyWaves* includes text waves, or the /KNDX flag is used and the first wave in the *sortWaves* list is a text wave, the text encoding of the text waves' data is taken into account and sorting is done according to the sorting conventions of the current system locale. /LOC is ignored if the text waves' data encoding is unknown, binary, Symbol, or Dingbats. /LOC can not be used with the /A flag. See Details for more information.
/R	Reverses the sort, sorting from largest to smallest.

Details

Waves in the *keyWaves* list are not actually sorted unless they also appear in the *sortWaves* list.

All waves must have the same number of rows but can have different numbers of columns, layers and chunks.

keyWaves, or the first wave in the *sortWaves* list when /KNDX is used, must be either numeric or text waves.

When the *sortWaves* list includes 3D or 4D waves, the operation sorts all columns of all layers/chunks.

The sorting algorithm used does not maintain the relative position of rows with the same key value.

When the /LOC flag is used, the bytes stored in the text wave at each point are converted into a Unicode string using the text encoding of the text wave data. These Unicode strings are then compared using OS-specific text comparison routines based on the current locale as set in the operating system. This means that the order of sorted items may differ when the same sort is done with the same data under different operating systems or different system locales.

Examples

```
// Define a function that creates sample data
Function CreateSampleData()
   Make/O key1={3,1,0,2}
   Make/O/T text1={"Jack","Fred","Robin","Bob"}
   Make/O w1={{1,2,3,4},{11,12,13,14}}
End

// Create sample data and display in a table
CreateSampleData()
Edit key1,text1,w1

// Sort based on a numeric key
SortColumns keyWaves=key1,sortWaves=w1

// Revert the data
CreateSampleData()

// Sort based on text key
SortColumns keyWaves=text1,sortWaves=w1

// Revert the data
CreateSampleData()

// Sort using key index
SortColumns/kndx=0 sortWaves={text1,w1}
```

See Also

Sorting on page III-126, **Sort**, **Reverse**, **SortList**

SortList

`SortList(`*`listStr`*` [, `*`listSepStr`*` [, `*`options`*`])`

The SortList function returns *listStr* after sorting it according to the default or *listSepStr* and *options* parameters. *listStr* should contain items separated by *listSepStr*, such as "the first item;second item;".

Use SortList to sort the items in a string containing a list of items separated by a string, such as those returned by functions like **TraceNameList** or **WaveList**, or a line of text from a delimited text file, where listSepStr can be "\r" or "\r\n".

listSepStr and *options* are optional; their defaults are ";" and 0 (ascending alphabetic sort), respectively.

Details

listStr is treated as if it ends with a *listSepStr* even if it doesn't. The returned list will always have an ending *listSepStr* string.

In Igor6, SortList used only the first byte of listSepStr. As of Igor7, it uses the whole string.

options controls the sorting method, as follows:

0:	Default sort (ascending case-sensitive alphabetic ASCII sort).
1:	Descending sort.
2:	Numeric sort.
4:	Case-insensitive sort.
8:	Case-sensitive alphanumeric sort.
16:	Case-insensitive alphanumeric sort that sorts wave0 and wave9 before wave10.
32:	Unique sort in which duplicates are removed. Added in Igor Pro 7.00.
64:	Ignore + and - in the alphanumeric sort so that "Text-09" sorts before "Text-10". Set options to 80 or 81. Added in Igor Pro 7.00.

options may also be a bitwise combination of these values with the following restriction: only one of 2, 4, 8, or 16 may be specified. Thus the legal values are thus 0, 1, 2, 3, 4, 5, 8, 9, 16, 17, 32, 33, 34, 35, 36, 40, 41, 48, 49, 80 or 81. Other values will produce undefined sorting.

In a case-insensitive, unique sort (options=4+32), if two items differ only in case, which one is retained is not specified.

Examples

```
// Alphabetic sorts
Print SortList("c;a;a;b")                    // prints "a;a;b;c;"
Print SortList("you,me,More", ",", 0)        // prints "More,me,you,"
Print SortList("you,me,More", ",", 4)        // prints "me,More,you,"
Print SortList("9,93,91,33,15,3", ",")       // prints "15,3,33,9,91,93,"
Print SortList("Zx;abc;All;", ";", 0)        // prints "All;Zx;abc;"
Print SortList("Zx;abc;All;", ";", 8)        // prints "abc;All;Zx;"
Print SortList("w9;w10;w02;", ";", 16)       // prints "w02;w9;w10;"

// Unique sort
Print SortList("b;c;a;a;", ";", 32)          // prints "a;b;c;"
Print SortList("b;c;A;a;", ";", 4+32)        // prints "A;b;c;"
Print SortList("b;c;a;A;", ";", 4+32)        // prints "a;b;c;"

// Numeric sorts
Print SortList("9,93,91,33,15,3",",",2)      // prints "3,9,15,33,91,93,"
Print SortList("9,93,91,33,15,3",",",3)      // prints "93,91,33,15,9,3,"
```

See Also

Sort, **StringFromList**, **WaveList**, **RemoveEnding**

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

SoundInRecord

`SoundInRecord [/Z] `*`wave`*

The SoundInRecord operation records audio input at the sample rate obtained from the wave's X scaling and for the number of points determined by the length of the wave. The recording is done synchronously.

The number type of the wave must be one of the types reported by the **SoundInStatus** operation in the V_SoundInSampSize variable. On Windows this will typically be 8- or 16-bit integer while on Macintosh 16-bit integer and 32-bit floating point (the OS X native type) will be supported.

To record in stereo, provide a 2 column wave. (The software is designed to handle any number of channels but has not been tested on more than 2.)

Flags

/Z	Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SoundInRecord requires a computer with sound inputs. Several sample experiments using sound input can be found in your Igor Pro 7 Folder in the Examples folder.

See Also

The **SoundInSet**, **SoundInStartChart**, and **SoundInStatus** operations.

SoundInSet

`SoundInSet [/Z][gain=`*`g`*`, agc=`*`a`*`]`

The SoundInSet operation is used to setup the input device for recording.

Parameters

SoundInSet can accept multiple *keyword =value* parameters on one line.

agc=*a* Turns automatic gain control mode on (*a*=1) or off (*a*=0). Will generate an error if device does not support setting agc. Use SoundInStatus to check or use /Z flag to make errors nonfatal.

Windows: This is not supported and V_SoundInAGC from the SoundInStatus command always returns -1.

gain=*g* Sets input gain, 0 is lowest gain and 1 is highest. Will generate an error if device does not support setting gain. Use SoundInStatus to check or use /Z flag to make errors nonfatal.

Windows: SoundInSet attempts to adjust the master gain of the sound input device but not all sound cards have a master gain. If V_SoundInGain from the SoundInStatus command returns -1, you will have to use your sound card software to adjust the input gain for the particular input source your are using. On some cards there are separate line-in and microphone-in sources.

Flags

/Z	Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SoundInSet requires a computer with sound inputs. Several sample experiments using sound inputs are in your Igor Pro 7 Folder in the Examples folder.

See Also

The **SoundInRecord**, **SoundInStartChart**, and **SoundInStatus** operations.

SoundInStartChart

`SoundInStartChart [/Z]` ***`buffersize , destFIFOname`***

The SoundInStartChart operation starts audio data acquisition into the given FIFO.

Parameters

buffersize is the number of bytes to allocate for the interrupt time buffer which then feeds into the given Igor named FIFO *destFIFOname*. The FIFO must be set up with the correct number of channels and number type - use **SoundInStatus** to find legal values. The sample rate is read from the FIFO also, so that also needs to be correct.

Flags

/Z	Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SoundInStartChart requires a computer with sound inputs. Several sample experiments using sound inputs are in your Igor Pro 7 Folder in the Examples folder.

On systems where 32-bit floating point data is supported, you can use NewFIFOChan with no flags and a range of -1 to 1.

See Also

The **SoundInRecord**, **SoundInSet**, **SoundInStatus** and **SoundInStopChart** operations, and **FIFOs and Charts** on page IV-291.

SoundInStatus

`SoundInStatus`

The SoundInStatus operation creates and sets a set of variables and strings with information about the current sound input device. The variable V_flag is set to an error code and will be zero if the device is available. If not then none of the following are valid.

Variables	Contents
`S_SoundInName`	String with name of device.
`V_SoundInAGC`	Automatic gain control on or off (1 or 0). This is an optional item and if the current device does not support AGC then V_SoundInAGC will be set to -1.
`V_SoundInChansAv`	Available number of channels (e.g., 1 for mono, 2 for stereo).
`V_SoundInGain`	Current input gain. Ranges from 0 (lowest) to 1. This is an optional item and if the current device does not support gain then V_SoundInGain will be set to -1.
`V_SoundInSampSize`	Bits set depending on number of bits available in a sample. Bit 0: Set if can do 8 bits. Bit 1: Set if can do 16 bits. Bit 3: Set if 32-bit floating point is supported (range is -1 to 1).
`W_SoundInRates`	Wave containing sample rate info: if point zero contains zero then points 1 and 2 contain the lower and upper limits of a continuous range else point zero contains the number of discrete rates which follow in the wave. The usual rates are 44100 Hz and 4800 Hz.

See Also

The **SoundInRecord**, **SoundInSet**, and **SoundInStartChart** operations.

SoundInStopChart

`SoundInStopChart [/Z]`

The SoundInStopChart operation stops audio data acquisition started by SoundInStartChart.

Flags

/Z	Errors are not fatal. V_flag is set to zero if no error, else nonzero if error.

Details

SoundInStopChart requires a computer equipped with sound input hardware.

Audio data acquisition also stops automatically when an experiment is closed.

See Also

The **SoundInStartChart** and **SoundInStatus** operations.

SoundLoadWave

`SoundLoadWave [flags] waveName [ ,fileNameStr ]`

The SoundLoadWave operation loads sound data from the named file into a wave. Mono, stereo, surround-sound, and high-resolution sound formats are supported.

The SoundLoadWave operation was added in Igor Pro 7.00.

Parameters

waveName is the name of the wave to load the sound into.

If *fileNameStr* is omitted or is "", SoundLoadWave displays an Open File dialog.

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with pathName, or the name of a file in the folder associated with pathName. If SoundLoadWave can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to choose the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/I [= *filterStr*]	Force interactive mode. Use optional filter string to limit allowable file extensions. See **Open File Dialog File Filters** on page IV-137.
/O	Overwrite existing waves in case of a name conflict.
/P=*pathName*	Specifies the folder to load the file from. *pathName* is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\`". See **Symbolic Paths** on page II-21 for details.
/Q	Quiet: Doesn't print message to history area, and doesn't abort, if the sound can not be loaded. V_Error is set to the returned error code, which will be zero if there was no error.
/S=(*startT*,*endT*)	Load a subrange of the sound resource. *startT* and *endT* are in seconds, clipped to the duration of the loaded sound.
/TMOT= *timeOut*	Aborts load if *timeOut*, in seconds, is exceeded.

Details

SoundLoadWave uses Core Audio on Macintosh and Qt framework calls on Windows. Note that some files can not be loaded due to digital rights managment issues even though they can be played.

If *waveName* specifies a wave that does not exist, it is created. The wave is redimensioned to a wave type that maintains the numeric precision of the sound data. If the wave can not be created or resized to fit the loaded data then SoundLoadWave returns an error.

If *waveName* does exist, the wave is overwritten only if the /O flag is specified. Without the /O flag SoundLoadWave returns an error.

Multi-channel audio is loaded into sequential columns of the destination wave.

On Macintosh, Core Audio provides only 32-bit floating point data and that is the data type used for the wave. The BITS value in S_info, described below, may be zero for some formats.

On Windows, SoundLoadWave uses the smallest Igor wave data type that preserves the number of bits in the audio. Igor doesn't have a 24-bit data type, so these values are stored in a 32-bit integer wave.

Output Variables

SoundLoadWave sets these output variables:

V_flag	Set to 1 if a sound is loaded and fits into available memory, 0 otherwise.

V_Error	Set if /Q is specified, V_Error is set to a non-zero error code if something went wrong or to zero on success. Negative returned codes are system-dependent, positive are Igor-defined errors.
	V_Error = 1 means there wasn't enough memory to load the (uncompressed) sound.
S_path	Set to the full file path of the loaded file, not including the file name.
S_fileName	Set to the name of the loaded file.
S_waveNames	Set to the name of loaded wave.
S_info	Information about the loaded sound.

If the sound file exists, SoundLoadWave sets the string variable S_info to:

```
"FILE:nameOfFile;FORMAT:soundFileFormat;CHANNELS:numChannels;CHANNEL_LAYOUT:ch
annelLayoutDescription;CHANNEL_ORDER:channelsList;BITS:numBits;SAMPLES:numSamp
les;RATE:samplesPerSec;"
```

The *soundFileFormat* and *channelLayoutDescription* values are text descriptions of the sound data in the file, and are written in the localized language. This information is available only on Macintosh and may or may not be present in a given sound file.

The *channelsList* value is a comma-separated list of channel names, always in English abbreviations, such as "L,R" or "L,R,C,LFE,Ls,Rs". The meaning of the abbrevations:

channelList Abbreviation	**Channel or Speaker Names**
L	Front Left
R	Front Right
C	Front Center
LFE	Low Frequency Effects
Ls	Left Surround (Back Left)
Rs	Right Surround (Back Right)
Lc	Left Center (Front Left of Center)
Rc	Right Center (Front Right of Center)
Cs	Center Surround (Back Center)
Lsd	Left Surround Direct (Side Left)
Rsd	Right Surround Direct (Side Right)
Ts	Top Center Surround (Top Center)
Vhl	Vertical Height Left (Top Front Left)
Vhc	Vertical Height Center (Top Front Center)
Vhr	Vertical Height Right (Top Front Right)
Rls	Rear Left Surround (Top Back Left)
Rcs	Rear Center Surround (Top Back Center)
Rrs	Rear Right Surround (Top Back Right)

Examples

```
// Display an Open File dialog and load the chosen file.
// Use file's name for wave, overwrite any pre-existing wave, print information to history
SoundLoadWave/O myDestWave

// SoundLoadWave stores following in S_Info and prints it to the history area
```

```
FILE:<file name>;FORMAT:MPEG Layer 3;CHANNELS:2;BITS:0;SAMPLES:524416;RATE:44100;

// Rename the wave to a cleaned up version of the file name
Rename myDestWave, $CleanupName(S_fileName,1)
```

See Also

SoundSaveWave, **PlaySound**

SoundSaveWave

SoundSaveWave [flags] typeStr, waveName [, fileNameStr]

The SoundSaveWave operation saves the named wave on disk as an Audio Interchange File Format (AIFF-C) or Microsoft WAVE sound file. AIFF-C is primarily used on Macintosh.

The SoundSaveWave operation was added in Igor Pro 7.00.

Parameters

typeStr must be either "AIFC" or "WAVE".

fileNameStr contains the name of the file in which the named wave is saved. If you omit fileNameStr , SoundSaveWave uses the wave name with the appropriate extension.

The file to be written is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with pathName, or the name of a file in the folder associated with pathName. If SoundSaveWave can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

Flags

/O	Overwrites the file if it already exists. If you omit /O and the file exists, SoundSaveWave displays a Save File dialog.
/P=*pathName*	Specifies the folder to store the file in. *pathName* is the name of an Igor symbolic path, created via **NewPath**. It is not a file system path like "`hd:Folder1:`" or "`C:\\Folder1\`". See **Symbolic Paths** on page II-21 for details.
/Q	Suppresses the normal messages in the history area of the command window. At present nothing is written to the history even if /Q is omitted.

Details

The sound file is always an uncompressed AIFF-C or WAVE file, with as many channels as the wave contains columns.

The sound file format is determined by the wave's data type. Signed 8-, 16- and 32-bit integers are supported as is 32-bit floating point. When writing floating point waves, the wave data should be scaled to +/- 1 as full scale.

Output Variables

SoundSaveWave sets these automatically created variables:

V_flag	Set to 1 if the wave was successfully saved to the file, else 0.
S_fileName	Set to the name of the saved file.
S_path	Set to the full path to the file's directory.

Examples

```
// Create a simple sound (1000 Hz tone burst)
Make/O/N=10000 mySound                   // Single-precision wave, 10,000 values
SetScale/P x, 0, 1/8000, "" mySound      // 8000 Hz sampling frequency (1.25 seconds)
mySound= sin(2*pi*1000*x)                // 1000 Hz tone
Hanning mySound                          // Fade in and out
```

```
// Save it to a file, chosen from the Save File dialog
SoundSaveWave "AIFC", mySound, "my sound.aif"

// Create a floating point stereo frequency sweep
Make/O/N=(20000,2) stereoSineSoundF32      // 32-bit float data
SetScale/P x,0,1e-4,stereoSineSoundF32     // Set sample rate to 10KHz
stereoSineSoundF32= sin(2*Pi*(1000 + (1-2*q)*150*x)*x)
NewPath sound                              // Create a symbolic path via dialog
SoundSaveWave/P=sound/O "WAVE", stereoSineSoundF32
```

See Also

SoundLoadWave, **PlaySound**, **WaveType**, **WaveInfo**

SpecialCharacterInfo

`SpecialCharacterInfo(`*`notebookNameStr, specialCharacterNameStr, whichStr`*`)`

The SpecialCharacterInfo function returns a string containing information about the named special character in the named notebook window.

Parameters

If *notebookNameStr* is "", the top visible notebook is used. Otherwise *notebookNameStr* contains either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-87 for details on host-child specifications.

specialCharacterNameStr is the name of a special character in the notebook.

If specialCharacterNameStr is "" and if exactly one special character is selected, the selected special character is used. If other than exactly one special character is selected, an error is returned.

whichStr identifies the information item you want. Because SpecialCharacterInfo can return several items that may contain semicolons, it does not return a semicolon-separated keyword-value list like other info functions. Instead it returns just one item as specified by *whichStr*.

Details

Here are the supported values for *whichStr*.

Keyword	Returned Information
NAME	The name of the special character.
FRAME	0: None 1: Single 2: Double 3: Triple 4: Shadow
LOC	Paragraph and character position (e.g., 1,3).
SCALING	Horizontal and vertical scaling in units of one tenth of a percent (e.g., 1000,1000).
TYPE	Special character type is: Picture, Graph, Table, Layout, Action, ShortDate, LongDate, AbbreviatedDate, Time, Page, TotalPages, or WindowTitle.

These keywords apply to Igor-object pictures only. If the specified character is not an Igor-object picture, “” is returned.

Keyword	Returned Information
WINTYPE	1 for graphs, 2 for tables, 3 for layouts.
OBJECTNAME	The name of the window with which the special character is associated.

The remaining keywords apply to notebook action characters only. If the specified special character is not a notebook action character, `""` is returned.

Keyword	Returned Information
BGRGB	Background color in RGB format (e.g., 65535,65534,49151).
COMMANDS	Command string.
ENABLEBGRGB	1 if the action's background color is enabled, 0 if not.
HELPTEXT	Help text string.
IGNOREERRORS	0 or 1.
LINKSTYLE	0 or 1.
PADDING	The value of the left, right, top, bottom and internal padding properties, in that order (.e.g, 4,4,4,4,8).
PICTURE	1 if the action has a picture, 0 if not.
PROCPICTNAME	The name of the action Proc Picture or `""` if none.
QUIET	0 or 1.
SHOWMODE	1: Title only 2: Picture only 3: Picture below title 4: Picture above title 5: Picture to the left of title 6: Picture to the right of title
TITLE	Title string.

If *whichStr* is an unknown keyword, SpecialCharacterInfo returns `""` but does not generate an error.

Examples

```
Function PrintSpecialCharacterInfo(notebookName, specialCharacterName)
   String notebookName, specialCharacterName

   String typeStr=SpecialCharacterInfo(notebookName, specialCharacterName, "TYPE")
   String locStr=SpecialCharacterInfo(notebookName, specialCharacterName, "LOC")

   Printf "TYPE: %s\r", typeStr
   Printf "LOC: %s\r", locStr
End
```

See Also

The **Notebook** and **NotebookAction** operations; the **SpecialCharacterList** function; **Using Igor-Object Pictures** on page III-18.

SpecialCharacterList

`SpecialCharacterList(`*`notebookNameStr, separatorStr, mask, flags`*`)`

The SpecialCharacterList function returns a string containing a list of names of special characters in a formatted text notebook.

Parameters

If *notebookNameStr* is "", the top visible notebook is used. Otherwise *notebookNameStr* contains either kwTopWin for the top notebook window, the name of a notebook window or a host-child specification (an hcSpec) such as Panel0#nb0. See **Subwindow Syntax** on page III-87 for details on host-child specifications.

separatorStr should contain a single character, usually semicolon, to separate the names.

mask determines which types of special characters are included. *mask* is a bitwise parameter with values:

1:	Pictures including graphs, tables and layouts.

2:	Notebook actions.
4:	All other special characters such as dates and times.

or a bitwise combination of the above for more than one type. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

flags is a bitwise parameter. Pass 0 to include all special characters or 1 to include only selected special characters. All other bits are reserved and should be passed as zero.

Details

Only formatted text notebooks have special characters. When called for a plain text notebook, SpecialCharacterList always returns `""`.

Examples

Print a list of all special characters in the top notebook:

```
Print SpecialCharacterList("", ";", -1, 0)
```

Prints a list of notebook action characters in Notebook0:

```
Print SpecialCharacterList("Notebook0", ";", 2, 0)
```

Print a list of selected notebook action characters in Notebook0:

```
Print SpecialCharacterList("Notebook0", ";", 2, 1)
```

See Also

The **Notebook** and **NotebookAction** operations; the **SpecialCharacterInfo** function.

SpecialDirPath

`SpecialDirPath(`*`dirIDStr, domain, flags, createDir`*`)`

The SpecialDirPath function returns a full path to a file system directory specified by *dirIDStr* and *domain*. It provides a programmer with a way to access directories of special interest, such as the preferences directory and the desktop directory.

The path returned always ends with a separator character which may be a colon, backslash, or forward slash depending on the operating system and the *flags* parameter.

SpecialDirPath depends on operating system behavior. The exact path returned depends on the locale, the operating system, the specific installation, the current user, and possibly other factors.

Parameters

dirIDStr is one of the following strings:

`"Packages"`	Place for advanced programmers to put preferences for their procedure packages.
`"Documents"`	The OS-defined place for users to put documents.
`"Preferences"`	The OS-defined place for applications to put preferences.
`"Desktop"`	The desktop.
`"Temporary"`	The OS-defined place for applications to put temporary files.

`"Igor Application"`	The Igor installation folder. This is typically:
	`/Applications/Igor Pro 7 Folder (`*`Macintosh`*`)`
	or
	`C:\Program Files\WaveMetrics\Igor Pro 7 Folder (`*`Windows`*`)`
	Use only with domain = 0 (the current user).

`"Igor Executable"`	The folder containing the current Igor executable. On Macintosh, this is the path to the executable itself, not to the application bundle. Use only with domain = 0 (the current user). Requires Igor Pro 7.00 or later.
`"Igor Preferences"`	The folder in which Igor's own preference files are stored.
`"Igor Pro User Files"`	A guaranteed-writable folder for the user to store their own Igor files, and to activate extensions, help, and procedure files by creating shortcuts or aliases in the appropriate subfolders. Use only with domain = 0 (the current user). This is the folder opened using the Show Igor Pro User Files menu item in the Help menu.

domain permits discriminating between, for example, the preferences folder for all users versus the preferences folder for the current user. It is supported only for certain *dirIDStrs*. It is one of the following:

0:	The current user (recommended value for most purposes).
1:	All users (may generate an error or return the same path as 0).
2:	System (may generate an error or return the same path as 1).

flags a bitwise parameter:

Bit 0:	If set, the returned path is a native path (Macintosh-style on Mac OS 9, Unix-style on Mac OS X, Windows-style on Windows). If cleared, the returned path is a Macintosh-style path regardless of the current platform. In most cases you should set this bit to zero since Igor accepts Macintosh-style paths on all operating systems. You must set this bit to one if you are going to pass the path to an external script.

All other bits are reserved and must be set to zero.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

createDir is 1 if you want the directory to be created if it does not exist or 0 if you do not want it to be created. This flag will not work if the current user does not have sufficient privileges to create the specified directory. In almost all cases it is not needed, you can't count on it, and you should pass 0.

Details

The *domain* parameter has no effect in most cases. In almost all cases you should pass 0 (current user) for this parameter. For values other than 0, SpecialDirPath might return an error which you must be prepared to handle.

In the event of an error, SpecialDirPath returns a NULL string and sets a runtime error code. You can check for an error like this:

```
String fullPath = SpecialDirPath("Packages", 0, 0, 0)
Variable len = strlen(fullPath)         // strlen(NULL) returns NaN
if (numtype(len) == 2)                  // fullPath is NULL?
   Print "SpecialDirPath returned error."
endif
```

Here is sample output from `SpecialDirPath("Packages",0,0,0)`:

Mac OS X	hd:Users:<*user*>:Library:Preferences:WaveMetrics:Igor Pro 6:Packages:
Windows	C:Documents and Settings:<*user*>:Application Data:WaveMetrics:Igor Pro 6:Packages:

where <*user*> is the name of the current user. The preferences directory may be hidden by some operating systems.

Example

For an example using SpecialDirPath, see **Saving Package Preferences** on page IV-237.

sphericalBessJ

```
sphericalBessJ(n, x [, accuracy])
```

The sphericalBessJ function returns the spherical Bessel function of the first kind and order *n*.

$$j_n(x) = \sqrt{\frac{\pi}{2x}} J_{n+1/2}(x).$$

For example:

$$j_0(x) = \frac{\sin(x)}{x}$$

$$j_1(x) = \frac{\sin(x)}{x^2} - \frac{\cos(x)}{x}$$

$$j_2(x) = \left(\frac{3}{x^3} - \frac{1}{x} \right) \sin(x) - \frac{3}{x^2} \cos(x).$$

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **sphericalBessJD** and **sphericalBessY** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

sphericalBessJD

```
sphericalBessJD(n, x [, accuracy])
```

The sphericalBessJD function returns the derivative of the spherical Bessel function of the first kind and order *n*.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **sphericalBessJ** and **sphericalBessY** functions.

sphericalBessY

```
sphericalBessY(n, x [, accuracy])
```

The sphericalBessY function returns the spherical Bessel function of the second kind and order *n*.

$$y_n(x) = \sqrt{\frac{\pi}{2x}} Y_{n+1/2}(x).$$

$$y_0(x) = -\frac{\cos(x)}{x}$$

$$y_1(x) = -\frac{\cos(x)}{x^2} - \frac{\sin(x)}{x}$$

$$y_2(x) = \left(\frac{1}{x} - \frac{3}{x^3}\right)\cos(x) - \frac{3}{x^2}\sin(x).$$

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **sphericalBessYD** and **sphericalBessJ** functions.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

sphericalBessYD

`sphericalBessYD(`*`n, x`*` [, `*`accuracy`*`])`

The sphericalBessYD function returns the derivative of the spherical Bessel function of the second kind and order *n*.

Details

See the **bessI** function for details on accuracy and speed of execution.

See Also

The **sphericalBessJ** and **sphericalBessY** functions.

sphericalHarmonics

`sphericalHarmonics(`*`L, M, q, f`*`)`

The sphericalHarmonics function returns the complex-valued spherical harmonics

$$Y_L^M(\theta,\phi) = (-1)^M \sqrt{\frac{2L+1}{4\pi}\frac{(L-M)!}{(L+M)!}} P_L^M(\cos\theta) e^{iM\phi}$$

$$Y_L^M(\theta,\phi) = (-1)^M \sqrt{\frac{2L+1}{4\pi}\frac{(L-M)!}{(L+M)!}} P_L^M(\cos(\theta)) e^{iM\phi},$$

where $P_L^M(\cos(\theta))$ is the associated Legendre function.

See Also

The **legendreA** function. The NumericalIntegrationDemo.pxp experiment.

Demos

Choose File→Example Experiments→Visualization→SphericalHarmonicsDemo.

Choose File→Example Experiments→Analysis→NumericalIntegrationDemo.

References

Arfken, G., *Mathematical Methods for Physicists*, Academic Press, New York, 1985.

SphericalInterpolate

`SphericalInterpolate` ***`triangulationDataWave, dataPointsWave, newLocationsWave`***

The SphericalInterpolate operation works in conjunction with the SphericalTriangulate operation to calculate interpolated values on a surface of a sphere. Given a set of $\{x_i, y_i, z_i\}$ points on the surface of a sphere with their associated values $\{v_i\}$, the SphericalTriangulate operation performs the Delaunay triangulation and creates an output that is used by the SphericalInterpolate operation to calculate values at any other point on the surface of a sphere. The interpolation calculation uses Voronoi polygons to weigh the contribution of the nearest neighbors to any given location on the sphere.

Parameters

triangulationDataWave is a 13 column wave that was created by the SphericalTriangulate operation.

dataPoints is a 4 column wave. The first 3 columns are the $\{x_i, y_i, z_i\}$ locations that were used to create the triangulation, and the last column corresponds to the $\{v_i\}$ values at the triangulation locations.

newLocationsWave is a 3 column wave that specifies the x, y, z locations on the sphere at which the interpolated values are calculated. Note that internally, each triplet is normalized to a point on the unit sphere before it is used in the interpolation.

Details

You will always need to use the SphericalTriangulate operation first to generate the *triangulationDataWave* input for this operation.

The result of the operation are put in the wave W_SphericalInterpolation.

See Also

The **SphericalTriangulate** operation.

Demo

Choose File→Example Experiments→Analysis→SphericalTriangulationDemo.

SphericalTriangulate

`SphericalTriangulate` `[/Z]` ***`tripletWaveName`***

The SphericalTriangulate operation triangulates an arbitrary XYZ triplet wave on a surface of a sphere.

It starts by normalizing the data to make sure that sqrt(x^2+y^2+z^2)=1, and then proceeds to calculate the Delaunay triangulation.

Flags

/Z	No error reporting.

Details

The result of the triangulation is the wave M_SphericalTriangulation. This 13 column wave is used in SphericalInterpolate to obtain the interpolated values.

Example

```
// Generates output waves that can be used in Gizmo to display the triangulation.
// triangulationData is the M_TriangulationData output from SphericalTriangulation.
// tripletWave is the source wave input to SphericalTriangulation.
// Output wave sphereTrianglesPath can be used to display the triangulation as a path.
// Output wave sphereTrianglesSurf can be used to display the triangulation as a surface.
Function BuildTriangleWaves(triangulationData,tripletWave)
	Wave triangulationData, tripletWave

	// Extract 3 columns from triangulationData that contain the index of the row.
	Duplicate/O/FREE/r=[][1,3] triangulationData,triIndices
	Variable finalNumTriangles=dimSize(triIndices,0),i,j,k

	// Initialize both waves to NaN so any unassigned point would appear as a hole.
	Make/O/N=(5*finalNumTriangles,3) sphereTrianglesPath=NaN
	Make/O/N=(3*finalNumTriangles,3) sphereTrianglesSurf=NaN

	// Assign the values of the vertices to the two waves:
	Variable rowIndex,rowIndex0,outRowCount=0,outcount2=0
	for(i=1;i<finalNumTriangles;i+=1)
```

```
        for(j=0;j<3;j+=1)
            rowIndex=triIndices[i][j]
            for(k=0;k<3;k+=1)
                sphereTrianglesPath[outRowCount][k]=tripletWave[rowIndex][k]
                sphereTrianglesSurf[outcount2][k]=tripletWave[rowIndex][k]
            endfor
            outRowCount+=1
            outcount2+=1
        endfor

        // Close the triangle path by returning to the first vertex:
        rowIndex0=triIndices[i][0]
        sphereTrianglesPath[outRowCount][0]=tripletWave[rowIndex0][0]
        sphereTrianglesPath[outRowCount][1]=tripletWave[rowIndex0][1]
        sphereTrianglesPath[outRowCount][2]=tripletWave[rowIndex0][2]
        outRowCount+=2          // Increment row count and skip the NaN
    endfor
End
```

See Also

The **SphericalInterpolate** operation.

Demo

Choose File→Example Experiments→Analysis→SphericalTriangulationDemo.

SplitString

SplitString /E=regExprStr str [***, substring1*** [***, substring2,… substringN***]]

The SplitString operation uses the regular expression *regExprStr* to split *str* into subpatterns. See **Subpatterns** on page IV-175 for details. Each matched subpattern is returned sequentially in the corresponding substring parameter.

Parameters

str is the input string to be split into subpatterns.

The *substring1*…*substringN* output parameters must be the names of existing string variables if you need to use the matched subpatterns. The first matched subpattern is returned in *substring1,* the second in *substring2,* etc.

Flags

/E=*regExprStr*	Specifies the Perl-compatible regular expression string containing subpattern definition(s).

Details

regExprStr is a regular expression with successive subpattern definitions, such as shown in the examples. (Subpatterns are regular expressions within parentheses.)

For unmatched subpatterns, the corresponding substring is set to `""`. If you specify more substring parameters than subpatterns, the extra parameters are also set to `""`.

The number of matched subpatterns is returned in V_flag.

The part of *str* that matches *regExprStr* (often all of *str*) is stored in S_value.

Examples

```
// Split the output of the date() function:
Print date()
  Mon, May 2, 2005

String expr="([[:alpha:]]+), ([[:alpha:]]+) ([[:digit:]]+), ([[:digit:]]+)"
String dayOfWeek, monthName, dayNumStr, yearStr
SplitString/E=(expr) date(), dayOfWeek, monthName, dayNumStr, yearStr
Print V_flag
  4
Print dayOfWeek
  Mon
Print monthName
  May
Print dayNumStr
  2
Print yearStr
```

```
  2005
Print S_value
  Mon, May 2, 2005

// Get the part of str that matches regExprStr
SplitString/E=",.*," "stuff in front,second value,stuff at end"
Print S_value
  ,second value,
```

See Also

Regular Expressions on page IV-164 and **Subpatterns** on page IV-175.

sscanf, **Grep**, **strsearch**, **str2num**, **RemoveEnding**, **TrimString**

SplitWave

SplitWave [*flags*] *srcWave*

The SplitWave operation creates new waves containing subsets of the data in *srcWave* which must be 2D or greater.

The newly generated waves have lower dimensionality than *srcWave*. The operation is ideal for splitting 2D waves into constituent columns, 3D waves into their layers, etc.

Added in Igor Pro 7.00.

Flags

/DDF=*destDataFolder*	Specifies the data folder where the generated waves are created. If the data folder does not exist the operation creates it. If the /DDF flag is not used, output goes into the current data folder.
/FREE	Generates free output waves. The /OREF flag must also be used when the /FREE flag is used. When you use this flag there is no need to use either /N or /NAME.
/N=*baseName*	Provides the base name for all output waves. The waves will be named sequentially, i.e., baseName0, baseName1...
/NAME=*strList*	*strList* is a semicolon-separated list of wave names to be used as the names of the output waves. If *strList* contains fewer names than the number needed, the operation terminates and returns an error. If the output data folder is the data folder containing *srcWave* then *strList* must not contain the name of *srcWave*. Only simple names, not full paths, are allowed in *strList*.
/O	Permits overwriting of existing destination waves. Overwriting *srcWave* is not permitted.
/OREF=*waveRefWave*	*waveRefWave* is a wave reference wave. SplitWave stores a wave reference for each of the output waves in *waveRefWave*. If the specified *waveRefWave* already exists it is overwritten and its size is changed as appropriate. If it does not already exist, it is created by the operation.
/SDIM=*n*	Specifies the dimensionality of the output waves. By default this is 1 less than the dimensionality of *srcWave*. The minimum value is *n*=1 which results in 1D output waves.
/Z[=*z*]	/Z or /Z=1 prevents procedure execution from aborting if there is an error. Use /Z if you want to handle this case in your procedures rather than having execution abort. /Z=0: Same as no /Z at all. This is the default. /Z=1: Same as /Z alone.

Details

The SplitWave operation is in some ways the inverse of the Concatenate operation. *srcWave* is decomposed into waves of lower dimensionality.

Splitting a 2D 10x15 wave results in 15 waves of 10 rows each.

Splitting a 3D 10x15x4 using /SDIM=2 results in 4 2D waves of dimension 10x15.

Splitting a 3D 10x15x4 using /SDIM=1 results in 60 1D waves of 10 rows each.

The SplitWave operation works on all wave types. *srcWave* must be 2D or greater.

The operation creates the string variable S_waveNames which contains a semicolon separated list of the names of the output waves. However if you use /FREE then S_waveNames will be empty as free waves can not be accessed by name; use /OREF to access the created waves.

Examples

```
// Create sample input
Make/N=(5,4,3,2) wave1 = p + 10*q + 100*r + 1000*s

// Split chunks into 2 3D waves and store them in data folder Chunks
SplitWave/DDF=Chunks/N=chunk wave1

// Split layers into 6 2D waves and store them in data folder Layers
SplitWave/DDF=Layers/N=Layers/SDIM=2 wave1

// Split into 24 1D waves and store them in data folder Columns
SplitWave/DDF=Columns/N=Columns/SDIM=1 wave1
```

See Also

Duplicate, **Redimension**, **Concatenate**

sprintf

sprintf *stringName*, *formatStr* [**,** ***parameter***]…

The sprintf operation is the same as printf except it prints the formatted output to the string variable *stringName* rather than to the history area.

Parameters

formatStr	See printf.
parameter	See printf.
stringName	Results are “printed” into the named string variable.

See Also

The **printf** operation for complete format and parameter descriptions and **Creating Formatted Text** on page IV-244.

sqrt

sqrt(*num*)

The sqrt function returns the square root of *num* or NaN if *num* is negative.

In complex expressions, *num* is complex, and sqrt(*num*) returns the complex value x + *i*y.

sscanf

sscanf *scanStr*, *formatStr*, *var* [**,** ***var***]

The sscanf operation is useful for parsing text that contains numeric or string data. It is based on the C sscanf function and provides a subset of the features available in C.

Here is a trivial example:

```
Variable v1
sscanf "Value= 1.234", "Value= %f", v1
```

This skips the text “Value=” and the following space and then converts the text “1.234” (or whatever number appeared there) into a number and stores it in the local variable v1.

The sscanf operation sets the variable V_flag to the number of values read. You can use this as an initial check to see if the *scanStr* is consistent with your expectations.

Note: The sscanf operation is supported in user functions only. It is not available using the command line, using a macro, or using the Execute operation.

Parameters

scanStr contains the text to be parsed.

formatStr is a format string which describes how the parsing is to be done.

formatStr is followed by the names of one or more local numeric or string variables or NVARs (references to global numeric variables) or SVARs (references to global string variables), which are represented by *var* above.

sscanf can handle a maximum of 100 *var* parameters.

Details

The format string consists of the following:

- Normal text, which is anything other than a percent sign ("%") or white space.
- White space (spaces, tabs, linefeeds, carriage returns).
- A percent ("%") character, which is the start of a conversion specification.

The trivial example illustrates all three of these components.

```
Variable v1
sscanf "Value= 1.234", "Value= %f", v1
```

sscanf attempts to match normal text in the format string to the identical normal text in the scan string. In the example, the text "Value=" in the format string skips the identical text in the scan string.

sscanf matches a single white space character in the format string to 0 or more white space characters in the scan string. In the example, the single space skips the single space in the scan string.

When sscanf encounters a percent character in the format string, it attempts to convert the corresponding text in the scan string into a number or string, depending on the conversion character following the percent, and stores the resulting number or string in the corresponding variable in the parameter list. In the example, "%f" converts the text "1.234" into a number which it stores in the local variable v1.

A conversion specification consists of:

- A percent character ("%").
- An optional "*", which is a conversion suppression character.
- An optional number, which is a maximum field width.
- A conversion character, which specifies how to interpret text in the scan string.

Don't worry about the suppression character and the maximum width specification for now. They will be explained later.

The sscanf operation supports a subset of the conversion characters supported by the C sscanf operation. The supported conversion characters, which are case-sensitive, are:

d	Converts text representing a decimal number into an integer numeric value.
i	Converts text representing a decimal, octal or hexadecimal number into an integer value. If the text starts with "0x" (zero-x), it is interpreted as hexadecimal. Otherwise, if it starts with "0" (zero), it is interpreted as octal. Otherwise it is interpreted as decimal.
o	Converts text representing an octal number into an integer numeric value.
u	Converts text representing an unsigned decimal number into an integer numeric value.
x	Converts text representing a hexadecimal number into an integer numeric value.
c	Converts a single character into an integer value which is the ASCII code representing that character.
e	Converts text representing a decimal number into a floating point numeric value.
f	Same as e.

g	Same as e.
s	Stores text up to the next white space into a string.
[	Stores text that matches a list of specific characters into a string. The list consists of the characters inside the brackets ("%[abc]"). If the first character is "^", this means to match any character that is **not** in the list. You can specify a range of characters to match. For example `"%[A-Z]"` matches all of the upper case letters and `"%[A-Za-z]"` matches all of the upper and lower case letters.

Here are some simplified examples to illustrate each of these conversions.

```
Variable v1
String s1
```

Convert text representing a decimal number to an integer value:

```
sscanf "1234", "%d", v1
```

Convert text representing a decimal, octal, or hexadecimal number:

```
sscanf "1.234", "%i", v1          // Convert from decimal.
sscanf "01234", "%i", v1          // Convert from octal.
sscanf "0x123", "%i", v1          // Convert from hex.
```

Convert text representing an octal number:

```
sscanf "1234", "%o", v1
```

Convert text representing an unsigned decimal number:

```
sscanf "1234", "%u", v1
```

Convert text representing a hexadecimal number:

```
sscanf "1FB9", "%x", v1
```

Convert a single character:

```
sscanf "A", "%c", v1
```

Convert text representing a decimal number to an floating point value:

```
sscanf "1.234", "%e", v1
sscanf "1.234", "%f", v1
sscanf "1.234", "%g", v1
```

Copy a string of text up to the first white space:

```
sscanf "Hello There", "%s", s1
```

Copy a string of text matching the specified characters:

```
sscanf "+4.27", "%[+-]", s1
```

In a C program, you will sometimes see the letters "l" (ell) or "h" between the percent and the conversion character. For example, you may see "%lf" or "%hd". These extra letters are not needed or tolerated by Igor's sscanf operation.

When sscanf matches the format string to the scan string, it reads from the scan string until a character that would be inappropriate for the section of the format string that sscanf is trying to match. In the following example, sscanf stops reading characters to be converted into a number when it hits the first character that is not appropriate for a number.

```
Variable v1
String s1, s2
sscanf "1234Volts DC", "%d%s %s", v1, s1, s2
```

sscanf stops matching text for "%d" when it hits "V" and stores the converted number in v1. It stops matching text for the first "%s" when it hits white space and stores the matched text in s1. It then skips the space in the scan string because of the corresponding space in the format string. Finally, it matches the remaining text to the second "%s" and stores the text in s2.

The maximum field width must appear just before the conversion character ("d" in this case).

```
Variable v1, v2
sscanf "12349876", "%4d%4d", v1, v2
```

The suppression character ("*") is used in a conversion specification to skip values in the scan string. It parses the value, but sscanf does not store the value in any variable. In the following example, we read one number into local variable v1, skip a colon, and read another number into local variable v2, skip a colon, and read another number into local variable v3.

```
Variable v1, v2, v3
sscanf "12:30:45", "%d%*[:]%d%*[:]%d", v1, v2, v3
```

Here "%*[:]" means "read a colon character but don't store it anywhere". The "*" character must appear immediately after the percent. Note that there is nothing in the parameter list corresponding to the suppressed strings.

If the text in the scan string is not consistent with the text in the format string, sscanf may not read all of the values that you expected. You can check for this using the V_flag variable, which is set to the number of values read. This kind of inconsistency does not cause sscanf to return an error to Igor, which would cause procedure execution to abort. It is a situation that you can deal with in your procedure code.

The sscanf operation returns the following kinds of errors:

- Out-of-memory.
- The number of parameters implied by *formatStr* does not match the number of parameters in the *var* list.
- *formatStr* calls for a numeric variable but the parameter list expects a string variable.
- *formatStr* calls for a string variable but the parameter list expects a numeric variable.
- *formatStr* includes an unsupported, unknown or incorrectly constructed conversion specification.
- The *var* list references a global variable that does not exist.

Examples

Here is a simple example to give you the general idea:

```
Function SimpleExample()
   Variable v1, valuesRead
   sscanf "Value=1.234", "Value=%g", v1
   valuesRead = V_flag
   if (valuesRead != 1)
      Printf "Error: Expected 1 value, got %d values\r", valuesRead
   else
      Printf "Value read = %g\r", v1
   endif
End
```

For an example that uses sscanf to load data from a text file, see the Load File Demo example in "Igor Pro 7 Folder:Examples:Programming".

See Also

str2num, **strsearch**, **StringMatch**, **SplitString**

Stack

`Stack [`*`flags`*`] [`*`objectName`*`][, `*`objectName`*`]…`

The Stack operation stacks the named layout objects in the top page layout.

Parameters

objectName is the name of a graph, table, picture or annotation object in the top page layout.

Flags

/A=(*rows*,*cols*)	/I	/O=*objTypes*	/S
/G=*grout*	/M	/R	/W=(*left*,*top*,*right*,*bottom*)

See Also

The **Tile** operation for details on the flags and parameters.

StackWindows

`StackWindows` `[`***`flags`***`] [`***`windowName`***` [, `***`windowName`***`]…]`

The StackWindows operation stacks the named windows on the desktop.

Flags

/A=(*rows*,*cols*)	/G=*grout*	/O=*objTypes*	/P	/W=(*left*,*top*,*right*,*bottom*)
/C	/I	/M	/R	

See Also

See the **TileWindows** operation for details on the flags and parameters.

StartMSTimer

`StartMSTimer`

The StartMSTimer function creates a new microsecond timer and returns a timer reference number.

Details

You can create up to ten different microsecond timers using StartMSTimer. A valid timer reference number is a number between 0 and 9. If StartMSTimer returns -1, there are no free timers available. StartMSTimer works in conjunction with StopMSTimer.

See Also

The **StopMSTimer** and **ticks** functions.

Static

`Static constant` ***`objectName`*** **`=`** ***`value`***

`Static strconstant` ***`objectName`*** **`=`** ***`value`***

`Static Function` ***`funcName`*** **`()`**

`Static Structure` ***`structureName`***

`Static Picture` ***`pictName`***

The Static keyword specifies that a constant, user-defined function, structure, or Proc Picture is local to the procedure file in which it appears. Static objects can only be used by other functions; they cannot be accessed from macros; they cannot be accessed from other procedure files or from the command line.

See Also

Static Functions on page IV-96, **Proc Pictures** on page IV-53, and **Constants** on page IV-47.

StatsAngularDistanceTest

`StatsAngularDistanceTest` `[`***`flags`***`] [`***`srcWave1, srcWave2, srcWave3`***`…]`

The StatsAngularDistanceTest operation performs nonparametric tests on the angular distance between sample data and reference directions for two or more samples in individual waves. The angular distance is the shortest distance between two points on a circle (in radians). Specify the sample waves using /WSTR or by listing them following the flags. Set reference directions with /ANG, /ANGW, or the sample mean direction.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/ANG={*d1*, *d2*}	Sets reference directions (in radians) for two samples; for more than two samples use /ANGW.
/ANGM	Computes the mean direction of each sample and uses it as the reference direction.
/ANGW=*dWave*	Sets reference directions (in radians) for more than two samples using directions in *dWave*, which must be single or double precision.
/APRX=*m*	Controls the approximation method for computing the P-value in the case of two samples (Mann-Whitney Wilcoxon). See **StatsWilcoxonRankTest** for more details. The default value is 0, which may require long computation times if your sample size is large. Use /APRX=1 if you have a large sample and you expect ties in the data.

/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

/TAIL=*tail* — *tail* is a bitwise parameter that specifies the tails tested.

Bit 0:	Lower tail.
Bit 1:	Upper tail (default).
Bit 2:	Two tail.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

The P value corresponding to the last tail calculated will be entered in the table.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The inputs for StatsAngularDistanceTest are two or more waves each corresponding to individual sample. The waves must be single or double precision expressing the angles in radians. There is no restriction on the number of points or dimensionality of the waves but the data should not contain NaNs or INFs. We recommend that you use double precision waves, especially if there are ties in the data. The reference directions should also be in radians. For two samples, StatsAngularDistanceTest computes the angular distances between the input data and the reference directions and then uses the Mann-Whitney-Wilcoxon test (**StatsWilcoxonRankTest**). Results are stored in the W_WilcoxonTest wave and in the corresponding table. For more than two samples, StatsAngularDistanceTest uses the Kruskal-Wallis test, storing results in the wave W_KWTestResults wave in the current data folder.

V_flag will be set to -1 for any error and to zero otherwise.

References

See, in particular, Chapter 27 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWilcoxonRankTest** and **StatsKWTest**.

Examples:Statistics:Circular Statistics:AngularDistanceTest.pxp.

StatsANOVA1Test

`StatsANOVA1Test` `[`***`flags`***`] [`***`wave1, wave2,… wave100`***`]`

The StatsANOVA1Test operation performs a one-way ANOVA test (fixed-effect model). The standard ANOVA test results are stored in the M_ANOVA1 wave in the current data folder.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/BF	Performs the Brown and Forsythe test computing F'' and degrees of freedom. The W_ANOVA1BnF wave in the current data folder contains the output.
/Q	No results printed in the history area.
/T=*k*	Displays results in a table; additional tables are created with /BF and /W.

k specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

/W	Performs the Welch test F' and computes degrees of freedom. The W_ANOVA1Welch wave in the current data folder contains the output.
/WSTR=*waveListString*	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Inputs to StatsANOVA1Test are two or more 1D numerical waves containing (one wave for each group of samples). Use NaN for missing entries or use waves with different numbers of points. The standard ANOVA results are in the M_ANOVA1 wave with corresponding row and column labels. Use /T to display the results in a table. In each case you will get the two degrees of freedom values, the F value, the critical value Fc for the choice of alpha and the degrees of freedom, and the P-value for the result. V_flag will be set to -1 for any error and to zero otherwise.

In some cases the ANOVA test may not be appropriate. For example, if groups do not exhibit sufficient homogeneity of variances. Although this may not be fatal for the ANOVA test, you may get more insight by performing the variances test in **StatsVariancesTest**.

If there are only two groups this test should be equivalent to **StatsTTest**.

You can evaluate the power of an ANOVA test for a given set of degrees of freedom and noncentrality parameter using:

```
power=1-StatsNCFCDF(StatsInvFCDF((1-alpha),n1,n2),n1,n2,delta)
```

Here n1 is the Groups' degrees of freedom, n2 is the Error degrees of freedom, and delta is the noncentrality parameter. For more information see ANOVA Power Calculations Panel and the associated example experiment.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsVariancesTest**, **StatsTTest**, **StatsNCFCDF**, and **StatsInvFCDF**.

StatsANOVA2NRTest

`StatsANOVA2NRTest [`*`flags`*`]` ***`srcWave`***

The StatsANOVA2NRTest operation performs a two-factor analysis of variance (ANOVA) on the data that has no replication where there is only a single datum for every factor level. *srcWave* is a 2D wave of any numeric type. Output is to the M_ANOVA2NRResults wave in the current data folder or optionally to a table.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/FOMD	Estimates one missing value. You will also have to use a single or double precision wave for *srcWave* and designate the single missing value as NaN. The estimated value is printed to the history as well as the bias used to correct the sum of the squares of factor A.
/INT=*val*	Sets the degree of interactivity.

Sets the degree of interactivity.

val=0:	No interaction between the factors (default).
val=1:	Significant interaction effect between factors.

Combination with /MODL determines which factors to test:

val	Model 1	Model 2	Model 3
1		A&B	A
0	A&B	A&B	A&B

As indicated in the table, factor B is not tested for significant interaction under Model 3 and neither factor A nor factor B are tested for Model 1. If you are willing to accept an increase in Type II error you can obtain the relevant values by specifying Model 2. None of the models support a test for interaction A x B.

/MODL=*m* Sets the model number.

m=1:	Factor A and factor B are fixed.
m=2:	Both factors are random.
m=3:	Factor A is fixed and factor B is random (default).

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Input to StatsANOVA2NRTest is a 2D wave in which the Factor A corresponds to rows and Factor B corresponds to columns. H0 provides that there is no difference in the means of the respective populations, i.e., if H0 is rejected for Factor A but accepted for Factor B that means that there is no difference in the means of the columns but the means of the rows are different.

NaN and INF entries are not supported although you may use a single NaN value in combination with the /FOMD flag. If srcWave contains dimension labels they will be used to designate the two factors in the output.

The contents of the M_ANOVA2NRResults output wave columns are as follows:

Column 0	Sum of the squares (SS) values
Column 1	Degrees of freedom (DF)
Column 2	Mean square (MS) values
Column 3	Computed F value for this test
Column 4	Critical F value (Fc) for the specified alpha
Column 5	Conclusion with 0 to reject H0 or 1 to accept it

The variable V_flag is set to zero if the operation succeeds or to -1 otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test** and **StatsANOVA2Test**.

StatsANOVA2RMTest

`StatsANOVA2RMTest` `[`***`flags`***`]` ***`srcWave`***

The StatsANOVA2RMTest operation performs analysis of variance (ANOVA) on *srcWave* where replicates consist of multiple measurements on the same subject (repeated measures). *srcWave* is a 2D wave of any numeric type. Output is to the M_ANOVA2RMResults wave in the current data folder or optionally to a table.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Input to StatsANOVA2RMTest is the 2D *srcWave* in which the factor A (Groups) are columns and the different subjects are rows. It does not support NaNs or INFs.

The contents of the M_ANOVA2RMResults output wave columns are: the first contains the sum of the squares (SS) values, the second contains the degrees of freedom (DF), the third contains the mean square (MS) values, the fourth contains the single F value for this test, the fifth contains the critical F value for the specified alpha and degrees of freedom, and the last column contains the conclusion with 0 to reject H_0 or 1 to accept it. In each case H_0 corresponds to the mean level, which is the same for all subjects.

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA2NRTest** and **StatsANOVA2Test**.

StatsANOVA2Test

`StatsANOVA2Test` `[`***`flags`***`]` ***`srcWave`***

The StatsANOVA2Test operation performs a two-factor analysis of variance (ANOVA) on *srcWave*. Output is to the M_ANOVA2Results wave in the current data folder or optionally to a table.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/FAKE=*num*	Specifies the number of points in *srcWave* obtained by “estimation”. *num* is subtracted from the total and error degrees of freedom.
/MODL=*m*	Sets the model number. *m*=1: Factor A and factor B are fixed. *m*=2: Both factors are random. *m*=3: Factor A is fixed and factor B is random (default).
/Q	No results printed in the history area.

/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Input to StatsANOVA2Test is the single or double precision 3D *srcWave* in which the factor A levels are columns, the factor B levels are rows, and the replicates are layers. If *srcWave* contains dimension labels they will be used to designate the factors in the output.

Ideally, the number of replicates must be equal for each factor and each level. StatsANOVA2Test supports both equal replication and proportional replication. Proportional replication allows for different number of data in each cell with missing data represented as NaN and the number of points in each cell is given by

```
Nij=(sum of data in row i)*(sum of data in column j)/number of samples.
```

If you have no replicates (a single datum per cell) use **StatsANOVA2NRTest** instead. If the number of replicates in your data does not satisfy these conditions you may be able to "estimate" additional replicates using various methods. In that case use the /FAKE flag so that the operation can account for the estimated data by reducing the total and error degrees of freedom. /FAKE only accounts for the number of estimates being used. You must provide an appropriate number of estimated values.

The contents of the M_ANOVA2Results output wave columns are: the first contains the sum of the squares (SS) values, the second the degrees of freedom (DF), the third contains the mean square (MS) values, the fourth contains the computed F value for this test, the fifth contains the critical Fc value for the specified alpha and degrees of freedom, and the last contains the conclusion with 0 to reject H_0 or 1 to accept it. In each case H_0 corresponds to the mean level, which is the same for all populations.

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test** and **StatsANOVA2NRTest**.

StatsBetaCDF

`StatsBetaCDF(x, p, q [, a, b])`

The StatsBetaCDF function returns the beta cumulative distribution function

$$F(x,p,q,a,b)=\frac{1}{B(p,q)}\int_0^{\frac{x-a}{b-a}} t^{p-1}(1-t)^{q-1}\,dt, \qquad \begin{matrix} p,q>0 \\ a \le x \le b \end{matrix}$$

where *B*(*p*,*q*) is the beta function

$$B(p,q)=\int_0^1 t^{p-1}(1-t)^{q-1}\,dt.$$

The defaults (*a*=0 and *b*=1) correspond to the standard beta distribution were *a* is the location parameter, (*b*-*a*) is the scale parameter, and *p* and *q* are shape parameters.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBetaPDF** and **StatsInvBetaCDF**.

StatsBetaPDF

`StatsBetaPDF(x, p, q [, a, b])`

The StatsBetaPDF function returns the beta probability distribution function

$$f(x;p,q,a,b)=\frac{(x-a)^{p-1}(b-x)^{q-1}}{B(p,q)(b-a)^{p+q-1}}, \qquad \begin{matrix} a\le x\le b \\ p,q>0 \end{matrix}$$

where $B(p,q)$ is the beta function

$$B(p,q)=\int_0^1 t^{p-1}(1-t)^{q-1}\,dt.$$

The defaults (a=0 and b=1) correspond to the standard beta distribution were a is the location parameter, (b-a) is the scale parameter, and p and q are shape parameters. When p<1, $f(x=a)$ returns Inf.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBetaCDF** and **StatsInvBetaCDF**.

StatsBinomialCDF

`StatsBinomialCDF(x, p, N)`

The StatsBinomialCDF function returns the binomial cumulative distribution function

$$F(x;p,N)=\sum_{i=1}^{x}\binom{N}{i}p^i(1-p)^{N-i}, \qquad x=1,2,...$$

where

$$\binom{N}{i}=\frac{N!}{i!(N-i)!}.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBinomialCDF** and **StatsBinomialPDF**.

StatsBinomialPDF

`StatsBinomialPDF(x, p, N)`

The StatsBinomialPDF function returns the binomial probability distribution function

$$f(x;p,N)=\binom{N}{x}p^x(1-p)^{N-x}, \qquad x=0,1,2,...$$

where

$$\binom{N}{x}=\frac{N!}{x!(N-x)!}.$$

is the probability of obtaining x good outcomes in N trials where the probability of a single successful outcome is p.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBinomialCDF** and **StatsInvBinomialCDF**.

StatsCauchyCDF

`StatsCauchyCDF(x, μ, σ)`

The StatsCauchyCDF function returns the Cauchy-Lorentz cumulative distribution function

$$F(x;\mu,\sigma)=\frac{1}{2}+\frac{1}{\pi}\tan^{-1}\left(\frac{x-\mu}{\sigma}\right).$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCauchyCDF** and **StatsCauchyPDF**.

StatsCauchyPDF

`StatsCauchyPDF(x, μ, σ)`

The StatsCauchyPDF function returns the Cauchy-Lorentz probability distribution function

$$f(x;\mu,\sigma)=\frac{1}{\sigma\pi}\frac{1}{1+\left(\frac{x-\mu}{\sigma}\right)^2},$$

where μ is the location parameter and σ is the scale parameter. Use μ=0 and σ=1 for the standard form of the Cauchy-Lorentz distribution.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCauchyCDF** and **StatsInvCauchyCDF**.

StatsChiCDF

`StatsChiCDF(x, n)`

The StatsChiCDF function returns the chi-squared cumulative distribution function for the specified value and degrees of freedom *n*.

$$F(x;n)=\frac{\gamma\left(\frac{n}{2},\frac{x}{2}\right)}{\Gamma\left(\frac{n}{2}\right)}.$$

where is γ(*a,b*) the incomplete gamma function. The distribution can also be expressed as

$$F(x;n)=1-gammq\left(\frac{n}{2},\frac{x}{2}\right).$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsChiPDF**, **StatsInvChiCDF**, and **gammq**.

StatsChiPDF

`StatsChiPDF(x, n)`

The StatsChiPDF function returns the chi-squared probability distribution function for the specified value and degrees of freedom as

$$f(x;n) = \frac{\exp\left(-\frac{x}{2}\right)x^{\frac{n}{2}-1}}{2^{\frac{n}{2}}\Gamma\left(\frac{n}{2}\right)}.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsChiCDF** and **StatsChiPDF**.

StatsChiTest

`StatsChiTest [`*`flags`*`]` ***`srcWave1, srcWave2`***

The StatsChiTest operation computes a χ^2 statistic for comparing two distributions or a χ^2 statistic for comparing a sample distribution with its expected values. In both cases the comparison is made on a bin-by-bin basis. Output is to the W_StatsChiTest wave in the current data folder or optionally to a table.

Flags

/ALZR	Allows zero entries in source waves. If you are using /S zero entries in *srcWave2* are skipped.
/NCON=*nCon*	Specifies the number of constraints (0 by default), which reduces the number degrees of freedom and the critical value by *nCon*.
/S	Sets the calculation mode to a single distribution where *srcWave1* represents an array of binned measurements and *srcWave2* represents the corresponding expected values.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The source waves, *srcWave1* and *srcWave2*, must have the same number of points and can be any real numeric data type. Any nonpositive values (including NaN) in either wave removes the entry in both waves from consideration and reduces the degrees of freedom by one. The number degrees of freedom is initially the number of points in *srcWave1*-1-*nCon*. By default it is assumed that *srcWave1* and *srcWave2* represent two distributions of binned data.

When you specify /S, *srcWave1* must consist of binned values of measured data and *srcWave2* must contain the corresponding expected values. The calculation is:

$$\chi^2 = \sum_{i=0}^{n-1} \frac{\left(Y_i - V_i\right)^2}{V_i}.$$

Here Y_i is the sample point from *srcWave1*, V_i is the expected value of Y_i based on an assumed distribution (*srcWave2*), and n is the number of points in the each wave. If you do not use /S, it calculates:

$$\chi^2 = \sum_{i=0}^{n-1} \frac{\left(Y_{1i} - Y_{2i}\right)^2}{Y_{1i} + Y_{2i}},$$

where Y_{1i} and Y_{2i} are taken from *srcWave1* and *srcWave2* respectively.

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsContingencyTable**.

StatsCircularCorrelationTest

```
StatsCircularCorrelationTest [flags] waveA, waveB
```

The StatsCircularTwoSampleTest operation peforms a number of tests for two samples of circular data. Using the appropriate flags you can choose between parametric or nonparametric, unordered or paired tests. The input consists of two waves that contain one or two columns. The first column contains angle data expressed in radians and an optional second column contains associated vector lengths. The waves must be either single or double precision floating point. Results are stored in the W_StatsCircularCorrelationTest wave in the current data folder and optionally displayed in a table. Some flags generate additional outputs, described below.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/NAA	Performs a nonparametric angular-angular correlation test.
/PAA	Performs a parametric angular-angular correlation test.
/PAL	Performs a parametric angular-linear correlation test. In this case the angle wave is *waveA* and the linear data corresponds to *waveB*.
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors.

Details

The nonparametric test (/NAA) follows Fisher and Lee's modification of Mardia's statistic, which is an analogue of Spearman's rank correlation. The test ranks the angles of each sample and computes the quantities *r*' and *r*'' as follows:

$$r' = \frac{\left\{\sum_{i=0}^{n-1}\cos\left[\frac{2\pi}{n}\left(r_{ai}-r_{bi}\right)\right]\right\}^2 + \left\{\sum_{i=0}^{n-1}\sin\left[\frac{2\pi}{n}\left(r_{ai}-r_{bi}\right)\right]\right\}^2}{n^2},$$

$$r'' = \frac{\left\{\sum_{i=0}^{n-1}\cos\left[\frac{2\pi}{n}\left(r_{ai}+r_{bi}\right)\right]\right\}^2 + \left\{\sum_{i=0}^{n-1}\sin\left[\frac{2\pi}{n}\left(r_{ai}+r_{bi}\right)\right]\right\}^2}{n^2}.$$

Here n is the number of data pairs and r_{ai} and r_{bi} are the ranks of the *i*th member in the first and second samples respectively.

The test statistic is $(n-1)(r'-r'')$, which is compared with the critical value (for one and two tails). The CDF of the statistic is a highly irregular function. The critical value is computed by a different methods according to n. For $3 \le n \le 8$, a built-in table of CDF transitions gives a "conservative" estimate of the critical value. For $9 \le n \le 30$, the CDF is approximated by a 7th order polynomial in the region $x > 0$. For $n \ge 30$, the CDF is from the asymptotic expression. For $3 \le n \le 30$, CDF values are obtained by Monte-Carlo simulations using 1e6 random samples for each n.

The parametric test for angular-angular correlation (/PAA) involves computation of a correlation coefficient r_{aa} and then evaluating the mean $\overline{r_{aa}}$ and variance $s^2_{r_{aa}}$ of equivalent correlation coefficients computed from the same data but by deleting a different pair of angles each time. The mean and variance are then used to compute confidence limits L1 and L2:

$$L1 = nr_{aa} - (n-1)\overline{r_{aa}} - Z_{\alpha(2)}\sqrt{\frac{s^2_{r_{aa}}}{n}},$$

$$L2 = nr_{aa} - (n-1)\overline{r_{aa}} + Z_{\alpha(2)}\sqrt{\frac{s^2_{r_{aa}}}{n}}$$

where $Z_{\alpha(2)}$ is the normal distribution two-tail critical value at the a level of significance. H_0 (corresponding to no correlation) is rejected if zero is not contained in the interval [L1,L2].

The parametric test for angular-linear correlation (/PAL) involves computation of the correlation coefficient r_{al} which is then compared with a critical value from χ^2 for alpha significance and two degrees of freedom.

$$r_{al} = \sqrt{\frac{r^2_{xc} + r^2_{xs} - 2r_{xc}r_{xs}r_{cs}}{1 - r^2_{cs}}},$$

where:

$$r_{xc} = \frac{\sum_{i=0}^{n-1} X_i \cos(a_i) - \frac{1}{n}\sum_{i=0}^{n-1} X_i \sum_{i=0}^{n-1}\cos(a_i)}{\sqrt{\left(\sum_{i=0}^{n-1} X_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1} X_i\right)^2\right)\left(\sum_{i=0}^{n-1}\cos^2(a_i) - \frac{1}{n}\left(\sum_{i=0}^{n-1}\cos(a_i)\right)^2\right)}},$$

$$r_{xs} = \frac{\sum_{i=0}^{n-1} X_i \sin(a_i) - \frac{1}{n}\sum_{i=0}^{n-1} X_i \sum_{i=0}^{n-1}\sin(a_i)}{\sqrt{\left(\sum_{i=0}^{n-1} X_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1} X_i\right)^2\right)\left(\sum_{i=0}^{n-1}\sin^2(a_i) - \frac{1}{n}\left(\sum_{i=0}^{n-1}\sin(a_i)\right)^2\right)}},$$

$$r_{cs} = \frac{\sum_{i=0}^{n-1} \cos(a_i)\sin(a_i) - \frac{1}{n}\sum_{i=0}^{n-1} \sin(a_i) \sum_{i=0}^{n-1}\cos(a_i)}{\sqrt{\left(\sum_{i=0}^{n-1} \sin^2(a_i) - \frac{1}{n}\left(\sum_{i=0}^{n-1} \sin(a_i)\right)^2\right)\left(\sum_{i=0}^{n-1}\cos^2(a_i) - \frac{1}{n}\left(\sum_{i=0}^{n-1}\cos(a_i)\right)^2\right)}}.$$

References

Fisher, N.I., and A.J. Lee, Nonparametric measures of angular-angular association, *Biometrica*, *69*, 315-321, 1982.

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvChiCDF**, **StatsInvNormalCDF**, and **StatsKendallTauTest**.

StatsCircularMeans

`StatsCircularMeans [`*`flags`*`]` ***`srcWave`***

The StatsCircularMeans operation calculates the mean of a number of circular means, returning the mean angle (grand mean), the length of the mean vector, and optionally confidence interval around the mean angle. Output is to the history area and to the W_CircularMeans wave in the current data folder.

Flags

/ALPH=*val* Sets the significance level (default 0.05).

/CI Calculates the confidence interval (labeled CI_t1 and CI_t2) around the mean angle.

/NSOA Performs nonparametric second order analysis according to Moore's version of Rayleigh's test where H_0 corresponds to uniform distribution around the circle.

Moore's test ranks entries by the lengths of the mean radii (second column of the input) from smallest (rank 1) to largest (rank *n*) and then computes the statistic:

$$R' = \sqrt{\frac{\left(\frac{1}{n}\sum_{i=0}^{n-1}(i+1)\cos(a_i)\right)^2 + \left(\frac{1}{n}\sum_{i=0}^{n-1}(i+1)\sin(a_i)\right)^2}{n}},$$

where a_i are the mean angle entries (from column 1) corresponding to vector length rank (*i*+1). The critical value is obtained from Moore's distribution **StatsInvMooreCDF**.

/PSOA Perform parametric second order analysis where H_0 corresponds to no mean population direction. It assumes that the second order quantities are from a bivariate normal distribution. If this is not the case, use /NSOA above. The test statistic is:

$$F = \frac{k(k-2)}{2}\left[\frac{\bar{X}^2 S_{y^2} - 2\bar{X}\bar{Y}S_{xy} + \bar{Y}^2 S_{x^2}}{S_{x^2}S_{y^2} - S_{xy}^2}\right.$$

where

$$\bar{X} = \frac{1}{n}\sum_{i=0}^{n-1}X_i = \frac{1}{n}\sum_{i=0}^{n-1}r_i\cos(a_i),$$

$$\bar{Y} = \frac{1}{n}\sum_{i=0}^{n-1}Y_i = \frac{1}{n}\sum_{i=0}^{n-1}r_i\sin(a_i),$$

$$S_{x^2} = \sum_{i=0}^{n-1}X_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1}X_i\right)^2,$$

$$S_{y^2} = \sum_{i=0}^{n-1}Y_i^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1}Y_i\right)^2,$$

$$S_{xy} = \sum_{i=0}^{n-1}X_iY_i - \frac{1}{n}\sum_{i=0}^{n-1}X_i\sum_{i=0}^{n-1}Y_i.$$

Here *n* is the number of means in *srcWave* and the critical value is computed from the F distribution, equivalent to executing:

```
Print StatsInvFCDF(1-alpha,2,n-2)
```

/Q No results printed in the history area.

/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The *srcWave* input to StatsCircularMeans must be a single or double precision two column wave containing in each row a mean angle (radians) and the length of a mean radius (the first column contains mean angles and the second column contains mean vector lengths). *srcWave* must not contain any NaNs or INFs. The confidence interval calculation follows the procedure outlined by Batschelet.

V_flag will be set to -1 for any error and to zero otherwise.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCircularMoments**, **StatsInvMooreCDF**, and **StatsInvFCDF**.

StatsCircularMoments

```
StatsCircularMoments [flags] srcWave
```

The StatsCircularMoments operation computes circular statistical moments and optionally performs angular uniformity tests for the data in *srcWave*. The extent of the calculation is determined by the requested moment. The default results are stored in the W_CircularStats wave in the current data folder and are optionally displayed in a table. Additional results are listed under the corresponding flags.

Flags

/ALPH=*alpha*	Sets an *alpha* value for computing confidence intervals (default is 0.05).
/AXD=*p*	Designates the input as p-axial data. For example, if the input represents undirected lines then *p* =2 and the operation multiplies the angles by a factor *p* (after shifting /ORGN and accounting for /CYCL). It does not back-transform the mean or median axis.
/CYCL=*cycle*	Specifies the length of the data cycle. You do not need to do so if you are using one of the built-in modes, but this is still a useful option, as for setting the length of a particular month when using /MODE=5.
/GRPD={*start*, *delta*}	
	Computes circular statistics for grouped data. In this case *srcWave* contains frequencies or the number of events that belong to a particular angle group. There are as many groups as there are elements in *srcWave*. The first group is centered at *start* radians and each consecutive group is centered *delta* radians away. You must set both the *start* and *delta* to sensible values. *srcWave* may contain NaNs but it is an error if all values are NaN. The only other flags that work in combination with this flag are /Q, /T, and /Z.
/KUPR	Tests the uniformity of the distribution for ungrouped data using Kuiper statistic. The data are converted into a set $\{x_i\}$ by normalizing the input angles to the range [0,1], ranking the results then using the two quantities D_+ and D_- to compute the Kuiper statistic $$\mathrm{V}=\left(D_+ + D_-\right)\left(\sqrt{\mathrm{n}}+0.155+0.24/\sqrt{\mathrm{n}}\right),$$ where

$$D_{+} = \text{Max of: } \frac{1}{n} - x_0, \frac{2}{n} - x_1, \ldots, 1 - x_{n-1},$$

$$D_{-} = \text{Max of: } x_0, x_{1-\frac{1}{n}}, \ldots, x_{n-1-\frac{n-1}{n}},$$

and *n* is the number of valid points in *srcWave*. You can find the results in the wave W_CircularStats under row label “Kuiper V” and “Kuiper CDF(V)”. See Fisher and Press *et al*. for more information.

/LOS — Computes Linear Order Statistics by sorting the angle values from small to large, dividing each angle by 2π and shifting the origin so that the output range is [0,1]. The results are stored in the wave W_LinearOrderStats in the current data folder. The X scaling of the wave is set so that the offset and the delta are 1/(*n*+1) where *n* is the number of non-NaN points in the input.

/M=*moment* — Computes specified moments. By default, it computes the second order moments as well as skewness, kurtosis, median, and mean deviation. Use /M=1 for the first moment. For higher moments, both the specified moment and all the default quantities are computed.

/MODE=*mode* — Handles special types of data.

mode	**Data in *srcWave***
0	Angles in radians [0,2π]
1	Angles in radians [-π, π]
2	Angles in degrees [0,360]
3	Angles in degrees [-180,180]
4	Igor date format for one year cycles.
5	Igor date format for one month cycles.
6	Igor date format for one week cycles.
7	Igor date format for one day cycles.
8	Igor date format for one hour cycles.

/ORGN=*origin* — Specifies the origin of the data (the value corresponding to an angle of zero degrees). For example, if you are using Igor date format and you want the origin to be the first second in year YYYY, use `/ORGN=(date2secs(YYYY,1,1))`.

/Q — No results printed in the history area.

/RAYL[=*meanDirection*]

Performs the Rayleigh test for uniformity. If the “alternative” mean direction is specified (in radians), the test computes

`r0Bar=rBar cos(tBar-`*`meanDirection`*`)`

and then computes the significance probability of r0Bar. The null hypothesis H_0 corresponds to uniformity. It is rejected when r0Bar is too large. If the mean direction is not specified then r0Bar is rBar which is always calculated as part of the first moments so the operation only computes the relevant significance probability (P-Value). The critical values for both cases are computed according to Durand and Greenwood.

/SAW — Saves the translated angle data in the wave W_AngleWave in the current data folder.

/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results unless you moved the output wave to a different data folder. If the named table exists, but does not display the output wave from the current data folder, the table is renamed and a new table is created.

/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsCircularMoments is equivalent to **WaveStats** but it applies to circular data, which are distributed on the perimeter of a circle representing some period or cycle. If your data are not described by one of the built-in modes, you can specify the value of the origin (/ORGN), which is mapped to zero degrees and the size of a cycle or period.

When you use Igor date formats with the built-in modes for dates, the default origin is set to zero. The default cycle in the case of Mode 4 is 366. This is done in order to handle both leap and nonleap years. Similarly, Mode 5 uses a cycle of 31 days. Note that the internal conversion from Igor date to (year, month, day) is independent of the cycle specification and is therefore not affected by this choice. You should use the /CYCL flag if you use one of these modes with a fixed size of year or month.

The parameters listed below are computed and displayed (see row labels) in the table. Here *N* is the number of valid (non-NaN) angles $\{\theta_i\}$

$$\mathrm{C}=\sum_{i=1}^{n}\cos\theta_i$$

$$\mathrm{S}=\sum_{i=1}^{n}\sin\theta_i$$

$$R=\sqrt{C^2+S^2}$$

$$cBar=\overline{C}=C/n$$

$$sBar=\overline{S}=S/n$$

$$rBar=\overline{R}=R/n$$

$$tBar=\overline{\theta}=\begin{cases}\mathrm{atan}(S/C) & S>0, C>0\\ \mathrm{atan}(S/C)+\pi & C<0\\ \mathrm{atan}(S/C)+2\pi & S<0, C>0\end{cases}$$

$$V=1-\overline{R}$$

$$v=\sqrt{-2\log(1-V)}$$

median is the value which minimizes

$$d(\theta) = \pi - \frac{1}{n}\sum_{i=1}^{n}\left|\pi - \left|\theta_i - \bar{\theta}\right|\right|$$

mean deviation = The minimum of the last equation when θ → median.

Higher order moments are denoted with the moment number such that t3Bar is the uncentered third moment of the angle while primed quantities are relative to mean direction tBar. Using this notation

$$\widehat{\rho_2} = \frac{1}{n}\sum_{i=1}^{n}\cos 2\left(\theta_i - \bar{\theta}\right)$$

$$circular\ \ dispersion = \frac{1-\widehat{\rho_2}}{2\bar{R}^2}$$

$$skewness = \frac{\widehat{\rho_2}\sin\left(\hat{\mu}_2' - 2\bar{\theta}\right)}{\left(1-\bar{R}\right)^{3/2}}$$

$$kurtosis = \frac{\widehat{\rho_2}\cos\left(\hat{\mu}_2' - 2\bar{\theta}\right) - \bar{R}^4}{\left(1-\bar{R}\right)^2}$$

where

$$\hat{\mu}_p' = \begin{cases} \operatorname{atan}\left(S_p/C_p\right) & S_p > 0, C_p > 0 \\ \operatorname{atan}\left(S_p/C_p\right) + \pi & C_p < 0 \\ \operatorname{atan}\left(S_p/C_p\right) + 2\pi & S_p < 0, C_p > 0 \end{cases}$$

and

$$C_p = \frac{1}{n}\sum_{i=1}^{n}\cos p\theta_i, \qquad S_p = \frac{1}{n}\sum_{i=1}^{n}\sin p\theta_i.$$

References

Fisher, N.I., *Statistical Analysis of Circular Data*, 295pp., Cambridge University Press, New York, 1995.

Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

Durand, D., and J.A. Greenwood, Modifications of the Rayleigh test for uniformity in analysis of two-dimensional orientation data, *J. Geol.*, *66*, 229-238, 1958.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

WaveStats, **StatsAngularDistanceTest**, **StatsCircularCorrelationTest**, **StatsCircularMeans**, **StatsHodgesAjneTest**, **StatsWatsonUSquaredTest**, **StatsWatsonWilliamsTest**, and **StatsWheelerWatsonTest**.

StatsCircularTwoSampleTest

`StatsCircularTwoSampleTest [`*`flags`*`]` ***`waveA, waveB`***

The StatsCircularTwoSampleTest operation performs second order analysis of angles. Using the appropriate flags you can choose between parametric or nonparametric, unordered or paired tests. The input consists of two waves that contain one or two columns. The first column contains angle data (mean angles) expressed in radians and an optional second column that contains associated vector lengths. The waves must be either single or double precision. Results are stored in the W_StatsCircularTwoSamples wave in the current data folder and optionally displayed in a table. Some of the tests may have additional outputs.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/NPR	Performs nonparametric paired-sample test (Moore). The input waves must contain paired angular data so both must have single column and the same number of points.
/NSOA	Perform nonparametric second order two-sample test. Input waves must each contain two columns.
/PPR	Performs parametric paired-sample test. Input waves must contain paired data and must have the same number of points.
/PSOA	Performs parametric second order analysis of two samples. The input waves must each contain two columns.
/Q	No information printed in the history area.
/T= *k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores any errors.

Details

The nonparametric paired-sample test (/NPR) is Moore's test for paired angles applied in second order analysis. The input can consist of one or two column waves. When both waves contain a single column the operation proceeds as if all the vector length were identically 1. The Moore statistic ($H_0 \rightarrow$ pair equality) is computed and compared to the critical value from the Moore distribution (see **StatsInvMooreCDF**).

The nonparametric second-order two-sample test (/NSOA) consists of pre-processing where the grand mean is subtracted from the two inputs followed by application of Watson's U^2 test (**StatsWatsonUSquaredTest**) with H_0 implying that the two samples came from the same population. The results of this test are stored in the wave W_WatsonUtest.

The parametric paired-sample test (/PPR) is due to Hotelling. In this test the input should consist of both angular and vector length data. The test statistic is compared with a critical value from the F distribution (**StatsInvFCDF**).

The parametric second order two-sample test (/PSOA) is an extension of Hotelling one-sample test to second order analysis where an F-like statistic is computed corresponding to H_0 of equal mean angles.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvMooreCDF**, **StatsWatsonUSquaredTest**, and **StatsInvFCDF**.

StatsCMSSDCDF

`StatsCMSSDCDF(`*`C, n`*`)`

The StatsCMSSDCDF function returns the cumulative distribution function of the C distribution (mean square successive difference), which is

$$f(C,n)=\frac{\Gamma(2m+2)}{a2^{2m+1}\left[\Gamma(m+1)\right]^2}\left(1-\frac{C^2}{a^2}\right)^m,$$

where

$$a^2=\frac{\left(n^2+2n-12\right)\left(n-2\right)}{\left(n^3-13n+24\right)},$$

$$m=\frac{\left(n^4-n^3-13n^2+37n-60\right)}{2\left(n^3-13n+24\right)}.$$

The distribution (*C*>0) can then be expressed as

$$F(C,n)=\frac{\Gamma(2m+2)}{a2^{2m+1}\left[\Gamma(m+1)\right]^2}C\,{}_2F_1\left(\frac{1}{2},-m,\frac{3}{2},\frac{C^2}{a^2}\right),$$

where ${}_2F_1$ is the hypergeometric function **hyperG2F1**.

References

Young, L.C., On randomness in ordered sequences, *Annals of Mathematical Statistics, 12*, 153-162, 1941.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCMSSDCDF** and **StatsSRTest**.

StatsCochranTest

```
StatsCochranTest [flags] [wave1, wave2,… wave100]
```

The StatsCochranTest operation performs Cochran's (Q) test on a randomized block or repeated measures dichotomous data. Output is to the M_CochranTestResults wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results unless you moved the output wave to a different data folder. If the named table exists but it does not display the output wave from the current data folder, the table is renamed and a new table is created.
/WSTR=*waveListString*	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsCochranTest computes Cochran's statistic and compares it to a critical value from a Chi-squared distribution, which depends only of the significance level and the number of groups (columns). The null hypothesis for the test is that all columns represent the same proportion of the effect represented by a non-zero data.

The Chi-square distribution is appropriate when there are at least 4 columns and at least 24 total data points.

Dichotomous data are presumed to consist of two values 0 and 1, thus StatsCochranTest distinguishes only between zero and any nonzero value, which is considered to be 1; it does not allow NaNs or INFs. Input waves can be a single 2D wave or a list of 1D numeric waves, which can also be specified in a string list with /WSTR. In the standard terminology, data rows represent blocks and data columns represent groups. H_0 corresponds to the assumption that all groups have the same proportion of 1's.

With the /T flag, it displays the results in a table that contains the number of rows, the number of columns, the Cochran statistic, the critical value, and the conclusion (1 to accept H_0 and 0 to reject it).

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFriedmanTest**.

StatsContingencyTable

`StatsContingencyTable [flags] srcWave`

The StatsContingencyTable operation performs contingency table analysis on 2D and 3D tables. Output is to the W_ContingencyTableResults wave in the current data folder or optionally to a table or the history area.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/COR=*mode*	Sets the correction type for 2x2 tables. By default there is no correction. Use *mode*=1 for Yates and *mode*=2 for Haber correction.
/FEXT={*row*, *col*}	Computes Fisher's Exact P-value with 2x2 contingency tables. *row* and *col* are zero-based indices of the table entry where it computes the probability of getting the results in the table or more extreme values. Without the /Q flag, it prints the probabilities of each individual table in the history.

Given the contingency table:

	Succeeded	**Failed**
Group1	11	8
Group2	4	9

Example 1: When you use `/FEXT={0,0}` the P-value represents the sum of the probabilities of the first group having in the Succeeded column 11 or more extreme values, i.e., 12, 13, 14, and 15. In each case the remaining table elements are adjusted so that row and column sums remain constant.

Example 2: When you needed to evaluate the sum of the probabilities of Group2 having 4 counts or less in the Succeeded column, then the appropriate flag is `/FEXT={1,1}`, which effectively computes the equivalent of having 9, 10, 11, 12, and 13 Failed counts. In each case it computes the upper, the lower, and the two-tail probabilities.

/HTRG	Tests for heterogeneity between tables stored as layers of 3D wave.
/LLIK	Computes log likelihood statistic.
/Q	No results printed in the history area.

/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results unless you moved the output wave to a different data folder. If the named table exists but it does not display the output wave from the current data folder, the table is renamed and a new table is created.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsContingencyTable supports 2D waves representing single contingency tables or 3D waves representing multiple 2D tables (where each table is a layer) or a single 3D table. Each entry in the wave must contain a frequency value and must be a positive number; it does not support 0's, NaNs, or INFs. In the special case of 2x2 tables, use the /COR flag to compute the statistic using either the Yates or Haber corrections. Except for the heterogeneity option you can also compute the log likelihood statistic. In all the tests, H_0 corresponds to independence between the tested variables.

For 3D tables StatsContingencyTable provides Chi-squared, degrees of freedom, the critical value, and optionally the log likelihood G statistic (/LLIK flag) for each of the following cases:

- Mutual independence by testing if all three variables are independent of each other.
- Partial dependence (rows) by testing if rows independent of columns and layers.
- Partial dependence (columns) by testing if columns independent of rows and layers.
- Partial dependence (layers) by testing if layers independent of rows and columns.

In each case you should compare the statistic with the critical value and reject H_0 if the statistic exceeds or equals the critical value.

You should examine the table entries to determine if the Chi-square statistic is appropriate (if the frequency is smaller than 6 for /ALPH=0.05 you should consider computing the Fisher exact test).

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvChiCDF**.

StatsCorrelation

`StatsCorrelation(`*`waveA`*` [, `*`waveB`*`])`

The StatsCorrelation function computes Pearson's correlation coefficient between two real valued arrays of data of the same length. Pearson r is give by:

$$r = \frac{\sum_{i=0}^{n-1}\left(waveA[i]-A\right)\left(waveB[i]-B\right)}{\sqrt{\sum_{i=0}^{n-1}\left(waveA[i]-A\right)^2 \sum_{i=0}^{n-1}\left(waveB[i]-B\right)^2}}$$

Here *A* is the average of the elements in *waveA*, *B* is the average of the elements of *waveB* and the sum is over all wave elements.

Details

If you use both *waveA* and *waveB* then the two waves must have the same number of points but they could be of different number type. If you use only the *waveA* parameter then *waveA* must be a 2D wave. In this case StatsCorrelation will return 0 and create a 2D wave M_Pearson where the (*i,j*) element is Pearson's r corresponding to columns *i* and *j*.

Fisher's z transformation converts Person's r above to a normally distributed variable *z*:

$$z = \frac{1}{2}\ln\left(\frac{1+r}{1-r}\right),$$

with a standard error

$$\sigma_z = \frac{1}{\sqrt{n-3}}.$$

You can convert between the two representations using the following functions:

```
Function pearsonToFisher(inr)
   Variable inr
   return 0.5*(ln(1+inr)-ln(1-inr))
End

Function fisherToPearson(inz)
   Variable inz
   return tanh(inz)
End
```

See Also

Correlate, **StatsLinearCorrelationTest**, and **StatsCircularCorrelationTest**.

StatsDExpCDF

`StatsDExpCDF(`*`x, m, s`*`)`

The StatsDExpCDF function returns the double-exponential cumulative distribution function

$$F(x;\mu,\sigma) = \begin{cases} \exp\left(\dfrac{x-\mu}{\sigma}\right) & \textit{when } x<\mu \\ 1-\dfrac{1}{2}\exp\left(-\left|\dfrac{x-\mu}{\sigma}\right|\right) & \textit{when } x \ge \mu \end{cases}$$

for σ>0. It returns NaN when σ=0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsDExpPDF** and **StatsInvDExpCDF**.

StatsDExpPDF

`StatsDExpPdf(`*`x, m, s`*`)`

The StatsDExpPdf function returns the double-exponential probability distribution function

$$\mathrm{f}(\mathrm{x};\mu,\sigma)=\frac{1}{2\sigma}\exp\left[-\left|\frac{x-\mu}{\sigma}\right|\right],$$

where μ is the location parameter and σ>0 is the scale parameter. Use μ=0 and σ=1 for the standard form of the double exponential distribution. It returns NaN when σ=0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsDExpCDF** and **StatsInvDExpCDF**.

StatsDIPTest

`StatsDIPTest [/Z]` ***`srcWave`***

The StatsDIPTest operation performs Hartigan test for unimodality.

Flags

/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The input to the operation *srcWave* is any real numeric wave. Outputs are: V_Value contains the dip statistic; V_min is the lower end of the modal interval; and V_max is the higher end of the modal interval. Percentage points or critical values for the dip statistic can be obtained from simulations using an identical sample size as in this example:

```
Function getCriticalValue(sampleSize,alpha)
Variable sampleSize,alpha

   Make/O/N=(sampleSize) dataWave
   Make/O/N=100000  dipResults
   Variable i
   for(i=0;i<100000;i+=1)
      dataWave=enoise(100)
      StatsDipTest dataWave
      dipResults[i]=V_Value
   endfor
   Histogram/P/B=4 dipResults               // Compute the PDF.
   Wave W_Histogram
   Integrate/METH=1 W_Histogram/D=W_INT     // Compute the CDF.
   Findlevel/Q  W_int,(1-alpha)             // Find the critical value.
   return V_LevelX
End
```

References

Hartigan, P. M., Computation of the Dip Statistic to Test for Unimodality, *Applied Statistics, 34*, 320-325, 1985.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsDunnettTest

`StatsDunnettTest [`*`flags`*`] [`*`wave1, wave2,… wave100`*`]`

The StatsDunnettTest operation performs the Dunnett test by comparing multiple groups to a control group. Output is to the M_DunnettTestResults wave in the current data folder or optionally to a table. StatsDunnettTest usually follows StatsANOVA1Test.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/CIDX=*cIndex*	Specifies the (zero based) index of the input wave corresponding to the control group. The default is zero (the first wave corresponds to the control group).
/Q	No results printed in the history area.
/SWN	Creates a text wave, T_DunnettDescriptors, containing wave names corresponding to each row of the comparison table (Save Wave Names). Use /T to append the text wave to the last column.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

/TAIL=*tc*	Specifies H_0. *tc*=1: Default; one tailed test ($\mu_c \le \mu_a$). *tc*=2: One tailed test ($\mu_c \ge \mu_a$). *tc*=4: Two tailed test ($\mu_c = \mu_a$). Code combinations are not allowed.
/WSTR=*waveListString*	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsDunnettTest inputs are two or more 1D numeric waves (one wave for each group of samples). The input waves may contain different number of points, but they must contain two or more valid entries per wave.

For output to a table (using /T), each labelled row represents the results of the test for comparing the means of one group to the control group, and rows are ordered so that all comparisons are computed sequentially starting with the group having the smallest mean. The contents of the labeled columns are:

First	The difference between the group means
Second	SE (which is computed for possibly unequal number of points)
Third	The q statistic for the pair which may be positive or negative
Fourth	The critical q' value
Fifth	0 if the conclusion is to reject H_0 or 1 to accept H_0
Sixth	The P-value

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsTukeyTest**, **StatsANOVA1Test**, **StatsScheffeTest**, and **StatsNPMCTest**.

StatsErlangCDF

`StatsErlangCDF(x, b, c)`

The StatsErlangCDF function returns the Erlang cumulative distribution function

$$F(x;b,c) = 1 - \frac{\Gamma\left(c, \frac{x}{b}\right)}{\Gamma(c)}.$$

where *b*>0 (also as λ=1/*b*) is the scale parameter, *c*> 0 the shape parameter, Γ(*x*) the **gamma** function, and Γ(*a*,*x*) the incomplete gamma function **gammaInc**.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsErlangPDF**.

StatsErlangPDF

`StatsErlangPDF(x, b, c)`

The StatsErlangPDF function returns the Erlang probability distribution function

$$f(x;b,c)=\frac{\left(\frac{x}{b}\right)^{c-1}\exp\left(-\frac{x}{b}\right)}{b(c-1)!}.$$

where b>0 (also as λ=1/b) is the scale parameter and c> 0 the shape parameter.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsErlangCDF**.

StatsErrorPDF

`StatsErrorPDF(x, a, b, c)`

The StatsErrorPDF function returns the error probability distribution function or the exponential power distribution

$$f(x;a,b,c)=\frac{\exp\left[-\frac{1}{2}\left(\frac{|x-a|}{b}\right)^{\frac{2}{c}}\right]}{b2^{\frac{c}{2}+1}\Gamma\left(1+\frac{c}{2}\right)}.$$

where a is the location parameter, b> 0 is the scale parameter, c> 0 is the shape parameter, and $\Gamma(x)$ is the **gamma** function.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsEValueCDF

`StatsEValueCDF(x, μ, σ)`

The StatsEValueCDF function returns the extreme-value (type I, Gumbel) cumulative distribution function

$$F(x;\mu,\sigma)=1-\exp\left(-\exp\left(\frac{x-\mu}{\sigma}\right)\right),$$

where σ>0. This is also known as the “minimum” form or distribution of the smallest extreme. To obtain the distribution of the largest extreme reverse the sign of σ.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsEValuePDF, **StatsInvEValueCDF**, **StatsGEVCDF**, **StatsGEVPDF**

StatsEValuePDF

`StatsEValuePDF(x, μ, σ)`

The StatsEValuePDF function returns the extreme-value (type I, Gumbel) probability distribution function

$$F(x;\mu,\sigma)=1-\exp\left(-\exp\left(\frac{x-\mu}{\sigma}\right)\right),$$

where σ>0. This is also known as the “minimum” form or the distribution of the smallest extreme. To obtain the distribution of the largest extreme reverse the sign of σ.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsEValueCDF, **StatsInvEValueCDF**, **StatsGEVCDF**, **StatsGEVPDF**

StatsExpCDF

```
StatsExpCDF(x, μ, σ)
```

The StatsExpCDF function returns the exponential cumulative distribution function

$$F(x;\mu,\sigma) = 1 - \exp\left(-\frac{x-\mu}{\sigma}\right),$$

where $x \geq \mu$ and $\sigma > 0$. It returns NaN for $\sigma = 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsExpPDF** and **StatsInvExpCDF**.

StatsExpPDF

```
StatsExpPDF(x, μ, σ)
```

The StatsExpPDF function returns the exponential probability distribution function

$$f(x;\mu,\sigma) = \frac{1}{\sigma}\exp\left(-\frac{x-\mu}{\sigma}\right),$$

where μ is the location parameter and $\sigma>0$ is the scale parameter. Use $\mu=0$ and $\sigma=1$ for the standard form of the exponential distribution. It returns NaN for $\sigma=0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsExpCDF** and **StatsInvExpCDF**.

StatsFCDF

```
StatsFCDF(x, n1, n2)
```

The StatsFCDF function returns the cumulative distribution function for the F distribution with shape parameters *n1* and *n2*

$$F(x;n_1,n_2) = 1 - Betai\left(\frac{n_2}{2},\frac{n_1}{2},\frac{n_2}{n_2+n_1x}\right),$$

where *Betai* is the incomplete beta function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFPDF** and **StatsInvFCDF**.

StatsFPDF

```
StatsFPDF(x, n1, n2)
```

The StatsFPDF function returns the probability distribution function for the F distribution with shape parameters *n1* and *n2*

$$f(x;n_1,n_2) = \frac{\Gamma\left(\frac{n_1+n_2}{2}\right)\left(\frac{n_1}{n_2}\right)^{\frac{n_1}{2}} x^{\frac{n_1}{2}-1}}{\Gamma\left(\frac{n_1}{2}\right)\Gamma\left(\frac{n_2}{2}\right)\left(1+\frac{n_1x}{n_2}\right)^{\frac{n_1+n_2}{2}}}.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFCDF** and **StatsInvFCDF**.

StatsFriedmanCDF

`StatsFriedmanCDF(`*`x, n, m, method, useTable`*`)`

The StatsFriedmanCDF function returns the cumulative probability distribution of the Friedman distribution with *m* rows and *n* columns. The exact Friedman distribution is computationally intensive, taking on the order of $(n!)^m$ iterations. You may be able to use a range of precomputed exact values by passing a nonzero value for *useTable*, which will use *method* only if the value is not in the table. For large *m*, consider using the Chi-squared or the Monte-Carlo approximations. To abort execution, press the **User Abort Key Combinations**.

method	**What It Does**
0	Exact computation.
1	Chi-square approximation.
2	Monte-Carlo approximation.
3	Use built-table only and return NaN if not in table.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvFriedmanCDF** and **StatsFriedmanTest**.

StatsFriedmanTest

`StatsFriedmanTest` `[`***`flags`***`] [`***`wave1, wave2,… wave100`***`]`

The StatsFriedmanTest operation performs Friedman's test on a randomized block of data. It is a nonparametric analysis of data contained in either individual 1D waves or in a single 2D wave. Output is to the M_FriedmanTestResults wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/RW	Saves the ranking wave M_FriedmanRanks, which contains the rank values corresponding to each input datum.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results.
/WSTR=*waveListString*	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The Friedman test ranks the input data on a row-by-row basis, sums the ranks for each column, and computes the Friedman statistic, which is proportional to the sum of the squares of the ranks.

Input waves can be a single 2D wave or a list of 1D numeric waves, which can also be specified in a string list with /WSTR. All 1D waves must have the same number of points. A 2D wave must not contain any NaNs.

The critical value for the Friedman distribution is fairly difficult to compute when the number of rows and columns is large because it requires a number of permutations on the order of (numColumns!)^numRows. A certain range of these critical values are supported by precomputed tables. When the exact critical value is not available you can use one of the two approximations that are always computed: the Chi-squared approximation or the Iman and Davenport approximation, which converts the Friedman statistic is converted to a new value Ff then compares it with critical values from the F distribution using weighted degrees of freedom.

With the /T flag, it displays the results in a table that contains the number of rows, the number of columns, the Friedman statistic, the exact critical value (if available), the Chi-squared approximation, the Iman and Davenport approximation, and the conclusion (1 to accept H_0 and 0 to reject it).

V_flag will be set to -1 for any error and to zero otherwise.

References

Iman, R.L., and J.M. Davenport, Approximations of the critical region of the Friedman statistic, *Comm. Statist. A9*, 571-595, 1980.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFriedmanCDF** and **StatsInvFriedmanCDF**.

StatsFTest

`StatsFTest [`*`flags`*`] `*`wave1, wave2`*

The StatsFTest operation performs the F-test on the two distributions in *wave1* and *wave2*, which can be any real numeric type, must contain at least two data points each, and can have an arbitrary number of dimensions. Output is to the W_StatsFTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).	
/Q	No results printed in the history area.	
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed.	
	k=0:	Normal with dialog (default).
	k=1:	Kills with no dialog.
	k=2:	Disables killing.
/TAIL=*tc*	Specifies the tail tested.	
	tc=1:	Lower one-tail test with H_a: sigma1>sigma2.
	tc=2:	Upper one-tail test with H_a: sigma1<sigma2.
	tc=3:	Default; the null hypothesis H_0: sigma1=sigma2 with H_a: sigma1!=sigma2.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.	

Details

The F statistic is the ratio of the variance of *wave1* to the variance of *wave2*. We assume the waves have equal wave variances and that H_0 is sigma1=sigma2. For the upper one-tail test we reject H_0 if F is greater than the upper critical value or if F is smaller than the lower critical value in the lower one-tail test. In the two-tailed test we reject H_0 if F is either greater than the upper critical value or smaller than the lower critical value. The critical values are computed by numerically solving for the argument at which the cumulative distribution function (CDF) equals the appropriate values for the tests. The CDF is given by

$$F(x,n_1,n_2) = 1 - betai\left(\frac{n_2}{2},\frac{n_1}{2},\frac{n_2}{n_2+n_1x}\right),$$

where the degrees of freedom n_1 and n_2 equal the number of valid (non-NaN) points in each wave -1, and *betai* is the incomplete beta function. To get the critical value for the upper one-tail test we solve F(x)=1-

alpha. For the lower one-tail test we solve F(x)=alpha. In the two-tailed test the lower critical value is a solution for F(x)=alpha/2 and the upper critical value is a solution for F(x)=1-alpha/2.

The F-test requires that the two samples are from normally distributed populations.

See Also
Chapter III-12, **Statistics** for a function and operation overview; **StatsVariancesTest**, **StatsFCDF**, and **betai**.

StatsGammaCDF

```
StatsGammaCDF(x, μ, σ, γ)
```

The StatsGammaCDF function returns the gamma cumulative distribution function

$$F(x;\mu,\sigma,\gamma)=\frac{\Gamma_{inc}\left(\gamma,\frac{x-\mu}{\sigma}\right)}{\Gamma(\gamma)}. \qquad \begin{matrix} x\geq\mu \\ \sigma,\gamma>0 \end{matrix}$$

where Γ is the gamma function and Γ_{inc} is the incomplete gamma function **gammaInc**.

See Also
Chapter III-12, **Statistics** for a function and operation overview; **StatsGammaPDF** and **StatsInvGammaCDF**.

StatsGammaPDF

```
StatsGammaPDF(x, μ, σ, γ)
```

The StatsGammaPDF function returns the gamma probability distribution function

$$f(x;\mu,\sigma,\gamma)=\frac{\left(\frac{x-\mu}{\sigma}\right)^{\gamma-1}\exp\left(-\frac{x-\mu}{\sigma}\right)}{\sigma\Gamma(\gamma)}. \qquad \begin{matrix} x\geq\mu \\ \sigma,\gamma>0 \end{matrix}$$

where μ is the location parameter, σ is the scale parameter, γ is the shape parameter, and Γ is the gamma function.

See Also
Chapter III-12, **Statistics** for a function and operation overview; **StatsGammaCDF** and **StatsInvGammaCDF**.

StatsGeometricCDF

```
StatsGeometricCDF(x, p)
```

The StatsGeometricCDF function returns the geometric cumulative distribution function

$$F(x,p)=1-(1-p)^{x+1}.$$

where *p* is the probability of success in a single trial and *x* is the number of trials for $x \geq 0$.

See Also
Chapter III-12, **Statistics** for a function and operation overview; **StatsGeometricPDF** and **StatsInvGeometricCDF**.

StatsGeometricPDF

```
StatsGeometricPDF(x, p)
```

The StatsGeometricPDF function returns the geometric probability distribution function

$$f(x,p)=p(1-p)^{x},$$

where the p is the probability of success in a single trial and x is the number of trials $x \geq 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGeometricCDF** and **StatsInvGeometricCDF**.

StatsHodgesAjneTest

`StatsHodgesAjneTest [`*`flags`*`]` ***`srcWave`***

The StatsHodgesAjneTest operation performs the Hodges-Ajne nonparametric test for uniform distribution around a circle. Output is to the W_HodgesAjne wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/SA=*specAngle*	Uses the Batschelet modification of the Hodges-Ajne test to test for uniformity against the alternative of concentration around the specified angle. *specAngle* must be expressed in radians modulus 2π.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The input *srcWave* must contain angles in radians, can be any number of dimensions, can be single or double precision, and should not contain NaNs or INFs.

StatsHodgesAjneTest performs the standard Hodges-Ajne test, which simply tests for uniformity against the hypothesis that the population is not uniformly distributed around the circle. This test finds a diameter that divides the circle into two halves such that one contains the least number of data *m*, the test statistic.

Use /SA to perform the modified (Batschelet) test, which tests against the alternative that the population is concentrated somehow about the specified angle. The modified test counts the number of points *m*' in 90-degree neighborhoods around the specified angle. The test statistic is given by C=*n*-*m*' where *n* is the number of points in the wave. The critical value is computed from the binomial probability density.

In both cases H_0 is rejected if the statistic is smaller than the critical value.

V_flag will be set to -1 for any error and to zero otherwise.

References

Ajne, B., A simple test for uniformity of a circular distribution, *Biometrica*, *55*, 343-354, 1968.

See, in particular, Chapter 27 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsCircularMeans, **StatsCircularMoments**, **StatsWatsonUSquaredTest**, **StatsWatsonWilliamsTest**, and **StatsWheelerWatsonTest**.

StatsGEVCDF

`StatsGEVCDF(`*`x`*`, `μ`, `σ`, `ξ`)`

The StatsGEVCDF function returns the generalized extreme value cumulative distribution function.

$$F(x,\mu,\sigma,\xi) = \exp\left\{-\left[1+\xi\left(\frac{x-\mu}{\sigma}\right)^{-1/\xi}\right]\right\},$$

where

$$1+\xi\left(\frac{x-\mu}{\sigma}\right) > 0,$$

and σ>0.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsGEVPDF, **StatsEValuePDF**, **StatsEValueCDF**, **StatsInvEValueCDF**

StatsGEVPDF

`StatsGEVPDF(`*x, μ, σ, ξ*`)`

The StatsGEVPDF function returns the generalized extreme value probability distribution function.

$$f(x,\mu,\sigma,\xi) = \frac{1}{\sigma}\left[1+\xi\left(\frac{x-\mu}{\sigma}\right)\right]^{(-1/\xi)-1} \exp\left\{-\left[1+\xi\left(\frac{x-\mu}{\sigma}\right)^{-1/\xi}\right]\right\},$$

where

$$1+\xi\left(\frac{x-\mu}{\sigma}\right) > 0,$$

and σ>0.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsGEVCDF, **StatsEValuePDF**, **StatsEValueCDF**, **StatsInvEValueCDF**

StatsHyperGCDF

`StatsHyperGCDF(`*x, m, n, k*`)`

The StatsHyperGCDF function returns the hypergeometric cumulative distribution function, which is the probability of getting *x* marked items when drawing (without replacement) *k* items out of a population of *m* items when *n* out of the *m* are marked.

Details

The hypergeometric distribution is

$$F(x;m,n,k) = \sum_{L=0}^{x} \frac{\begin{pmatrix} n \\ L \end{pmatrix}\begin{pmatrix} m-L \\ k-L \end{pmatrix}}{\begin{pmatrix} m \\ k \end{pmatrix}},$$

where $\binom{a}{b}$ is the **binomial** function. All parameters must be positive integers and must have *m*>*n* and *x*<*k*; otherwise it returns NaN.

References

Klotz, J.H., *Computational Approach to Statistics*, <`http://www.stat.wisc.edu/~klotz/Book.pdf`>.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsHyperGPDF**.

StatsHyperGPDF

`StatsHyperGPDF(`*`x, m, n, k`*`)`

The StatsHyperGPDF function returns the hypergeometric probability distribution function, which is the probability of getting *x* marked items when drawing without replacement *k* items out of a population of *m* items where *n* out of the *m* are marked.

Details

The hypergeometric distribution is

$$f(x;m,n,k) = \frac{\begin{pmatrix} n \\ x \end{pmatrix}\begin{pmatrix} m-n \\ k-x \end{pmatrix}}{\begin{pmatrix} m \\ k \end{pmatrix}},$$

where $\binom{a}{b}$ is the **binomial** function. All parameters must be positive integers and must have *m*>*n* and *x*<*k*.

References

Klotz, J.H., *Computational Approach to Statistics*, <`http://www.stat.wisc.edu/~klotz/Book.pdf`>.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsHyperGCDF**.

StatsInvBetaCDF

`StatsInvBetaCDF(`*`cdf, p, q`*` [, `*`a, b`*`])`

The StatsInvBetaCDF function returns the inverse of the beta cumulative distribution function. There is no closed form expression for the inverse beta CDF; it is evaluated numerically.

The defaults (*a*=0 and *b*=1) correspond to the standard beta distribution.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBetaCDF** and **StatsBetaPDF**.

StatsInvBinomialCDF

`StatsInvBinomialCDF(`*`cdf, p, N`*`)`

The StatsInvBinomialCDF function returns the inverse of the binomial cumulative distribution function. The inverse function returns the value at which the binomial *CDF* with probability *p* and total elements *N*, has the value 0.95. There is no closed form expression for the inverse binomial CDF; it is evaluated numerically.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsBinomialCDF** and **StatsBinomialPDF**.

StatsInvCauchyCDF

`StatsInvCauchyCDF(`*`cdf,`*` μ, σ)`

The StatsInvCauchyCDF function returns the inverse of the Cauchy-Lorentz cumulative distribution function

$$x = \mu + \sigma \tan\left[\pi\left(cdf - \frac{1}{2}\right)\right].$$

It returns NaN for *cdf* <0 or *cdf*> 1.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCauchyCDF** and **StatsCauchyPDF**.

StatsInvChiCDF

`StatsInvChiCDF(`*`x, n`*`)`

The StatsInvChiCDF function returns the inverse of the chi-squared distribution of *x* and shape parameter *n*. The inverse of the distribution is also known as the percent point function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsChiCDF** and **StatsChiPDF**.

StatsInvCMSSDCDF

`StatsInvCMSSDCDF(`*`cdf, n`*`)`

The StatsInvCMSSDCDF function returns the critical values of the C distribution (mean square successive difference distribution), which is given by

$$f(C,n) = \frac{\Gamma(2m+2)}{a2^{2m+1}\left[\Gamma(m+1)\right]^2}\left(1-\frac{C^2}{a^2}\right)^m,$$

where

$$a^2 = \frac{\left(n^2+2n-12\right)\left(n-2\right)}{\left(n^3-13n+24\right)},$$

$$m = \frac{\left(n^4-n^3-13n^2+37n-60\right)}{2\left(n^3-13n+24\right)}.$$

Critical values are computed from the integral of the probability distribution function.

References

Young, L.C., On randomness in ordered sequences, *Annals of Mathematical Statistics*, *12*, 153-162, 1941.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCMSSDCDF** and **StatsSRTest**.

StatsInvDExpCDF

`StatsInvDExpCDF(`*`cdf`*`, μ, σ)`

The StatsInvDExpCDF function returns the inverse of the double-exponential cumulative distribution function

$$x = \begin{cases} \mu + \sigma \ln(2cdf) & \text{when } cdf < 0.5 \\ \mu - \sigma \ln\left[2\left(1-cdf\right)\right] & \text{when } cdf \geq 0.5 \end{cases}$$

It returns NaN for *cdf* <0 or *cdf* > 1.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsDExpCDF** and **StatsDExpPDF**.

StatsInvEValueCDF

```
StatsInvEValueCDF(cdf, μ, σ)
```

The StatsInvEValueCDF function returns the inverse of the extreme-value (type I, Gumbel) cumulative distribution function

$$x = \mu - \sigma \ln\left(1 - cdf\right)$$

where σ>0. It returns NaN for *cdf*<0 or *cdf*>1. This inverse applies to the “minimum” form of the distribution. Reverse the sign of σ to obtain the inverse distribution of the maximum form.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsEValueCDF, **StatsEValuePDF**, **StatsGEVCDF**, **StatsGEVPDF**

StatsInvExpCDF

```
StatsInvExpCDF(cdf, μ, σ)
```

The StatsInvExpCDF function returns the inverse of the exponential cumulative distribution function

$$x = \mu - \sigma \ln\left(1 - cdf\right).$$

It returns NaN for *cdf* <0 or *cdf* > 1.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsExpCDF** and **StatsExpPDF**.

StatsInvFCDF

```
StatsInvFCDF(x, n1, n2)
```

The StatsInvFCDF function returns the inverse of the F distribution cumulative distribution function for *x* and shape parameters *n1* and *n2*. The inverse is also known as the percent point function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFCDF** and **StatsFPDF**.

StatsInvFriedmanCDF

```
StatsInvFriedmanCDF(cdf, n, m, method, useTable)
```

The StatsInvFriedmanCDF function returns the inverse of the Friedman distribution cumulative distribution function of *cdf* with *m* rows and *n* columns. Use this typically to compute the critical values of the distribution

```
Print StatsInvFriedmanCDF(1-alpha,n,m,0,1)
```

where alpha is the significance level of the associated test.

The complexity of the computation of Friedman CDF is on the order of $(n!)^m$. For nonzero values of *useTable,* searches are limited to the built-in table for distribution values. If *n* and *m* are not in the table the calculation may still proceed according to the *method*.

method	**What It Does**
0	Exact computation(slow, not recommended).
1	Chi-square approximation.
2	Monte-Carlo approximation (slow).
3	Use built-in table only and return a NaN if not in table.

For large *m* and *n*, consider using the Chi-squared or the Iman and Davenport approximations. To abort execution, press the **User Abort Key Combinations**.

Note: Table values are different from computed values for both methods. Table values use more conservative criteria than computed values. Table values are more consistent with published values because the Friedman distribution is a highly irregular function with multiple steps of arbitrary sizes. The standard for published tables provides the X value of the next vertical transition to the one on which the specified P is found.

Precomputed tables use these values:

n	*m*
3	2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
4	2, 3, 4, 5, 6, 7, 8, 9
5	2, 3, 4, 5, 6
6	2, 3, 4, 5
7	2, 3, 4
8	2, 3
9	2, 3

References

Iman, R.L., and J.M. Davenport, Approximations of the critical region of the Friedman statistic, *Comm. Statist.*, *A9*, 571-595, 1980.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsFriedmanCDF** and **StatsFriedmanTest**.

StatsInvGammaCDF

`StatsInvGammaCDF(`*`cdf`*`, μ, σ, γ)`

The StatsInvGammaCDF function returns the inverse of the gamma cumulative distribution function. There is no closed form expression for the inverse gamma distribution; it is evaluated numerically.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGammaCDF** and **StatsGammaPDF**.

StatsInvGeometricCDF

`StatsInvGeometricCDF(`*`cdf, p`*`)`

The StatsInvGeometricCDF function returns the inverse of the geometric cumulative distribution function

$$x = \frac{\ln(1 - cdf)}{\ln(1 - p)} - 1$$

where *p* is the probability of success in a single trial and *x* is the number of trials.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsGeometricCDF** and **StatsGeometricPDF**.

StatsInvKuiperCDF

`StatsInvKuiperCDF(`*`cdf`*`)`

The StatsInvKuiperCDF function returns the inverse of Kuiper cumulative distribution function.

There is no closed form expression. It is mapped to the range of 0.4 to 4, with accuracy of 1e-10.

References

See in particular Section 14.3 of

Press, William H., *et al.*, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsKuiperCDF**.

StatsInvLogisticCDF

```
StatsInvLogisticCDF(cdf, a, b)
```

The StatsInvLogisticCDF function returns the inverse of the logistic cumulative distribution function

$$x = a + b\log\left(\frac{cdf}{1-cdf}\right).$$

where the scale parameter b>0 and the shape parameter is a.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogisticCDF** and **StatsLogisticPDF** functions.

StatsInvLogNormalCDF

```
StatsInvLogNormalCDF(cdf, sigma, theta, mu)
```

The StatsInvLogNormalCDF function returns the numerically evaluated inverse of the lognormal cumulative distribution function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogNormalCDF** and **StatsLogNormalPDF** functions.

StatsInvMaxwellCDF

```
StatsInvMaxwellCDF(cdf, k)
```

The StatsInvMaxwellCDF function returns the evaluated numerically inverse of the Maxwell cumulative distribution function. There is no closed form expression.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsMaxwellCDF** and **StatsMaxwellPDF** functions.

StatsInvMooreCDF

```
StatsInvMooreCDF(cdf, N)
```

The StatsInvMooreCDF function returns the inverse cumulative distribution function for Moore's R*, which is used as a critical value in nonparametric version of the Rayleigh test for uniform distribution around the circle. It supports the range $3 \le N \le 120$ and does not change appreciably for $N > 120$.

The inverse distribution is computed from polynomial approximations derived from simulations and should be accurate to approximately three significant digits.

References

Moore, B.R., A modification of the Rayleigh test for vector data, *Biometrica*, *67*, 175-180, 1980.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsCircularMeans** function.

StatsInvNBinomialCDF

`StatsInvNBinomialCDF(`*`cdf, k, p`*`)`

The StatsInvNBinomialCDF function returns the numerically evaluated inverse of the negative binomial cumulative distribution function. There is no closed form expression.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNBinomialCDF** and **StatsNBinomialPDF** functions.

StatsInvNCChiCDF

`StatsInvNCChiCDF(`*`cdf, n, d`*`)`

The StatsInvNCChiCDF function returns the inverse of the noncenteral chi-squared cumulative distribution function. It is computationally intensive because the inverse is computed numerically and involves multiple evaluations of the noncentral distribution, which is evaluated from a series expansion.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCChiCDF**, **StatsNCChiPDF**, **StatsChiCDF**, and **StatsChiPDF** functions.

StatsInvNCFCDF

`StatsInvNCFCDF(`*`cdf, n1, n2, d`*`)`

The StatsInvNCFCDF function returns the numerically evaluated inverse of the cumulative distribution function of the noncentral F distribution. *n1* and *n2* are the shape parameters and *d* is the noncentrality measure. There is no closed form expression for the inverse.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCFCDF** and **StatsNCFPDF** functions.

StatsInvNormalCDF

`StatsInvNormalCDF(`*`cdf, m, s`*`)`

The StatsInvNormalCDF function returns the numerically computed inverse of the normal cumulative distribution function. There is no closed form expression.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNormalCDF** and **StatsNormalPDF** functions.

StatsInvParetoCDF

`StatsInvParetoCDF(`*`cdf, a, c`*`)`

The StatsInvParetoCDF function returns the inverse of the Pareto cumulative distribution function

$$x = \frac{a}{\left(1 - cdf\right)^{(1/c)}}$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsParetoCDF** and **StatsParetoPDF** functions.

StatsInvPoissonCDF

`StatsInvPoissonCDF(`*`cdf`*`, λ)`

The StatsInvPoissonCDF function returns the numerically evaluated inverse of the Poisson cumulative distribution function. There is no closed form expression for the inverse Poisson distribution.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPoissonCDF** and **StatsPoissonPDF** functions.

StatsInvPowerCDF

`StatsInvPowerCDF(`*`cdf, b, c`*`)`

The StatsInvPowerCDF function returns the inverse of the Power Function cumulative distribution function

$$x = b / cdf^{(1/c)}.$$

where the scale parameter *b* and the shape parameter *c* satisfy *b*,*c*>0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPowerCDF**, **StatsPowerPDF** and **StatsPowerNoise** functions.

StatsInvQCDF

`StatsInvQCDF(`*`cdf, r, c, df`*`)`

The StatsInvQCDF function returns the critical value of the Q cumulative distribution function for *r* the number of groups, *c* the number of treatments, and *df* the error degrees of freedom (*df*=*r***c**(*n*-1) with sample size *n*).

Details

The Q distribution is the maximum of several Studentized range statistics. For a simple Tukey test, use *r*=1.

Examples

The critical value for a Tukey test comparing 5 treatments with 6 samples and 0.05 significance is:

```
Print StatsInvQCDF(1-0.05,1,5,5*(6-1))
```

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTukeyTest** function.

StatsInvQpCDF

`StatsInvQpCDF(`*`ng, nt, df, alpha, side, sSizeWave`*`)`

The StatsInvQpCDF function returns the critical value of the Q' cumulative distribution function for *ng* the number of groups, *nt* the number of treatments, and *df* the error degrees of freedom. *side*=1 for upper-tail or *side*=2 for two-tailed critical values.

sSizeWave is an integer wave of *ng* columns and *nt* rows specifying the number of samples in each treatment. If *sSizeWave* is a null wave (`$""`) StatsInvQpCDF computes the number of samples from *df*=*ng***nt**(*n*-1) with *n* truncated to an integer.

Details

StatsInvQpCDF is a modified Q distribution typically used with Dunnett's test, which compares the various means with the mean of the control group or treatment.

StatsInvQpCDF differs from other StatsInv*XXX* functions in that you do not specify a *cdf* value for the inverse (usually 1-*alpha* for the critical value). Here *alpha* selects one- or two-tailed critical values.

It is computationally intensive, taking longer to execute for smaller *alpha* values.

Examples

The critical value for a Dunnett test comparing 4 treatments with 4 samples and (upper tail) 0.05 significance is:

```
// n=4 because 12=1*4*(4-1).
Print StatsInvQpCDF(1,4,12,0.05,1,$"")
  2.28734
```

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsDunnettTest** and **StatsInvQCDF** functions.

StatsInvRayleighCDF

`StatsInvRayleighCDF(`*`cdf`*` [, `*`s`*` [, `*`m`*`]])`

The **`StatsInvRayleighCDF`** function returns the inverse of the Rayleigh cumulative distribution functiongiven by

$$x = \mu + \sigma\sqrt{-2\ln(1-cdf)},$$

with defaults *s*=1 and *m*=0. It returns NaN for *s* ≤ 0 and zero for x ≤ *m*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRayleighCDF** and **StatsRayleighPDF** functions.

StatsInvRectangularCDF

`StatsInvRectangularCDF(`*`cdf`*`, `*`a`*`, `*`b`*`)`

The StatsInvRectangularCDF function returns the inverse of the rectangular (uniform) cumulative distribution function

$$x = a + cdf(b-a), \qquad a < b.$$

where *a*< *b*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRectangularCDF** and **StatsRectangularPDF** functions.

StatsInvSpearmanCDF

`StatsInvSpearmanCDF(`*`cdf`*`, `*`N`*`)`

The StatsInvSpearmanCDF function returns the inverse cumulative distribution function for Spearman's *r*, which is used as a critical value in rank correlation tests.

The inverse distribution is computed by finding the value of *r* for which it attains the *cdf* value. The result is usually lower than in published tables, which are more conservative when the first derivative of the distribution is discontinuous.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRankCorrelationTest**, **StatsSpearmanRhoCDF**, and **StatsKendallTauTest** functions.

StatsInvStudentCDF

`StatsInvStudentCDF(`*`cdf`*`, `*`n`*`)`

The StatsInvStudentCDF function returns the numerically evaluated inverse of Student cumulative distribution function. There is no closed form expression.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentCDF** and **StatsStudentPDF** functions.

StatsInvTopDownCDF

`StatsInvTopDownCDF(cdf, N)`

The StatsInvTopDownCDF function returns the inverse cumulative distribution function for the top-down distribution. For 3 ≤ *N* ≤ 7 it uses a lookup table CDF and returns the next higher value of *r* for which the distribution value is larger than *cdf*. For 8 ≤ *N* ≤ 50 it returns the nearest value for which the built-in distribution returns *cdf*. For *N*>50 it returns the scaled normal approximation.

Tabulated values are from Iman and Conover who pick as the critical value the very first transition of the distribution following the specified *cdf* value. These tabulated values tend to be slightly higher than calculated values for 7<*N*<15.

References

Iman, R.L., and W.J. Conover, A measure of top-down correlation, *Technometrics*, *29*, 351-357, 1987.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRankCorrelationTest** and **StatsTopDownCDF** functions.

StatsInvTriangularCDF

```
StatsInvTriangularCDF(cdf, a, b, c)
```

The StatsInvTriangularCDF function returns the inverse of the triangular cumulative distribution function

$$x = \begin{cases} a + \sqrt{cdf(b-a)(c-a)} & 0 \le cdf \le \dfrac{c-a}{b-a} \\ b - \sqrt{(1-cdf)(b-a)(b-c)} & \dfrac{c-a}{b-a} \le cdf \le 1 \end{cases}$$

where $a<c<b$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTriangularCDF** and **StatsTriangularPDF** functions.

StatsInvUSquaredCDF

```
StatsInvUSquaredCDF(cdf, n, m, method, useTable)
```

The StatsInvUSquaredCDF function returns the inverse of Watson's U^2 cumulative distribution function integer sample sizes *n* and *m*. Use a nonzero value for *useTable* to search a built-in table of values. If *n* and *m* cannot be found in the table, it will proceed according to *method*:

method	What It Does
0	Exact computation using Burr algorithm (could be slow).
1	Tiku approximation using chi-squared.
2	Use built-in table only and return a NaN if not in table.

For large *n* and *m*, consider using the Tiku approximation. To abort execution, press the **User Abort Key Combinations**. Because *n* and *m* are interchangeable, *n* should always be the smaller value. For *n*>8 the upper limit in the table matched the maximum that can be computed using the Burr algorithm. There is no point in using method 0 with *m* values exceeding these limits.

The inverse is obtained from precomputed tables of Watson's U^2 (see **StatsUSquaredCDF**).

Note: Table values are different from computed values. These values use more conservative criteria than computed values. Table values are more consistent with published values because the U^2 distribution is a highly irregular function with multiple steps of arbitrary sizes. The standard for published tables provides the X value of the next vertical transition to the one on which the specified P is found. See **StatsInvFriedmanCDF**.

References

Burr, E.J., Small sample distributions of the two sample Cramer-von Mises' W^2 and Watson's U^2, *Ann. Mah. Stat. Assoc.*, *64*, 1091-1098, 1964.

Tiku, M.L., Chi-square approximations for the distributions of goodness-of-fit statistics, *Biometrica*, *52*, 630-633, 1965.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWatsonUSquaredTest** and **StatsUSquaredCDF** functions.

StatsInvVonMisesCDF

`StatsInvVonMisesCDF(`*`cdf, a, b`*`)`

The StatsInvVonMisesCDF function returns the numerically evaluated inverse of the von Mises cumulative distribution function where the value of the integral of the distribution matches *cdf*. Parameters are as for **StatsVonMisesCDF**.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsVonMisesPDF** and **StatsVonMisesNoise** functions.

StatsInvWeibullCDF

`StatsInvWeibullCDF(`*`cdf, m, s, g`*`)`

The StatsInvWeibullCDF function returns the inverse of the Weibull cumulative distribution function

$$x = \mu + \sigma\left[-\ln\left(1 - cdf\right)\right]^{1/\gamma}.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWeibullCDF** and **StatsWeibullPDF** functions.

StatsJBTest

`StatsJBTest [`*`flags`*`]` ***`srcWave`***

The StatsJBTest operation performs the Jarque-Bera test on *srcWave*. Output is to the W_JBResults wave in the current data folder.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

StatsJBTest computes the Jarque-Bera statistic

$$JB = \frac{n}{6}\left(S^2 + \frac{K^2}{4}\right),$$

where *S* is the skewness, *K* is the kurtosis, and *n* is the number of points in the input wave. We can express *S* and *K* terms of the *j*th moment of the distribution for n samples X_i

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^j$$

as

$$S = \frac{\mu_3}{(\mu_2)^{3/2}},$$

and

$$K = \frac{\mu_4}{(\mu_2)^2} - 3.$$

The Jarque-Bera statistic is asymptotically distributed as a Chi-squared with two degrees of freedom. For values of *n* in the range [7,2000] the operation provides critical values obtained from Monte-Carlo simulations. For further details or if you would like to run your own simulation to obtain critical values for other values of *n*, use the JarqueBeraSimulation example experiment.

StatsJBTest reports the number of finite data points, skewness, kurtosis, Jarque-Bera statistic, asymptotic critical value, and the critical value obtained from Monte-Carlo calculations as appropriate; it ignores NaNs and INFs.

References

Jarque, C., and A. Bera, A test of normality of observations and regression residuals, *International Statistical Review*, *55*, 163-172, 1987.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsKSTest**, **WaveStats**, and **StatsCircularMoments**.

StatsKDE

`StatsKDE [`*`flags`*`]` ***`srcWave`***

StatsKDE can be used to estimate a PDF from original data distribution. Unlike histograms, this method produces a smooth result as it constructs the PDF from a normalized superposition of kernel functions.

The StatsKDE operation was added in Igor Pro 7.00.

Flags

/BWM=*m*	Sets the bandwidth selection method. *m*=0: User-specified via /H flag *m*=1: Silverman *m*=2: Scott *m*=3: Bowmann and Azzolini
/DEST=*destWave*	Specifies the output destination. Creates a real wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-66 for details.
/FREE	Makes the destination wave (specified by /DEST) a free wave.
/H=bw	Specifies a fixed user-defined bandwidth.

/KT=*kernel*	Specifies the kernel type. *kernel*=1: Epanechnikov *kernel*=2: Bi-weight *kernel*=3: Tri-weight *kernel*=4: Triangular *kernel*=5: Gaussian *kernel*=6: Rectangular
/Q	No results printed in the history area. In the case of univariate KDE this flags suppresses the printing of the bandwidth value.
/S={*x0*,*dx*,*xn*}	Specifies the range of the output starting from x=*x0* to x=*xn* in increments of *dx*.
/Z	Ignores errors. V_flag is set to zero if there are no errors.

Details

StatsKDE estimates the PDF of a distribution of values using a smoothing kernel and a bandwidth paramter which affects the degree of smoothing.

Theory suggests that the Epanechnikov kernel is the most efficient but many expressions for the optimal bandwidth are derived for the Gaussian kernel. If *srcWave* contains N points and the requested output (/S flag) has M points then the computational complexity is O(NM). For large problems it may be beneficial to use the Gaussian kernel via the FastGaussTransform operation.

References

Wand M.P. and Jones M.C. (1995) Monographs on Statistics and Applied Probability, London: Chapman and Hall

Bowman, A.W., and Azzalini, A. (1997), Applied Smoothing Techniques for Data Analysis, London: Oxford University Press.

See Also

Statistics on page III-337, **Histogram**, **FastGaussTransform**

StatsKendallTauTest

`StatsKendallTauTest [`*`flags`*`] `*`wave1`*` [, `*`wave2`*`]`

The StatsKendallTauTest operation performs the nonparametric Mann-Kendall test, which computes a correlation coefficient τ (similar to Spearman's correlation) from the relative order of the ranks of the data. Output is to the W_StatsKendallTauTest wave in the current data folder.

Flags

/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Inputs may be a pair of XY (1D) waves of any real numeric type or a single 1D wave, which is equivalent to using a pair of XY waves where the X wave is monotonically increasing function of the point number. StatsKendallTauTest ignores wave scaling.

Kendall's τ is 1 for a monotonically increasing input and -1 for monotonically decreasing input. The significance of the test is computed from the normal approximation

$$Var(\tau) = \frac{4n+10}{9n(n-1)},$$

where *n* is the number of data points in each wave. The significance is expressed as a P-value for the null hypothesis of no correlation.

References

Kendall, M.G., *Rank Correlation Methods*, 3rd ed., Griffin, London, 1962.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsRankCorrelationTest**.

For small values of *n* you can compute the exact probability using the procedure WM_KendallSProbability().

StatsKSTest

`StatsKSTest [`*`flags`*`] `*`srcWave`*` [, `*`distWave`*`]`

The StatsKSTest operation performs the Kolmogorov-Smirnov (KS) goodness-of-fit test for two continuous distributions. The first distribution is *srcWave* and the second distribution can be expressed either as the optional wave *distWave* or as a user function with /CDFF. Output is to the W_KSResults wave in the current data folder.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/CDFF=*func*	Specifies a user function expressing the cumulative distribution function. See Details.
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The Kolmogorov-Smirnov (KS) goodness-of-fit test applies only to continuous distributions and cases where the compared distribution (expressed as a user function) is completely specified without estimating parameters from the data. It compares the cumulative distribution function (CDF) of two distributions and sets the test statistic D to the largest difference between the CDFs. Because CDFs are in the range [0,1], D is also bound by this range.

When specifying the distributions with two waves, StatsKSTest first sorts the data in the waves and then computes the CDFs and D. You can also specify one of the distributions with a user function. For example, the following function tests if the data in *srcWave* is normally distributed with zero mean and stdv=5:

```
Function GetUserCDF(inX) : CDFFunc
	Variable inX
	return StatsNormalCDF(inX,0,5)
End
```

The "`: CDFFunc`" designation, which requires Igor7 or later, tells Igor to make the function accessible from the Kolmogorov-Smirnov Test dialog.

Outputs are the number of elements, the KS statistic D, and the critical value. When both distributions are specified by waves, the number of elements is the weighted value (n1*n2)/(n1+n2).

References

Critical values are based on:

Birnbaum, Z. W., and Fred H. Tingey, One-sided confidence contours for probability distribution functions, *The Annals of Mathematical Statistics*, *22*, 592–596, 1951.

A statistically more powerful modification of the classic KS test can be found in:

Khamis, H.J., The two-stage delta-corrected Kolmogorov-Smirnov test, *Journal of Applied Statistics*, *27*, 439-450, 2000.

StatsKSTest implements the original KS test. The difficulty in implementing the modified tests for all the cases defined by Stephens is in obtaining the critical values which have to be derived by time consuming Monte-Carlo simulations.

Critiques can be found in:

D'Agostino, R.B., and M. Stephens, eds., *Goodness-Of-Fit Techniques*, Marcel Dekker, New York, 1986.

NIST/SEMATECH, Kolmogorov-Smirnov Goodness-of-Fit Test, in *NIST/SEMATECH e-Handbook of Statistical Methods*, `<http://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm>`, 2005.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsJBTest**, **WaveStats**, and **StatsCircularMoments**.

StatsKuiperCDF

`StatsKuiperCDF(`*`V`*`)`

The StatsKuiperCDF function returns the Kuiper cumulative distribution function

$$F(V) = 1 - 2\sum_{j=1}^{\infty}\left(4j^2V^2 - 1\right)\exp\left(-2j^2V^2\right).$$

Accuracy is on the order of 1e-15. It returns 0 for values of *V*<0.4 or 1 for *V*>3.1.

References

See in particular Section 14.3 of

Press, William H., *et al*., *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsInvKuiperCDF**.

StatsKWTest

`StatsKWTest [`*`flags`*`] [`*`wave1, wave2,… wave100`*`]`

The StatsKWTest operation performs the nonparametric Kruskal-Wallis test which tests variances using the ranks of the data. Output is to the W_KWTestResults wave in the current data folder.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/E	Computes the exact P-value using the Klotz and Teng algorithm, which may require long computation times for large data sets. You can stop the calculation by pressing the **User Abort Key Combinations** after which all remaining results remain valid and the exact P-value is set to NaN.
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/WSTR=*waveListString*	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Inputs are two or more 1D numerical waves (one for each group of samples). Use NaNs for missing data or use waves with different number of points.

StatsKWTest always computes the critical values using both the Chi-squared and Wallace approximations. If appropriate (small enough data set) you can also use /E to obtain the exact P value. When the calculation involves many waves or many data points the calculation of the exact critical value can be very lengthy. All the results are saved in the wave W_KWTestResults in the current data folder and are optionally displayed in a table (/T). The wave contains the following information:

Row	Data
0	Number of groups
1	Number of valid data points (excludes NaNs)
2	Alpha
3	Kruskal-Wallis Statistic H
4	Chi-squared approximation for the critical value Hc
5	Chi-squared approximation for the P value
6	Wallace approximation for the critical value Hc
7	Wallace approximation for the P value
8	Exact P value (requires /E)

H_0 for the Kruskal-Wallis test is that all input waves are the same. If the test fails and the input consisted of more than two waves, there is no indication for possible agreement between some of the waves. See **StatsNPMCTest** for further analysis.

V_flag will be set to -1 for any error and to zero otherwise.

References

Klotz, J.H., *Computational Approach to Statistics*, <`http://www.stat.wisc.edu/~klotz/Book.pdf`>.

Klotz, J., and Teng, J., One-way layout for counts and the exact enumeration of the Kruskal-Wallis H distribution with ties, *J. Am. Stat. Assoc*, *72*, 165-169, 1977.

Wallace, D.L., Simplified Beta-Approximation to the Kruskal-Wallis H Test, *J. Am. Stat. Assoc.*, *54*, 225-230, 1959.

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWilcoxonRankTest**, **StatsNPMCTest**, and **StatsAngularDistanceTest**.

StatsLinearCorrelationTest

`StatsLinearCorrelationTest [`*`flags`*`]` ***`waveA, waveB`***

The StatsLinearCorrelationTest operation performs correlation tests on *waveA* and *waveB*, which must be real valued numeric waves and must have the same number of points. Output is to the W_StatsLinearCorrelationTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/CI	Computes confidence intervals for the correlation coefficient.
/Q	No results printed in the history area.
/RHO=*rhoValue*	Tests hypothesis that the correlation has a nonzero value $|r| \leq 1$.

/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed.
	k=0: Normal with dialog (default).
	k=1: Kills with no dialog.
	k=2: Disables killing.
/Z	Ignores errors.

Details

The linear correlation tests start by computing the linear correlation coefficient for the *n* elements of both waves:

$$r = \frac{\sum_{i=1}^{n} X_i Y_i - \frac{1}{n}\sum_{i=1}^{n} X_i \sum_{i=1}^{n} Y_i}{\sqrt{\left(\sum_{i=1}^{n} X_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} X_i\right)^2\right)\left(\sum_{i=1}^{n} Y_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} Y_i\right)^2\right)}}$$

Next it computes the standard error of the correlation coefficient

$$sr = \sqrt{\frac{1-r^2}{n-2}}$$

The basic test is for hypothesis H_0: the correlation coefficient is zero, in which case *t* and *F* statistics are applicable. It computes the statistics:

$$t = r / sr$$

and

$$F = \frac{1+|r|}{1-|r|},$$

and then the critical values for one and two tailed hypotheses (designated by t_{c1}, t_{c2}, F_{c1}, and F_{c2} respectively). Critical value for *r* are computed using

$$\mathrm{rc_i} = \sqrt{\frac{t_c^2}{t_c^2 + n}}$$

where *i* takes the values 1 or 2 for one and two tailed hypotheses. Finally, it computes the power of the test at the alpha significance level for both one and two tails (Power1 and Power2).

If you use /RHO it uses the Fisher transformation to compute

$$\mathrm{FisherZ} = \frac{1}{2}\ln\left(\frac{1+r}{1-r}\right)$$

$$\mathrm{zeta} = \frac{1}{2}\ln\left(\frac{1+\rho}{1-\rho}\right)$$

the standard error approximation

$$\text{sigmaZ}=\sqrt{\frac{1}{n-3}},$$

$$\text{Zstatistic}=\frac{FisherZ-zeta}{sigmaZ},$$

and the critical values from the normal distribution Z_{ci}.

The confidence intervals are calculated differently depending on the hypothesis for the value of the correlation coefficient. If /RHO is not used the confidence intervals are computed using the critical value F_{c2}, otherwise they are computed using the critical Z_{c2} and sigmaZ.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsCircularCorrelationTest**, **StatsMultiCorrelationTest**, and **StatsRankCorrelationTest**.

StatsLinearRegression

`StatsLinearRegression [`*`flags`*`] [`*`wave0, wave1,…`*`]`

The StatsLinearRegression operation performs regression analysis on the input wave(s). Output is to the W_StatsLinearRegression wave in the current data folder or optionally to a table. Additionally, the M_DunnettMCElevations, M_TukeyMCSlopes, and M_TukeyMCElevations waves may be created as specified.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/B=*beta0*	Tests the hypothesis that the slope *b*= *beta0* (default is 0). The results are expressed by the t-statistic, which can be compared with the tc value for the two-tailed test. Get the critical value for a one-tailed test using `StatsStudentCDF(1-alpha,N-2)`. It does not work with /MYVW.
/BCIW	Computes two confidence interval waves for the high side and the low side of the confidence interval. The new waves are named with _CH and _CL suffixes respectively appended to the Y wave name and are created in the current data folder. For multiple runs a numeric suffix will also be appended to the names.
/BPIW[=*mAdditional*]	Computes prediction interval waves for the high side and the low side of the confidence interval on a single additional measurement (default). Use *mAdditional* to specify additional measurements. The new waves are named with _PH and _PL suffixes respectively appended to the Y wave name and are created in the current data folder. For multiple runs a numeric suffix will also be appended to the names.
/DET=*controlIndex*	Performs Dunnett’s multicomparison test for the elevations. The test requires more than two Y waves for regression, the test for the slopes should not reject the equal slope hypothesis, and the test for the elevations should reject the equal elevation hypothesis. *controlIndex* is the zero-based index of the Y wave representing the control (X waves do not count in the index specification). The test compares the elevation of every Y wave with the specified control. Output is to the M_DunnettMCElevations wave in the current data folder or optionally to a table. For every Y wave and control Y wave combination, the results include SE, q, q' (shown as qp), and the conclusion with 1 to accept the hypothesis of equal elevations or 0 to reject it. Use /TAIL to determine the critical value and the sense of the test. If you use /TUK you will also get the Tukey test for the set of elevations.

/MYVW={*xWave*, *yWave*}

Specifies that the input consists of multiple Y values for each X value. It ignores all other inputs and the results are appropriate only for multiple Y values at each X point.

yWave is a 2D wave of values arranged in columns. Use NaNs for padding where rows do not have the same number of entries as others. It will use the X scaling of *yWave* when *xWave* is null, `/MYVW={*,yWave}`.

It first tests the hypothesis (H_0) that the population regression is linear in an analysis of variance calculation. It generates results 1-7 (see Details) as well as: Among Groups SS, Among Groups DF, Within Groups SS, Within Groups DF, Deviations from Linearity SS, Deviations from Linearity DF, F statistic defined by the ratio of Deviation from Linearity MS to Within Groups MS, and the critical value Fc.

Next, it tests the hypothesis that the slope beta=0. If the original H_0 was accepted, the new F statistic=regressionMS/residualMS. Otherwise the with the critical F=regressionMS/WithinGroupsMS with a corresponding critical value. Finally, it reports the values of the coefficient of determination r2 and the standard error of the estimate S_{YX}.

/PAIR Specifies that the input waves are XY pairs, where each pair must be an X wave followed by a Y wave.

/Q No results printed in the history area.

/RTO Reflects the regression through the origin.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

/TAIL=*tCode* Sets the sense of the test when applying Dunnett's test (see /DET). *tCode* is 1 or 2 for a one-tail critical value and 4 for a two-tail critical value.

/TUK Performs a Tukey-type test on multiple regressions on two or more Y waves. There are two possible Tukey-type tests: The first is performed if the hypothesis of equal slopes is rejected. It compares all combinations of two Y waves to identify if some of the waves have equal slopes. Output is to the M_TukeyMCSlopes wave in the current data folder or optionally to a table. For every Y wave pair, the results include the difference between slopes (absolute value), q, the critical value qc, and the conclusion set to 1 for accepting the equality of the pair of slopes or 0 for rejecting the hypothesis.

The second Tukey-type test is performed if all the slopes are the same but the elevations are not. The test (see /DET) compares all possible pairs of elevations to determine which satisfy the hypothesis of equality. Output is to the M_TukeyMCElevations wave in the current data folder.

/WSTR=*waveListString*

Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.

/Z Ignores errors.

Details

Inputs may consist of Y waves or XY wave pairs. If X data are not used, the X values are inferred from the Y wave scaling. For multiple waves where only some have pairs, use the /PAIR flag and enter * in each place where the X values should be computed.

For each input StatsLinearRegression calculates:

1. Least squares regression line y=a+b*x.
2. Mean value of X: xBar.
3. Mean value of Y: yBar.
4. Sum of the squares (x_i-xBar)^2.
5. Sum of the squares (y_i-yBar)^2.
6. Sum of the product ($x_i y_i$-xyBar).
7. Standard error of the estimate $S_{yx}^2 = \frac{\sum\left(Y_i - \hat{Y}_i\right)^2}{n-2}$.
8. F statistic for the hypothesis beta=0.
9. Critical F value Fc.
10. Coefficient of determination r2.
11. Standard error of the regression coefficient S_b.
12. t-statistic for the hypothesis beta=*beta0*, NaN if /B is not specified.
13. Critical value tc for the t-statistic above (used to calculate L1 and L2).
14. Lower confidence interval boundary (L1) for the regression coefficient.
15. Upper confidence interval boundary (L2) for the regression coefficient.

For two Y waves with the same slope, it computes a common slope (bc) and then tests the equality of the elevations (a). In both cases it computes a t-statistic and compares it with a critical value. If the elevations are also the same then it computes the common elevation (ac) and the pooled means of X and Y in (xp) and (yp).

For more than two Y waves it computes:

$$A_c = \sum_{j=1}^{W} A_j; \qquad A_j \equiv \sum x_i^2 = \sum_{i=0}^{n_j-1} X_i^2 - \frac{1}{n_j}\left(\sum_{i=0}^{n_j-1} X_i\right)^2$$

$$B_c = \sum_{j=1}^{W} B_j; \qquad B_j \equiv \sum xy = \sum_{i=0}^{n_j-1} XY - \frac{1}{n_j}\left(\sum_{i=0}^{n_j-1} X_i\right)\left(\sum_{i=0}^{n_j-1} Y_i\right)$$

$$C_c = \sum_{j=1}^{W} C_j; \qquad C_j \equiv \sum y^2 = \sum_{i=0}^{n_j-1} Y_i^2 - \frac{1}{n_j}\left(\sum_{i=0}^{n_j-1} Y_i\right)^2$$

$$SSp = \sum_{j=1}^{W} C_j - \frac{B_j^2}{A_j}$$

$$SSc = Cc - \frac{B_c^2}{A_c^2}$$

$$SSt = \sum_{j=1}^{W}\sum_{i=0}^{n_j} Y_{ji}^2 - \frac{1}{N}\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} Y_{ji}\right)^2 - \frac{\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} X_{ji}Y_{ji} - \frac{1}{N}\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} X_{ji}\right)\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} Y_{ji}\right)\right)^2}{\sum_{j=1}^{W}\sum_{i=0}^{n_j} X_{ji}^2 - \frac{1}{N}\left(\sum_{j=1}^{W}\sum_{i=0}^{n_j} X_{ji}\right)^2}$$

$$DFp = \sum_{j=1}^{W} \left(n_i - 2\right)$$

$$DFt = \sum_{j=1}^{W} n_i - 2$$

Here *W* is the number of Y-waves and $N = \sum_{j=1}^{W} n_j$ is the total number of data points in all Y-waves.

The test statistic F for equality of slopes is given by:

$$F = \left(\frac{SSc - SSp}{numWaves - 1}\right) \Big/ \frac{SSp}{DFp}.$$

Fc is the corresponding critical value.

Output is to the W_LinearRegressionMC wave in the current data folder.

V_flag will be set to -1 for any error and to zero otherwise.

References

See, in particular, Chapter 18 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; curvefit.

StatsLogisticCDF

`StatsLogisticCDF(x, a, b)`

The StatsLogisticCDF function returns the logistic cumulative distribution function

$$F(x;a,b) = \frac{1}{1 + \exp\left(-\frac{x-a}{b}\right)}.$$

where the scale parameter *b*>0 and the shape parameter is *a*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogisticPDF** and **StatsInvLogisticCDF** functions.

StatsLogisticPDF

`StatsLogisticPDF(x, a, b)`

The StatsLogisticPDF function returns the logistic probability distribution function

$$f(x;a,b) = \frac{\exp\left(-\frac{x-a}{b}\right)}{b\left[1 + \exp\left(-\frac{x-a}{b}\right)\right]^2},$$

where the scale parameter *b*>0 and the shape parameter is *a*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogisticCDF** and **StatsInvLogisticCDF** functions.

StatsLogNormalCDF

`StatsLogNormalCDF(x, σ [, θ, μ])`

The StatsLogNormalCDF function returns the lognormal cumulative distribution function

$$F(x;\sigma,\theta,\mu)=\frac{1}{\sigma\sqrt{2\pi}}\int_0^x \frac{1}{t-\theta}\exp\left\{-\left[\ln\left(\frac{t-\theta}{\mu}\right)\right]^2 \Big/ 2\sigma^2\right\}dt,$$

for $x \geq \theta$ and σ, μ>0. The standard lognormal distribution is for θ=0 and μ=1, which are the optional parameter defaults.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogNormalPDF** and **StatsInvLogNormalCDF** functions.

StatsLogNormalPDF

`StatsLogNormalPDF(x, σ [, θ, μ])`

The StatsLogNormalPDF function returns the lognormal probability distribution function

$$f(x;\sigma,\theta,\mu)=\frac{1}{\sigma\sqrt{2\pi}}\frac{1}{x-\theta}\exp\left\{-\left[\ln\left(\frac{x-\theta}{\mu}\right)\right]^2 \Big/ 2\sigma^2\right\},$$

for $x \geq \theta$ and σ, μ > 0, where θ is the location parameter, μ is the scale parameter and, σ is the shape parameter. The standard lognormal distribution is for θ=0 and μ=1, which are the optional parameter defaults.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsLogNormalCDF** and **StatsInvLogNormalCDF** functions.

StatsMaxwellCDF

`StatsMaxwellCDF(x, k)`

The StatsMaxwellCDF function returns the Maxwell cumulative distribution function

$$F(x;k)=gammp\left(\frac{3}{2},\frac{kx^2}{2}\right), \qquad x>0.$$

where **gammp** is the regularized incomplete gamma function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsMaxwellPDF** and **StatsInvMaxwellCDF** functions.

StatsMaxwellPDF

`StatsMaxwellPDF(x, k)`

The StatsMaxwellPDF function returns Maxwell's probability distribution function

$$f(x;k)=\sqrt{\frac{2}{\pi}}k^{3/2}x^2\exp\left(-\frac{kx^2}{2}\right), \qquad x>0.$$

The Maxwell distribution describes, for example, the speed distribution of molecules in an ideal gas.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsMaxwellCDF** and **StatsInvMaxwellCDF** functions.

StatsMedian

`StatsMedian(`*`waveName`*`)`

The StatsMedian function returns the median value of a numeric wave *waveName*, which must not contain NaNs.

Example

```
Make/N=5 sample1={1,2,3,4,5}
Print StatsMedian(sample1)
3
Make/N=6 sample2={1,2,3,4,5,6}
Print StatsMedian(sample2)
3.5
```

See Also

Chapter III-12, **Statistics** for a function and operation overview

median, **WaveStats**, **StatsQuantiles**

StatsMooreCDF

`StatsMooreCDF(`*`x, N`*`)`

The StatsMooreCDF function returns the cumulative distribution function for Moore's R*, which is used in a nonparametric version of the Rayleigh test for uniform distribution around the circle. It supports the range $3 \le N \le 120$ and does not change appreciably for N>120.

The distribution is computed from polynomial approximations derived from simulations and should be accurate to approximately three significant digits.

References

Moore, B.R., A modification of the Rayleigh test for vector data, *Biometrica*, *67*, 175-180, 1980.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsCircularMeans** function.

StatsMultiCorrelationTest

`StatsMultiCorrelationTest [`*`flags`*`]`*`corrWave, sizeWave`*

The StatsMultiCorrelationTest operation performs various tests on multiple correlation coefficients. Inputs are two 1D waves: *corrWave*, containing correlation coefficients, and *sizeWave*, containing the size (number of elements) of the corresponding samples. Although you can do all the tests at the same time, it rarely makes sense to do so.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/CON={*controlRow*,*tails*}	Performs a multiple comparison test using the *controlRow* element of *corrWave* as a control. It is one- or two-tailed test according to the tails parameter. Output is to the M_ControlCorrTestResults wave in the current data folder.

/CONT=*cWave*	Performs a multiple contrasts test on the correlation coefficients. The contrasts wave, *cWave*, contains the contrast factor, c_i, entry for each of the *n* correlation coefficients r_i in *corrWave*, and satisfying the condition that the sum of the entries in *cWave* is zero. H_0 corresponds to

$$\sum_{i=0}^{n-1} c_i r_i = 0.$$

The test statistic *S* is

$$S = \frac{1}{\sqrt{\dfrac{c_i^2}{n_i - 3}}} \left| \sum_{i=0}^{n-1} c_i z_i \right|,$$

where z_i is the Fisher z transform of the correlation coefficient r_i:

$$z_i = \frac{1}{2} \ln\left(\frac{1+r_i}{1-r_i} \right).$$

It produces the SE value, the contrast statistic S, and the critical value, which are labeled ContrastSE, ContrastS, and Contrast_Critical, respectively, in the W_StatsMultiCorrelationTest wave.

/Q No results printed in the history area.

/T=*k* Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

/TUK Performs a Tukey-type multi comparison testing between the correlation coefficients by comparing every possible combination of pairs of correlation coefficients, computing the difference in their z-transforms, the SE, and the *q* statistic:

$$q = \frac{\left| z_j - z_i \right|}{\sqrt{\dfrac{1}{2}\left(\dfrac{1}{n_i - 3} + \dfrac{1}{n_j - 3} \right)}}.$$

The critical value is computed from the *q* CDF (**StatsInvQCDF**) with degrees of freedom *numWaves* and infinity. Output is to the M_TukeyCorrTestResults wave in the current data folder or optionally to a table.

/Z Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

Without any flags, StatsMultiCorrelationTest computes χ^2 for the correlation coefficients and compares it with the critical value.

$$\chi^2 = \sum_{i=0}^{n-1} z_i^2 \left(n_i - 3\right) - \frac{\left(\sum_{i=0}^{n-1} z_i \left(n_i - 3\right)\right)^2}{\sum_{i=0}^{n-1} \left(n_i - 3\right)},$$

where z_i is the Fisher's z transform of the correlation coefficients and n_i is the corresponding sample size. It computes the common correlation coefficient rw and its transform zw.

$$z_w = \frac{\sum_{i=0}^{n-1} z_i \left(n_i - 3\right)}{\sum_{i=0}^{n-1} \left(n_i - 3\right)}$$

These values are calculated even when not appropriate, such as when χ^2 exceeds the critical value and H_0 (all samples came from populations of identical correlation coefficients) is rejected.

The operation also computes ChiSquaredP (due to S.R. Paul), a different variant of χ^2 that is corrected for bias and should be compared with the same critical value. Output is to the W_StatsMultiCorrelationTest wave in the current data folder or optionally to a table.

References

See, in particular, Chapters 19 and 11 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsLinearCorrelationTest, **StatsCircularCorrelationTest**, **StatsDunnettTest**, **StatsTukeyTest**, **StatsInvQCDF**, and **StatsScheffeTest**.

StatsNBinomialCDF

`StatsNBinomialCDF(x, k, p)`

The StatsNBinomialCDF function returns the negative binomial cumulative distribution function

$$F(x;k,p) = Betai(k, x+1; p),$$

where **betai** is the regularized incomplete beta function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNBinomialPDF** and **StatsInvNBinomialCDF** functions.

StatsNBinomialPDF

`StatsNBinomialPDF(x, k, p)`

The StatsNBinomialPDF function returns the negative binomial probability distribution function

$$f(x;k,p) = \binom{x+k-1}{k-1} p^k (1-p)^x, \qquad x = 0,1,2...$$

where $\binom{a}{b}$ is the **binomial** function.

The binomial distribution expresses the probability of the *k*th success in the *x*+*k* trial for two mutually exclusive results (success and failure) and *p* the probability of success in a single trial.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNBinomialCDF** and **StatsInvNBinomialCDF** functions.

StatsNCChiCDF

`StatsNCChiCDF(x, n, d)`

The StatsNCChiCDF function returns the noncentral chi-squared cumulative distribution function

$$F(x;n,d) = \sum_{i=1}^{\infty} \exp\left(d/2\right) \frac{\left(d/2\right)^i}{i!} F_c(x;n+2i),$$

where n>0 corresponds to degrees of freedom, $d \geq 0$ is the noncentrality parameter, and F_c is the central chi-squared distribution.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsChiCDF**, **StatsNCChiPDF**, and **StatsChiPDF** functions.

StatsNCChiPDF

`StatsNCChiPDF(x, n, d)`

The StatsNCChiPDF function returns the noncentral chi-squared probability distribution function

$$f(x;n,d) = \frac{\sqrt{d}\exp\left(-\frac{x+d}{2}\right)x^{(n-1)/2}}{2(dx)^{n/4}} I_{n/2-1}\left(\sqrt{dx}\right).$$

where n>0 is the degrees of freedom, $d \geq 0$ is the noncentrality parameter, and $I_k(x)$ is the modified Bessel function of the first kind, **bessI**.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCChiCDF**, **StatsInvNCChiCDF**, **StatsChiCDF**, and **StatsChiPDF** functions.

StatsNCFCDF

`StatsNCFCDF(x, n1, n2, d)`

The StatsNCFCDF function returns the cumulative distribution function of the noncentral F distribution. *n1* and *n2* are the shape parameters and *d* is the noncentrality measure. There is no closed form expression for the distribution.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCFPDF** and **StatsInvNCFCDF** functions.

StatsNCFPDF

`StatsNCFPDF(x, n1, n2, d)`

The StatsNCFPDF function returns the probability distribution function of the noncentral F distribution

$$f(x;n_1,n_2,d)=\frac{\exp(-d/2)}{B\left(\frac{n_1}{2},\frac{n_2}{2}\right)}x^{n_1/2-1}(xn_1+n_2)^{-(n_1+n_2)/2}n_1^{n_1/2}n_2^{n_2/2}\,{}_1F_1\left(\frac{n_1+n_2}{2},\frac{n_1}{2},\frac{xdn_1}{2(xn_1+n_2)}\right),$$

where $B()$ is the **beta** function and ${}_1F_1()$ is the hypergeometric function **hyperG1F1**.

References

Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, 446 pp., Dover, New York, 1972.

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNCFCDF** and **StatsInvNCFCDF** functions.

StatsNCTCDF

StatsNCTCDF(*x, df, d*)

The StatsNCTCDF function returns the cumulative distribution function of the noncentral Student-T distribution. *df* is the degrees of freedom (positive integer) and *d* is the noncentrality measure. There is no closed form expression for the distribution.

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentCDF**, **StatsStudentPDF**, and **StatsNCTPDF** functions.

StatsNCTPDF

StatsNCTPDF(*x, df, d*)

The StatsNCTPDF function returns the probability distribution function of the noncentral Student-T distribution. *df* is the degrees of freedom (positive integer) and *d* is the noncentrality measure.

$$f(x;n,\delta)=\frac{n^{n/2}n!}{2^n e^{\delta^2/2}(n+x^2)^{n/2}\Gamma\left(\frac{n}{2}\right)}\left\{\frac{\sqrt{2}\delta x\,{}_1F_1\left(\frac{n}{2}+1;\frac{3}{2};\frac{\delta^2x^2}{2(n+x^2)}\right)}{(n+x^2)\Gamma\left(\frac{n+1}{2}\right)}+\frac{{}_1F_1\left(\frac{n+1}{2};\frac{1}{2};\frac{\delta^2x^2}{2(n+x^2)}\right)}{\sqrt{(n+x^2)}\Gamma\left(\frac{n}{2}+1\right)}\right\}$$

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentPDF**, **StatsStudentCDF**, and **StatsNCTCDF** functions.

StatsNormalCDF

StatsNormalCDF(*x, m, s*)

The StatsNormalCDF function returns the normal cumulative distribution function

$$F(x,\mu,\sigma)=\frac{1}{2}+\frac{1}{2}erf\left(\frac{x-\mu}{\sigma\sqrt{2}}\right),$$

where *erf* is the error function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **erf**, **StatsNormalPDF** and **StatsInvNormalCDF** functions.

StatsNormalPDF

`StatsNormalPDF(`***`x, m, s`***`)`

The StatsNormalPDF function returns the normal probability distribution function

$$f(x,\mu,\sigma)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsNormalCDF** and **StatsInvNormalCDF** functions.

StatsNPMCTest

`StatsNPMCTest [`***`flags`***`] [`***`wave1, wave2,… wave100`***`]`

The StatsNPMCTest operation performs a number of nonparametric multiple comparison tests. Output waves are saved in the current data folder according to the test(s) performed. Some tests are only appropriate when you have the same number of samples in all groups. StatsNPMCTest usually follows **StatsANOVA1Test** or **StatsKWTest**.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/CIDX=*controlIndex*	Performs nonparametric multiple comparisons on a control group specified by the zero-based *controlIndex* wave in the input list. Output is to the M_NPCCResults wave in the current data folder or optionally to a table. The output column contents are: the first contains the difference between the rank sums of the control and each of the other waves; the second contains the standard error (SE); the third contains the statistic q, defined as the ratio of the difference in rank sums to SE; the fourth contains the critical value which also depends on the tails specification (see /TAIL); and the fifth contains the conclusion with 0 to reject H_0 and 1 to accept it. One version of this test applies when all inputs contain the same number of samples. When that is not the case, it uses the Dunn-Hollander-Wolfe approach to compute an appropriate SE and to handle possible ties.
/CONW=*cWave*	Performs a nonparametric multiple contrasts tests. *cWave* has one point for each input wave. The *cWave* value is 1 to include the corresponding (zero based) input wave in the first group, 2 to include the wave in the second group, or zero to exclude the wave. The contrast is defined as the difference between the normalized sum of the ranks of the first group and that of the second group. If *cWave*={0,1,1,1,2}, then the contrast is computed as $\text{contrast: } \frac{1}{3}[R_{n1}+R_{n2}+R_{n3}]\text{-}R_{n4}$ where R_{ni} is the normalized rank sum of the samples from the corresponding input wave. Note the significance of allowing zeros in the contrast wave because the actual ranking is performed on the pool of all the samples.

Output is to the M_NPMConResults wave in the current data folder or optionally to a table. The output column contents are: the first is the contrast value; the second is the standard error (SE); the third is the statistic S, which is the ratio of the absolute value of the contrast to SE; the fourth is the critical value (from χ^2 the approximation); and the fifth is the conclusion with 0 to reject H_0 and 1 indicating acceptance.

This test supports input waves with different number of samples and can also handle tied ranks. Note that the contrast wave used here is structured differently than for **StatsMultiCorrelationTest**.

/DHW

Performs the Dunn-Holland-Wolfe test, which supports unequal number of samples and accounts for ties in the rank sums. Output is to the M_NPMCDHWResults wave in the current data folder or optionally to a table. The output column contents are: the first contains the difference between the means of the rank sums (rank sums divided by the number of samples in the group), the second contains the standard error (SE), the third contains the DHW statistic Q, the fourth contains the critical value, and the fifth contains the conclusion (0 to reject H_0 and 1 to accept).

/Q

No results printed in the history area.

/SWN

Creates a text wave containing wave names corresponding to each row of the comparison table. Depending on your choice of tests, the following wave names are created:

/CIDX test: T_NPCCResultsDescriptors

/DHW test: T_NPMCDHWDescriptors

/SNK test: T_NPMCSNKResultsDescriptors

/TUK test: T_NPMCTukeyDescriptors

/T=*k*

Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

The table is associated with the test and not with the data. If you repeat the test, it will update the table with the new results.

/TAIL=*tc*

Specifies H_0 with /CIDX.

tc=1:	One tailed test ($\mu_c \leq \mu_a$).
tc=2:	One tailed test ($\mu_c \geq \mu_a$).
tc=4:	Default; two tailed test ($\mu_c = \mu_a$).

Code combinations are not allowed.

/SNK

Performs a nonparametric variation on the Student-Newman-Keuls test where the standard error SE is a function of p (the rank difference). This test requires equal numbers of samples in all groups; use /DHW for unequal sizes.

Output is to the M_NPMCSNKResults wave in the current data folder. The output column contents are: the first contains the difference between rank sums, the second contains the standard error (SE), the third contains the p value (rank difference), the fourth the statistic, the fifth contains the critical value, and the sixth contains the conclusion (0 to reject H_0 and 1 to accept). This test is more sensitive to differences than the Tukey test (/TUK).

/TUK	Perform a Tukey-type (Nemenyi) multiple comparison test using the difference between the rank sums. This is the default that is performed if you do not specify any of the test flags. This test requires equal numbers of points in all waves; use /DHW for unequal sizes. Output is to the M_NPMCTukeyResults wave in the current data folder. The output column contents are: the first contains the difference between the rank sums, the second contains the SE values, the third contains the statistic q, the fourth contains the critical value for this specific alpha and the number of groups; and the last contains a conclusion flag with 0 indicating a rejection of H_0 and 1 indicating acceptance. H_0 postulates that the paired means are the same.
/WSTR=*waveListString*	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.
/Z	Ignores errors.

Details

Inputs to StatsNPMCTest are two or more 1D numerical waves (one wave for each group of samples) containing two or more valid entries. The waves must have the same number of points for the use /SNK and /TUK tests, otherwise, for waves of differing lengths you must use the Dunn-Hollander-Wolfe test (/DHW).

V_flag will be set to zero for no execution errors. Individual tests may fail if, for example, there are different number of samples in the input waves for a test that requires an equal number of points. StatsNPMCTest skips failed tests and V_flag will be a binary combination identifying the failed test(s):

`V_flag & 1`	Tukey method failed (/TUK).
`V_flag & 2`	Student-Newman-Keuls failed (/SNK).

V_flag will be set to -1 for any other errors.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test** and **StatsKWTest**.

For multiple comparisons in parametric tests see: **StatsDunnettTest** and **StatsScheffeTest**.

StatsNPNominalSRTest

StatsNPNominalSRTest [***flags***] [***srcWave***]

The StatsNPNominalSRTest operation performs a nonparametric serial randomness test for nominal data consisting of two types. The null hypothesis is that the data are randomly distributed. Output is to the W_StatsNPSRTest wave in the current data folder.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/P={*m*,*n*,*u*}	Provides a summary of the data instead of providing the nominal series. *m* is the number of elements of the first type, *n* is the number of elements of the second type, and *u* is the number of runs or contiguous sequences of each type. Do not use *srcWave* with /P.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors.

Details

The input wave to StatsNPNominalSRTest is specified with *srcWave* or /P. The wave must contain exactly two values. If *srcWave* is a text wave, then each type can be designated by a letter or by a short string (less than 200 bytes). If *srcWave* is numeric, you should avoid the usual floating point waves, which can give rise to internal representations of more than two distinct values. Output to W_StatsNPSRTest includes the total number of points (*N*), the number of occurrences (*m*) of the first variable, the number of occurrences (*n*) of the second variable, and the number of runs (*u*). When both *m* and *n* are less than 300, it computes the P value (probability P(*u*'<*u*)) and the critical values using the Swed and Eisenhart algorithm. When *m* or *n* are larger than 300, it computes the mean and standard deviation of an equivalent normal distribution with the corresponding critical value.

References

Swed, F.S., and C. Eisenhart, Tables for testing randomness of grouping in a sequence of alternatives, *Ann. Math. Statist.*, *14*, 66-87, 1943.

See, in particular, Chapter 25 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsSRTest**.

StatsParetoCDF

`StatsParetoCDF(x, a, c)`

The StatsParetoCDF function returns the Pareto cumulative distribution function

$$F(x;a,c) = 1 - \left(\frac{a}{x}\right)^c .$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsParetoPDF** and **StatsInvParetoCDF** functions.

StatsParetoPDF

`StatsParetoPDF(x, a, c)`

The StatsParetoPDF function returns the Pareto probability distribution function

$$\mathrm{f(x;a,c)} = \frac{\mathrm{c}}{\mathrm{x}}\left(\frac{a}{x}\right)^c , \qquad \begin{matrix} a,c > 0 \\ x \ge a. \end{matrix}$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsParetoCDF** and **StatsInvParetoCDF** functions.

StatsPermute

`StatsPermute(waveA, waveB, dir)`

The StatsPermute function permutes elements in *waveA* based on the lexicographic order of *waveB* and the direction *dir*. It returns 1 if a permutation is possible and returns 0 otherwise. Use *dir*=1 for the next permutation and *dir*=-1 for a previous permutation.

Details

Both *waveA* and *waveB* must be numeric. The lexicographic order of elements in the index wave is set so that permutations start with the index wave *waveB* in ascending order and end in descending order. Elements of *waveA* are permuted in place according to the order of the indices in *waveB* which are clipped (after permutation) to the valid range of entries in *waveA*. *waveB* is also permuted in place in order to allow you to obtain sequential permutations. If *waveA* consists of real numbers you can permute them using the lexicographic value of the entries directly. To do so pass `$""` for *waveB*. Whenever it returns 0, neither *waveA* and *waveB* are changed.

Examples

```
Function AllPermutations(num)
   Variable num

   Variable i,nf=factorial(num)
   Make/O/N=(num) wave0=p+1,waveA,waveB=p

   Print wave0
   for(i=0;i<nf;i+=1)
      waveA=wave0
      if(statsPermute(waveA,waveB,1)==0)
         break
      endif
      print waveA
   endfor
end

Executing AllPermutations(3) prints:
  wave0[0]= {1,2,3}
  waveA[0]= {1,3,2}
  waveA[0]= {2,1,3}
  waveA[0]= {2,3,1}
  waveA[0]= {3,1,2}
  waveA[0]= {3,2,1}
```

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsPoissonCDF

`StatsPoissonCDF(`*`x`*`, λ)`

The StatsPoissonCDF function returns the Poisson cumulative distribution function

$$F(x;\lambda)=\sum_{i=0}^{x}\frac{\exp(-\lambda)\lambda^{i}}{i!}, \qquad x=0,1,2...$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPoissonPDF** and **StatsInvPoissonCDF** functions.

StatsPoissonPDF

`StatsPoissonPDF(`*`x`*`, λ)`

The StatsPoissonPDF function returns the Poisson probability distribution function

$$f(x;\lambda)=\frac{\exp(-\lambda)\lambda^{x}}{x!}, \qquad x=0,1,2...$$

where λ is the shape parameter.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPoissonCDF** and **StatsInvPoissonCDF** functions.

StatsPowerCDF

`StatsPowerCDF(`*`x`*`, `*`b`*`, `*`c`*`)`

The StatsPowerCDF function returns the Power Function cumulative distribution function

$$F(x;b,c)=\left(\frac{x}{b}\right)^{c}$$

where the scale parameter *b* and the shape parameter *c* satisfy $b,c > 0$ and $b \geq x \geq 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPowerPDF**, **StatsInvPowerCDF** and **StatsPowerNoise** functions.

StatsPowerNoise

```
StatsPowerNoise(b, c)
```

The StatsPowerNoise function returns a pseudorandom value from the power distribution function with probability distribution:

$$f(x;b,c) = \frac{c}{x}\left(\frac{x}{b}\right)^c .$$

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use SetRandomSeed. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

The **StatsPowerPDF StatsInvPowerCDF** and **StatsInvPowerCDF** functions.

Noise Functions on page III-344.

Chapter III-12, **Statistics** for a function and operation overview.

StatsPowerPDF

```
StatsPowerPDF(x, b, c)
```

The StatsPowerPDF function returns the Power Function probability distribution function

$$f(x,b,c) = \frac{|c|}{x}\left(\frac{\mathrm{x}}{b}\right)^c ,$$

where b is a scale parameter and c is a shape parameter.

For b,c > 0, x is drawn from b >= x >= 0.

For b>0, c<0, x is drawn from x>b.

For b<0, c>0, x is drawn from -b <= x <= 0.

For b<0, c<0, x is drawn from x<-b.

Note that for -1<c<0 the average diverges and the magnitude of a mean calculated from N samples will increase indefinitely with N.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsPowerCDF**, **StatsInvPowerCDF** and **StatsPowerNoise** functions.

StatsQCDF

```
StatsQCDF(q, r, c, df)
```

The StatsQCDF function returns the value of the Q cumulative distribution function for *r* the number of groups, *c* the number of treatments, and *df* the error degrees of freedom (*f*=*rc*(*n*-1) with sample size *n*).

Details

The Q distribution is the maximum of several Studentized range statistics. For a simple Tukey test, use *r*=1.

References

Copenhaver, M.D., and B.S. Holland, Multiple comparisons of simple effects in the two-way analysis of variance with fixed effects, *Journal of Statistical Computation and Simulation*, *30*, 1-15, 1988.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTukeyTest** function.

StatsQpCDF

`StatsQpCDF(`*`q, nr, nt, dt, side, sSizeWave`*`)`

The StatsQpCDF function returns the Q' cumulative distribution function associated with Dunnett's test.

Here *nr* is the number of groups (should be set to 1), *nt* is the number of treatments, *df* is the error degrees of freedom.

Set *side*=1 for upper-tail or *side*=2 for two-tailed CDF.

sSizeWave is an integer wave of nt rows specifying the number of samples in each treatment.

Details

StatsQpCDF is a modified Q distribution typically used with Dunnett's test, which compares the various means with the mean of the control group or treatment

References

"Algorithm AS 251: Multivariate Normal Probability Integrals with Product Correlations Structure", C. W. Dunnett, *Appl. Stat.*, 38 (1989) 564-579.

A short correction for the algorithm was published in: *Appl. Stat.*, 42 (1993) 709.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsDunnettTest**, **StatsInvQpCDF**, and **StatsInvQCDF** functions.

StatsQuantiles

`StatsQuantiles` `[`*`flags`*`]` *`srcWave`*

The StatsQuantiles operation computes quantiles and elementary univariate statistics for a set of data in *srcWave*.

Flags

/ALL	Invokes all flags except /Q, /QM, and /Z.
/BOX	Computes parameters necessary to construct a box plot.
/iNaN	Ignores NaNs, which are sorted to the end of the array by default.
/IW	Creates an index wave W_QuantilesIndex. W_QuantilesIndex[*i*] corresponds to the position of *srcWave*[*i*] when sorted from minimum to maximum.
/Q	No information printed in the history area.
/QM=*qMethod*	Specifies the method for computing quartiles. *qMethod* has one of these values: 0: Tukey (default). 1: Minitab. 2: Moore and McCabe. 3: Mendenhall and Sincich. See Details for more information.
/QW	Creates a single precision wave W_QuantileValues containing the quantile value corresponding to each entry in *srcWave*.
/STBL	Uses a stable sort, which may require significant computation time for multiple entries with the same value.

/T=*k*	Displays the result wave W_StatsQuantiles in a table and specifies window behavior when the user attempts to close the table. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. If you use /K=2 you can still kill the window using the **KillWindow** operation.
/TM	Computes the tri-mean: 0.25*(V_Q25+2*median+V_Q75).
/TRIM=*tVal*	Computes the trimmed mean which is the mean value of the entries between the quantiles *tVal* (in %) and 100-*tVal*. By default *tVal*=25 and the trimmed mean corresponds to the midmean.
/Z	Ignores any errors.

Details

StatsQuantiles produces quick five-number summaries or more detailed results for univariate data. Values are returned in the wave W_StatsQuantiles and in the variables:

V_min	Minimum value.
V_max	Maximum value.
V_Median	Median value.
V_Q25	Lower quartile.
V_Q75	Upper quartile.
V_IQR	Inter-quartile range V_Q75-VQ25, which is also known as the H-spread.
V_MAD	Median absolute deviation.
V_mode	The most frequent value. If there is a tie and several values have the highest frequency then the lowest value among them is returned as the mode. If all values in srcWave are unique or if the number of points in srcWave is less than 3, V_mode is set to NaN. This output was added in Igor Pro 7.00.

Entries in the wave W_StatsQuantiles depend on your choice of flags. Each row has a row label explicitly defining its value. If you use the /ALL flag, W_StatsQuantiles will contain the following row labels:

minValue	lowerInnerFence
maxValue	lowerOuterFence
Median	upperInnerFence
Q25	upperOuterFence
Q75	triMean
IQR	trimmedMean
MedianAbsoluteDeviation	

Otherwise, W_StatsQuantiles will contain the first five entries and any additionally requested value. You should always access values using the dimension labels (see **Dimension Labels** on page II-85).

There is frequently some confusion in comparing statistical results computed by different programs because each may use a different definition of quartiles. You can specify the method of computing the quartiles as you prefer with the /QM flag. If you neglect to choose a method, StatsQuantiles uses Tukey's method, which computes quartiles (also called hinges) as the lower and upper median values between the median of the data and the edges of the array. The Moore and McCabe method is similar to Tukey's method except you do not include the median itself in computing the quartiles. Mendenhall and Sincich compute the quartiles using 1/4 and 3/4 of (numDataPoints+1) and round to the nearest integer (if the fraction part is exactly 0.5 they round up for the lower quartile and down for the upper quartile). Minitab uses the same expressions but instead of rounding it uses linear interpolation.

StatsQuantiles uses a stable index sorting routine so that

```
IndexSort W_QuantilesIndex,srcWave
```

is a monotonically increasing wave.

References

Tukey, J. W., *Exploratory Data Analysis*, 688 pp., Addison-Wesley, Reading, Massachusetts, 1977.

Mendenhall, W., and T. Sincich, *Statistics for Engineering and the Sciences*, 4th ed., 1008 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1995.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **WaveStats**, **StatsMedian**, **Sort**, and **MakeIndex**.

StatsRankCorrelationTest

`StatsRankCorrelationTest` `[`***`flags`***`]` ***`waveA, waveB`***

The StatsRankCorrelationTest operation performs Spearman's rank correlation test on *waveA* and *waveB*, 1D waves containing the same number of points. Output is to the W_StatsRankCorrelationTest wave in the current data folder.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors.

Details

StatsRankCorrelationTest ranks *waveA* and *waveB* and then computes the sum of the squared differences of ranks for all rows. Ties are assigned an average rank and the corrected Spearman rank correlation coefficient is computed with ties. It reports the sum of the squared ranks (sumDi2), the sums of the ties coefficients (sumTx and sumTy respectively), the Spearman rank correlation coefficient (in the range [-1,1]), and the critical value. H_0 corresponds to zero correlation against the alternative of nonzero correlation. The critical value is usually lower than the one in published tables. When the first derivative of the CDF is discontinuous, tables tend to use a more conservative value by choosing the next transition of the CDF as the critical value. StatsRankCorrelationTest is not as powerful as **StatsLinearCorrelationTest**.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsLinearCorrelationTest, **StatsCircularCorrelationTest**, **StatsKendallTauTest**, **StatsSpearmanRhoCDF**, and **StatsInvSpearmanCDF**.

StatsRayleighCDF

`StatsRayleighCDF(`*`x`*`[, `*`s`*`[, `*`m`*`]])`

The StatsRayleighCDF function returns the Rayleigh cumulative distribution function

$$F(x;\sigma,\mu) = 1 - \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \qquad \sigma > 0, x > \mu.$$

with defaults *s*=1 and *m*=0. It returns NaN for *s* ≤ 0 and zero for x ≤ *m*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRayleighPDF** and **StatsInvRayleighCDF** functions.

StatsRayleighPDF

```
StatsRayleighPDF(x [, s [, m]])
```

The StatsRayleighPDF function returns the Rayleigh probability distribution function

$$f(x;\sigma,\mu) = \frac{x-\mu}{\sigma^2}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \qquad \sigma > 0, x > \mu.$$

with defaults *s*=1 and *m*=0. It returns NaN for *s* ≤ 0 and zero for x ≤ *m*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRayleighCDF** and **StatsInvRayleighCDF** functions.

StatsRectangularCDF

```
StatsRectangularCDF(x, a, b)
```

The StatsRectangularCDF function returns the rectangular (uniform) cumulative distribution function

$$F(x,a,b) = \begin{cases} 0 & x \le a \\ \dfrac{x-a}{b-a} & a \le x \le b \\ 1 & x \ge b \end{cases}$$

where *a*< *b*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRectangularPDF** and **StatsInvRectangularCDF** functions.

StatsRectangularPDF

```
StatsRectangularPDF(x, a, b)
```

The StatsRectangularPDF function returns the rectangular (uniform) probability distribution function

$$f(x;a,b) = \begin{cases} \dfrac{1}{b-a} & a \le x \le b \\ 0 & otherwise \end{cases}$$

where *a*< *b*.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRectangularCDF** and **StatsInvRectangularCDF** functions.

StatsResample

StatsResample /N=*numPoints* [*flags*] *srcWave*

The StatsResample operation resamples *srcWave* by drawing (with replacement) *numPoints* values from *srcWave* and storing them in the wave W_Resampled or M_Resampled if /MC is used. You can iterate the process and compute various statistics on the data samples.

Flags

/ITER=*n* Repeats the resampling for *n* iterations, which is useful only when combined with /WS or /SQ.

/JCKN=*ufunc* Performs Jack-Knife analysis. Here ufunc is a user function of the format:

```
Function ufunc(inWave)
    wave inWave
    ... compute some statistic for inWave
    return someValue
End
```

The results are stored in the wave W_JackKnifeStats in the current data folder. Use

```
Edit W_JackKnifeStats.ld
```

to display the wave with dimension labels.

The idea behind this method is that *ufunc* returns some statistic z for inWave which is a subsample of srcWave of size (n-1). There are exactly n iterations and in each iteration the operation calls *ufunc* with one element of srcWave missing and stores the result in an internal array. At the end of iterations it uses the array to compute the various Jack-Knife estimates.

The standard estimator is defined as:

$$Z = ufunc(srcWave).$$

The Jack-Knife estimator is simply:

$$\hat{z} = \frac{1}{n}\sum_{i=1}^{n} z_i .$$

The Jack-Knife t-estimator is slightly less biased. It is given by:

$$t = nZ - (n-1)\hat{z},$$

The estimate of the standard error is given by:

$$\hat{\sigma}_{\hat{z}} = \sqrt{\frac{n-1}{n}\sum_{i=1}^{n}(z_i - \hat{z})^2} .$$

/K Kills W_Resampled after passing it to WaveStats. When /ITER is used, W_Resampled is not saved.

/MC Use /MC when you want to sample random (complete) rows from a multi-column 2D *srcWave*. The combination of /N=n with /MC results in the wave M_Resampled in the current data folder. M_Resampled will have n rows, the same number of columns and the same data type as *srcWave*.

/N=*numPoints* Specifies the number of points sampled from *srcWave*.

/Q	No information printed in the history area.
/SQ=*m*	Uses StatsQuantiles to compute the data quartiles. The methods are: *m*=0: Tukey (default). *m*=1: Minitab. *m*=2: Moore and McCabe. *m*=3: Mendenhall and Sincich. See Details for information about how the results are stored. The default trim value is 25%.
/WS=*m*	Uses WaveStats operation to calculate data statistics. *m*=0: Creates a new wave containing the samples (default). *m*=1: Creates the new wave and passes it to `WaveStats/Q/M=1`. *m*=2: Creates the new wave and passes it to `WaveStats/Q/M=2`. See Details for information about how the results are stored.
/Z	Ignores any errors.

Details

StatsResample can perform Bootstrap Analysis, permutations tests, and Monte-Carlo simulations. It draws the specified number of data points (with replacement) from *srcWave* and places them in a destination wave W_Resampled.

Specify /WS or /SQ to use the WaveStats or StatsQuantiles operations, respectively, to compute results directly from the data. StatsResample normally creates the wave W_Resampled and, optionally, the M_WaveStats and W_StatsQuantiles waves. Both options also create various V_ variables described below. If you use more than one iteration, StatsResample creates instead the waves M_WaveStatsSamples and M_StatsQuantilesSamples for the results.

M_WaveStatsSamples (with /WS) contains a column for each iteration. Each column is equivalent to the contents of M_WaveStats for that iteration. You can use the command

```
Edit M_WaveStatsSamples.ld
```

to display the results in a table using row labels, and, for example, to display a graph of the rms of the samples as a function of iteration number execute:

```
Display M_WaveStatsSamples[5][]
```

M_StatsQuantilesSamples (with /SQ) contains a column for each iteration. Each column consists of the contents of W_StatsQuantiles for the corresponding data. Here again you can execute the command

```
Edit M_StatsQuantilesSamples.ld
```

to display the wave in a table using row labels. To display a graph of the median as a function of iteration execute:

```
Display M_statsQuantilesSamples[2][]
```

Output Variables

StatsResample creates the following variables: V_Median, V_Q25, V_Q75, V_IQR, V_min, V_max, V_numNaNs, V_numINFs, V_avg, V_sdev, V_rms, V_adev, V_skew, V_kurt, and V_Sum.

These variables are valid only if you use either /SQ or /WS, but not both, and only if you do not use /ITER. Unused variables are set to NaN.

If you use /SQ the operation sets V_Median, V_Q25, V_Q75, V_IQR, V_min, and V_max.

If you use /WS the operation sets V_min, V_max, V_numNaNs, V_numINFs, V_avg, V_sdev, V_rms, V_adev, V_skew, V_kurt, and V_Sum.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsSample**, **WaveStats** and **StatsQuantiles**.

StatsSample

`StatsSample /N=`*`numPoints`*` [`*`flags`*`] `*`srcWave`*

StatsSample creates a random, non-repeating sample from *srcWave*.

It samples *srcWave* by drawing without replacement *numPoints* values from *srcWave* and storing them in the output wave W_Sampled or M_Sampled if /MC or /MR are used.

Flags

/ACMB	Creates a wave containing all unique combinations of *numPoints* values from srcWave. It is assumed that *srcWave* is a 1D numeric wave containing more than *numPoints* elements. The results are stored in the wave M_Combinations in the current data folder. Each row in the result wave corresponds to a unique combination of samples. Added in Igor Pro 7.00.
/N=*numPoints*	Specifies the number of points sampled from *srcWave*. When combined with /MC, *numPoints* is the number of sampled rows and when combined with /MR, it is the number of sampled columns.
/MC	Use /MC (multi-column) to randomly sample full rows from *srcWave*, i.e., the output consists of all columns of each selected row. /MC and /MR are mutually exclusive flags.
/MR	Use /MR (multi-row) to randomly sample full columns from *srcWave*, i.e., the output consists of all rows of each of the selected columns. /MC and /MR are mutually exclusive flags.
/Z	Ignores errors.

Details

If you omit /MC and /MR, the output is a 1D wave named W_Sampled where the samples are chosen from *srcWave* without regard to its dimensionality.

If you use either /MC or /MR the output is a 2D wave named M_Sampled which will have either the same number of columns (/MC) as *srcWave* or the same number of rows (/MR) as *srcWave*.

See Also

Chapter III-12, **Statistics**, **StatsResample**

StatsRunsCDF

`StatsRunsCDF(`*`n, r`*`)`

The StatsRunsCDF function returns the cumulative distribution function for the up and down runs distribution for total number of runs *r* in a random linear arrangement of *n* unequal elements. There is no closed form expression. It is computed numerically from the recursion of the probability density

$$f(r,n) = \frac{rf(r,n-1) + 2f(r-1,n-1) + (n-r)f(r-2,n-1)}{n},$$

with the initial condition

$$f(1,n) = \frac{2}{n!}.$$

References

Bradley, J.V., *Distribution-Free Statistical Tests*, Prentice Hall, Englewood Cliffs, New Jersey, 1968.

Olmstead, P.S., Distribution of sample arrangements for runs up and down, *Annals of Mathematical Statistics*, *17*, 24-33, 1946.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsSRTest** function.

StatsScheffeTest

`StatsScheffeTest [`***`flags`***`] [`***`wave1, wave2,… wave100`***`]`

The StatsScheffeTest operation performs Scheffe's test for the equality of the means. It supports two basic modes: the default tests all possible combinations of pairs of waves; the second tests a single combination where the precise form of H_0 is determined by the coefficients of a contrast wave (see /CONT). Output is to the M_ScheffeTestResults wave in the current data folder.

Flags

/ALPH=*val* — Sets the significance level (default 0.05).

/CONW=*cWave* — Performs a multiple contrasts test. *cWave* has one point for each input wave. The *cWave* value is 1 to include the corresponding (zero based) input wave in the first group, 2 to include the wave in the second group, or zero to exclude the wave.

The contrast is defined as the difference between the normalized sum of the ranks of the first group and that of the second group. If *cWave*={0,1,1,1,2}, then the contrast hypothesis H_0 corresponds to:

$$\frac{\bar{X}_1+\bar{X}_2+\bar{X}_3}{3}-\bar{X}_4=0.$$

For each pair of waves (*i*, *j*) with *i* ¦ *j*, it computes

$$SE_{ij}=\sqrt{s^2\left(\frac{1}{n_j}+\frac{1}{n_i}\right)}, \qquad s^2=\sum_{i=1}^{W}\sum_{j=0}^{n_j-1}X_j^2-\sum_{i=1}^{W}\frac{\left(\sum_{j=0}^{n_j-1}X_j\right)^2}{n_j},$$

the statistic

$$S=\frac{\left|\sum_{i=0}^{n-1}c_i\bar{X}_i\right|}{SE},$$

the critical value, and a result field which is set to 1 if H_0 should be accepted or 0 if it should be rejected. *W* is the total number of waves, n_i and $\bar{X}_i$ are respectively the number of data points and the average of wave *i*.

/Q — No results printed in the history area.

/SWN — Creates a text wave, T_ScheffeDescriptors, containing wave names corresponding to each row of the comparison table (Save Wave Names). Use /T to append the text wave to the last column.

/T=*k* — Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/WSTR=*waveListString*	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The default of StatsScheffeTest (also known as the S test) tests the hypotheses of equality of means for each possible pair of samples. It is not as powerful as Tukey's test (**StatsTukeyTest**) and is more useful for hypotheses formulated as multiple contrasts (see /CONT).

References

See, in particular, Chapter 11 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test**, **StatsDunnettTest** and **StatsTukeyTest**.

StatsShapiroWilkTest

`StatsShapiroWilkTest [`*`flags`*`]` ***`srcWave`***

The StatsShapiroWilkTest computes Shapiro-Wilk statistic W and its associated P-value and stores them in V_statistic and V_prob respectively.

Flags

/Q	No results printed in the history area.
/Z	Ignores errors.

Details

The Shapiro-Wilk tests the null hypothesis that the population is normally distributed. If the P-value is less than the selected alpha then the null hypothesis, normality, is rejected.

The test is valid only for waves containing 3 to 5000 data points. The operation ignores any NaNs or INFs in *srcWave*.

Example

```
// Test normally distributed data
Make/O/N=(200) ggg=gnoise(5)
StatsShapiroWilkTest ggg
W=0.995697 p=0.846139            // p>alpha so accept normality

// Test uniform distribution
Make/O/N=(200) eee=enoise(5)
StatsShapiroWilkTest eee
W=0.959616 p=1.7979e-05          // p<alpha so reject normality
```

StatsSignTest

`StatsSignTest` `[`***`flags`***`]` ***`wave1, wave2`***

The StatsSignTest operation performs the sign test for paired-sample data contained in *wave1* and *wave2*.

Flags

/ALPH=*val*	Sets the significance level (default 0.05).
/Q	No results printed in the history area.

/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

The input waves must be the equal length, real numeric waves and must not contain any NaNs or INFs. Results are saved in the wave W_SignTest and are optionally displayed in a table. StatsSignTest computes the differences in each pair and counts the total number of entries with positive and negative differences, and tests the results using a binomial distribution. When the number of data pairs exceeds 1024 it uses a normal approximation to the binomials for calculating the probabilities and the power of the test.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsWilcoxonRankTest

StatsSpearmanRhoCDF

`StatsSpearmanRhoCDF(`*`r, N`*`)`

The StatsSpearmanRhoCDF function returns the cumulative distribution function for Spearman's *r*, which is used in rank correlation test. It is valid for *N*>1 and -1 ≤ *r* ≤ 1. The distribution is mostly computed using the Edgeworth series expansion.

References

Algorithm AS 89, *Appl. Statist*., *24*, 377, 1975.

van de Wiel, M.A., and A. Di Bucchianico, Fast computation of the exact null distribution of Spearman's rho and Page's L statistic for samples with and without ties, *J. of Stat. Plan. and Inference*, *92*, 133-145, 2001.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRankCorrelationTest**, **StatsInvSpearmanCDF**, and **StatsKendallTauTest** functions.

StatsSRTest

`StatsSRTest [`*`flags`*`]` ***`srcWave`***

The StatsSRTest operation performs a parametric or nonparametric serial randomness test on *srcWave*, which must contain finite numerical data. The null hypothesis of the test is that the data are randomly distributed. Output is to the W_StatsSRTest wave in the current data folder.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/GCD	Tests the output of a random number generator (RNG). *srcWave* consists of values between 0 and 2^{32} (converted to unsigned 32-bit integers). GCD computes the gcd for consecutive pairs of data in *srcWave*. The number of steps in the GCD and the distribution of the GCD's are compared with ideal distributions and corresponding P values are reported. This test is part of Marsaglia's Die-Hard battery of tests. P-values close to either 0 or 1 indicate a nonideal RNG. You should use the reported minimum and maximum values to check that the input is indeed in the proper range. Typically *srcWave* consists of at least1e6 entries.
/NAPR	Use the normal approximation even when the number of points is below 150.

/NP	Performs a nonparametric serial randomness test by counting the numbers of runs up and down and computing the probability that such a value is obtained by chance.
/P	Performs a parametric serial randomness test.
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

/Z	Ignores errors.

Details

The parametric test for serial randomness is according to Young. *C* is given by

$$C = 1 - \frac{\sum_{i=0}^{n-2}\left(X_i - X_{i+1}\right)^2}{2\sum_{i=0}^{n-1}\left(X_i - \overline{X}\right)^2},$$

where $\bar{X}$ is the mean and n is the number of points in *srcWave*. The critical value is obtained from mean square successive difference distribution **StatsInvCMSSDCDF**. For more than 150 points, StatsSRTest uses the normal approximation and provides the critical values from the normal distribution. For samples from a normal distribution, *C* is symmetrically distributed about 0 with positive values indicating positive correlation between successive entries and negative values corresponding to negative correlation.

The nonparametric test consists of counting the number of runs that are successive positive or successive negative differences between sequential data. If two sequential data are the same it computes two numbers of runs by considering the two possibilities where the equality is replaced with either a positive or a negative difference. The results of the operation include the number of runs up and down, the number of unchanged values (the number of places with no difference between consecutive entries), the size of the longest run and its associated probability, the number of converted equalities, and the probability that the number of runs is less than or equal to the reported number (**StatsRunsCDF**). When equalities are encountered the operation computes the probabilities that the computed number of runs or less can be found in an equivalent random sequence.

Converted equalities are those with the same sign on both sides so that when we replace the equality by the opposite sign we increase the number of runs. The equalities that are not converted are found between two different signs and therefore regardless of the sign that we give them they do not affect the total number of runs. We implicitly assume that the data does not contain more than one sequential equalities.

The longest run is determined without taking into account equalities or their conversions. The probability of the longest run is computed from Equation 6 of Olmstead, which is accurate within 0.001 when the number of runs is 5 or more. This probability applies to either positive or negative differences and should be divided by two if a specific sign is selected.

References

Bradley, J.V., *Distribution-Free Statistical Tests*, Prentice Hall, Englewood Cliffs, New Jersey, 1968.

Olmstead, P.S., Distribution of sample arrangements for runs up and down, *Annals of Mathematical Statistics*, *17*, 24-33, 1946.

Wallis, W.A., and G.H. Moore, A significance test for time series, *J. Amer. Statist. Assoc.*, *36*, 401-409, 1941.

Young, L.C., On randomness in ordered sequences, *Annals of Mathematical Statistics*, *12*, 153-162, 1941.

See, in particular, Chapter 25 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

`<http://www.csis.hku.hk/cisc/projects/va/index.htm>`

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsNPNominalSRTest** and **StatsRunsCDF**.

StatsStudentCDF

`StatsStudentCDF(t, n)`

The StatsStudentCDF function returns the Student (uniform) cumulative distribution function

$$F(t,n)=\begin{cases}\frac{1}{2}\left\{1+I\left(\frac{n}{2},\frac{1}{2};1\right)-I\left(\frac{n}{2},\frac{1}{2};\frac{n}{n+t^2}\right)\right\} & t>0\\ \frac{1}{2}\left\{1+I\left(\frac{n}{2},\frac{1}{2};\frac{n}{n+t^2}\right)-I\left(\frac{n}{2},\frac{1}{2};1\right)\right\} & t<0\\ \frac{1}{2} & t=0\end{cases}$$

where n>0 is degrees of freedom and is the incomplete beta function **betai**.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentPDF** and **StatsInvStudentCDF** functions.

StatsStudentPDF

`StatsStudentPDF(t, n)`

The StatsStudentPDF function returns the Student (uniform) probability distribution function

$$f(t,n)=\frac{\left(\frac{n}{n+t^2}\right)^{(n+1)/2}}{\sqrt{n}B\left(\frac{n}{2},\frac{1}{2}\right)}.$$

where n>0 is degrees of freedom and B() is the **beta** function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsStudentCDF** and **StatsInvStudentCDF** functions.

StatsTopDownCDF

`StatsTopDownCDF(r, N)`

The StatsTopDownCDF function returns the cumulative distribution function for the top-down correlation coefficient. It is computationally intensive because it must evaluate many permutations [O((n!)^2)]. It exactly calculates the distribution for $3 \le N \le 7$; outside this range it uses Monte-Carlo estimation for $8 \le N \le 50$ and asymptotic Normal approximation for N>50. The Monte-Carlo estimate uses 1e6 random permutations fitted with two 9-order polynomials for the range [-1,0] and [0,1]. The results are within 0.2% of exact values where known.

References

Iman, R.L., and W.J. Conover, A measure of top-down correlation, *Technometrics*, *29*, 351-357, 1987.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsRankCorrelationTest** and **StatsInvTopDownCDF** functions.

StatsTriangularCDF

`StatsTriangularCDF(`*`x, a, b, c`*`)`

The StatsTriangularCDF function returns the triangular cumulative distribution function

$$F(x;a,b,c) = \begin{cases} \dfrac{(x-a)^2}{(b-a)(c-a)} & a \le x \le c \\ 1 - \dfrac{(b-x)^2}{(b-a)(c-a)} & c \le x \le b. \end{cases}$$

where $a<c<b$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTriangularPDF** and **StatsInvTriangularCDF** functions.

StatsTriangularPDF

`StatsTriangularPDF(`*`x, a, b, c`*`)`

The StatsTriangularPDF function returns the triangular probability distribution function

$$f(x;a,b,c) = \begin{cases} \dfrac{2(x-a)}{(b-a)(c-a)} & a \le x \le c \\ \dfrac{2(b-x)}{(b-a)(c-a)} & c < x < b \\ 0 & otherwise. \end{cases}$$

where $a<c<b$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsTriangularCDF** and **StatsInvTriangularCDF** functions.

StatsTrimmedMean

`StatsTrimmedMean(`*`waveName, trimValue`*`)`

The StatsTrimmedMean function returns the mean of the wave *waveName* after removing *trimValue* fraction of the values from both tails of the distribution. *trimValue* is a number in the range [0, 0.5]. *waveName* can be any real numeric type.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsQuantiles** and **mean**.

StatsTTest

`StatsTTest [`*`flags`*`]` ***`wave1`*** **`[,`** ***`wave2`*****`]`**

The StatsTTest operation performs two kinds of T-tests: the first compares the mean of a distribution with a specified mean value (/MEAN) and the second compares the means of the two distributions contained in *wave1* and *wave2*, which must contain at least two data points, can be any real numeric type, and can have an arbitrary number of dimensions. Output is to the W_StatsTTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/CI	Computes the confidence intervals for the mean(s).

/DFM=*m*	Specifies method for calculating the degrees of freedom.

m=0:	Default; computes equivalent degrees of freedom accounting for possibly different variances.
m=1:	Computes equivalent degrees of freedom but truncates to a smaller integer.
m=2:	Computes degrees of freedom by $DF=n_1+n_2-2$, where *n* is the sum of points in the wave. Appropriate when variances are equal.

/MEAN=*meanV* — Compares *meanV* with the mean of the distribution in *wave1*. Outputs are the number of points in the wave, the degrees of freedom (accounting for any NaNs), the average, standard deviation (σ),

$$s_{\overline{X}} = \frac{\sigma}{\sqrt{DF+1}},$$

the statistic

$$t = \frac{\overline{X} - meanV}{s_{\overline{X}}}$$

and the critical value, which depends on /TAIL.

/PAIR — Specifies that the input waves are pairs and computes the difference of each pair of data to get the average difference $\bar{d}$ and the standard error of the difference $S_{\bar{d}}$. The t statistic is the ratio of the two

$$t = \frac{\bar{d}}{s_{\bar{d}}}.$$

In this case H_0 is that the difference $\bar{d}$ is zero.

This mode does not support /CI and /DFM.

/Q — No results printed in the history area.

/T=*k* — Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/TAIL=*tailCode*	Specifies H_0.	
	tailCode=1:	One tailed test ($\mu_1 \leq \mu_2$).
	tailCode=2:	One tailed test ($\mu_1 \geq \mu_2$).
	tailCode=4:	Default; two tailed test ($\mu_1 = \mu_2$).
	When performing paired tests using /PAIR:	
	tailCode=1:	One tailed test ($\mu_d \leq 0$).
	tailCode=2:	One tailed test ($\mu_d \geq 0$).
	tailCode=4:	Default; two tailed test ($\mu_d = 0$).
	Here μ_d is the mean of the difference population.	
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.	

Details

When comparing the mean of a single distribution with a hypothesized mean value, you should use /MEAN and only one wave (*wave1*). If you use two waves StatsTTest performs the T-test for the means of the corresponding distributions (which is incompatible with /MEAN).

When comparing the means of two distributions, the default t-statistic is computed from Welch's approximate t:

$$t' = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}},$$

where s_i^2 are variances, n_i the number of samples, and $\bar{X}_i$ the averages of the respective waves. This expression is appropriate when the number of points and the variances of the two waves are different. If you want to compute the t-statistic using pooled variance you can use the /AEVR flag. In this case the pooled variance is given by

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2},$$

and the t-statistic is

$$t = \frac{\bar{x}_1 - \bar{x}_2}{s_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}.$$

The different test are:

H_0	Rejection Condition
$\mu_1 = \mu_2$	$\lvert t \rvert \geq$ Tc(alpha,ν)
$\mu_1 > \mu_2$	$t \leq$ Tc(alpha, ν)
$\mu_1 < \mu_2$	$t \geq$ Tc(alpha, ν)

Tc is the critical value and ν is the effective number of degrees of freedom (see /DFM flag).When accounting for possibly unequal variances, ν is given by

$$\nu = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2 - 1}}.$$

The critical values (Tc) are computed by numerically by solving for the argument at which the cumulative distribution function (CDF) equals the appropriate values for the tests. The CDF is given by

$$F(x) = \begin{cases} \frac{1}{2} betai\left(\frac{\nu}{2}, \frac{1}{2}, \frac{\nu}{\nu + x^2}\right) & x < 0 \\ 1 - \frac{1}{2} betai\left(\frac{\nu}{2}, \frac{1}{2}, \frac{\nu}{\nu + x^2}\right) & x \geq 0. \end{cases}$$

To get the critical value for the upper one-tail test we solve F(x)=1-alpha. For the lower one-tail test we solve for x the equation F(x)=alpha. In the two-tailed test the lower critical value is a solution for F(x)=alpha/2 and the upper critical value is a solution for F(x)=1-alpha/2.

The T-test assumes both samples are randomly taken from normal population distributions.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsStudentCDF**, **StatsStudentPDF**, and **StatsInvStudentCDF**.

References

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999. See in particular Section 8.1.

StatsTukeyTest

`StatsTukeyTest [`*`flags`*`] [`*`wave1, wave2,… wave100`*`]`

The StatsTukeyTest operation performs multiple comparison Tukey (HSD) test and optionally the Newman-Keuls test. Output is to the M_TukeyTestResults wave in the current data folder. StatsTukeyTest usually follows **StatsANOVA1Test**.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/NK	Computes the Newman-Keuls test.
/Q	No results printed in the history area.
/SWN	Creates a text wave, T_TukeyDescriptors, containing wave names corresponding to each row of the comparison table (Save Wave Names). Use /T to append the text wave to the last column.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing.

/WSTR=*waveListString*

	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.
/Z	Ignores errors.

Details

Inputs to StatsTukeyTest are two or more 1D numeric waves (one wave for each group of samples) containing any numbers of points but with at least two or more valid entries.

The contents of the M_TukeyTestResults columns are: the first contains the difference between the group means $\bar{X}_i - \bar{X}_i$, the second contains SE (supports unequal number of points), the third contains the q statistic for the pair, and the fourth contains the critical q value, the fifth contains the conclusion with 0 to reject H_0 ($\mu_i == \mu_j$) or 1 to accept H_0, with /NK, the sixth contains the *p* values

$$p = rank[\bar{X}_i] - rank[\bar{X}_j] + 1,$$

the seventh contains the critical values, and the eighth contains the Newman-Keuls conclusion (with 0 to reject and 1 to accept H_0). The order of the rows is such that all possible comparisons are computed sequentially starting with the comparison of the group having the largest mean with the group having the smallest mean.

V_flag will be set to -1 for any error and to zero otherwise.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsANOVA1Test**, **StatsScheffeTest**, and **StatsDunnettTest**.

StatsUSquaredCDF

`StatsUSquaredCDF(`*`u2, n, m, method, useTable`*`)`

The StatsUSquaredCDF function returns the cumulative distribution function for Watson’s U^2 with parameters *u2* (U^2 statistic) and integer sample sizes *n* and *m*. The calculation is computationally intensive, on the order of binomial(*n*+*m*, *m*). Use a nonzero value for *useTable to* search a built-in table of values. If *n* and *m* cannot be found in the table, it will proceed according to *method*:

method	What It Does
0	Exact computation using Burr algorithm (could be slow).
1	Tiku approximation using chi-squared.
2	Use built-in table only and return a NaN if not in table.

For large *n* and *m*, consider using the Tiku approximation. To abort execution, press the **User Abort Key Combinations**.

Precomputed tables, using the algorithm described by Burr, contain these values:

n	*m*
4	4-30
5	5-30
6	6-30
7	7-30
8	8-26
9	9-22
10	10-18

n	*m*
11	11-16
12	12-14
13	13

Because *n* and *m* are interchangeable, *n* should always be the smaller value. For *n*>8 the upper limit in the table matched the maximum that can be computed using the Burr algorithm. There is no point in using method 0 with *m* values exceeding these limits.

References

Burr, E.J., Small sample distributions of the two sample Cramer-von Mises' W^2 and Watson's U^2, *Ann. Mah. Stat. Assoc.*, *64*, 1091-1098, 1964.

Tiku, M.L., Chi-square approximations for the distributions of goodness-of-fit statistics, *Biometrica*, *52*, 630-633, 1965.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWatsonUSquaredTest** and **StatsInvUSquaredCDF** functions.

StatsVariancesTest

`StatsVariancesTest [`*`flags`*`] [`*`wave1, wave2,… wave100`*`]`

The StatsVariancesTest operation performs Bartlett's or Levene's test to determine if wave variances are equal. Output is to the W_StatsVariancesTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/METH=*m*	Specifies the test type. *m*=0: Bartlett test (default). *m*=1: Levene's test using the mean. *m*=2: Modified Levene's test using the median. *m*=3: Modified Levene's test using the 10% trimmed mean.
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.
/WSTR=*waveListString*	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.
/Z	Ignores errors. V_flag will be set to -1 for any error and to zero otherwise.

Details

All tests define the null hypothesis by

$$H_0: \qquad \sigma_1^2 = \sigma_2^2 = ... = \sigma_k^2,$$

against the alternative

$$H_a: \quad \sigma_i^2 \neq \sigma_j^2 \text{ for at least one } i \neq j.$$

Bartlett's test computes:

$$T = \frac{(n-k)\ln\left(\sigma_w^2\right) - \sum_{i=1}^{k}(n_i - 1)\ln\left(\sigma_i^2\right)}{1 + \frac{1}{3(k-1)}\left[\sum_{i=1}^{k}\frac{1}{n_i - 1} - \frac{1}{N-k}\right]}.$$

Here σ^2_i is the variance of the *i*th wave, *N* is the sum of the points of all the waves, n_i is the number of points in wave *i*, and *k* is the number of waves. The weighted variance is given by

$$\sigma_w^2 = \sum_{i=1}^{k}\frac{(n_i - 1)\sigma_i^2}{N-k}.$$

H_0 is rejected if T is greater than the critical value taken from the χ^2 distribution computed by solving for *x*:

$$1 - alpha = 1 - gammq\left(\frac{k-1}{2}, \frac{x}{2}\right).$$

Levene's test computes:

$$W = \frac{(N-k)\sum_{i=1}^{k} n_i\left(\overline{Z}_i - \overline{Z}\right)^2}{(k-1)\sum_{i=1}^{k}\sum_{j=1}^{k}\left(Z_{ij} - \overline{Z}_i\right)^2},$$

where

$$Z_{ij} = \left|Y_{ij} - \overline{Y}_i\right|,$$

$$\overline{Z}_i = \frac{1}{n_i}\sum_{j=1}^{k} Z_{ij},$$

$$\overline{Z} = \frac{1}{N}\sum_{i=1}^{k}\sum_{j=1}^{k} Z_{ij}.$$

$\bar{Y}_i$ depends on /METH.

H_0 is rejected if *W* is greater than the critical value from the F distribution computed by solving for *x*:

$$1 - alpha = 1 - betai\left(\frac{\nu_2}{2}, \frac{\nu_1}{2}, \frac{\nu_2}{\nu_2 + \nu_1 x}\right).$$

References

NIST/SEMATECH, Bartlett's Test, in *NIST/SEMATECH e-Handbook of Statistical Methods*, <http://www.itl.nist.gov/div898/handbook/eda/section3/eda357.htm>, 2005.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsVonMisesCDF

`StatsVonMisesCDF(`*`x, a, b`*`)`

The StatsVonMisesCDF function returns the von Mises cumulative distribution function

$$F(\theta;a,b)=\frac{1}{2\pi I_0(b)}\int_0^\theta \exp\big(b\cos(x-a)\big)dx.$$

where $I_0(b)$ is the modified Bessel function of the first kind (**bessI**), and

$$0<\theta\le 2\pi$$
$$0<a\le 2\pi$$
$$b>0.$$

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsVonMisesPDF**, **StatsInvVonMisesCDF**, and **StatsVonMisesNoise** functions.

StatsVonMisesNoise

`StatsVonMisesNoise(`*`a, b`*`)`

The StatsVonMisesNoise function returns a pseudo-random number from a von Mises distribution whose probability density is

$$f(\theta;a,b)=\frac{\exp\big[b\cos(\theta-a)\big]}{2\pi I_0(b)},$$

where I_0 is the zeroth order modified Bessel function of the first kind.

References

Best, D.J., and N. I. Fisher, Efficient simulation of von Mises distribution, *Appl. Statist*., *28*, 152-157, 1979.

See Also

StatsVonMisesCDF, **StatsVonMisesPDF**, and **StatsInvVonMisesCDF**.

Noise Functions on page III-344.

Chapter III-12, **Statistics** for a function and operation overview

StatsVonMisesPDF

`StatsVonMisesPDF(`*`q, a, b`*`)`

The StatsVonMisesPDF function returns the von Mises probability distribution function

$$f(\theta;a,b)=\frac{\exp\big(b\cos\big(\theta-a\big)\big)}{2\pi I_0(b)}.$$

where $I_0(b)$ is the modified Bessel function of the first kind **bessI**, and

$$0<\theta\le 2\pi$$
$$0<a\le 2\pi$$
$$b>0.$$

References

Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed., Wiley, New York, 2000.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsVonMisesCDF**, **StatsInvVonMisesCDF**, and **StatsVonMisesNoise** functions.

StatsWaldCDF

`StatsWaldCDF(x, m, l)`

The StatsWaldCDF function returns the numerically evaluated inverse Gaussian or Wald cumulative distribution function.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWaldPDF** function.

StatsWaldPDF

`StatsWaldPDF(x, m, l)`

The StatsWaldPDF function returns the inverse Gaussian or Wald probability distribution function

$$f(x;\mu,\lambda)=\sqrt{\frac{\lambda}{2\pi x^3}}\exp\left[-\frac{\lambda(x-\mu)^2}{2\mu^2 x}\right]$$

where *x*, *m*, *l*> 0.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWaldCDF** function.

StatsWatsonUSquaredTest

`StatsWatsonUSquaredTest [flags] srcWave1, srcWave2`

The StatsWatsonUSquaredTest operation performs Watson's nonparametric two-sample U^2 test for samples of circular data. Output is to the W_WatsonUtest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.
/Z	Ignores errors.

Details

The input waves, *srcWave1* and *srcWave2*, each must contain at least two angles in radians (mod 2π), can have any number of dimensions, and can be single or double precision. They must not contain any NaNs or INFs.

The Watson U^2 H_0 postulates that the two samples came from the same population against the different populations alternative. In the calculation, StatsWatsonUSquaredTest ranks the two inputs, accounts for possible ties, computes the test statistic U^2, and compares it with the critical value. Because of the difficulty of computing the critical values, it always computes first the approximation due to Tiku and if possible it computes the exact critical value using the method outlined by Burr. You can evaluate the U2 CDF to get more information about the critical region.

V_flag will be set to -1 for any error and to zero otherwise.

References

We have found that this method leads to slightly different results depending on the compiler and the system on which it is implemented:

Burr, E.J., Small sample distributions of the two sample Cramer-von Mises' W2 and Watson's U2, *Ann. Mah. Stat. Assoc.*, *64*, 1091-1098, 1964.

Tiku, M.L., Chi-square approximations for the distributions of goodness-of-fit statistics, *Biometrica*, *52*, 630-633, 1965.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWatsonWilliamsTest**, **StatsWheelerWatsonTest**, **StatsUSquaredCDF**, and **StatsInvUSquaredCDF**.

StatsWatsonWilliamsTest

`StatsWatsonWilliamsTest [flags] [srcWave1, srcWave2, srcWave3,…]`

The StatsWatsonWilliamsTest operation performs the Watson-Williams test for two or more sample means. Output is to the W_WatsonWilliams wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.
/WSTR=*waveListString*	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.
/Z	Ignores errors.

Details

The StatsWatsonWilliamsTest must have at least two input waves, which contain angles in radians, can be single or double precision, and can be of any dimensionality; the waves must not contain any NaNs or INFs.

The Watson-Williams H_0 postulates the equality of the means from all samples against the simple inequality alternative. The test computes the sums of the sines and cosines from which it obtains a weighted r value (rw). According to Mardia, you should use different statistics depending on the size of rw: for rw>0.95 use the simple F statistic, but for 0.95>rw>0.7 you should use the F-statistic with the K correction factor. Otherwise you should use the t-statistic. StatsWatsonWilliamsTest computes both the (corrected) F-statistic and the t-statistic as well as their corresponding critical values.

V_flag will be set to -1 for any error and to zero otherwise.

References

See, in particular, Section 6.3 of:

Mardia, K.V., *Statistics of Directional Data*, Academic Press, New York, New York, 1972.

See, in particular, Chapter 27 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWatsonUSquaredTest** and **StatsWheelerWatsonTest**.

StatsWeibullCDF

`StatsWeibullCDF(x, m, s, g)`

The StatsWeibullCDF function returns the Weibull cumulative distribution function

$$F(x;\mu,\sigma,\gamma)=1-\exp\left[-\left(\frac{x-\mu}{\sigma}\right)^{\gamma}\right], \qquad x\geq\mu \ \ and\ \ \sigma,\gamma>0.$$

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWeibullPDF** and **StatsInvWeibullCDF** functions.

StatsWeibullPDF

`StatsWeibullPDF(x, m, s, g)`

The StatsWeibullPDF function returns the Weibull probability distribution function

$$f(x;\mu,\sigma,\gamma)=\frac{\gamma}{\sigma}\left(\frac{x-\mu}{\sigma}\right)^{\gamma-1}\exp\left[-\left(\frac{x-\mu}{\sigma}\right)^{\gamma}\right],$$

where *m* is the location parameter, *s* is the scale parameter, and *g* is the shape parameter with $x \geq m$ and *s*, $g > 0$.

See Also

Chapter III-12, **Statistics** for a function and operation overview; the **StatsWeibullCDF** and **StatsInvWeibullCDF** functions.

StatsWheelerWatsonTest

`StatsWheelerWatsonTest [flags] [srcWave1, srcWave2, srcWave3,…]`

The StatsWheelerWatsonTest operation performs the nonparametric Wheeler-Watson test for two or more samples. Output is to the W_WheelerWatson wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed.

Displays results in a table. *k* specifies the table behavior when it is closed.

k=0:	Normal with dialog (default).
k=1:	Kills with no dialog.
k=2:	Disables killing.

The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.

/WSTR=*waveListString*	Specifies a string containing a semicolon-separated list of waves that contain sample data. Use *waveListString* instead of listing each wave after the flags.
/Z	Ignores errors.

Details

The StatsWatsonWilliamsTest must have at least two input waves, which contain angles in radians (mod 2π), can be single or double precision, and can be of any dimensionality; the waves must not contain any NaNs or INFs.

The Wheeler-Watson H_0 postulates that the samples came from the same population. The extension of the test to more than two samples is due to Mardia. The Wheeler-Watson test is not valid for data with ties, in which case you should use Watson's U^2 test.

V_flag will be set to -1 for any error and to zero otherwise.

References

Mardia, K.V., *Statistics of Directional Data*, Academic Press, New York, New York, 1972.

See, in particular, Chapter 27 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsWatsonUSquaredTest** and **StatsWheelerWatsonTest**.

StatsWilcoxonRankTest

StatsWilcoxonRankTest [*flags*] *waveA, waveB*

The StatsWilcoxonRankTest operation performs the nonparametric Wilcoxon-Mann-Whitney two-sample rank test or the Wilcoxon Signed Rank test (for paired data) on *waveA* and *waveB*. Output is to the W_WilcoxonTest wave in the current data folder or optionally to a table.

waveA and *waveB* must not contain NaNs or INFs.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/APRX=*m*	Sets the approximation method. It computes an exact critical value by default. *m*=1: Standard normal approximation with ties (Zar P. 151). *m*=2: Improved normal approximation (Zar P. 152). Approximations may be appropriate for large sample sizes when computation may take a long time.
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.
/TAIL=*tail*	*tail* is a bitwise parameter that specifies the tails tested. Bit 0: Lower tail. Bit 1: Upper tail (default). Bit 2: Two tail. See **Setting Bit Parameters** on page IV-12 for details about bit settings. You can perform any combination of tests by adding their corresponding tail values (/TAIL=7 tests all tail possiblities). Note that H0 changes according to the selected tail.

/WSRT	Performs the Wilcoxon Signed Rank Test for paired data. The testh computes statistics Tp and Tm, lower-tail, upper-tail, and two-tail P-values. If the number of samples is less than 200 it computes exact P-values, otherwise they are computed using the normal approximation. Do not use /ALPH, /APRX, and /TAIL with this flag.
/Z	Ignores errors.

Details

The Wilcoxon-Mann-Whitney test combines the two samples and ranks them to compute the statistic U. If *waveA* has *m* points and *waveB* has *n* points, then *U* is given by

$$U = mn + \frac{m(m+1)}{2} - R_1,$$

with the corresponding statistic *U*' given by

$$U' = nm + \frac{n(n+1)}{2} - R2.$$

where R_i is the ranks of data in the *i*th wave (ranked in ascending order).

The distribution of *U* is difficult to compute, requiring the number of possible permutations of *m* elements of *waveA* and *n* elements of *waveB* that give rise to *U* values that do not exceed the one computed. The distribution is computed according to the algorithm developed by Klotz. With increasing sample size one can avoid the time consuming distribution computation and use a normal approximation instead. Klotz recommends this approximation for *N*=*m*+*n*~100.

Use /APRX=2 for the best approximation. The two approximations are discussed by Zar.

The Wilcoxon Signed Rank Test, or Wilcoxon Paired-Sample Test, ranks the difference between pairs of values and computes the sums of the positive ranks (Tp) and the negative ranks (Tm). It calculates Tp and Tm and P-values for all tail combinations. The P-values are:

P_lower_tail	P(Wp<=Tp)
P_upper_tail	P(Wp>=Tp)
P_two_tail	2*Min(P_lower_tail,P_upper_tail)

Wp is the generic symbol for the sum of positive ranks for the given number of pairs.

V_flag will be set to -1 for any error and to zero otherwise.

In both Wilcoxon-Mann-Whitney two-sample rank test and the Wilcoxon Signed Rank test H0 is that the data in the two input waves are statistically the same.

References

Cheung, Y.K., and J.H. Klotz, The Mann Whitney Wilcoxon distribution using linked lists, *Statistica Sinica*, *7*, 805-813, 1997.

See in particular Chapter 15 of:

Klotz, J.H., *Computational Approach to Statistics*, <`http://www.stat.wisc.edu/~klotz/Book.pdf`>.

Streitberg, B., and J. Rohmel, Exact distributions for permutations and rank tests: An introduction to some recently published algorithms, *Statistical Software Newsletter*, *12*, 10-17, 1986.

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

StatsAngularDistanceTest, **StatsKWTest**, **StatsWilcoxonRankTest**

StatsWRCorrelationTest

`StatsWRCorrelationTest [`*`flags`*`]  `*`waveA, waveB`*

The StatsWRCorrelationTest operation performs a Weighted Rank Correlation test on *waveA* and *waveB*, which contain the ranks of sequential factors. The waves are 1-based, integer ranks of factors in the range 1-2^31.

StatsWRCorrelationTest computes a top-down correlation coefficient using Savage sums as well as the critical and P-values. Output is to the W_StatsWRCorrelationTest wave in the current data folder or optionally to a table.

Flags

/ALPH = *val*	Sets the significance level (default *val*=0.05).
/Q	No results printed in the history area.
/T=*k*	Displays results in a table. *k* specifies the table behavior when it is closed. *k*=0: Normal with dialog (default). *k*=1: Kills with no dialog. *k*=2: Disables killing. The table is associated with the test, not the data. If you repeat the test, it will update any existing table with the new results.
/Z	Ignores errors.

Details

The StatsWRCorrelationTest input waves must be one-dimensional and have the same length. The waves are 1-based, integer ranks of factors corresponding to the point number. Ranks may have ties in which case you should repeat the rank value. For example, if the second and third entries have the same rank you should enter {1,2,2,4}. H_0 stipulates that the same factors are most important in both groups represented by *waveA* and *waveB*.

The top-down correlation is the sum of the product of Savage sums for each row:

$$r_{TD} = \frac{\sum_{i=1}^{n} S_{iA} S_{iB} - n}{n - S_1},$$

where n is the number of rows and the Savage sum S_i is

$$S_i = \sum_{j=i}^{n} \frac{1}{j},$$

and S_{iA} corresponds to the S_i value of the rank of the data in row (i-1) of *waveA*.

References

Iman, R.L., and W.J. Conover, A measure of top-down correlation, *Technometrics*, *29*, 351-357, 1987.

See, in particular, Chapter 19 of:

Zar, J.H., *Biostatistical Analysis*, 4th ed., 929 pp., Prentice Hall, Englewood Cliffs, New Jersey, 1999.

See Also

Chapter III-12, **Statistics** for a function and operation overview; **StatsLinearCorrelationTest**, **StatsRankCorrelationTest**, **StatsTopDownCDF**, and **StatsInvTopDownCDF**.

StopMSTimer

`StopMSTimer(`*`timerRefNum`*`)`

The StopMSTimer function frees up the timer associated with the *timerRefNum* and returns the number of elapsed microseconds since StartMSTimer was called for this timer.

Parameters

timerRefNum is the value returned by StartMSTimer or the special values -1 or -2. If *timerRefNum* is not valid then StopMSTimer returns 0.

On Windows, passing -1 returns the clock frequency of the timer and. On Macintosh, it returns NaN.

Passing -2 returns the time in microseconds since the computer was started.

Details

If you want to make sure that all timers are free, call StopMSTimer ten times with *timerRefNum* equal to 0 through 9. It is OK to stop a timer that you never started.

Examples

How long does an empty loop take on your computer?

```
Function TestMSTimer()
   Variable timerRefNum
   Variable microSeconds
   Variable n

   timerRefNum = StartMSTimer
   if (timerRefNum == -1)
      Abort "All timers are in use"
   endif
   n=10000
   do
      n -= 1
   while (n > 0)
   microSeconds = StopMSTimer(timerRefNum)
   Print microSeconds/10000, "microseconds per iteration"
End
```

See Also

The **StartMSTimer** and **ticks** functions.

str2num

`str2num(`*`str`*`)`

The str2num function returns a number represented by the string expression *str*.

Details

str2num returns NaN if *str* does not contain the text for a number.

str2num skips leading spaces and tabs and then reads up to the first non-numeric character.

See Also

The **char2num**, **num2char** and **num2str** functions.

The **sscanf** operation for more complex parsing jobs.

Strconstant

`Strconstant `*`ksName`*`="`*`literal string`*`"`

The Strconstant declaration defines the string *literal string* under the name *ksName* for use by other code, such as in a switch construct.

See Also

The **Constant** keyword for numeric types, **Constants** on page IV-47, and **Switch Statements** on page IV-41.

String

`String [/G] *strName* [=*strExpr*][, *strName* [=*strExpr*]…]`

The String operation creates string variables and gives them the specified names.

Flags

/G	Creates a global string. Overwrites any existing string with the same name.

Details

The string variable is initialized when it is created if you supply the =*strExpr* initializer. However, when String is used to declare a function parameter, it is an error to attempt to initialize it.

You can create more than one string variable at a time by separating the names and optional initializers with commas.

If used in a procedure, the new string is local to that procedure unless the /G (global) flag is used. If used on the command line, String is equivalent to String/G.

strName can optionally include a data folder path.

See Also

String Variables on page II-97, **Working With Strings** on page IV-12

StringByKey

`StringByKey(*keyStr*, *kwListStr* [, *keySepStr* [, *listSepStr* [, *matchCase*]]])`

The StringByKey function returns a substring extracted from *kwListStr* based on the specified key contained in *keyStr*. *kwListStr* should contain keyword-value pairs such as `"KEY=value1,KEY2=value2"` or `"Key:value1;KEY2:value2"`, depending on the values for *keySepStr* and *listSepStr*.

Use StringByKey to extract a string value from a string containing a `"key1:value1;key2: value2;"` style list such as those returned by functions like **AxisInfo** or **TraceInfo**.

If the key is not found or if any of the arguments is `""` then a zero-length string is returned.

keySepStr, *listSepStr*, and *matchCase* are optional; their defaults are ":", ";", and 0 respectively.

Details

keyStr is limited to 255 bytes.

kwListStr is searched for an instance of the key string bound by *listSepStr* on the left and a *keySepStr* on the right. The text up to the next *listSepStr* is returned.

kwListStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *keySepStr* and *listSepStr* are always case-sensitive. Searches for *keyStr* in *kwListStr* are usually case-insensitive. Setting the optional *matchCase* parameter to 1 makes the comparisons case sensitive.

In Igor6, only the first byte of *keySepStr* and *listSepStr* was used. In Igor7 and later, all bytes are used.

If *listSepStr* is specified, then *keySepStr* must also be specified. If *matchCase* is specified, *keySepStr* and *listSepStr* must be specified.

Examples

```
Print StringByKey("BKEY", "AKEY:hello;BKEY:nok-nok")  // prints "nok-nok"
Print StringByKey("KY", "KX=1;ky=hello", "=")         // prints "hello"
Print StringByKey("KY", "KX:1,KY:joey,", ":", ",")    // prints "joey"
Print StringByKey("kz", "KZ:1st,kz:2nd,", ":", ",")   // prints "1st"
Print StringByKey("kz", "KZ:1st,kz:2nd,", ":", ",", 1)// prints "2nd"
```

See Also

The **NumberByKey**, **RemoveByKey**, **ReplaceNumberByKey**, **ReplaceStringByKey**, **ItemsInList**, **AxisInfo**, **IgorInfo**, **SetWindow**, and **TraceInfo** functions.

StringCRC

`StringCRC(*inCRC*,*str*)`

The StringCRC function returns a 32-bit cyclic redundancy check value of bytes in *str* starting with *inCRC*.

Pass 0 for *inCRC* the first time you call StringCRC for a particular stream of bytes as represented by the string data.

Pass the last-returned value from StringCRC for *inCRC* if you are creating a CRC value for a given stream of bytes through multiple calls to StringCRC.

Details

Polynomial used is:

x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x+1

See crc32.c in the public domain source code for zlib for more information.

See Also

The **WaveCRC** function.

StringFromList

StringFromList(*index, listStr* [, *listSepStr*] [, *offset*])

The StringFromList function returns the *index*th substring extracted from *listStr* starting *offset* bytes into *listStr*. *listStr* should contain items separated by *listSepStr*, such as "abc;def;".

Use StringFromList to extract an item from a string such as those returned by functions like **TraceNameList** and **AnnotationList**.

Parameters

index is the zero-based index of the list item that you want to get. If *index* < 0, or *index* ≥ the number of items in list, or if *listStr* or *listSepStr* is `""`, then a zero-length string is returned.

listStr contains a series of text items separated by *listSepStr*. The trailing separator is optional though recommended. For example, these are both valid lists:

```
"First;Second;"
"First;Second"
```

listSepStr is optional. If omitted it defaults to ";". Prior to Igor Pro 7, only the first byte of *listSepStr* was used. Now all bytes are used.

offset is optional and requires Igor Pro 7 or later. If omitted it defaults to 0. The search begins *offset* bytes into *listStr*. When iterating through lists containing large numbers of items, using the *offset* parameter provides dramatically faster execution.

Details

For optimal performance, especially with lists larger than 100 items, provide the *separatorStr* and *offset* parameters as shown in the DemoStringFromList example below. When using this technique, the *index* parameter must be 0 and the *offset* parameter controls which list item is returned.

Examples

```
Print StringFromList(0, "wave0;wave1;")              // Prints "wave0"
Print StringFromList(2, "wave0;wave1;")              // Prints ""
Print StringFromList(1, "wave0;;wave2")              // Prints ""

// Iterate quickly over a list using the offset parameter
Function DemoQuickStringFromList(list)
    String list      // A semicolon-separated string list

    String separator = ";"
    Variable separatorLen = strlen(separator)
    Variable numItems = ItemsInList(list)

    Variable offset = 0
    Variable i
    for(i=0; i<numItems; i+=1)
        // When using offset, the index parameter is always 0
        String item = StringFromList(0, list, separator, offset)
        // Do something with item
        offset += strlen(item) + separatorLen
    endfor
End
```

See Also

The **AddListItem**, **ItemsInList**, **FindListItem**, **RemoveListItem**, **RemoveFromList**, **WaveList**, **WhichListItem**, **StringByKey**, **ListMatch**, **ControlNameList**, **TraceNameList**, **StringList**, **VariableList**, and **FunctionList** functions.

StringList

`StringList(`*`matchStr`*`, `*`separatorStr`*`)`

The StringList function returns a string containing a list of global string variables selected based on the *matchStr* parameter. The string variables listed are all in the current data folder.

Details

For a string variable name to appear in the output string, it must match *matchStr*. The first character of *separatorStr* is appended to each string variable name as the output string is generated.

The name of each string variable is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all string variable names
"xyz"	Matches name xyz only
"*xyz"	Matches names which end with xyz
"xyz*"	Matches names which begin with xyz
"*xyz*"	Matches names which contain xyz
"abc*xyz"	Matches names which begin with abc and end with xyz

The list contains names only, without data folder paths. Thus, they are not suitable for accessing string variables outside the current data folder.

matchStr may begin with the ! character to return windows that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches variable names which *do not* end with xyz

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

Examples

`StringList("*",";")`	Returns a list of all string variables in the current data folder.
`StringList("S_*", ";")`	Returns a list of all string variables in the current data folder whose names begin with “S_”.

See Also

See the **VariableList** and **WaveList** functions.

StringMatch

`StringMatch(`*`string`*`, `*`matchStr`*`)`

The StringMatch function tests *string* for a match to *matchStr*. You may include asterisks in *matchStr* as a wildcard character.

StringMatch returns 1 to indicate a match, 0 for no match or NaN if it ran out of memory.

Details

matchStr is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches any string.
"xyz"	Matches the string “xyz” only.

"*xyz"	Matches strings ending with "xyz", for instance "abcxyz".
"xyz*"	Matches strings beginning with xyz, for instance "xyzpqr".
"*xyz*"	Matches strings containing xyz, for instance "abcxyzpqr".
"abc*xyz"	Matches strings beginning with abc and ending with xyz, for instance "abcpqrxyz".

If *matchStr* begins with the ! character, a match is indicated if *string* does *not* match *matchStr*. For example:

"!*xyz"	Matches strings which *do not* end with xyz.

The ! character is considered to be a normal character if it appears anywhere else.

Note that matching is case-insensitive, so "xyz" also matches "XYZ" or "Xyz".

Also note that it is impossible to match an asterisk in *string*: use **GrepString** instead.

Among other uses, the StringMatch function can be used to build your own versions of the **WaveList** function, using **NameOfWave** and stringmatch to qualify names of waves found by **WaveRefIndexedDFR**.

See Also

The **GrepString**, **cmpstr**, **strsearch**, **Demo**, **ListMatch**, and **ReplaceString** functions and the **sscanf** operation.

strlen

strlen(*str*)

The strlen function returns the number of bytes in the string expression *str*.

strlen returns NaN if the *str* is NULL. A local string variable or a string field in a structure that has never been set is NULL. NULL is not the same as zero length. Use **numtype** to test if the result from strlen is NaN.

Examples

```
String zeroLength = ""
String neverSet
Print strlen(zeroLength), strlen(neverSet)

// Test if a string is null
Variable len = strlen(neverSet)   // NaN if neverSet is null
if (numtype(len) == 2)            // strlen returned NaN?
   Print "neverSet is null"
endif
```

See Also

Characters Versus Bytes on page III-433, **Character-by-Character Operations** on page IV-162

strsearch

strsearch(*str*, *findThisStr*, *start* [, *options*])

The strsearch function returns the numeric position of the string expression *findThisStr* in the string expression *str*.

Details

strsearch performs a case-sensitive search.

strsearch returns -1 if *findThisStr* does not occur in *str*.

The search starts from the character position in *str* specified by *start*; 0 is the first character in *str*.

strsearch limits *start* to one less than the length of *str*, so it is useful to use `Inf` for *start* when searching backwards to ensure that the search is from the end of *str*.

The optional *options* parameter is a bitmask specifying the search options:

1:	Search backwards from *start*.
2:	Ignore case.

3: Search backwards and ignore case.

Examples

```
String str="This is a test isn't it?"
Print strsearch(str,"test",0)                  // prints 10
Print strsearch(str,"TEST",0)                  // prints -1
Print strsearch(UpperStr(str),"TEST",0)        // prints 10
Print strsearch(str,"TEST",0,2)                // prints 10
Print strsearch(str,"is",0)                    // prints 2
Print strsearch(str,"is",3)                    // prints 5
Print strsearch(str,"is",Inf,1)                // prints 15
```

See Also

sscanf, **FindListItem**, **ReplaceString**, **Character-by-Character Operations**

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

strswitch-case-endswitch

```
strswitch(<string expression>)
   case <literal><constant>:
      <code>
      [break]
   [default:
      <code>]
endswitch
```

A strswitch-case-endswitch statement evaluates a string expression and compares the result to the case labels using a case-insensitive comparison. If a case label matches *string expression*, then execution proceeds with *code* following the matching case label. When none of the cases match, execution will continue at the default label, if it is present, or otherwise the strswitch will be exited with no action taken. Note that although the break statement is optional, in almost all case statements it will be required for the strswitch to work correctly.

See Also

Switch Statements on page IV-41, **default** and **break** for more usage details.

STRUCT

STRUCT *structureName localName*

STRUCT is a reference that creates a local reference to a Structure accessed in a user-defined function. When a Structure is passed to a user function, it can only be passed by reference, so in the declaration within the function you must use `&localStructName` to define the function input parameter.

See Also

Structures in Functions on page IV-91 for further information.

See the **Structure** keyword for creating a Structure definition.

StructGet

StructGet [/B=*b*] *structVar*, *waveStruct*[[*colNum*]]

StructGet /S [/B=*b*] *structVar*, *strStruct*

The StructGet operation reads binary numeric data from a specified column of a wave or from a string variable and copies the data into the designated structure variable. The source wave or string will have been filled beforehand by **StructPut**.

Parameters

structVar is the name of an existing structure that is to be filled with new data values.

waveStruct is the name of a wave containing binary numeric data that will be used to fill *structVar*. Use the optional *colNum* parameter to specify a column from the structure wave. The contents of *waveStruct* are created beforehand using StructPut.

strStruct is the name of a string variable containing binary numeric data. The contents of *strStruct* are created beforehand using StructPut.

Flags

/B=*b*	Sets the byte ordering for reading of structure data.
	b=0: Reads in native byte order.
	b=1: Reads bytes in reversed order.
	b=2: Default; reads data in big-endian, high-byte-first order (Motorola).
	b=3: Reads data in little-endian, low-byte-first order (Intel).
/S	Reads binary data from a string variable, which was set previously with StructPut.

Details

The data that are stored in *waveStruct* and *strStruct* are in binary format so you can not directly view a meaningful representation of their contents by printing them or viewing the wave in a table. To view the contents of *waveStruct* or *strStruct* you must use StructGet to export them back into a structure and then retrieve the members.

If *colNum* is out of bounds it will be clipped to valid values and an error reported. If the row dimension does not match the structure size, as much data as possible will be copied to the structure.

By default, data are read in big-endian, high-byte order (Motorola). This allows data written on one platform to be read on the other.

See Also

The **StructPut** operation for writing structure data to waves or strings.

StructPut

`StructPut [/B=b] structVar, waveStruct[[colNum]]`

`StructPut /S [/B=b] structVar, strStruct`

The StructPut operation copies the binary numeric data in a structure variable to a specified column in a wave or to a string variable. The data in the wave or string can be read out into another structure using **StructGet**.

Parameters

structVar is the name of a structure from which data will be exported.

waveStruct is the name of an existing wave to which data will be exported. Use the optional *colNum* parameter to specify a column in *waveStruct* to contain the data. The first column of *waveStruct* will be filled if *colNum* is omitted.

strStruct is the name of an existing string variable to which data will be exported.

Flags

/B=*b*	Sets the byte ordering for writing of structure data.
	b=0: Writes in native byte order.
	b=1: Writes bytes in reversed order.
	b=2: Default; writes data in big-endian, high-byte-first order (Motorola).
	b=3: Writes data in little-endian, low-byte-first order (Intel).
/S	Writes binary data to a string variable.

Details

The structure to be exported must contain only numeric data in either integer, floating point, or double precision format. If the structure contains any objects such as String, NVAR, WAVE, etc., then an error will result at compile time.

If needed, StructPut will redimension *waveStruct* to unsigned byte format, will set the number of rows to equal the size of the structure, and set the column dimension large enough to accommodate the size specified by *colNum*. You can think of *waveStruct* as a one-dimensional array of structure contents indexed by *colNum* although the wave is actually two-dimensional with each column containing a copy of a separate structure.

By default, data are written in big-endian, high-byte order (Motorola). This allows data written on one platform to be read on the other.

After you have exported the structure data to *waveStruct* or *strStruct* they will contain binary data that you cannot inspect directly. To view the contents of *waveStruct* or *strStruct*, you must use the original structure or use StructGet to export them into another structure.

See Also

The **StructGet** operation for reading structure data from waves or strings.

Structure

```
Structure structureName
    memType memName [arraySize] [, memName [arraySize]]
    …
EndStructure
```

The Structure keyword introduces a structure definition in a user function. Within the body of the structure you declare the member type (*memType*) and the corresponding member name(s) (*memName*). Each *memName* may be declared with an optional array size.

Details

Structure member types (*memType*) can be any of the following Igor objects: Variable, String, WAVE, NVAR, SVAR, DFREF, FUNCREF, or STRUCT.

Igor structures also support additional member types, as given in the next table, for compatibility with C programming structures and disk files.

Igor Member Type	C Equivalent	Size	Note
char	signed 8-bit int	1 byte	
uchar	unsigned 8-bit int	1 byte	
int16	signed 16-bit int	2 bytes	
uint16	unsigned 16-bit int	2 bytes	
int32	signed 32-bit int	4 bytes	
uint32	unsigned 32-bit int	4 bytes	
int64	signed 64-bit int	8 bytes	Requires Igor Pro 7.00 or later
uint64	unsigned 64-bit int	8 bytes	Requires Igor Pro 7.00 or later
float	float	4 bytes	
double	double	8 bytes	

The Variable and double types are identical although Variable can be also specified as complex (using the /C flag).

Each structure member may have an optional *arraySize* specification, which gives the number of elements contained by the structure member. The array size is an integer number from 1 to 400 except for members of type STRUCT for which the upper limit is 100.

See Also

Structures in Functions on page IV-91 for further information.

See the **STRUCT** declaration for creating a local reference to a Structure.

StrVarOrDefault

StrVarOrDefault(pathStr, defStrVal)

The StrVarOrDefault function checks to see if *pathStr* points to a string variable and if so, it returns its value. If the string variable does not exist, returns *defStrVal* instead.

Details

StrVarOrDefault initializes input values of macros so they can remember their state without needing global variables to be defined first. Numeric variables use the corresponding numeric function, **NumVarOrDefault**.

Examples

```
Macro foo(nval,sval)
   Variable nval=NumVarOrDefault("root:Packages:mypack:nvalSav",2)
   String sval=StrVarOrDefault("root:Packages:mypack:svalSav","Hi")

   DFREF dfSav= GetDataFolderDFR()
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S mypack
   Variable/G nvalSav= nval
   String/G svalSav= sval
   SetDataFolder dfSav
End
```

StudentA

StudentA(t, DegFree)

Note: This function is deprecated. New code should use the more accurate **StatsStudentCDF**.

The StudentA function returns the area from -*t* to *t* under the Student's T distribution having *DegFree* degrees of freedom. That is, it returns the probability that a random sample from Student's T is between -*t* and *t*.

Note that this is the *bi-tail* result. That is, it gives the area from -*t* to *t*, rather than the cumulative area from -∞ to *t*. It is this latter number that is commonly tabulated- StudentA returns the probability 1-α where the area from -∞ to t is the probability 1-α/2.

StudentA tests whether a normally-distributed statistic is significantly different from a certain value. You could use it to test whether an intercept from a line fit is significantly different from zero:

```
Make/O/N=20 Data=0.5*x+2+gnoise(1)      // line with Gaussian noise
Display Data
CurveFit line Data /D
Print "Prob = ", StudentA(W_coef[0]/W_sigma[0], V_npnts-2)
```

Because the noise is random, the results will differ slightly each time this is tried. When we did it, the result was:

```
Prob = 0.999898
```

which indicates that the intercept of the line fit was different from zero with 99.99 per cent probability.

See Also

StatsStudentCDF, **StatsStudentPDF**, **StatsInvStudentCDF**

StudentT

StudentT(Prob, DegFree)

Note: This function is deprecated. New code should use the more accurate **StatsInvStudentCDF**.

The StudentT function returns the t value corresponding to an area *Prob* under the Student's T distribution from -*t* to *t* for *DegFree* degrees of freedom.

Note that this is a *bi-tail* result, which is what is usually desired. Tabulated values of the Student's T distribution are commonly the one-sided result.

StudentT calculates confidence intervals from standard deviations for normally-distributed statistics. For instance, you can use it to calculate a confidence interval for the coefficients from a curve fit:

```
Make/O/N=20 Data=0.5*x+2+gnoise(1)      // line with Gaussian noise
Display Data
```

```
CurveFit line Data /D
print "intercept = ", W_coef[0], "±", W_sigma[0]*StudentT(0.95, V_npnts-2)
print "slope = ", W_coef[1], "±", W_sigma[1]*StudentT(0.95, V_npnts-2)
```

See Also

StatsStudentCDF, **StatsStudentPDF**, **StatsInvStudentCDF**

Submenu

`Submenu` ***`menuNameStr`***

The Submenu keyword introduces a submenu definition. It is used inside a Menu definition. See Chapter IV-5, **User-Defined Menus** for further information.

sum

`sum(`*`waveName`*`[,`*`x1, x2`*`])`

The sum function returns the sum of the wave elements for points from x=*x1* to x=*x2*.

Details

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with `pnt2x(`*`waveName`*`,`*`pointNumber1`*`)`, and a similar expression for *x2*.

If *x1* and *x2* are not specified, they default to -∞ and +∞, respectively.

If the points nearest to *x1* or *x2* are not within the point range of 0 to numpnts(*waveName*)-1, sum limits them to the nearest of point 0 or point numpnts(*waveName*)-1.

If any values in the point range are NaN, sum returns NaN.

Examples

```
Make/O/N=100 data; SetScale/I x 0,Pi,data
data=sin(x)
Print sum(data,0,Pi)          // the entire point range, and no more
Print sum(data)               // same as -infinity to +infinity
Print sum(data,Inf,-Inf)      // +infinity to -infinity
```

The following is printed to the history area:

```
Print sum(data,0,Pi)          // the entire point range, and no more
  63.0201
Print sum(data)               // same as -infinity to +infinity
  63.0201
Print sum(data,Inf,-Inf)      // +infinity to -infinity
  63.0201
```

See Also

mean, **area**, **SumSeries**, **SumDimension**

SumDimension

`SumDimension [`*`flags`*`]` ***`srcWave`***

The SumDimension operation sums values in *srcWave* along the specified dimension.

The SumDimension operation was added in Igor Pro 7.00.

Flags

/D=*dimension* Specifies a zero-based dimension number.

dimension=0:	Rows
dimension=1:	Columns
dimension=2:	Layers
dimension=3:	Chunks

If you omit /D the operation sums the highest dimension in the wave.

/DEST=*destWave*	Specifies the output wave created by the operation. If *destWave* already exists it is overwritten by the new results. If you omit /DEST the operation saves the data in W_SumDimension if the output wave is 1D or M_SumDimension otherwise.
/Y=type	Specifies the data type of the output wave. See **WaveType** for the supported values of type. If you omit /Y, the output wave is double precision. Pass -1 for type to force the output wave to have the same data type as *srcWave*.

Details

The operation sums one dimension of an N dimensional wave producing an output wave with N-1 dimensions except if *srcWave* is 1D wave in which case SumDimension produces a single point 1D output wave. For example, given a 4D wave of dimensions dim0 x dim1 x dim2 x dim3 and the command:

```
SumDimension/D=1/DEST=wout wave4d
```

creates a wave wout that satisfies

$$wout[i][k][l] = \sum_{j=0}^{\dim 1-1} wave4d[i][j][k][l],$$

and wout has dimensions dim0 x dim2 x dim3.

If any values in *srcWave* are NaN, the corresponding sum element will be NaN.

See Also

sum

MatrixOp keywords sumRows, sumCols, sumBeams

ImageTransform keywords sumAllCols, sumAllRows, sumPlane, sumPlanes

SumSeries

`SumSeries [`*`flags`*`] `*`keyword=value`*

The SumSeries operation computes the sum of the results returned from a user-defined function for input values between two specified index limits.

SumSeries was added in Igor Pro 7.00.

Flags

/CCNT=*nc*	When summing with one or two infinite limits you can use this flag to specify the minimum number of calls to the summand function which, when added to the sum, produce a change that is less than the tolerance. By default *nc*=10. If you are summing a well-behaved monotonic series it is sufficient to set *nc*=1. In some pathological cases it is useful to check that the sum remains effectively unchanged even after many terms are added to the series.
/INAN	Ignore NaNs returned from the user function. In the case of a complex valued summand, a NaN in either the real or imaginary components excludes the contribution of the term to the sum.
/Q	Quiet mode; do not print in the history.
/Z[=z]	/Z or /Z=1 prevents reporting any errors. If the operation encounters an error it sets V_Flag to the error code.

Keywords

lowerLimit=*n1*	Specifies the starting index at which the summand is evaluated. *n1* must be either an integer -INF.

series=*userFunc*	Specifies the name of the user function that returns the summand (i.e., a single term in the sum that corresponds to the input index). See **The SumSeries Summand Function** below for details.
upperLimit=*n2*	Specifies the last value at which the summand is evaluated. *n2* must be either an integer INF.
tolerance=*tol*	Specifies a tolerance value used when one or both of the limits are infinite. By default, the tolerance value is 1e-10. *tol* must be finite. If both limits are finite this keyword is ignored.
paramWave=*pw*	*pw* is a single-precision or double-precision wave that is passed to the summand function. This is useful if you need to provide the summand function external/global data. If you omit the paramWave keyword then the summand function receives a null wave as the parameter wave.

The SumSeries Summand Function

You specify the summand function using the series keyword. The form of the user-defined summand function is:

```
Function summandReal(inW,index)
    Wave inW
    Variable index
    ... compute something
    return result
End
```

The index changes by 1 for each successive call to the summand.

You can also define a complex summand function:

```
Function/C summandComplex(inW,index)
    Wave inW
    Variable index
    ... compute something
    Variable/C result
    return result
End
```

Details

The SumSeries operation is primarily intended for use with one or two infinite limits. If both limits are finite the operation performs the straightforward sum by calling the summand function once for every index from lowerLimit to upperLimit, inclusive.

If one limit is infinite the sum is evaluated by starting from the finite limit and proceeding in the direction of the infinite limit index until convergence is reached. Convergence in this context is defined as multiple (*nc*) consecutive calls to the summand which do not change the value of the sum by more than the tolerance value. By default *nc*=10 but you can change it using the /CCNT flag.

When both limits are infinite the operation first computes the sum for indices 0 to INF and then the sum from -1 to -INF. The two calculations are independent and require that the same convergence condition is met independently in each case. When the summand function is complex the convergence condition must hold for the real and imaginary components independently.

The operation does not perform any test on the summand function to estimate its rate of convergence. If you provide a non-converging summand function the operation can run indefinitely. You can abort it by pressing the **User Abort Key Combinations** or by clicking the Abort button.

The result of the sum is stored in V_resultR and, if the summand function returns a complex result, V_resultI.

If the calculation completes without error V_Flag is set to 0. Otherwise it contains an error code.

Examples

A simple test case is the geometric series for powers of 1/2. The sum of x^i for i=0 to i=INF where 0<x<1 is given by 1/(1-x). For x=1/2, this sum is 2.

```
Function s1(inW,index)
   Wave/z inW
   Variable index

   return 0.5^index
End
```

```
// Execute:
SumSeries series=s1,lowerLimit=0,upperLimit=INF
Print V_resultR
```

In the following example we use the series expansion of cosine and sine to evaluate exp(i*pi).

```
Function/C s2(inW,index)
   Wave/z inW
   Variable index

   Variable n2=2*index
   Variable xx=pi^n2
   Variable sn=(-1)^index
   Variable fn=Factorial(n2)
   return cmplx(sn*xx/fn,sn*xx*pi/(fn*(n2+1)))
End
```

```
// Execute:
SumSeries series=s2,lowerLimit=0,upperLimit=INF
Print V_resultR,V_resultI
```

See Also

Integrate1D, **sum**

SVAR

SVAR [**/Z**] ***localName*** [**=** ***pathToStr***][**,** ***localName1*** [**=** ***pathToStr1***]]…

SVAR is a declaration that creates a local reference to a global string variable accessed in a user-defined function.

The SVAR reference is required when you access a global string variable in a function. At compile time, the SVAR statement specifies a local name referencing a global string variable. At runtime, it makes the connection between the local name and the actual global variable. For this connection to be made, the global string variable must exist when the SVAR statement is executed.

When *localName* is the same as the global string variable name and you want to reference a global variable in the current data folder, you can omit *pathToStr*.

pathToStr can be a full literal path (e.g., root:FolderA:var0), a partial literal path (e.g., :FolderA:var0) or $ followed by string variable containing a computed path (see **Converting a String into a Reference Using $** on page IV-57).

You can also use a data folder reference or the /SDFR flag to specify the location of the string variable if it is not in the current data folder. See **Data Folder References** on page IV-72 and **The /SDFR Flag** on page IV-74 for details.

If the global variable may not exist at runtime, use the /Z flag and call **SVAR_Exists** before accessing the variable. The /Z flag prevents Igor from flagging a missing global variable as an error and dropping into the Igor debugger. For example:

```
SVAR/Z nv=<pathToPossiblyMissingStringVariable>
if( SVAR_Exists(sv) )
   <do something with sv>
endif
```

Note that to create a global string variable, you use the **String**/G operation.

Flags

/Z	An SVAR reference to a null string variable does not cause an error or a debugger break.

See Also

SVAR_Exists function.

Accessing Global Variables and Waves on page IV-59.

Converting a String into a Reference Using $ on page IV-57.

SVAR_Exists

`SVAR_Exists(`*`name`*`)`

The SVAR_Exists function returns 1 if the specified SVAR reference is valid or 0 if not. It can be used only in user-defined functions.

For example, in a user function you can test if a global string variable exists like this:

```
SVAR /Z str1 = gStr1        // /Z prevents debugger from flagging bad SVAR
if (!SVAR_Exists(str1))     // No such global string variable?
   String/G gStr1 = ""      // Create and initialize it
endif
```

See Also

WaveExists, **NVAR_Exists**, and **Accessing Global Variables and Waves** on page IV-59.

switch-case-endswitch

```
switch(<numeric expression>)
   case <literal><constant>:
      <code>
      [break]
   [default:
      <code>]
endswitch
```

A switch-case-endswitch statement evaluates a numerical expression. If a case label matches *numerical expression*, then execution proceeds with *code* following the matching case label. When no cases match, execution continues at the default label, if present, or otherwise the switch exits with no action taken. Note that although the break statement is optional, in almost all case statements it is required for the switch to work correctly.

See Also

Switch Statements on page IV-41, **default** and **break** for more usage details.

t

`t`

The t function returns the T value for the current chunk of the destination wave when used in a multidimensional wave assignment statement. T is the scaled chunk index while **s** is the chunk index itself.

Details

Unlike **x**, outside of a wave assignment statement, t does not act like a normal variable.

See Also

Waveform Arithmetic and Assignments on page II-69.

For other dimensions, the **p**, **q**, **r**, and **s** functions.

For scaled dimension indices, the **x**, **y**, and **z** functions.

TabControl

`TabControl` `[/Z]` ***`ctrlName`*** `[`***`keyword = value`*** `[,` ***`keyword = value`*** `…]]`

The TabControl operation creates tab panels for controls.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the TabControl to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}

	Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings. *kind* can be one of `default`, `native`, or `os9`. *platform* can be one of `Mac`, `Win`, or `All`. See **DefaultGUIControls Default Fonts and Sizes** for how enclosed controls are affected by native TabControl appearance. See **Button** for more appearance details.
disable=*d*	Sets user editability of the control. *d*=0: Normal. *d*=1: Hide. *d*=2: Draw in gray state; disable control action.
fColor=(*r*,*g*,*b*)	Sets the initial color of the tab labels. *r*, *g*, and *b* can range from 0 to 65535. To further change the color of the tab labels text, use escape sequences in the text specified by the tabLabel keyword. fColor defaults to black (0,0,0).
focusRing=*fr*	Enables or disables the drawing of a rectangle indicating keyboard focus: *fr*=0: Focus rectangle will not be drawn. *fr*=1: Focus rectangle will be drawn (default). On Macintosh, regardless of this setting, the focus ring appears if you have enabled full keyboard access via the Shortcuts tab of the Keyboard system preferences.
font= "*fontName*"	Sets the font used for tabs, e.g., `font="Helvetica"`.
fsize= *s*	Sets the font size for tabs.
fstyle=*fs*	*fs* is a bitwise parameter with each bit controlling one aspect of the font style as follows: Bit 0: Bold Bit 1: Italic Bit 2: Underline Bit 4: Strikethrough See **Setting Bit Parameters** on page IV-12 for details about bit settings.
labelBack=(*r*,*g*,*b*) or 0	Sets fill color for current tab and the interior. *r*, *g*, and *b* are integers from 0 to 65535. If not set, then interior is transparent and the current tab is filled with the window background. Note that if you use a fill color, draw objects can not be used because they will be covered up.
noproc	Specifies that no function is to run when clicking a tab.
pos={*left*,*top*}	Sets the position of the control in pixels.
pos+={*dx*,*dy*}	Offsets the position of the control in pixels.
proc=*procName*	Specifies the function to run when the tab is pressed. Your function must hide and show other controls as desired. The TabControl does not do this automatically.
size={*width*,*height*}	Sets TabControl size in pixels.
tabLabel(*n*)=*lbl*	Sets *n*th tab label to *lbl*. Set the label of the last tab to `""` to reduce the number of tabs. Using escape codes you can change the font, size, style, and color of the label. See **Annotation Escape Codes** on page III-53 or details.

userdata(*UDName*)=*UDStr*

Sets the unnamed user data to *UDStr*. Use the optional (*UDName*) to specify a named user data to create.

userdata(*UDName*)+=*UDStr*

Appends *UDStr* to the current unnamed user data. Use the optional (*UDName*) to append to the named *UDStr*.

value=*v*	Sets current tab number. Tabs count from 0.
win=*winName*	Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.

When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Flags

/Z	No error reporting.

Tab Control Action Procedure

The action procedure for a TabControl takes a predefined `WMTabControlAction` structure as a parameter to the function:

```
Function ActionProcName(TC_Struct) : TabControl
   STRUCT WMTabControlAction &TC_Struct
   …
   return 0
End
```

The “`: TabControl`” designation tells Igor to include this procedure in the Procedure pop-up menu in the Tab Control dialog.

See **WMTabControlAction** for details on the WMTabControlAction structure.

Although the return value is not currently used, action procedures should always return zero.

When clicking a TabControl with the selector arrow, click in the title region. The control is not selected if you click in the body. This is to make it easier to select controls in the body rather than the TabControl itself.

Example

Designing a TabControl with all the accompanying interior controls can be somewhat difficult. Here is a suggested technique:

First, create and set the size and label for one tab. Then create the various controls for this first tab. Before starting on the second tab, create the TabControl’s procedure so that it can be used to hide the first set of controls. Then add the second tab, click it to run your procedure and start adding controls for this new tab. When done, update your procedure so the new controls are hidden when you start on the third tab.

Here is an example:

1. Create a panel and a TabControl:

```
NewPanel /W=(150,50,478,250)
ShowTools
TabControl MyTabControl,pos={29,38},size={241,142},tabLabel(0)="First Tab",value=0
```

2. Add a few controls to the interior of the TabControl:

```
Button button0,pos={52,72},size={80,20},title="First"
CheckBox check0,pos={52,105},size={102,15},title="Check first",value=0
```

3. Write an action procedure:

```
Function TabActionProc(tc) : TabControl
   STRUCT WMTabControlAction& tc

   switch(tc.eventCode)
      case 2:                // Mouse up
      Button button0, disable=(tc.tab!=0)
      CheckBox check0, disable=(tc.tab!=0)
      break
```

```
    endswitch
End
```

4. Set the action procedure and add a new tab:

```
TabControl MyTabControl,proc=TabActionProc,tabLabel(1)="Second Tab"
```

5. Click the second tab, which hides the first tab's controls, and then add new controls like this:

```
Button button1,pos={58,73},size={80,20},title="Second"
CheckBox check1,pos={60,105},size={114,15},title="Check second",value= 0
```

6. Finally, change the action procedure by adding these lines at the end:

```
Button button1,disable=(tc.tab!=1)
CheckBox check1,disable=(tc.tab!=1)
```

See Also

The **ControlInfo** operation for information about the control along with the **ModifyControl** and **ModifyControlList** operations. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

TabControl

`TabControl`

TabControl is a procedure subtype keyword that identifies a macro or function as being an action procedure for a user-defined tab control. See **Procedure Subtypes** on page IV-193 for details. See **TabControl** for details on creating a tab control.

Table

`Table`

Table is a procedure subtype keyword that identifies a macro as being a table recreation macro. It is automatically used when Igor creates a window recreation macro for a table. See **Procedure Subtypes** on page IV-193 and **Killing and Recreating a Table** on page II-176 for details.

TableStyle

`TableStyle`

TableStyle is a procedure subtype keyword that puts the name of the procedure in the Style pop-up menu of the New Table dialog and in the Table Macros menu. See **Table Style Macros** on page II-205 for details.

TableInfo

`TableInfo(`*`winNameStr, itemIndex`*`)`

The TableInfo function returns a string containing a semicolon-separated list of keywords and values that describe a column in a table or overall properties of the table. The main purpose of TableInfo is to allow an advanced Igor programmer to write a procedure which formats or arranges a table or which manipulates the table selection.

Parameters

winNameStr is the name of an existing table window or `""` to refer to the top table.

itemIndex is one of the following:

itemIndex **Value**	**Returns**
-2	Information about the table as a whole.
-1	Information about the Point column
≥0	Information about a column other than the Point column. 0 refers to the first column after the Point column, 1 refers to the second column after the Point column, and so on.

TableInfo returns `""` in the following situations:

- *winNameStr* is `""` and there are no table windows.
- *winNameStr* is a name but there are no table windows with that name.
- *itemIndex* not -2 and is out of range for an existing column.

Details

If *itemIndex* is -2, the returned string describes the table as a whole and contains the following keywords, with a semicolon after each keyword-value pair.

Keyword	Information Following Keyword
TABLENAME	The name of the table.
HOST	The host specification of the table's host window if it is a subwindow or "" if it is a top-level table window.
ROWS	Number of used rows in the table.
COLUMNS	Number of used columns in the table including the Point column.
SELECTION	A description of the table selection as you would specify it when invoking the ModifyTable operation's selection keyword.
FIRSTCELL	An identification of the first visible data cell in the top/left corner of the table in row-column format. The first data cell is at location 0, 0.
LASTCELL	An identification of the last visible data cell in the bottom/right corner of the table in row-column format.
TARGETCELL	An identification of the target (highlighted) data cell in row-column format.
ENTERING	1 if an entry has been started in the entry line, 0 if not.

If *itemIndex* is -1 up to but not including the number of used columns to the right of the Point column, the returned string describes the specified column and contains the following keywords, with a semicolon after each keyword-value pair.

Keyword	Information Following Keyword
TABLENAME	The name of the table.
HOST	The host specification of the table's host window if it is a subwindow or "" if it is a top-level table window.
COLUMNNAME	Name of the column as you would specify it to the Edit operation if you were creating a table showing just the column of interest.
TYPE	Column's type which will be one of the following: Unused, Point, Index, Label, Data, RealData, ImagData. "Index" identifies a index column such as the X values of a wave. "Label" identifies a column of dimension labels. "Data" identifies a data column of a scalar wave. RealData and ImagData identify a real or imaginary column of a complex wave.
INDEX	Column's position. -1 refers to the Point column, 0 to the first data column, and so on.
DATATYPE	Numeric data type of the wave or zero for text waves. See WaveType for a definition of data type codes.
WAVE	A full data folder path to the wave displayed in the column or "" for the Point column.
COLUMNS	The total number of columns in the table from the wave for the column for which you are getting information. This can be used to skip over all of the columns of a multidimensional wave.
HDIM	The wave dimension displayed horizontally as you move from one column to the next. 0 means rows, 1 means columns, 2 means layers, 3 means chunks.
VDIM	The wave dimension displayed vertically in the column. 0 means rows, 1 means columns, 2 means layers, 3 means chunks.
TITLE	As specified for the ModifyTable operation's title keyword.

Keyword	Information Following Keyword
WIDTH	Column's width in points.
FORMAT	As specified for the ModifyTable operation's format keyword.
DIGITS	As specified for the ModifyTable operation's digits keyword.
SIGDIGITS	As specified for the ModifyTable operation's sigDigits keyword.
TRAILINGZEROS	As specified for the ModifyTable operation's trailingZeros keyword.
SHOWFRACSECONDS	As specified for the ModifyTable operation's showFracSeconds keyword.
FONT	The name of the column's font.
SIZE	Column's font size.
STYLE	As specified for the ModifyTable operation's style keyword.
ALIGNMENT	0=left, 1=center, 2=right.
RGB	The column's color in R,G,B format.
ELEMENTS	As specified for the ModifyTable operation's elements keyword.

Examples

This example makes the table's target cell advance by one position within the range of selected cells each time it is called. To try it, create a table, select a range of cells and then run the function using the Macros menu.

```
Menu "Macros"
   "Test/1", /Q, AdvanceTargetCell("")
End

Function AdvanceTargetCell(tableName)
   String tableName             // Name of table or "" for top table.

   String info = TableInfo(tableName, -2)
   if (strlen(info) == 0)
      return -1                 // No such table
   endif

   String selectionInfo
   selectionInfo = StringByKey("SELECTION", info)

   Variable fRow, fCol, lRow, lCol, tRow, tCol
   sscanf selectionInfo, "%d,%d,%d,%d,%d,%d", fRow, fCol, lRow, lCol, tRow, tCol

   tCol += 1
   if (tCol > lCol)
      tCol = fCol
      tRow += 1
      if (tRow > lRow)
         tRow = fRow
      endif
   endif

   ModifyTable selection=(-1, -1, -1, -1, tRow, tCol)
End
```

See Also

The **ModifyTable** operation.

Tag

Tag [*flags*] [***traceOrAxisName*, *xAttach*** [, ***textStr***]]

The Tag operation puts a tag on the target or named graph window or subwindow. A tag is an annotation that is attached to a particular point on a trace, image, waterfall plot, or axis in a graph.

Parameters

traceOrAxisName is an optional trace or axis name. A trace name can be optionally followed by the # character and an instance number in order to distinguish multiple instances of the same wave in a graph. It identifies the trace or image to which the tag is to be attached. An axis name can be one of the standard axis names (Bottom, Top, Left, or Right) or a user-defined custom axis name.

A string containing *traceOrAxisName* must be used with the $ operator to specify *traceOrAxisName*.

xAttach is the X value of the point on the trace to which the tag is to be attached. For a multidimensional image, it is the linear index into the matrix array. For an axis, *xAttach* can be the X or Y point depending on the particular axis to which the tag is attached; specifying NaN for *xAttach* will center the tag on the axis.

textStr is the text that is to appear in the tag.

Flags

/A=*anchorCode* Specifies position of tag anchor point. *anchorCode* is a literal, *not* a string.

LT	left top
LC	left center
LB	left bottom
MT	middle top
MC	middle center (default)
MB	middle bottom
RT	right top
RC	right center
RB	right bottom

The anchor point is on the tag itself. Any line or arrow drawn from the tag to the wave starts at the tag’s anchor point. The anchor point also determines the precise spot on the tag which represents its position.

/AO=*ao* Sets the text's auto-orientation mode. A non-zero *a0* value overrides the /O value.

/AO is for trace tags only. Setting /AO for any other kind of annotation has no effect.

An auto-oriented tag's text rotates whenever it is redrawn, usually when the underlying data changes, the graph is resized, or when the tag is attached to a new point.

The values for *ao* are:

ao=0:	No auto-orientation. Use the /O value (default).
ao=1:	Tangent to the trace line at the attachment point.
ao=2:	Tangent to the trace line, snaps to vertical or horizontal if within 2 degrees of vertical or horizontal.
ao=3:	Perpendicular to the trace line.
ao=4:	Perpendicular to the trace line, snaps to vertical or horizontal if within 2 degrees of vertical or horizontal.

/B=(*r*,*g*,*b*) Sets color of the tag’s background. *r*, *g*, and *b* specify the amount of red, green, and blue as an integer from 0 to 65535.

/B=*b* Controls the tag background.

b=0:	Opaque background.
b=1:	Transparent background.
b=2:	Same background as the graph plot area background.
b=3:	Same background as the window background.

/C Changes the existing tag.

/F=*frame* Controls the tag frame.

frame=0:	No frame.
frame=1:	Underline frame.
frame=2:	Box frame.

/G=(*r*,*g*,*b*)	Sets color of the text in the tag. *r*, *g*, and *b* specify the amount of red, green, and blue as an integer from 0 to 65535.
/H=*legendSymbolWidth*	*legendSymbolWidth* sets width of the legend symbol (the sample line or marker) in points. Use 0 for the default, automatic width.
/I=*i*	Controls the tag visibility. *i*=1: Tag will be invisible if it is "off screen". "Off screen" means that its attachment point or any part of the tag's text is off screen. This is esthetically pleasing but gives you nothing to grab if you want to drag the tag back on screen. *i*=0: Tag will always be visible. If it is "off screen", it appears at the extreme edge of the graph.
/K	Kills existing tag.
/L=*line*	Controls the line attaching the tag to the tagged point. *line*=0: No line from tag to attachment point. *line*=1: Line connecting tag to attachment point. *line*=2: Line with arrow pointing from tag to attachment point. *line*=3: Line with arrow pointing from attachment point to tag. *line*=4: Line with arrows at both ends.
/LS= *linespace*	Specifies a tweak to the normal line spacing where *linespace* is points of extra (plus or minus) line spacing. For negative values, a blank line may be necessary to avoid clipping the bottom of the last line.
/M[=*sameSize*]	/M or /M=1 specifies that legend markers should be the same size as the marker in the graph. /M=0 turns same-size mode off so that the size of the marker in the legend is based on text size.
/N=*name*	Specifies name of the tag to create or change.
/O=*rot*	Sets the text's rotation. *rot* is in (integer) degrees, counterclockwise and must be a number from -360 to 360. 0 is normal horizontal left-to-right text, 90 is vertical bottom-to-top text. If the tag is attached to a trace (not an image or axis), any non-zero /AO value will overwrite this rotation value.
/P=*tipOffset*	Sets the offset from the tip of a tag's line or arrow to the point on the wave that it is tagging. *tipOffset* is a positive number from 0 to 200 in points. If *tipOffset*=0 (default), it automatically chooses an appropriate offset.
/Q[=*contourInstance*]	Associates a tag with a particular contour level trace in a graph recreation macro. Of interest mainly to hard-core programmers. When "=*contourInstance*" is present, /Q associates the tag with the contour wave. Igor will feel free to change or delete the tag, as appropriate, when it recalculates the contour (because you changed the contour data or appearance, the graph size or the axis range). *contourInstance* is a contour instance name, such as zWave or zWave#1 if you have the same wave contoured twice in the graph. /Q by itself, with "=*contourInstance*" not present, disassociates the tag from the contour wave. Igor will no longer modify or delete the tag (unless the contour level to which it is attached is deleted). If you manually tweak a contour label, using the Modify Annotation dialog, Igor uses this flag.
/R=*newName*	Renames the tag.

/S=*style*	Controls the tag frame style. *style*=0: Single frame. *style*=1: Double frame. *style*=2: Triple frame. *style*=3: Shadow frame.
/T=*tabSpec*	*tabSpec* is a single number in points, such as `/T=72`, for evenly spaced tabs or a list of tab stops in points such as `/T={50, 150, 225}`.
/TL=*extLineSpec*	Specifies extended tag line parameters similar to the **SetDrawEnv** arrow settings. *extLineSpec* = {*keyword* = *value*,…} or zero to turn off all extended specifications. Valid *keyword-value* pairs are: len=*l* Length of arrow head in points (*l*=0 for auto). fat=*f* Width to length ratio of arrow head (default is 0.5 same as *f*=0). style=*s* Sets barb side mode (see **SetDrawEnv** astyle for values). shar =*s* Sets sharpness between -1 and 1 (default is 0; blunt). frame=*f* Sets frame thickness in outline mode. lThick=*l* Sets line thickness in points (default is 0.5 for *l*=0). lineRGB=(*r*,*g*,*b*) Sets color for lines. Default is all zeros (black); the same as the tag frame. dash=*d* Specifies dash pattern number between 0 and 17 (see **SetDashPattern** for patterns).
/W=*winName*	Operates on the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/V=*vis*	Controls annotation visibility. *vis*=0: Invisible annotation; not selectable. The annotation is still listed in **AnnotationList**. *vis*=1: Visible annotation (default).
/X=*xOffset*	Distance from point to tag as percentage of graph width. For axis tags, the offsets are proportional to the size of the text used for the axis labels.
/Y=*yOffset*	Distance from point to tag as percentage of graph height. For axis tags, the offsets are proportional to the size of the text used for the axis labels.
/Z=*freeze*	Controls freezing of tag position. *freeze*=1: Freezes tag position (you can't move it with the mouse). *freeze*=0: Unfreezes it.

Details

If the /C flag is used, it must be the first flag in the command and must be followed immediately by the /N=*name* flag.

If the /K flag is used, it must be the first flag in the command and must be followed immediately by the /N=*name* flag with no further flags or parameters.

traceOrAxisName, xAttach, and *textStr* are all optional. If *traceOrAxisName* is specified, then *xAttach* must be specified, and vice versa. *textStr* may be specified only if *traceOrAxisName* and *xAttach* are specified.

This syntax allows changes to the tag to be made through the *flags* parameters without needing to respecify the other parameters. Similarly, the tag's attachment point can be changed without needing to respecify the *textStr* parameter.

xAttach is in terms of the wave's X scaling. If *traceOrAxisName* is displayed as an XY pair, we recommend that you use "point scaling" for the waves, so that *xAttach* can be a point number (because *xAttach* will not be an X axis value).

A tag can have at most 100 lines.

textStr can contain escape codes which affect subsequent characters in the text. An escape code is introduced by a backslash character. In a literal string, you must enter two backslashes to produce one. See **Backslashes in Annotation Escape Sequences** on page III-57 for details.

Using escape codes you can change the font, size, style and color of text, create superscripts and subscripts, create dynamically-updated text, insert legend symbols, and apply other effects. See **Annotation Escape Codes** on page III-53 for details.

Some escape codes insert text based on the wave point or axis to which a tag is attached. See **Tag Escape Codes** on page III-55 and **Axis Label Escape Codes** on page III-57 for details.

The characters "<??>" in a tag indicate that you specified an invalid escape code or used a font that is not available.

Examples

```
Tag/C/N=t1/X=25/Y=50
```

moves the tag named t1 to the location defined by X=25 and Y=50.

```
Tag/C/N=t1 wave1, 50
```

moves the tag named t1 to wave1 at x=50.

```
Tag/N=t2 wave1, 50,"\\JC\\{numpnts(wave1)} points\rin this wave"
```

creates a new tag on wave1 that shows the number of points in the wave.

```
Tag w,0,"\\{\"%g is first, %g is last\"\rw[0],w[numpnts(w)-1]}"
```

creates a new tag on w that shows the value of the first and last points in the wave.

`"\r"` inserts a carriage-return character which starts a new line of text in the annotation.

Following is an example of various ways in which axis tags can be used:

```
Make/O jack=sin(x/8)
SetScale x,0,14e9,"y" jack
Display jack
Label bottom "\\u#2"                // turn off default axis label
ModifyGraph axOffset(bottom)=1.16667 // make room for tag (manual adustment)
Tag/N=text0/F=0/A=MT/X=0.20/Y=-4.29/L=0 bottom, Nan, "\\JCTime (\\U)\r2nd line"

// now a few "important location" tags...
Tag/N=text1/F=0/A=LB/X=1.20/Y=3.00 bottom, 0, "Big Bang"
Tag/N=text2/F=0/A=MB/X=0.00/Y=2.86 bottom, 8000000000, "Earth formed"
Tag/N=text3/F=0/A=RB/X=-0.80/Y=4.71 bottom, 13040000000, "Dinosaurs ruled"
```

See Also

TextBox, **Legend**, **AppendText**, **AnnotationInfo**, **AnnotationList**

TagVal, **TagWaveRef**

Annotation Escape Codes on page III-53

Label, **Axis Labels** on page II-246

Trace Names on page II-216, **Programming With Trace Names** on page IV-81

TagVal

`TagVal(`*`code`*`)`

TagVal is a very specialized function that is only valid when called from within the text of a tag as part of a \{} dynamic text escape sequence. It returns a number reflecting some property of the tag and helps you to display information about the tagged wave. The property is selected by the *code* parameter:

code	**Return Value**
0	Similar to \OP, returns the tag attach point number.
1	Similar to \OX, returns the X coordinate of tag attachment in the graph. When a tag is attached to an XY pair of traces, the X coordinate will most likely be different than the tag's X scaling attachment value specified in the Tag command.
2	Similar to \OY, returns the Y coordinate of tag attachment in the graph or the Y axis value in a Waterfall plot.
3	Similar to \OZ, returns the Z coordinate of tag attachment in a contour, image, or Waterfall plot.
4	Similar to \Ox, returns the trace x offset.
5	Similar to \Oy, returns the trace y offset.
6	Returns the X muloffset (with the not set value 0 translated to 1).
7	Returns the Y muloffset (with the not set value 0 translated to 1).

Because TagVal returns a numeric value, the result can be formatted any way you wish using the **printf** formatting codes. In contrast, the \O codes insert preformatted text, and you don't have control over the format.

TagVal is sometimes used in conjunction with the **TagWaveRef** function. For example, you might write a user-defined function that calculates a value as a function of a wave and a point number.

Examples

```
Tag wave0, 0, "Y value is \\{\"%g\",TagVal(2)}"
Tag wave0, 0, "Y value is \\{\"%g\",TagWaveRef()[TagVal(0)]}"
Tag wave0, 0, "Y value is \\OY"
```

These examples all produce identical results.

See Also

The **Tag** operation, the **TagWaveRef** function.

For a discussion of wave references, see **Wave Reference Functions** on page IV-186.

TagWaveRef

`TagWaveRef()`

TagWaveRef is a very specialized function that is only valid when called from within the text of a tag as part of a \{} dynamic text escape sequence. It returns a wave reference to the wave that the tag is on and helps you to display information about the tagged wave. It is often used in conjunction with the **TagVal** function. You can pass the result of TagWaveRef to any function that takes a Wave parameter.

Examples

Show the name of the data folder containing the tagged wave:

```
Tag wave0, 0,"\\ON is in \\{\"%s\",GetWavesDataFolder(TagWaveRef(),0)}"
```

See Also

The **Tag** operation, the **TagVal** function

For a discussion of wave references, see **Wave Reference Functions** on page IV-186.

tan

`tan(`*`angle`*`)`

The tan function returns the tangent of *angle* which is in radians.

In complex expressions, *angle* is complex, and tan(*angle*) returns a complex value:

$$\tan(x+iy)=\frac{\sin(\mathrm{x+iy})}{\cos(x+iy)}=\frac{\sin(2x)+i\sinh(2y)}{\cos(2x)+\cosh(2y)}.$$

See Also
atan, **atan2**, **sin**, **cos**, **sec**, **csc**, **cot**

tanh

`tanh(num)`

The tanh function returns the hyperbolic tangent of *num*:

$$\tanh(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}.$$

In complex expressions, *num* is complex, and tanh(*num*) returns a complex value.

See Also
sinh, **cosh**, **coth**

TextBox

`TextBox [flags] [textStr]`

The TextBox operation puts a textbox on the target or named graph window. A textbox is an annotation that is not associated with any particular trace.

Parameters

textStr is the text that is to appear in the textbox. It is optional.

Flags

/A=*anchorCode* Specifies position of textbox anchor point.

anchorCode	**Position**	*anchorCode*	**Position**
LT	left top	RT	right top
LC	left center	RC	right center
LB	left bottom	RB	right bottom
MT	middle top		
MC	middle center		
MB	middle bottom		

anchorCode is a literal, *not* a string.

For interior textboxes, the anchor point is on the rectangular edge of the plot area of the graph window (where the left, bottom, right, and top axes are drawn).

For exterior textboxes, the anchor point is on the rectangular edge of the entire graph window.

/B=(*r*,*g*,*b*) Sets the color of the tag's background. *r*, *g*, and *b* specify the amount of red, green, and blue as an integer from 0 to 65535.

/B=*b*	Controls the textbox background. *b*=0: Opaque background. *b*=1: Transparent background. *b*=2: Same background as the graph plot area background. *b*=3: Same background as the window background.
/C	Changes existing textbox.
/D={*thickMult* [, *shadowThick* [, *haloThick*]]}	*thickMult* multiplies the normal frame thickness of a text-box. The thickness may be set using just /D=*thickMult*. *shadowThick*, if present, overrides Igor's normal shadow thickness. It is in units of fractional points. *haloThick* governs the annotation's halo thickness (a surrounding band of the annotation's background color), which can be -1 to 10 points wide. The default *haloThick* value is -1, which preserves the behavior of previous versions of Igor where the halo of all annotations was set by the global variable root:V_TBBufZone. Any negative value of *haloThick* (-0.5, for example) will be overridden by V_TBBufZone if it exists, otherwise the absolute value of *haloThick* will be used. A zero or positive value overrides V_TBBufZone. Any of the parameters may be missing. To set *haloThick* to 0 without changing other parameters, use `/D={,,0}`.
/E[=*exterior*]	/E or /E=1 forces textbox (or legend) to be exterior to graph (provided *anchorCode* is not MC) and pushes the graph margins away from the anchor edge(s). /E=2 also forces exterior mode but does not push the margins. /E=0 returns it to the default (an "interior textbox" which can be anywhere in the graph window).
/F=*frame*	Controls the textbox frame. *frame*=0: No frame. *frame*=1: Underline frame. *frame*=2: Box frame.
/G=(*r*,*g*,*b*)	Sets color of the text in the tag. *r*, *g,* and *b* specify the amount of red, green, and blue as an integer from 0 to 65535.
/H=*legendSymbolWidth*	*legendSymbolWidth* sets width of the legend symbol (the sample line or marker) in points. Use 0 for the default, automatic width.
/K	Kills existing textbox.
/LS= *linespace*	Specifies a tweak to the normal line spacing where *linespace* is points of extra (plus or minus) line spacing. For negative values, a blank line may be necessary to avoid clipping the bottom of the last line.
/M[=*sameSize*]	/M or /M=1 specifies that legend markers should be the same size as the marker in the graph. /M=0 turns same-size mode off so that the size of the marker in the legend is based on text size.
/N=*name*	Specifies the name of the textbox to change or create.
/O=*rot*	Sets the text's rotation. *rot* is in (integer) degrees, counterclockwise and must be a number from -360 to 360. 0 is normal horizontal left-to-right text, 90 is vertical bottom-to-top text.

/R=*newName*	Renames the textbox.
/S=*style*	Controls the textbox frame style. *style*=0: Single frame. *style*=1: Double frame. *style*=2: Triple frame. *style*=3: Shadow frame.
/T=*tabSpec*	*tabSpec* is a single number in points, such as `/T=72`, for evenly spaced tabs or a list of tab stops in points such as `/T={50, 150, 225}`.
/V=*vis*	Controls annotation visibility. *vis*=0: Invisible annotation; not selectable. The annotation is still listed in **AnnotationList**. *vis*=1: Visible annotation (default).
/W=*winName*	Operates in the named graph window or subwindow. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
/X=*xOffset*	For interior textboxes *xOffset* is the distance from anchor to textbox as a percentage of the plot area width. For exterior textboxes *xOffset* is the distance from anchor to textbox as a percentage of the graph window width. See /E and /A.
/Y=*yOffset*	*yOffset* is the distance from anchor to textbox as a percentage of the plot area height (interior textboxes) or graph window height (exterior textboxes). See /E and /A.
/Z=*freeze*	Controls freezing of textbox position. *freeze*=1: Freezes textbox position (you can't move it with the mouse). *freeze*=0: Unfreezes it.

Details

Use the optional /W=*winName* flag to specify a specific graph or layout window. When used on the command line or in a Macro, Proc, or Window procedure, /W must precede all other flags.

If the /C flag is used, it must be the first flag in the command (except that if may follow an initial /W) and must be followed immediately by the /N=*name* flag.

If the /K flag is used, it must be the first flag in the command (or follow an initial /W) and must be followed immediately by the /N=*name* flag with no further flags or parameters.

textStr is optional. If missing, the textbox text is unchanged. This allows changes to the textbox to be made through the flags without changing the text.

A textbox can have at most 100 lines.

textStr can contain escape codes which affect subsequent characters in the text. An escape code is introduced by a backslash character. In a literal string, you must enter two backslashes to produce one. See **Backslashes in Annotation Escape Sequences** on page III-57 for details.

Using escape codes you can change the font, size, style and color of text, create superscripts and subscripts, create dynamically-updated text, insert legend symbols, and apply other effects. See **Annotation Escape Codes** on page III-53 for details.

The characters "<??>" in a textbox indicate that you specified an invalid escape code or used a font that is not available.

Examples

```
TextBox/C/N=t1/X=25/Y=50
```

moves the textbox named t1 to the location defined by X=25 and Y=50.

```
TextBox/C/N=t1 "New Text"
```

changes the text for t1.

See Also

Tag, **Legend**, **AppendText**, **AnnotationInfo**, **AnnotationList**

Annotation Escape Codes on page III-53

See the **printf** operation for formatting codes used in *formatStr*.

Programming with Annotations on page III-52.

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

TextEncoding

```
#pragma TextEncoding = "<text encoding name>"
```

#pragma TextEncoding is a compiler directive that tells Igor the text encoding used by a procedure file. Igor needs to know this in order to correctly interpret non-ASCII characters in the file. We recommend that you add a TextEncoding pragma to your procedure files.

See **Text Encoding Names and Codes** on page III-434 for a list of accepted text encoding names.

This statement must be flush against the left edge of the procedure file with no indentation. It is usually placed at or near the top of the file.

The TextEncoding pragma was added in Igor Pro 7.00 and is ignored by earlier versions.

See **The TextEncoding Pragma** on page IV-51 for further explanation.

TextEncodingCode

`TextEncodingCode(`*`textEncodingNameStr`*`)`

The TextEncodingCode function returns the Igor text encoding code for the named text encoding or 0 if the text encoding is unknown.

The TextEncodingCode function was added in Igor Pro 7.00.

Parameters

textEncodingNameStr is an Igor text encoding name as listed under **Text Encoding Names and Codes** on page III-434.

Details

Igor ignores all non-alphanumeric characters in text encoding names so "Shift JIS", "ShiftJIS", "Shift_JIS" and "Shift-JIS" are equivalent.

It also ignores leading zeros in numbers embedded in text encoding names so "ISO-8859-1" and "ISO-8859-01" are equivalent.

TextEncodingCode does a case-insensitive comparison.

See Also

Text Encodings on page III-409, **Text Encoding Names and Codes** on page III-434, **TextEncodingName**

TextEncodingName

`TextEncodingName(`*`textEncoding, index`*`)`

The TextEncodingName function returns one or more text encoding names corresponding to the specified text encoding code. The result is returned as a string value.

If *textEncoding* is not a valid Igor text encoding code or if index is out of range, TextEncodingName returns "Unknown".

This function is mainly useful for providing a human-readable string corresponding to a given text encoding code for display purposes. You might use it to generate some Internet-compatible text, such as an HTML page, if you need a string to specify the charset.

The TextEncodingName function was added in Igor Pro 7.00.

Parameters

textEncoding is an Igor text encoding code as listed under **Text Encoding Names and Codes** on page III-434.

index specifies which text encoding name you want. A given text encoding can be identified by more than one name. Normally you will pass 0 to get the first text encoding name for the specified text encoding code. This is the preferred text encoding name. You can pass 1 for the second name, if any, 2 for the third, if any, and so on. You can pass -1 to get a semicolon-separated list of all text encoding names for the specified text encoding.

Details

Internally Igor has a table of text encoding codes and the corresponding text encoding names. For a given code there may be more than one acceptable name. For example, for the code 2 (MacRoman), the names "macintosh", "MacRoman" and "x-macroman" are accepted, with "macintosh" being preferred. The TextEncodingName function returns a text encoding name from the internal table.

The preferred name is usually the name recognized by the Internet Assigned Numbers Authority (IANA) as listed at http://www.iana.org/assignments/character-sets.

Examples

```
// Get the preferred name for the MacRoman text encoding (2)
String firstName = TextEncodingName(2, 0); Print firstName

// Get the second name for the MacRoman text encoding (2)
String secondName = TextEncodingName(2, 1); Print secondName

// Get a semicolon-separated list of all text encoding names for MacRoman
String names = TextEncodingName(2, -1); Print names
```

See Also

Text Encodings on page III-409, **Text Encoding Names and Codes** on page III-434, **TextEncodingCode**

TextFile

`TextFile(`*`pathName`*`, `*`index`*` [, `*`creatorStr`*`])`

Note: TextFile is antiquated. Use **IndexedFile** instead.

The TextFile function returns a string containing the name of the *index*th TEXT file from the folder specified by *pathName*.

On Macintosh, TextFile returns only files whose file type property is TEXT, regardless of the file's extension.

On Windows, Igor considers files with ".txt" extensions to be of type TEXT.

Details

TextFile returns an empty string (`""`) if there is no such file.

pathName is the name of an Igor symbolic path; it is *not* a string.

index starts from zero.

creatorStr is an optional string argument containing four ASCII characters such as "IGR0". Only files of the specified Macintosh creator code are indexed. Set *creatorStr* to "`????`" to index all text files (or omit the argument altogether). This argument is ignored on Windows systems.

The order of files in a folder is determined by the operating system.

Examples

You can use TextFile in a procedure to sequence through each TEXT file in a folder, put the name of the text file into a string variable, and use this string variable as a parameter to the **LoadWave** or **Open** operations:

```
Function/S PrintFirstLineOfTextFiles(pathName)
   String pathName                  // Name of an Igor symbolic path.

   Variable refNum, index
   String str, fileName
   index = 0
   do
      fileName = TextFile($pathName, index)
      if (strlen(fileName) == 0)
```

```
            break                       // No more files
        endif
        Open/R/P=$pathName refNum as fileName
        FReadLine refNum, str           // Read first line including CR/LF
        Print fileName +":" + str       // Print file name and first line
        Close refNum
        index += 1                      // Next file
    while (1)
End
```

See Also

See the **IndexedFile** function, which is similar to TextFile but works on files of any type, and also **IndexedDir**. Also see the **LoadWave** and **Open** the operations.

ThreadGroupCreate

`ThreadGroupCreate(`*`nt`*`)`

The ThreadGroupCreate function creates a thread group containing *nt* threads and returns a thread ID number. Use the number of computer processors for *nt* when trying to improve computation speed using parallel threads. A background worker might use just one thread regardless of the number of processors.

See Also

ThreadSafe Functions on page IV-97 and **ThreadSafe Functions and Multitasking** on page IV-308.

ThreadGroupGetDF

`ThreadGroupGetDF(`*`tgID, waitms`*`)`

ThreadGroupGetDFR should be used instead of ThreadGroupGetDF which causes memory leaks.

The ThreadGroupGetDF function retrieves a data folder path string from a thread group queue and removes the data folder from the queue.

When called from a preemptive thread it returns a data folder from the thread group's input queue. When called from the main thread it returns a data folder from the thread group's output queue.

tgID is a thread group ID returned by **ThreadGroupCreate**. You can pass 0 for *tgID* when calling ThreadGroupGetDF from a preemptive thread. You must pass a valid thread group ID when calling ThreadGroupGetDF from the main thread.

waitms is the maximum number of milliseconds to wait for a data folder to become available in the queue. Pass 0 to test if a data folder is available immediately. Pass INF to wait indefinitely or until a user abort.

ThreadGroupGetDF returns "" if the timeout period specified by *waitms* expires and no data folder is available in the queue.

See Also

ThreadSafe Functions on page IV-97 and **ThreadSafe Functions and Multitasking** on page IV-308.

The **ThreadGroupGetDFR** function.

ThreadGroupGetDFR

`ThreadGroupGetDFR(`*`tgID, waitms`*`)`

The ThreadGroupGetDF function retrieves a data folder reference from a thread group queue and removes the data folder from the queue. The data folder becomes a free data folder.

When called from a preemptive thread it returns a data folder from the thread group's input queue. When called from the main thread it returns a data folder from the thread group's output queue.

tgID is a thread group ID returned by **ThreadGroupCreate**. You can pass 0 for *tgID* when calling ThreadGroupGetDFR from a preemptive thread. You must pass a valid thread group ID when calling ThreadGroupGetDFR from the main thread.

waitms is the maximum number of milliseconds to wait for a data folder to become available in the queue. Pass 0 to test if a data folder is available immediately. Pass INF to wait indefinitely or until a user abort.

ThreadGroupGetDFR returns a NULL data folder reference if the timeout period specified by *waitms* expires and no data folder is available in the queue. You can test for NULL using **DataFolderRefStatus**.

See Also

ThreadSafe Functions on page IV-97, **ThreadSafe Functions and Multitasking** on page IV-308 and **Free Data Folders** on page IV-88.

ThreadGroupPutDF

`ThreadGroupPutDF` ***`tgID, datafolder`***

The ThreadGroupPutDF operation posts data to a preemptive thread group.

Parameters

tgID is thread group ID returned by **ThreadGroupCreate**, datafolder is the data folder you wish to send to the thread group.

datafolder can be just the name of a child data folder in the current data folder, a partial path (relative to the current data folder) and name or an absolute path (starting from root) and name.

Details

When you call it from the main thread, **ThreadGroupPutDF** removes *datafolder* from the main thread's data hierarchy and posts to the input queue of the thread group specified by *tgID*.

When you call it from a preemptive thread, use 0 for *tgID* and the data folder will be posted to the output queue of thread group to which thread belongs.

Input and output data folders may be retrieved from the queues by calling the string function **ThreadGroupGetDF** or **ThreadGroupGetDFR**.

Warning: Take care not to use any stale WAVE, NVAR, or SVAR variables that might contain references to objects in the data folder. Use **WAVEClear** on all WAVE reference variables that might contain references to waves that are in the data folder being posted before calling **ThreadGroupPutDF**. An error will occur if any waves in the data folder are in use or referenced in a WAVE variable.

Warning: Any DFREF variables that refer to the data folder (or any child thereof) must be cleared prior to executing this command. You can clear a DFREF using `dfr=$""`.

From the standpoint of the source thread, ThreadGroupPutDF is conceptually similar to KillDataFolder and, like KillDataFolder, if the current data folder is within *datafolder*, the current data folder is set to the parent of datafolder. You can not pass `root:` as *datafolder*.

See Also

The **ThreadGroupCreate** function, **ThreadSafe Functions** on page IV-97, and **ThreadSafe Functions and Multitasking** on page IV-308.

ThreadGroupRelease

`ThreadGroupRelease(`*`tgID`*`)`

The ThreadGroupRelease function releases thread group (and *tgID* is no longer valid). *tgID* is the thread group ID returned by **ThreadGroupCreate**.

If threads are still running, they are killed. An attempt is made to safely stop running threads but, if they continue to run, they will be force quit.

ThreadGroupRelease returns zero if successful, -1 if an error occurred (probably invalid *tgID*), or -2 if a force quit was needed. In the latter case, you should restart Igor Pro.

Any data folders remaining in the group's input or output queues will be discarded.

See Also

The **ThreadGroupCreate** function, **ThreadSafe Functions** on page IV-97, and **ThreadSafe Functions and Multitasking** on page IV-308.

ThreadGroupWait

`ThreadGroupWait(`*`tgID`*`, `*`waitms`*`)`

The ThreadGroupWait function returns index+1 of the first thread found still running after *waitms* milliseconds or returns zero if all are done.

tgID is the thread group ID returned by **ThreadGroupCreate** and *waitms* is milliseconds to wait.

If any of the threads of the group encountered a runtime error, the first such error will be reported now.

Use zero for *waitms* to just test or provide a large value to cause the main thread to sleep until the threads are finished. You can use INF to wait forever or until a user abort. If you know the maximum time the threads should take, you can use that value so you can print an error message or take other action if the threads don't return in time.

When ThreadGroupWait is called, Igor updates certain internal variables including variables that track whether a thread has finished and what result it returned. Therefore you must call ThreadGroupWait before calling **ThreadReturnValue**.

ThreadGroupWait updates the internal state of all threads in the group.

Finding a Free Thread

If you pass -2 for *waitms*, ThreadGroupWait returns index+1 of the first free (not running) thread or 0 if all threads in the group are running.

This allows you to dispatch a thread anytime a free thread is available. See **Parallel Processing - Thread-at-a-Time Method** on page IV-311 for an example.

See Also

The **ThreadGroupCreate** function, **ThreadSafe Functions** on page IV-97, and **ThreadSafe Functions and Multitasking** on page IV-308.

ThreadProcessorCount

`ThreadProcessorCount`

The ThreadProcessorCount function returns the number of processors in your computer. For example, on a Macintosh Core Duo, it would return 2.

ThreadReturnValue

`ThreadReturnValue(`*`tgID`*`, `*`index`*`)`

The ThreadReturnValue function returns the value that the specified thread function returned when it exited. Returns NAN if thread is still running. *tgID* is the thread group ID returned by **ThreadGroupCreate** and *index* is the thread number.

When **ThreadGroupWait** is called, Igor updates certain internal variables including variables that track whether a thread has finished and what result it returned. Therefore you must call ThreadGroupWait before calling ThreadReturnValue.

See Also

The **ThreadGroupCreate** function, **ThreadSafe Functions** on page IV-97, and **ThreadSafe Functions and Multitasking** on page IV-308.

ThreadSafe

`ThreadSafe Function `*`funcName`*`()`

The ThreadSafe keyword declaration specifies that a user function can be used for preemptive multitasking background tasks on multiprocessor computer systems.

A ThreadSafe function is one that can operate correctly during simultaneous execution by multiple threads. Such functions are generally limited to numeric or utility functions. Functions that access windows are not ThreadSafe. To determine if an operation is ThreadSafe, use the Command Help tab of the Help Browser and choose ThreadSafe from the pop-up menu.

ThreadSafe functions can call other ThreadSafe functions but may not call non-ThreadSafe functions. Non-ThreadSafe functions can call ThreadSafe functions.

See Also

ThreadSafe Functions on page IV-97 and **ThreadSafe Functions and Multitasking** on page IV-308.

ThreadStart

`ThreadStart` ***`tgID, index, WorkerFunc`*****`(`*****`param1, param2,…`*****`)`**

The ThreadStart operation starts the specified function running in a preemptive thread.

Parameters

tgID is thread group ID returned by **ThreadGroupCreate**, *index* is the desired thread of the group to set up to execute the specified ThreadSafe *WorkerFunc*.

Details

The worker function starts running immediately.

The worker function must be defined as ThreadSafe and must return a real or complex numeric result.

The worker function's return value can be obtained after the function finishes by calling **ThreadReturnValue**. Igor records the fact that a thread has terminated when you call ThreadGroupWait so you must call ThreadGroupWait before calling ThreadReturnValue.

The worker function can take variable and wave parameters. It can not take pass-by-reference parameters or data folder reference parameters.

Any waves you pass to the worker are accessible to both the main thread and to your preemptive thread. Such waves are marked as being in use by a thread and Igor will refuse to perform any manipulations that could change the size of the wave.

See Also

The **ThreadGroupCreate** and **ThreadReturnValue** functions; **ThreadSafe Functions** on page IV-97, and **ThreadSafe Functions and Multitasking** on page IV-308.

ticks

`ticks`

The ticks function returns the number of ticks (approximately 1/60 second) elapsed since the operating system was initialized.

See Also

The **StopMSTimer** function.

Tile

`Tile [`***`flags`*****`] [`*****`objectName`*****` [, `*****`objectName`*****`]…]`**

The Tile operation tiles the specified objects in the top page layout.

Parameters

objectName is the name of a graph, table, picture or annotation object in the top page layout.

Flags

/A=(*rows*,*cols*)	Specifies number of rows/columns in which to tile objects.
/G=*grout*	Specifies grout, the spacing between window tiles, in prevailing coordinates (points unless preceded by /I, /M or /R).
/I	Specifies coordinates in inches.
/M	Specifies coordinates in centimeters.

/O=*objTypes*	Adds objects of type(s) specified by bitwise mask to list of objects to be tiled: Bit 0: Tile graphs. Bit 1: Tile tables. Bit 3: Tile pictures. Bit 5: Tile textboxes. See **Setting Bit Parameters** on page IV-12 for details about bit settings.
/R	Specifies coordinates measured in percent of the printable page.
/S	Adds selected objects to objects to be tiled.
/W=(*left*,*top*,*right*,*bottom*)	Specifies page layout area in which to tile objects. Coordinates are in points unless /I, /M or /R are specified before /W.

Details

If /A=(*rows*,*cols*) is not used, Tile uses an appropriate number of rows and columns. If /A=(*rows*,*cols*) is used, objects are tiled in a grid of that many rows and columns. If *rows* or *cols* is zero, it substitutes an appropriate number for the zero parameter.

Objects to be tiled are determined by the /S and /O=*objTypes* flags and by any *objectName*s.

If no /S or /O flags are present and there are no *objectName*s, then all objects in the layout are tiled.

Otherwise the objects to be tiled are determined as follows:

- All objects specified by *objectName*s are tiled.
- If the /S flag is present, the selected objects, if any, are also tiled.
- If the /O=*objTypes* flag is present then any objects specified by *objTypes* are also tiled. *objTypes* is a bitwise mask, so /O=3 tiles both graphs and tables.

See Also

The **Stack** operation.

TileWindows

`TileWindows [`*`flags`*`] [`*`windowName`*` [, `*`windowName`*`]…]`

The TileWindows operation tiles the specified windows on the desktop (*Macintosh*) or in the Igor frame window (*Windows*).

Flags

/A=(*rows*,*cols*)	Specifies number of rows/columns in which to tile windows.
/C	Adds the command window to the windows to be tiled.
/G=*grout*	Specifies grout, the spacing between tiles, in prevailing units (points unless /I or /M are used).
/I	Specifies coordinates in inches.
/M	Specifies coordinates in centimeters.
/O=*objTypes*	Adds windows of types specified by *objTypes* to windows to be tiled.

objTypes is a bitwise mask where:

Bit 0:	Graphs
Bit 1:	Tables
Bit 2:	Page layouts
Bit 4:	Notebooks
Bit 6:	Control panels
Bit 7:	Procedure windows
Bit 9:	Help windows
Bit 12:	XOP target windows
Bit 14:	Camera windows
Bit 16:	Gizmo windows

Other bits should always be zero. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

/P Adds the main procedure window to the windows to be tiled.

/R Specifies coordinates measured as % of tiling rectangle.

/W=(*left*,*top*,*right*,*bottom*)

Specifies tiling rectangle on the screen. Coordinates are in points unless /I, /M, or /R are specified before /W.

Details

The windows to be tiled are determined by the /C, /P, and /O=*objTypes* flags and by the *windowName*s. If no /C, /P or /O flags are present and there is no *windowName*s then all windows are tiled.

Otherwise the windows to be tiled are determined as follows:

- All named windows are tiled.
- If the /C flag is present, the command window is also tiled.
- If the /P flag is present, the procedure window is also tiled.
- If the /O=*objTypes* flag is present, any windows specified by *objTypes* are also tiled.

Examples

To tile all the procedure windows, including the main one, use:

```
TileWindows/P/O=128        // 2^7=128
```

See Also

The **StackWindows** operation.

time

`time()`

The time function returns a string containing the current time. The empty parentheses are required.

See Also

The **date**, **date2secs** and **DateTime** functions.

TitleBox

`TitleBox` `[`**`/Z`**`]` ***`ctrlName`*** `[`***`keyword = value`*** `[,` ***`keyword = value`*** `…]]`

The TitleBox operation creates the named title box in the target window.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the TitleBox control to be created or changed.

The following keyword=value parameters are supported:

anchor= *hv*	Specifies the anchor mode using a two letter code, *hv*. *h* may be L, M, or R for left, middle, and right. *v* may be T, C, or B for top, center and bottom. Default is LT. If fixedSize=1, the anchor code sets the positioning of text within the frame.
appearance={*kind* [, *platform*]}	Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings. *kind* can be one of `default`, `native`, or `os9`. *platform* can be one of `Mac`, `Win`, or `All`. See **Button** and **DefaultGUIControls** for more appearance details.
disable=*d*	Sets user editability of the control. *d*=0: Normal. *d*=1: Hide. *d*=2: Draw in gray state.
fColor=(*r*,*g*,*b*)	Sets color of the titlebox. *r*, *g*, and *b* are integers from 0 to 65535.
fixedSize=*f*	Controls title box sizing: *f* =0: The titlebox automatically sizes itself to fit the title text (default). *f* =1: The size settings are honored, and the titlebox does not automatically size itself to fit the title text.
font="*fontName*"	Sets the font used for the control, e.g., `font="Helvetica"`.
frame= *f*	Sets frame style: *f*=0: No frame. *f*=1: Default (same as *f*=3). *f*=2: Simple box. *f*=3: 3D sunken frame. *f*=4: 3D raised frame. *f*=5: Text well.
fsize=*s*	Sets font size.
fstyle=*fs*	*fs* is a bitwise parameter with each bit controlling one aspect of the font style as follows: Bit 0: Bold Bit 1: Italic Bit 2: Underline Bit 4: Strikethrough See **Setting Bit Parameters** on page IV-12 for details about bit settings.
labelBack=(*r*,*g*,*b*) or 0	Sets background color for title box. *r*, *g*, and *b* are integers from 0 to 65535. If not set (or labelBack=0), then background is transparent (not erased).
pos={*left*,*top*}	Sets the location of the top left corner in pixels.
pos+={*dx*,*dy*}	Offsets the position of the control in pixels.
size={*w*,*h*}	Set the width and height in pixels.

title=*titleStr*	Sets the text of the title box to *titleStr*. *titleStr* is limited to 100 bytes.
	Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.
variable= *svar*	Specifies an optional global string variable from which to get the TitleBox text.
win=*winName*	Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed.
	When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Flags

/Z	No error reporting.

Details

The text can come from either the title=*titleStr* or variable=*svar* method. Whichever is used last is the current method. The maximum length of text with the title=*titleStr* method is 100 bytes while the variable=*svar* method has no limit.

Using escape codes you can change the font, size, style and color of text, and apply other effects. See **Annotation Escape Codes** on page III-53 for details.

By default, the titlebox automatically resizes itself relative to the anchor point on the rectangle that encloses the text. Therefore you can specify a size of 0,0 along with a pos value in order to place the anchor point at the desired position. When fixedSize=1 is used, the titlebox does not resize itself and instead honors the values specified via the size keyword.

TitleBoxes can be used not only for titles but also as status or results readout areas, especially in conjunction with the variable= *svar* mode. When using a titlebox like this, you may find it useful to use fixedSize=1 so that the titlebox doesn't change size as the text changes.

Examples

```
NewPanel /W=(94,72,459,294)
DrawLine 150,32,150,140
DrawLine 70,100,213,100          // draw crossing lines at 150,100

// illustrate a default box
TitleBox tb1,title="A title box\rwith 2 lines",pos={150,100}

// Move center to 150,100
TitleBox tb1,pos={150,100},size={0,0},anchor=MC

// Set background color and therfore opaque mode
TitleBox tb1,labelBack=(55000,55000,65000)

// Now a few frame styles. Run these one at a time
TitleBox tb1,frame= 0            // no frame
TitleBox tb1,frame= 2            // plain frame
TitleBox tb1,frame= 3            // 3D sunken
TitleBox tb1,frame= 4            // 3D raised
TitleBox tb1,frame= 5            // text well

// Now some fancy text…
TitleBox tb1,frame= 1            // back to default (3D raised)
TitleBox tb1,title= "\Z18\[020 log\\B10\\M|[1 + 2K(jwt) + (jwt)\\S2\\M]|\\S-1"

// Create a string variable and hook up to the TitleBox
String s1= "text from a string variable"
TitleBox tb1,variable=s1

// Change string variable contents & note automatic update of TitleBox
s1= "something new"
```

See Also

Annotation Escape Codes on page III-53.

ToCommandLine

`ToCommandLine` ***`commandsStr`***

The ToCommandLine operation sends command text to the command line without executing the command(s).

The intended usage is for user-created panel windows with "To Cmd Line" buttons that are mimicking built-in Igor dialogs. You'll usually want to use Execute, instead.

Parameters

commandsStr	The text of one or more commands.

Details

To send more than one line of commands, separate the commands with "\r" characters.

Note: ToCommandLine does not work when typed on the command line; use it only in a Macro, Proc, or Function.

Examples

```
Macro CmdPanel()
	PauseUpdate; Silent 1
	NewPanel /W=(150,50,430,229)
	Button toCmdLine,pos={39,148},size={103,20},title="To Cmd Line"
	Button toCmdLine,proc=ToCmdLineButtonProc
End

Function ToCmdLineButtonProc(ctrlName) : ButtonControl
	String ctrlName

	String cmd="MyFunction(xin,yin,\"yResult\")"// line 1: generate results
	cmd +="\rDisplay yOutput vs wx as \"results\"" // line 2: display results
	ToCommandLine cmd

	return 0
End
```

See Also

The **Execute** and **DoIgorMenu** operations.

ToolsGrid

`ToolsGrid [/W=`*`winName`*`]` ***`keyword = value`*** **`[,`** ***`keyword = value`*** **`…]`**

The ToolsGrid operation controls the grid you can use for laying out draw or control objects.

Parameters

ToolsGrid can accept multiple *keyword* = *value* parameters on one line.

snap=*n*	Turns snap to grid on (*n*=1) or off (*n*=0).
visible=*n*	Turns on grid visibility (*n*=1) or hides it (*n*=0).
grid=(*xy0*,*dxy*,*ndiv*)	Defines both X and Y grids where *ndiv* is the number of subdivisions between major grid lines and *xy0* and *dxy* define the origin and spacing. Units are in points.
gridx=(*x0*,*dx*,*ndiv*)	Defines the X grid where *ndiv* is the number of subdivisions between major grid lines and *x0* and *dx* define the origin and spacing. Units are in points.
gridy=(*y0*,*dy*,*ndiv*)	Defines the Y grid where *ndiv* is the number of subdivisions between major grid lines and *y0* and *dy* define the origin and spacing. Units are in points.

Flags

/W=*winName*	Sets the named window or subwindow for drawing. When omitted, action will affect the active window or subwindow. This must be the first flag specified when used in a Proc or Macro or on the command line. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The default grid is 1 inch with 8 subdivisions. The grid is visible only in draw or selector mode and appears in front of the currently active draw layer.

TraceFromPixel

`TraceFromPixel(`*`xpixel, ypixel, optionsString`*`)`

The TraceFromPixel function returns a string based on an attempt to hit test the provided X and Y coordinates. Used to determine if the mouse was clicked on a trace in a graph.

When a trace is found, TraceFromPixel returns a string containing the following `KEY:value;` pairs:

`TRACE:`*`tracename`*

`HITPOINT:`*`pnt`*

tracename will be quoted if necessary and may contain instance notation. *pnt* is the point number index into the trace's wave when the hit was detected. If a trace is not found near the coordinate point, a zero length string is returned.

Parameters

xpixel and *ypixel* are the X and Y pixel coordinates.

optionsString can contain the following:

`WINDOW:`*`winName`*`;`

`PREF:`*`traceName`*`;`

`ONLY:`*`traceName`*`;`

`DELTAX:`*`dx`*`;DELTAY:`*`dy`*`;`

Use the `WINDOW` option to hit test in a graph other than the top graph. Use the `ONLY` option to search only for a special target trace. If the PREF option is used then the search will start with the specified trace but if no hit is detected, it will go on to the others.

When identifying a subwindow with `WINDOW:`*`winName`*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

The DELTAX and DELTAY values must both be specified to alter the region that Igor searches for traces. The *dx* and *dy* values are in pixels and the region searched is the rectangle from *xpixel-dx* to *xpixel+dx* and *ypixel-dy* to *ypixel+dy*.

If DELTAX or DELTAY are omitted, the search region depends on whether PREF or ONLY are specified. If either are specified then Igor first searches for the trace using *dx* = 3 and *dy* = 3. If the trace is not identified, Igor searches again using *dx* = 6 and *dy* = 6. If the trace is still not identified, Igor gives up and returns a zero-length result string.

If neither PREF nor ONLY are specified then Igor uses tries 3, 6, 12, and 24 for *dx* and *dy* until it finds a trace or gives up and returns a zero-length result string.

See Also

The **NumberByKey**, **StringByKey**, **AxisValFromPixel**, and **PixelFromAxisVal** functions.

ModifyGraph (traces) and **Instance Notation** on page IV-19 for discussions of trace names and instance notation.

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

TraceInfo

`TraceInfo(`*`graphNameStr, ywavenameStr, instance`*`)`

The TraceInfo function returns a string containing a semicolon-separated list of information about the trace in the named graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

yWaveNameStr is either the name of a wave containing data displayed as a trace in the named graph, or a trace name (wave name with “#n” appended to distinguish the nth image of the wave in the graph). You might get a trace name from the **TraceNameList** function.

If *yWaveNameStr* contains a wave name, *instance* identifies which trace of *yWaveNameStr* you want information about. *instance* is usually 0 because there is normally only one instance of a given wave displayed in a graph. Set *instance* to 1 for information about the second trace of the wave named by *yWaveNameStr*, etc. If *yWaveNameStr* is `""`, then information is returned on the *instance*th trace in the graph.

If *yWaveNameStr* is a trace name, and *instance* is zero, the instance is extracted from *yWaveNameStr*. If *instance* is greater than zero, the wave name is extracted from *yWaveNameStr*, and information is returned concerning the *instance*th instance of the wave.

Details

The string contains several groups of information. Each group is prefaced by a keyword and colon, and terminated with the semicolon. The keywords are as follows:

Keyword	Information Following Keyword
AXISFLAGS	Flags used to specify the axes. Usually blank because /L and /B (left and bottom axes) are the defaults.
AXISZ	Z value of a contour level trace or NaN if the trace is not a contour trace.
ERRORBARS	The ErrorBars command for the trace, as it would appear in the recreation macro (without the beginning tab character).
RECREATION	List of keyword commands as used by ModifyGraph command. The format of these keyword commands is: ***keyword*(x)=*modifyParameters*;**
XAXIS	X axis name.
XRANGE	Point subrange of the trace’s X data wave in “[*startPoint*,þ*endPoint* : *increment*]” format. **Note**: Unlike the actual syntax of a trace subrange specification where increment is preceded by a semicolon character, here it is preceded by a colon character to preserve the notion that semicolon is what separates the keyword-value groups. If the entire X wave is displayed (the usual case), the XRANGE value is “[*]”. If an X wave is not used to display the trace, then the XRANGE value is `""`.
XWAVE	X wave name if any, else blank.
XWAVEDF	Full path to the data folder containing the X wave or blank if no X wave.
YAXIS	Y axis name.
YRANGE	Point subrange of the trace’s Y data wave or “[*]”.

The format of the RECREATION information is designed so that you can extract a keyword command from the keyword and colon up to the ";", prepend "`ModifyGraph `", replace the "`x`" with the name of a trace ("`data#1`" for instance) and then **Execute** the resultant string as a command.

Note: The syntax of any subrange specifications in the RECREATION information are modified in the same way as for XRANGE and YRANGE. Currently only the zColor, zmrkSize, and zmrkNum keywords might have a subrange specification.

Examples

This example shows how to extract a string value from the keyword-value list returned by TraceInfo:

```
String yAxisName= StringByKey("YAXIS", TraceInfo("","",0))
```

This example shows how to extract a subrange and put the semicolon back:

```
String yRange= StringByKey("YRANGE", TraceInfo("","",0))
Print yRange                  // prints "[30,40:2]"
yRange= ReplaceString(":", yRange, ";")
Print yRange                  // prints "[30,40;2]"
```

The next example shows the trace information for the second instance of the wave "data" (which has an instance number of 1) displayed in the top graph:

```
Make/O data=x;Display/L/T data,data     // two instances of data: 0 and 1
Print TraceInfo("","data",1)[0,64]      // error if you try to print all
Print TraceInfo("","data",1)[65,128]
```

Prints the following in the history area:

```
XWAVE:;YAXIS:left;XAXIS:top;AXISFLAGS:/T;AXISZ:NAN(255);XWAVEDF:;
RECREATION:zColor(x)=0;zmrkSize(x)=0;zmrkNum(x)=0;textMarker(x)=
```

Following is a function that returns the marker code from the given instance of a named wave in the top graph. This example uses the convenient GetNumFromModifyStr() function provided by the #include <Readback ModifyStr> procedures, which are useful for parsing strings returned by TraceInfo.

```
#include <Readback ModifyStr>

Function MarkerOfWave(wv,instance)
   Wave wv
   Variable instance

   Variable marker
   String info = TraceInfo("",NameOfWave(wv),instance)

   marker = GetNumFromModifyStr(info,"marker","",0)

   return marker
End
```

See Also

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

The **Execute** operation.

TraceNameList

`TraceNameList(`*`graphNameStr, separatorStr, optionsFlag`*`)`

The TraceNameList function returns a string containing a list of trace names in the graph window or subwindow identified by *graphNameStr*.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

The parameter *separatorStr* should contain a single character such as "," or ";" to separate the names.

Details

The bits of *optionsFlag* have the following meanings:

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Bit Number	Bit Value	Meaning
0	1	Include normal graph traces
1	2	Include contour traces
2	4	Omit hidden traces (the default is to list even hidden traces)

A trace name is defined as the name of the Y wave that defines the trace with an optional #ddd suffix that distinguishes between two or more traces that have the same wave name. Since the trace name has to be parsed, it is quoted if necessary.

Commands that take a trace name as a parameter or in a keyword can use a string containing a trace name with the $ operator to specify traceName. For instance, to change the display mode of a wave, you might use

```
ModifyGraph mode(myWave#1)=3
```

but

```
String myTraceName="myWave#1"
ModifyGraph mode($myTraceName)=3
```

will also work.

Examples

```
Make/O jack,'jack # 2';Display jack,jack,'jack # 2','jack # 2'
Print TraceNameList("",";",1)
Prints: jack;jack#1;'jack # 2';'jack # 2'#1;
```

```
// Generate a list of hidden traces
Make/O jack,jill,joy;Display jack,jill,joy
ModifyGraph hideTrace(joy)=1// hide joy
// (hidden + visible) - visible = hidden
String visibleTraces=TraceNameList("",";",1+4)// only visible normal traces
String allNormalTraces=TraceNameList("",";",1)// hidden + visible normal traces
String hiddenTraces= RemoveFromList(visibleTraces,allNormalTraces)
Print hiddenTraces
// Prints: joy;
```

See Also

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

For other commands related to waves and traces: **WaveRefIndexed**, **XWaveRefFromTrace**, **TraceNameToWaveRef**, **CsrWaveRef**, and **CsrXWaveRef**.

For a description of traces: **ModifyGraph**. For a discussion of contour traces: **Contour Traces** on page II-283.

For commands referencing other waves in a graph: **ImageNameList**, **ImageNameToWaveRef**, **ContourNameList**, and **ContourNameToWaveRef**.

ModifyGraph (traces) and **Instance Notation** on page IV-19 for discussions of trace names and instance notation.

TraceNameToWaveRef

`TraceNameToWaveRef(`*`graphNameStr, traceNameStr`*`)`

The TraceNameToWaveRef function returns a wave reference to the Y wave corresponding to the given trace in the graph window or subwindow named by *graphNameStr*.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

The trace is identified by the string in *traceNameStr*, which could be a string determined using **TraceNameList**. Note that the same trace name can refer to different waves in different graphs.

Use **Instance Notation** (see page IV-19) to choose from traces in a graph that represent waves of the same name. For example, if *traceNameStr* is “myWave#2”, it refers to the third instance of wave “myWave” in the graph (“myWave#0” or just “myWave” is the first instance).

See Also

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

For other commands related to waves and traces: **WaveRefIndexed**, **XWaveRefFromTrace**, **TraceNameList**, **CsrWaveRef**, and **CsrXWaveRef**.

For a description of traces: **ModifyGraph**. For a discussion of contour traces, see **Contour Traces** on page II-283.

For a discussion of wave references, see **Wave Reference Functions** on page IV-186.

For commands referencing other waves in a graph: **ImageNameList**, **ImageNameToWaveRef**, **ContourNameList**, and **ContourNameToWaveRef**.

Triangulate3D

`Triangulate3D [/OUT=`*`format`*`]` ***`srcWave`***

The Triangulate3D operation creates a Delaunay "triangulation" of a 3D scatter wave. The output is a list of tetrahedra that completely span the convex volume defined by *srcWave*. Triangulate3D can also generate the triangulation needed for performing 3D interpolation for the same domain. Normally *srcWave* is a triplet wave (a 2D wave of 3 columns), but can use any 2D wave that has more than 3 columns (the operation ignores all but the first 3 columns).

Flags

/OUT=*format*	Specifies how to save the output triangulation data. *format*=1: Default; saves the triangulation result in the wave M_3DVertexList, which contains in each row, indices to rows in *srcWave* that describe the X, Y, Z coordinates of a single tetrahedral vertex. Each tetrahedron is described by one row in M_3DVertexList. *format*=2: Saves the triangulation result in the wave M_TetraPath, which is a triplet path wave describing the tetrahedra edges. For each tetrahedron, there are four rows (triangles) separated by row of NaNs. The total number of rows in M_TetraPath is 20 times the number of tetrahedra in the triangulation. *format*=4: Saves a wave containing internal diagnostic information generated during the triangulation process.
/VOL	Computes the volume of the full convex hull by summing the volumes of the tetrahedra generated in the triangulation. The result is stored in the variable V_value. This flag requires Igor Pro 7.00 or later.

Details

Triangulate3D implements Watson's algorithm for tetrahedralization of a set of points in three dimensions. It starts by creating a very large tetrahedron which inscribes all the data points followed by a sequential insertion of one datum at a time. With each new datum the algorithm finds the tetrahedron in which the datum falls. It then proceeds to subdivide the tetrahedron so that the datum becomes a vertex of new tetrahedra.

The algorithm suffers from two known problems. First, it may, due to numerical instabilities, result in tetrahedra that are too thin. You can get around this problem by introducing a slight random perturbation in the input wave. For example:

```
srcWave+=enoise(amp)
```

Here amp is chosen so that it is much smaller than the smallest cartesian distance between two input points.

The second problem has to do with memory allocations which may exhaust available memory for some pathological spatial distributions of data points. The operation reports both problems in the history area.

Examples

```
Make/O/N=(10,3) ddd=gnoise(5)     // create random 10 points in space
Triangulate3d/out=2 ddd

// now display the triangulation in Gizmo:
Window Gizmo0() : GizmoPlot
   PauseUpdate; Silent 1
```

```
    if(exists("NewGizmo")!=4)
        DoAlert 0, "Gizmo XOP must be installed"
        return
    endif
    NewGizmo/N=Gizmo0 /W=(309,44,642,373)
    ModifyGizmo startRecMacro
    ModifyGizmo scalingMode=2
    AppendToGizmo Scatter=root:ddd,name=scatter0
    ModifyGizmo ModifyObject=scatter0 property={ scatterColorType,0}
    ModifyGizmo ModifyObject=scatter0 property={ Shape,2}
    ModifyGizmo ModifyObject=scatter0 property={ size,0.2}
    ModifyGizmo ModifyObject=scatter0 property={ color,0,0,0,1}
    AppendToGizmo Path=root:M_TetraPath,name=path0
    ModifyGizmo ModifyObject=path0 property={ pathColor,0,0,1,1}
    ModifyGizmo setDisplayList=0, object=scatter0
    ModifyGizmo setDisplayList=1, object=path0
    ModifyGizmo autoscaling=1
    ModifyGizmo compile
    ModifyGizmo endRecMacro
End
```

References

Watson, D.F., Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes, *The Computer J., 24*, 167-172, 1981.

Further information about this algorithm can be found in:

Watson, D.F., *CONTOURING: A guide to the analysis and display of spatial data*, Pergamon Press, 1992.

See Also

The **Interpolate3D** operation and the **Interp3D** function.

TrimString

TrimString(*str* [, *simplifyInternalSpaces*])

The TrimString function returns a string identical to *str* except that leading and trailing whitespace characters are removed. The whitespace characters are space, tab, carriage-return and linefeed.

If the optional second parameter is non-zero, then each run of whitespace characters between words in *str* is "simplified" to a single space character.

TrimString was added in Igor Pro 7.00.

Examples

```
Print TrimString("   spaces   at  ends  ")        // Prints "spaces   at  ends"
Print TrimString("   spaces   at  ends  ", 1)     // Prints "spaces at ends"
```

See Also

SplitWave, **RemoveEnding**, **ReplaceString**

trunc

trunc(*num*)

The trunc function returns the integer closest to *num* in the direction of zero.

See Also

The **round**, **floor**, and **ceil** functions.

try

try

The try flow control keyword marks the beginning of the initial code block in a try-catch-entry flow control construct.

See Also

The **try-catch-endtry** flow control statement for details.

try-catch-endtry

```
try
   <code>
catch
   <code to handle abort>
endtry
```

A try-catch-endtry flow control statement provides a means for catching and responding to abort conditions in user functions. A programmatic abort is generated when the code executes Abort, AbortOnRTE or AbortOnValue. A user abort is generated when the user clicks the Abort button or presses the user abort key combination.

When code executes in the try-catch area, a programmatic abort immediately jumps to the code in the catch-endtry area rather than jumping to the end of the user function. A user abort jumps to the catch-entry area when a flow control keyword such as for or while executes or at the end of the try code. Normal flow (no aborts) skips all code within the catch-endtry area.

Details

During execution of code in the catch-endtry area, user aborts are suppressed. This means that, if the user attempts to abort procedure execution by pressing the **User Abort Key Combinations** or by clicking the Abort button, this will not abort the catch code itself.

When an abort occurs, information about the cause of the abort is returned via the `V_AbortCode` variable as follows:

-4:	Abort triggered by AbortOnRTE.
-3:	Abort caused by Abort operation.
-2:	Stack overflow abort.
-1:	User abort.
>=1:	Abort triggered by AbortOnValue.

See Also

Flow Control for Aborts on page IV-45 and **try-catch-endtry Flow Control** on page IV-45 for further details.

The **AbortOnRTE** and **AbortOnValue** keywords, and the **Abort** operation.

UInt64

`uint64` ***`localName`***

Declares a local unsigned 64-bit integer in a user-defined function or structure.

UInt64 is available in Igor Pro 7 and later. See **Integer Expressions** on page IV-36 for details.

See Also

Int, **Int64**

UniqueName

`UniqueName(`*`baseName`*`, `*`objectType`*`, `*`startSuffix`*`[, `*`windowNameStr`*`])`

The UniqueName function returns the concatenation of *baseName* and a number such that the result is not in conflict with any other object name.

windowNameStr is optional. If missing, it is taken to be the top graph, panel, layout, or notebook according to the value of *objectType*.

Details

baseName should be an unquoted name, such as you might receive from the user via a dialog or control panel.

objectType is one of the following:

1	Wave.	9	Control panel window.
2	Reserved.	10	Notebook window.
3	Numeric variable.	11	Data folder.
4	String variable.	12	Symbolic path.
5	XOP target window.	13	Picture.
6	Graph window.	14	Annotation in the named or topmost graph or layout.
7	Table window.	15	Control in the named or topmost graph or panel.
8	Layout window.	16	Notebook action character in the named or topmost notebook.

startSuffix is the number used as a starting point when generating the numeric suffix that makes the name unique. Normally you should pass zero for startSuffix. If you know that names of the form base0 through baseN are in use, you can make UniqueName run a bit faster by passing `N+1` as the *startSuffix*.

The *windowNameStr* argument is used only with *objectTypes* 14, 15, and 16. The returned name is unique only to the window (other windows might have objects with the same name). If a named window is given but does not exist, UniqueName returns *baseName startSuffix*. *windowNameStr* is ignored for other *objectTypes*

Examples

```
String uniqueWaveName = UniqueName(baseWaveName, 1, 0)
String uniqueControlName = UniqueName("ctrl", 15, 0, "Panel0")
```

See Also

CheckName and **CleanupName**.

UnPadString

`UnPadString(`*`str, padValue`*`)`

The UnPadString function undoes the action of PadString. It returns a string identical to *str* except that trailing bytes of *padValue* are removed.

See Also

PadString

UnsetEnvironmentVariable

`UnsetEnvironmentVariable(`*`varName`*`)`

The UnsetEnvironmentVariable function deletes the variable named *varName* from the environment of Igor's process, if it exists

The function returns 0 if it succeeds or a nonzero value if it fails.

The UnsetEnvironmentVariable function was added in Igor Pro 7.00.

Parameters

varName	The name of an environment variable which does not need to actually exist. It must not be an empty string and may not contain an equals sign (=).

Details

The environment of Igor's process is composed of a set of key=value pairs that are known as environment variables.

The environment of Igor's process is composed of a set of key=value pairs that are known as environment variables. Any child process created by calling ExecuteScriptText inherits the environment variables of Igor's process.

UnsetEnvironmentVariable changes the environment variables present in Igor's process and any future process created by ExecuteScriptText but does not affect any other processes already created.

On Windows, environment variable names are case-insensitive. On other platforms, they are case-sensitive.

Examples

```
Variable result
result = SetEnvironmentVariable("SOME_VARIABLE", "15")       // Sets the variable
result = UnsetEnvironmentVariable("SOME_VARIABLE")           // Unsets the variable
```

See Also

GetEnvironmentVariable, **SetEnvironmentVariable**

Unwrap

`Unwrap` ***`modulus, waveName`*** `[`***`, waveName`***`]…`

The Unwrap operation scans through each named wave trying to undo the effect of a modulus operation.

Parameters

modulus is the value applied to the named waves through the **mod** function as if the command:

```
waveName = mod(waveName,modulus)
```

had been executed. It is this calculation which Unwrap attempts to undo.

Details

The unwrap operation works with 1D waves only. See **ImageUnwrapPhase** for phase unwrapping in two dimensions.

Examples

If you perform an FFT on a wave, the result is a complex wave in rectangular coordinates. You can create a real wave that contains the phase of the result of the FFT with the command:

```
wave2 = imag(r2polar(wave1))
```

However, the rectangular to polar conversion leaves the phase information modulo 2π. You can restore the phase information with the command:

```
Unwrap 2*pi, wave2
```

Because the first point of a wave that has been FFTed has no phase information, in this example you would *precede* the Unwrap command with the command:

```
wave2[0] = wave2[1]
```

See Also

The **ImageUnwrapPhase** operation and **mod** function.

UpperStr

`UpperStr(`*`str`*`)`

The UpperStr function returns a string expression in which all lower-case ASCII characters in *str* are converted to upper-case.

See Also

The **LowerStr** function.

URLDecode

`URLDecode(`*`inputStr`*`)`

The URLDecode function returns a percent-decoded copy of the percent-encoded string *inputStr*. It is unlikely that you will need to use this function; it is provided for completeness.

For an explanation of percent-encoding, see **Percent Encoding** on page IV-253.

Example

```
String theURL = "http://google.com?key1=35%25%20larger"
theURL = URLDecode(theURL)
Print theURL
  http://google.com?key1=35% larger
```

See Also

URLEncode, **URLRequest**, **URLs** on page IV-252.

URLEncode

`URLEncode(`*`inputStr`*`)`

The URLEncode function returns a percent-encoded copy of *inputStr*.

Percent-encoding is useful when encoding the query part of a URL or when the URL contains special characters that might otherwise be misinterpreted by a web server. For an explanation of percent-encoding, see **Percent Encoding** on page IV-253.

Example

```
String baseURL = "http://google.com"
String key1 = "key1"
String value1 = URLEncode("35% larger")
String theURL = ""
sprintf theURL, "%s?%s=%s", baseURL, key1, value1
Print theURL
  http://google.com?key1=35%25%20larger
```

See Also

URLDecode, **URLRequest**, **URLs** on page IV-252.

URLRequest

`URLRequest [ flags ] url=urlStr [method=methodName, headers=headersStr]`

The URLRequest operation connects to a URL using the specified method and optionally stores the response from the server. *urlStr* can point to a remote server or to a local file. The server's response is stored in the S_serverResponse output variable or in a file if you use the /FILE flag.

The URLRequest operation was added in Igor Pro 7.00.

Keywords

The url=*urlStr* keyword is require. All others are optional.

url=*urlStr*	A string containing the URL to retrieve. See **URLs** on page IV-252 for details.
method=*methodName*	Specifies which method to use for the request. The get method is used by default.

This table shows the valid methods for each supported scheme. Not all servers support all of the listed methods.

Scheme	Supported Methods
http, https	get, post, put, head, delete
ftp	get, put
file	get, put

Because *methodName* is a name, not a string, you must not enclose it in quotes.

Before using the post method, you should read The **The HTTP POST Method** on page V-915 so that you know how to use the optional headers parameter.

If you use the head method, URLRequest sets the S_serverResponse output variable to "" because only the headers are retrieved. As with other methods, the headers are stored in the S_headers output variable.

headers=*headersStr* Specifies a string containing additional or replacement headers to use with the request. This parameter is ignored unless the scheme is http or https.

The headers parameter is provided primarily for use with the post method when making HTTP requests and is ignored for schemes other than http/https. The header consists of a colon-separated key:value pair (though see the next paragraph for an exception). Pairs must be separated by a carriage return (\r) character.

Certain standard headers (such as Content-Type and User-Agent) may automatically be set when making the request. You can override those default values by using this keyword and setting a different value. If you add a header with no content, as in "Accept:" (there is no data on the right side of the colon), the internally used header is disabled. To actually add a header with no content, use the form "MyHeader;" (note the trailing semicolon).

Any headers specified with this keyword are sent only to the http server, not to the proxy server, if one is in use.

See **The HTTP POST Method** on page V-915 for a detailed explanation and examples.

Flags

/AUTH={*username*, *password* }

Uses the specified *username* and *password* string parameters for authentication. Values provided here override any username and/or password provided as part of the URL. To specify a username but not a password, pass "" for the *password* parameter.

Note: See **Safe Handling of Passwords** on page IV-254 for more information on how to use URLRequest to prevent authentication information, such as passwords, from being accidentally revealed.

/DFIL=*dataFileNameStr* Specifies a file name to use as a source of data. Typically this flag is used only with the put or post methods, but it is accepted with all methods. Unless *dataFileNameStr* is a full path, the /P flag must also be used. When using the post or put methods, one and only one of the /DFIL or /DSTR flags must be used.

When you use /DFIL and the method is anything other than put, the "Content-Type: application/x-www-form-urlencoded" header is automatically added. If necessary, you can override this behavior using the headers keyword. See **The HTTP POST Method** on page V-915 for more information.

/DSTR=*dataStr* Specifies the string to use as a source of data. Typically you use this flag only with the put or post methods, but it is accepted with all methods. When using the post or put methods, one and only one of the /DFIL or /DSTR flags must be used.

When /DSTR is used and the method is anything other than put, the "Content-Type: application/x-www-form-urlencoded" header is automatically added. If necessary, you can override this behavior using the headers keyword. See **The HTTP POST Method** on page V-915 for more information.

/FILE=*destFileNameStr*	If present, URLRequest saves the server's response in a file instead of in the S_serverResponse output variable. URLRequest ignores /FILE if you include the /IGN flag and the *ignoreResponse* parameter is not 0. *destFileNameStr* can be a full path to the file, in which case /P=*pathName* is not needed, a partial path relative to the folder associated with pathName, or the name of a file in the folder associated with pathName. If the file already exists, URLRequest returns an error unless you include the /O flag. If you include /O and the file already exists, the existing file is overwritten. This happens even in the event of an empty response or transfer error. You should consider using the /FILE flag when you are expecting the server to return a large amount of data, such as when downloading a file.
/IGN[=*ignoreResponse*]	Ignore and do not store the server's response to the request. /IGN alone has the same effect as /IGN=1. If ignore is turned on, URLRequest sets the S_serverResponse output variable to "", regardless of whether the server responded to the request or not. If you include the /FILE flag, URLRequest does not create the output file and sets S_fileName is set to "". This flag is useful only when your goal is to establish a connection with a server and the server's response is not important. All error message codes and strings are still set when ignore is on. /IGN=0: Same as no /IGN. /IGN=1: Ignore server response completely. S_headers is set to "". /IGN=2: Ignore server response but capture the headers of the response. S_serverResponse is set to "" but S_headers will contain the headers.
/NRED=*maxNumRedirects*	Specifies the maximum number of redirects that are allowed. A redirect means that when a certain URL is requested, the server responds telling the client to try a different URL. Most web browsers automatically follow server redirects, up to a certain limit. For security purposes, it is sometimes useful to prevent redirects from being followed at all. *maxNumRedirects* is a number between -1 and 1000. To allow infinite redirection, set maxNumRedirects to -1. If you omit the /NRED flag, a moderate value (currently 20, but subject to change in the future) is used.
/O	Specifies that the file is to be overwritten when you use the /FILE flag and the output file already exists. If you omit /O and the file exists, URLRequest returns an error.
/P=*pathName*	Specifies the folder to use for the output file, specified by the /FILE flag, and/or the source data file, specified by the /DFIL flag. pathName is the name of an existing Igor symbolic path. The /P flag affects both the /FILE and /DFIL flags. If you use both flags and want to use different directories, you must provide a full path for one or both of the /FILE and /DFIL flag parameters. If the /P flag is used without one or both of the /FILE and /DFIL flags, it is ignored.

/PROX[={*proxyURLStr*, *proxyUserNameStr*, *proxyPassStr*, *proxyOptions*}]

NOTE: This flag is experimental and has not been extensively tested. The behavior of this flag may change in the future, or it may be eliminated entirely.

Designates a proxy server to be used when making the connection. *proxyURLStr* is either the host name or IP address of the proxy server, or a full URL. If a full URL is used, the scheme specifies which kind of proxy is used. Typically the scheme for a proxy server should be either http or socks5. If no scheme is provided, http is assumed. See **URLs** on page IV-252 for more information.

If the proxy server does not require authentication, or if the username and password for the proxy server are specified in proxyURLStr, the optional proxyUserNameStr and proxyPassStr parameters do not need to be provided. The following two examples do the same thing:

```
/PROX={"http://proxy.example.com:800"}
/PROX={"http://proxy.example.com:800", "", ""}
```

As with other URLs, you can also provide the username and password as part of the URL itself. The following two examples do the same thing:

```
/PROX={"http://proxy.example.com:800", "user", "pass"}
/PROX={"http://user:pass@proxy.example.com:800"}
```

proxyOptions is optional and for future use. It is currently ignored. If provided, you must set it to 0.

If you use the /PROX flag without any parameters, Igor attempts to get proxy information from the operating system's proxy configuration information. If Igor cannot get any proxy server information from the system, no proxy server is used.

See **Safe Handling of Passwords** on page IV-254 for more information on how to use URLRequest to prevent authentication information, such as passwords, from being accidentally revealed.

/TIME=*timeoutSeconds*

Forces the operation to time out after timeoutSeconds seconds if it has not completed by that time. If this flag is not provided, URLRequest runs until the server has finished sending and receiving data. For simple requests a value of a few seconds is appropriate. However, because uploading and/or downloading large amounts of data may take a long time, if *timeoutSeconds* is too small the request might be prematurely terminated. If you omit the /TIME flag, or if *timeoutSeconds* = 0, there is no timeout.

Regardless of whether or not you include /TIME, you can abort the operation by pressing the **User Abort Key Combinations**.

/V=*diagnosticMode*

This flag is useful only when your goal is to establish a connection with a server and the server's response is not important. All error message codes Controls diagnostic messages printed in the history area of the command window.

/V=0:	Do not print any diagnostic messages. This is the default if you omit /V.
/V=1:	Prints an error message if a run time error occurs.
/V=2:	Prints full debugging information.

/Z[=z]

Suppress error generation. Use this if you want to handle errors yourself.

/Z=0:	Do not suppress errors. This is the default if /Z is omitted.
/Z=1:	Any errors generated by Igor do not stop execution. If there is an error the error code is stored in V_Flag.

/Z alone has the same effect as /Z=1.

Output Variables

URLRequest sets the following output variables:

`V_flag`	V_flag is zero if the operation succeeds without an error or a non-zero Igor error code if it fails. Note that "succeeds without an error" does not necessarily mean that the operation performed as you intended. For example, if you attempt to connect to a server that requires a username and password for authentication but you do not provide this information, V_flag is likely to be 0, indicating no error. You need to inspect the value of V_responseCode to ensure that it is what you expect.
`V_responseCode`	Contains the status code provided by the server. V_responseCode is valid only if V_flag is 0, meaning that no error occured. If V_flag is nonzero, an error occured and V_responseCode will be 0. Different schemes use different sets of status codes. A list of http/https status codes and their definitions can be found at: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html A list of FTP status codes can be found at: https://en.wikipedia.org/wiki/List_of_FTP_server_return_codes
`S_serverResponse`	Contains the server's response to the request. S_serverResponse is set to "" if you use the /FILE flag or the /IGN=1 or /IGN=2 flags. We recommend that you use the /FILE flag when you expect the size of the server response to be large, such as when downloading a large file.
`S_headers`	Contains a list of all of the headers received as part of the response. Header names are separated from their value by a colon and pairs are separated by a carriage return followed by a line feed (\r\n). For schemes other than http and https, S_headers is set to "".
`S_fileName`	Contains the full Igor-style path to the output file if the /FILE flag was used. If there was an error saving to the specified file, or another kind of error, or if the /FILE flag was not used, S_fileName is set to "".

Basic Examples

```
// Retrieve the contents of the WaveMetrics home page.
// Output is stored in S_serverResponse
URLRequest url="http://www.wavemetrics.com"

// Download the Windows Igor6 installer and save it as a file on the desktop.
NewPath/O Desktop, SpecialDirPath("Desktop", 0, 0, 0)
String theURL = "http://www.wavemetrics.net/Downloads/Win/setupIgor6.exe"
URLRequest/FILE="setupIgor6.exe"/P=Desktop url=theURL

// Download a file to the desktop from an FTP server.
NewPath/O Desktop, SpecialDirPath("Desktop", 0, 0, 0)
String theURL = "ftp://ftp.wavemetrics.com/test/test.htm"
URLRequest/O/FILE="test.htm"/P=Desktop url=theURL

// Upload the same file to the FTP server.
theURL = "ftp://user:pass@ftp.wavemetrics.com/test.htm"
URLRequest/DFIL="test.htm"/P=Desktop method=put, url=theURL
```

Using the File Scheme

URLRequest supports the file scheme in URLs. You must provide the full path to the file as a native Windows-style or UNIX-style path, depending on which platform the code is running.

You must specify the schema as "file:///". Note that you must use three front slash characters (/) between the colon and the full file path (for most URLs you would only use two slashes).

```
URLRequest url="file:///C:\\Data\\Trial1\\control.ibw"      // Works on Windows only
URLRequest url="file:///Users/bob/Data/Trial1/control.ibw"  // Works on Macintosh only
URLRequest url="file:///C:\\Data\\Trial1\\control.ibw"      // Doesn't work
URLRequest url="file:///Users/bob/Data/Trial1/control.ibw"  // Doesn't work
```

The following example shows how to convert a full Igor file path to a native path suitable for use as a file URL:

```
String nativeFilePath
#ifdef WINDOWS
    String igorFilePath = "C:Documents:myFile.txt"
    nativeFilePath = ParseFilePath(5, igorFilePath, "*", 0, 0)
#endif
#ifdef MACINTOSH
    String igorFilePath = "MacintoshHD:Documents:myFile.txt"
    nativeFilePath = ParseFilePath(5, igorFilePath, "/", 0, 0)
#endif
URLRequest url="file:///" + nativeFilePath
```

The HTTP POST Method

Like the HTTP GET method, the HTTP POST method can be used to transmit information to a web server. When using the GET method, all information must be contained within the URL itself. As an example, making a GET request to the URL <http://www.google.com/search?q=Igor+Pro+WaveMetrics> searches Google using the keywords "Igor", "Pro", and "WaveMetrics".

Unlike the GET method, the POST method allows the client to send information to the server that is not contained within the URL itself. This is necessary in many situations, such as when uploading a file or transfering a large amount of data. In addition, because the data contained in POST requests is typically not stored in the log files of web servers, it is more appropriate for sending data that is secure, such as login credentials and form submissions which might contain sensitive information.

When using the POST method, the client must encode its data using one of several methods and must tell the server what type of data is being sent and how it is encoded. The client does this by setting the Content-Type header that is part of the request. The most commonly used content type is "application/x-www-form-urlencoded". Unless you provide your own Content-Type header using the optional headers parameter, URLRequest will set this header for you. This is true regardless of which data source (/DSTR for string or /DFIL for file) you use for the POST.

Here is a simple example that uses the POST method with URL encoded data (the default Content-Type).

```
String theURL = "http://www.example.com/process.php"
String nameData = URLEncode("name") + "=" + URLEncode("Dan P. Ikes, Jr.")
String address = "650 E. Chicago Ave."
String addressData = URLEncode("address") + "=" + URLEncode(address)
String postData = nameData + "&" + addressData
URLRequest/DSTR=postData method=post, url=theURL
```

There are two things to note in this example.

First, both the key name and value string are passed through **URLEncode** so that any special characters can be percent-encoded.

Second, keys and values are separated by an equal sign ("=") and key/value pairs are separated by an ampersand ("&"). These characters are not passed through URLEncode because doing so would cause them to lose their meaning as special characters.

For more information on using the application/x-www-form-urlencoded content type, see http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1

It is also possible to transmit more complicated data using the POST method, such as simulating a form that contains regular text fields as well as a file upload field. To do so you must set the Content-Type header using the headers parameter. You must also build your post data using a multi-part header. Here is an example:

```
String theURL = "http://www.example.com/process.php"
String postData = ""
// Note: The boundary is arbitrary. It needs to be
// unique enough that it is not contained within any
// of the actual data being transmitted in the post.
String boundary = "AaBbCcDd0987"

// name
postData += "--" + boundary + "\r\n"// beginning of this part
postData += "Content-Disposition: form-data; name=\"name\"\r\n"
postData += "\r\n"
postData += "Dan P. Ikes, Jr.\r\n"

// address
postData += "--" + boundary + "\r\n"
postData += "Content-Disposition: form-data; name=\"address\"\r\n"
postData += "\r\n"
postData += "650 E. Chicago Ave.\r\n"

// file
// Open the file we plan to send and store the binary
// contents of the file in a string.
Variable refNum
Open/R/Z/P=Igor refNum as "ReadMe.ihf"
FStatus refNum
Variable fileSize = V_logEOF
String fileContents = ""
fileContents = PadString(fileContents, fileSize, 0)
FBinRead refNum, fileContents
Close refNum
postData += "--" + boundary + "\r\n"
postData += "Content-Disposition: form-data; name=\"file\"; filename=\"ReadMe.ihf\"\r\n"
postData += "Content-Type: application/octet-stream\r\n"
postData += "Content-Transfer-Encoding: binary\r\n"
postData += "\r\n"
postData += fileContents + "\r\n"

postData += "--" + boundary + "--\r\n"      // end of this part

String headers
headers = "Content-Type: multipart/form-data; boundary=" + boundary
URLRequest/DSTR=postData method=post, url=theURL, headers=headers
```

For more information on using the multipart/form-data content type, see http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.2

See Also

URLEncode, **URLDecode**, **FetchURL**, **BrowseURL**

Network Communication on page IV-252, **URLs** on page IV-252

ValDisplay

`ValDisplay` [**`/Z`**] ***`ctrlName`*** [***`keyword = value`*** [, ***`keyword = value`*** …]]

The ValDisplay operation creates or modifies the named control that displays a numeric value in the target window. The appearance of the control varies; see the **Examples** section.

For information about the state or status of the control, use the **ControlInfo** operation.

Parameters

ctrlName is the name of the ValDisplay control to be created or changed.

The following keyword=value parameters are supported:

appearance={*kind* [, *platform*]}

	Sets the appearance of the control. *platform* is optional. Both parameters are names, not strings. *kind* can be one of `default`, `native`, or `os9`. *platform* can be one of `Mac`, `Win`, or `All`. See **Button** and **DefaultGUIControls** for more appearance details.
barBackColor=(*r*,*g*,*b*)	Sets the background color under the bar (if any). *r*, *g*, and *b* range from 0 to 65535.
barBackColor=0	Sets the background color under the bar to the default color, the standard document background color used on the current operating system, which is usually white.

barmisc={*lts*, *valwidth*}	Sets the “limits text size” and the size of the type showing the bar limits. If *lts* is zero, the bar limits are not displayed. Otherwise, *lts* must be between 5 and 100. *valwidth* is the “value readout width”. It claims the amount of horizontal space for the numeric part of the display. If *valwidth* equals or exceeds the control width available to it, the numeric readout uses all the room, and prevents display of any bar. If *valwidth* is zero, there is no numeric readout, and only the bar is displayed. *valwidth* can range from zero to 4000, and it defaults to 1000 (which usually leaves no room for the display bar).
bodyWidth=*width*	Specifies an explicit size for the body (nontitle) portion of a ValDisplay control. By default (bodyWidth=0), the body portion is the amount left over from the specified control width after providing space for the current text of the title portion. If the font, font size, or text of the title changes, then the body portion may grow or shrink. If you supply a bodyWidth>0, then the body is fixed at the size you specify regardless of the body text. This makes it easier to keep a set of controls right aligned when experiments are transferred between Macintosh and Windows, or when the default font is changed.
disable=*d*	Sets user editability of the control. *d*=0: Normal. *d*=1: Hide. *d*=2: Disable user input. Does not change control appearance because it is read-only. *d*=3: Hide and disable the control. This is useful to disable a control that is also hidden because it is in a hidden tab.
fColor=(*r*,*g*,*b*)	Sets the initial color of the title. *r*, *g*, and *b* range from 0 to 65535. fColor defaults to black (0,0,0). To further change the color of the title text, use escape sequences as described for title=*titleStr*.
font="*fontName*"	Sets the font used to display the value of the variable, e.g., `font="Helvetica"`
format=*formatStr*	Sets the numeric format of the displayed value. The default format is "%g". For a description of *formatStr*, see the **printf** operation.
frame=*f*	Sets frame style: *f*=0: Value is unframed. *f*=1: Default; value is framed (same as *f*=3). *f*=2: Simple box. *f*=3: 3D sunken frame. *f*=4: 3D raised frame. *f*=5: Text well.
fsize=*s*	Sets the size of the type used to display the value in the numeric readout. The default is 12 points.
fstyle=*fs*	*fs* is a bitwise parameter with each bit controlling one aspect of the font style as follows: Bit 0: Bold Bit 1: Italic Bit 2: Underline Bit 4: Strikethrough See **Setting Bit Parameters** on page IV-12 for details about bit settings.

help={*helpStr*}	Sets the help for the control. The help text is limited to a total of 255 bytes. On Macintosh, help appears when you turn Igor Tips on. On Windows, help for the first 127 bytes or up to the first line break appears in the status line. If you press F1 while the cursor is over the control, you will see the entire help text. You can insert a line break by putting "\r" in a quoted string.
highColor=(*r*,*g*,*b*)	Specifies the bar color when the value is greater than *base* in the limits keyword. *r*, *g*, and *b* are integers from 0 to 65535.
labelBack=(*r*,*g*,*b*) or 0	Specifies the background fill color for labels. *r*, *g*, and *b* are integers from 0 to 65535. The default is 0, which uses the window's background color.
limits={*low*,*high*,*base*}	Controls how the value is translated into a graphical representation when the display includes a bar (described fully in **Details**). Defaults are {0,0,0}, which aren't too useful.
limitsColor=(*r*,*g*,*b*)	Sets the color of the limits text, if any. *r*, *g*, and *b* range from 0 to 65535. limitsColor defaults to black (0,0,0).
limitsBackColor=(*r*,*g*,*b*)	Sets the background color under the limits text. *r*, *g*, and *b* range from 0 to 65535.
limitsBackColor=0	Sets the background color under the limits text to the default color, the standard document background color used on the current operating system, which is usually white.
lowColor=(*r*,*g*,*b*)	Specifies the bar color when the value is less than *base* in the limits keyword. *r*, *g*, and *b* are integers from 0 to 65535.
mode=*m*	Specifies the type of LED display to use, if any. *m*=0: Bar mode (default). *m*=1: Oval LED. *m*=2: Rectangular LED. *m*=3: Bar mode with no fractional part. *m*=4: Candy-stripe effect for the bar area to support indefinite-style progress windows. The value is taken to be the phase of the candy stripe. When using value= _NUM:n, n is taken as an increment value so you would normally just use 1. Uses the native platform appearance if the high and low colors are left as default. Note native formats may not fill vertical space. See **Progress Windows** on page IV-144 for an example.
pos={*left*,*top*}	Sets the position of the display in pixels, from 0 to 32767.
pos+={*dx*,*dy*}	Offsets the position of the display in pixels.
rename=*newName*	Gives the ValDisplay control a new name.
size={*width*,*height*}	Sets width and height of display in pixels. *width* can range from 10 to 200 pixels, *height* from 5 to 200 pixels. Default width is 50, default height is determined by the numeric readout font size.
title=*titleStr*	Sets title of display to the specified string expression. The title appears to the left of the display. If this title is too long, it won't leave enough room to display the bar or even the numeric readout! Defaults to "" (no title). Using escape codes you can change the font, size, style, and color of the title. See **Annotation Escape Codes** on page III-53 or details.
value=*valExpr*	Displays the numeric expression *valExpr*. It is *not* a string. As of version 6.1, you can use the syntax _NUM:num to specify a numeric value without using a dependency.
valueColor=(*r*,*g*,*b*)	Sets the color of the value readout text, if any. *r*, *g*, and *b* range from 0 to 65535. valueColor defaults to black (0,0,0).

valueBackColor=(*r*,*g*,*b*)	Sets the background color under the value readout text.*r*, *g*, and *b* range from 0 to 65535.
valueBackColor=0	Sets the background color under the value readout text to the default color, the standard document background color used on the current operating system, which is usually white.
win=*winName*	Specifies which window or subwindow contains the named control. If not given, then the top-most graph or panel window or subwindow is assumed. When identifying a subwindow with *winName*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.
zeroColor=(*r*,*g*,*b*)	Governs the LED color (in LED mode only). *r*, *g*, and *b* are integers from 0 to 65535. Used in conjunction with the limits keyword such that zeroColor determines one endpoint color when base is between *low* and *high*, or LED color when the value is less than *low*.

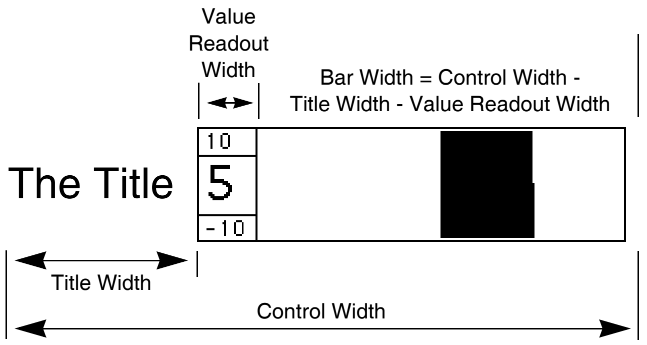
Flags

/Z	No error reporting.

Details

The target window must be a graph or panel.

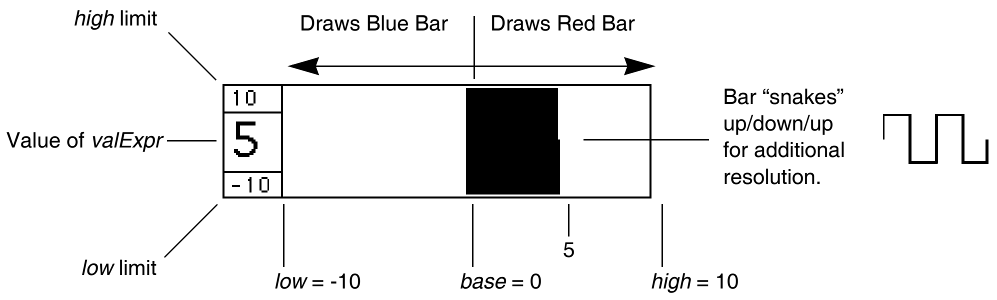
The appearance of the ValDisplay control depends primarily on the *width* and *valwidth* parameters and the width of the title. Space for the individual elements is allocated from left to right, with the title receiving first priority. If the control width hasn't all been used by the title, then the value display gets either *valwidth* pixels of room, or what is left. If the control width hasn't been used up, the bar is displayed in the remaining control width:


Value
Readout
Width
Bar Width = Control Width -
Title Width - Value Readout Width
10
The Title
5
-10
Title Width
Control Width

If you use the bodyWidth keyword, the value readout width and bar width occupy the body width. The total control width is then bodyWidth+title width, and the width from the size keyword is ignored.

The limits values *low*, *high,* and *base* and the value of *valExpr* control how the bar, if any, is drawn. The bar is drawn from a starting position corresponding to the *base* value to an ending position determined by the value of *valExpr*, *low* and *high*. *low* corresponds to the left side of the bar, and *high* corresponds to the right. The position that corresponds to the *base* value is linearly interpolated between *low* and *high*.

For example, with *low*= -10, *high*=10, and *base*= 0, a *valExpr* value of 5 will draw from the center of the bar area (0 is centered between -10 and 10) to the right, halfway from the center to the right of the bar area (5 is halfway from 0 to 10):



The *valExpr* must be executable at any time. The expression is stored and executed when the ValDisplay needs to be updated. However, execution will occur outside the routine that creates the ValDisplay, so you must not use local variables in the expression.

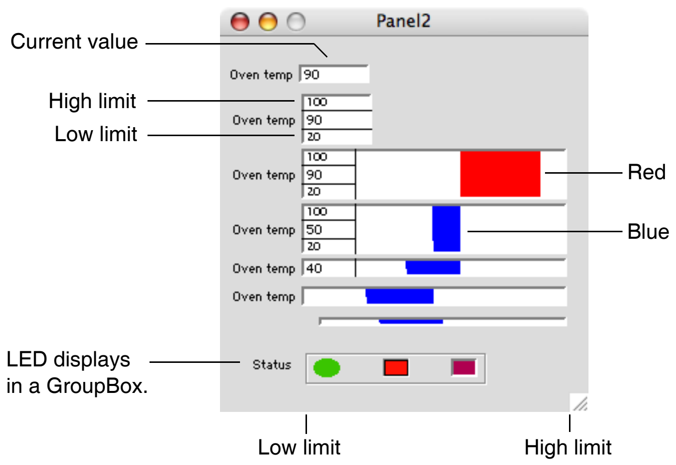
valExpr may be enclosed in quotes and preceded with a # character (see **When Dependencies are Updated** on page IV-217) to defer evaluation of the validity of the numeric expression, which may be needed if the expression references as-yet-nonexistent global variables or user-defined functions:

```
ValDisplay valdisp0 value=notAVar*2      // "unknown name or symbol" error
ValDisplay valdisp0 value=#"notAVar*2"      // still not valid, no error
Variable notAVar=3                       // now valid; ValDisplay works
```

In a ValDisplay, the #"" syntax permits use of a string expression. Normally, the # prefix signifies that the following text must be a literal quoted string. String expressions are evaluated at runtime to obtain the final expression for the ValDisplay. In other words, there is a level of indirection.

Examples

Here is a sampling of the various types of ValDisplay controls available:



You can use a ValDisplay to replace the bar mode with a solid color fill designed to look like an LED. Use the mode keyword with mode=1 to create an oval LED or mode=2 to create a rectangular LED. You can specify different frames with the rectangular LED but only a simple frame is available for the oval mode. Use mode=0 to revert to bar mode.

The color and brightness of the LED depends on the value that the ValDisplay is monitoring combined with the limits={*low*, *high*, *base*) setting, the two color settings used in bar mode along with a third color (zeroColor) that is used only in LED mode. When the value is between *low* and *high*, the color is a linear combination of endpoint colors. If *base* is between *low* and *high*, the endpoint colors are the low color and the zero color, or the zero color and the high color. For values outside the limits, the appropriate limiting color is chosen.

If *base* is less than the *low*, the endpoint colors are the low color and the high color. In this case, if the value is less than *low* the LED takes on the zero color.

You should use the bodyWidth setting in conjunction with LED mode to keep the LED from dramatically changing size or disappearing when the title is changed or if your experiment is moved to a different platform (Macintosh vs PC).

Try the ValDisplay Demo example experiment to see these different modes in action. The experiment file is in your Igor Pro 7 folder, in the “Examples:Feature Demos” subfolder.

See Also

See **Creating ValDisplay Controls** on page III-385 for more examples.

The **ControlInfo** operation for information about the control. Chapter III-14, **Controls and Control Panels**, for details about control panels and controls. The **GetUserData** operation for retrieving named user data.

printf for an explanation of *formatStr*.

Progress Windows on page IV-144 for an example of candy-stripe mode=4.

Variable

`Variable` `[`***`flags`***`]` ***`varName`*** `[=`***`numExpr`***`][,` ***`varName`*** `[=`***`numExpr`***`]]…`

The Variable operation creates real or complex variables and gives them the specified name.

Flags

/C	Declares a complex variable.
/D	Obsolete, included only for backward compatibility (see **Details**).
/G	Creates a variable with global scope and overwrites any existing variable.

Details

The variable is initialized when it is created if you supply the initial value. However, when Variable is used to declare a function parameter, it is an error to attempt to initialize it.

You can create more than one variable at a time by separating the names and optional initializers for multiple variables with a comma.

Numeric variables are double precision. In ancient times, variables could be single or double precision and the /D flag meant double precision. The /D flag is allowed for backward compatibility but is no longer needed and should not be used in new code.

If used in a macro or function the new variable is local to that macro or function unless the /G (global) flag is used. If used on the command line, the new variable is global.

varName can include a data folder path.

Examples

To initialize a complex variable, use the **cmplx** function. For example:

```
Variable/C cv1 = cmplx(1,2)
```

sets the real part of cv1 to 1 and the imaginary part to 2.

See Also

Numeric Variables on page II-96

Variance

`Variance(`*`inWave`*` [, `*`x1`*`, `*`x2`*` ] )`

Returns the variance of the real-valued *inWave*. The function ignores NaN and INF values in *inWave*.

Parameters

inWave is expected to be a real-valued numeric wave. If *inWave* is a complex or text wave, Variance returns NaN.

x1 and *x2* specify a range in *inWave* over which the variance is to be calculated. They are used only to locate the points nearest to x=*x1* and x=*x2* . The variance is then calculated over that range of points. The order of *x1* and *x2* is immaterial.

If omitted, *x1* and *x2* default to -∞ and +∞ respectively and the variance is calculated for the entire wave.

Details

The variance is defined by

$$\mathrm{var} = \frac{\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2}{n-1}$$

where

$$\overline{x} = \frac{\sum_{i=1}^{n} X_i}{n}.$$

Examples

```
Make/O/N=5 test = p
SetScale/P x, 0, .1, test

// Print variance of entire wave
Print Variance(test)

// Print variance from x=0 to x=.2
Print Variance(test, 0, .2)

// Print variance for points 1 through 3
Variable x1=pnt2x(test, 1)
Variable x2=pnt2x(test, 3)
Print Variance(test, x1, x2)
```

See Also

mean, **median**, **WaveStats**

VariableList

`VariableList(`*`matchStr`*`, `*`separatorStr`*`, `*`variableTypeCode`*`)`

The VariableList function returns a string containing a list of global variables selected based on the *matchStr* and *variableTypeCode* parameters. The variables listed are all in the current data folder.

Details

For a variable name to appear in the output string, it must match *matchStr* and also must fit the requirements of *variableTypeCode*. The first character of *separatorStr* is appended to each variable name as the output string is generated.

The name of each variable is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all variable names.
"xyz"	Matches variable name xyz only.
"*xyz"	Matches variable names which end with xyz.
"xyz*"	Matches variable names which begin with xyz.
"*xyz*"	Matches variable names which contain xyz.
"abc*xyz"	Matches variable names which begin with abc and end with xyz.

matchStr may begin with the ! character to return windows that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches variable names which *do not* end with xyz.

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

variableTypeCode is used to further qualify the variable. The variable name goes into the output string only if it passes the match test and its type is compatible with *variableTypeCode*. *variableTypeCode* is any one of:

2:	System variables (K0, K1 . . .)
4:	Scalar variables
5:	Complex variables

Examples

`VariableList("*",";",4)`	Returns a list of all scalar variables.
`VariableList("!V_*", ";",5)`	Returns a list of all complex variables except those whose names begin with “V_”.

See Also

See the **StringList** and **WaveList** functions.

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

vcsr

`vcsr(`*`cursorName`*` [, `*`graphNameStr`*`])`

The vcsr function returns the Y (vertical) value of the point which the specified cursor (A through J) is attached to in the top (or named) graph.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The result is computed from the coordinate system of the graph’s Y axis. The Y axis used is the one used to display the wave on which the cursor is placed.

See Also

The **hcsr**, **pcsr**, **qcsr**, **xcsr**, and **zcsr** functions.

Programming With Cursors on page II-249.

version

`#pragma version = `*`versNum`*

In the File Information dialog, `#pragma version=`*versNum* provides file version information that is displayed next to the file name in the dialog. This line must not be indented and must appear in the first fifty lines of the file. See **Procedure File Version Information** on page IV-155.

See Also

The **The version Pragma** on page IV-50, **Procedure File Version Information** on page IV-155, the **IgorInfo** function, and #**pragma**.

VoigtFunc

`VoigtFunc(X,Y)`

Computes the Voigt function using an approximation giving an accuracy better than one part in 10^5 over large range of input parameters.

VoigtFunc returns values from a normalized Voigt peak centered at X=0 for the given value of X. The X input is a normalized distance from the peak center:

$$X = \sqrt{\ln(2)}\,\frac{\nu - \nu_0}{\gamma_g}$$

where γ_g is the Gaussian component half-width, and $\nu - \nu_0$ is the distance from the peak center.

The parameter Y is the shape parameter: when Y is zero, the peak is pure Gaussian. When Y approaches infinity, the shape is pure Lorentzian. When Y is sqrt(ln(2)), the ratio of the Lorentzian and Gaussian half-widths is one.

VoigtFunc was added in Igor Pro 7.00.

Details

The VoigtFunc function returns values from a normalized peak that can be used as the basis for user-defined fitting functions. A typical fitting function might look like this:

```
Function VoigtFit(w,xx) : FitFunc
   Wave w
   Variable xx

   //CurveFitDialog/ These comments were created by the Curve Fitting dialog.
   //CurveFitDialog/ Equation:
   //CurveFitDialog/   f(xx) = Y0+Amp*VoigtFunc(width*(xx-x0),shape)
   //CurveFitDialog/ End of Equation
   //CurveFitDialog/ Independent Variables 1
   //CurveFitDialog/ xx
   //CurveFitDialog/ Coefficients 5
   //CurveFitDialog/ w[0] = Y0
   //CurveFitDialog/ w[1] = Amp
   //CurveFitDialog/ w[2] = width
   //CurveFitDialog/ w[3] = x0
   //CurveFitDialog/ w[4] = shape

   return w[0]+w[1]*VoigtFunc(w[2]*(xx-w[3]),w[4])
End
```

Parameter w[0] sets the vertical offset, w[1] sets the amplitude, w[2] affects the width, w[3] sets the location of the peak and w[4] adjusts the shape (but also affects the amplitude). Note that w[2] cannot be taken directly to be width, as the shape parameter also affects the width.

After the fit, you can use the returned coefficients to calculate the area (a) along with the half width at half max for the Gaussian (wg), Lorentzian (wl) and the Voigt (wv). Assuming the coefficient wave is named w_coef:

```
a = w_coef[1]*sqrt(pi)/w_coef[2]
wg = sqrt(ln(2))/w_coef[2]
wl = w_coef[4]/w_coef[2]
wv = wl/2 + sqrt( wl^2/4 + wg^2)
```

References

The approximation is described here:

Wells, R.J., Rapid Approximation to the Voigt/Faddeeva Function and Its Derivatives, *Journal of Quantitative Sprectroscopy and Radiative Transfer*, 1999.

WAVE

WAVE [**/C**][**/T**][**/Z**] ***localName*** [**=*pathToWave***][**,** ***localName1*** [**=*pathToWave1***]]…

WAVE is a declaration that identifies the nature of a user-defined function parameter or creates a local reference to a wave accessed in the body of a user-defined function.

The optional parameter *pathToWave* is used only in the body of a function, not in a parameter declaration.

The WAVE reference is required when you use a wave in an assignment statement in a function. At compile time, the WAVE statement specifies that the local name references a wave. At runtime, it makes the connection between the local name and the actual wave. For this connection to be made, the wave must exist when the WAVE statement is executed.

When *localName* is the same as the global wave name and you want to reference a wave in the current data folder, you can omit the *pathToWave*.

pathToWave can be a full literal path (e.g., root:FolderA:wave0), a partial literal path (e.g., :FolderA:wave0) or $ followed by string variable containing a computed path (see **Converting a String into a Reference Using $** on page IV-57).

You can also use a data folder reference or the /SDFR flag to specify the location of the wave if it is not in the current data folder. See **Data Folder References** on page IV-72 and **The /SDFR Flag** on page IV-74 for details.

If the wave may not exist at runtime, use the /Z flag and call **WaveExists** before accessing the wave. The /Z flag prevents Igor from flagging a missing wave as an error and dropping into the debugger. For example:

```
WAVE/Z wv=<pathToPossiblyMissingWave>
if( WaveExists(wv) )
    <do something with wv>
endif
```

Note that to create a wave, you use the **Make** operation.

Flags

/C	Complex wave.
/T	Text wave.
/Z	Ignores wave reference checking failures.

See Also

WaveExists function.

WAVE Reference Type Flags on page IV-68 for additional wave type flags and information.

Accessing Global Variables and Waves on page IV-59.

Accessing Waves in Functions on page IV-76.

Converting a String into a Reference Using $ on page IV-57.

WAVEClear

`WAVEClear` ***`localName`*** `[,` ***`localName1`*** `…]`

The WAVEClear operation clears out a WAVE reference variable. WAVEClear is equivalent to `WAVE/Z localName= $""`.

Details

Use WAVEClear to avoid unexpected results from certain operations such as **Duplicate** or **Concatenate**, which will reuse the contents of a WAVE reference variable and may not generate the wave in the desired data folder or with the desired name.

WAVEClear ensures that memory is deallocated after waves are killed as in this example:

```
Function foo()
    Make wave1
    FunctionThatKillsWave1()
    WAVEClear wave1
    AnotherFunction()
End
```

Although memory used for wave1 will be deallocated when `foo` returns, that memory will not be automatically released while the function executes because the WAVE variable still contains a reference to the wave. In this example, WAVEClear deallocates that memory before `AnotherFunction` executes.

You can also use WAVEClear before passing a data folder to preemptive threads using **ThreadGroupPutDF**.

See Also

Accessing Waves in Functions on page IV-76, **Wave Reference Counting** on page IV-194, and **ThreadSafe Functions and Multitasking** on page IV-308.

WaveCRC

`WaveCRC(`*`inCRC`*`,`*`waveName`*`[,`*`checkHeader`*`])`

The WaveCRC function returns a 32-bit cyclic redundancy check value of the bytes in the named wave starting with *inCRC*.

Pass 0 for *inCRC* the first time you call WaveCRC for a particular stream of bytes as represented by the wave data.

Pass the last-returned value from WaveCRC for *inCRC* if you are creating a CRC value for a given stream of bytes through multiple calls to WaveCRC.

waveName may be a numeric or text wave.

The optional *checkHeader* parameter determines how much of the wave is checked:

checkHeader	**What It Does**
0	Check only the wave data (default).
1	Check only the internal binary header.
2	Check both.

Details

Polynomial used is:

x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x+1

See crc32.c in the public domain source code for zlib for more information.

See Also

The **StringCRC** function.

WaveDims

`WaveDims(`*`wave`*`)`

The WaveDims function returns the number of dimensions used by *wave*.

Returns zero if wave reference is null. See **WaveExists** for a discussion of null wave references.

Also returns zero if wave has zero rows. A matrix will return 2.

WaveExists

`WaveExists(`*`wave`*`)`

The WaveExists function returns one if wave reference is valid or zero if the wave reference is null. For example if, in a user function, you have:

```
Wave w= $"no such wave"
```

then `WaveExists(w)` will return zero.

WaveExists should be used in functions only. In macros, use the exists function instead.

See Also

exists, **NVAR_Exists**, **SVAR_Exists**, and **Accessing Global Variables and Waves** on page IV-59.

WaveInfo

`WaveInfo(`*`waveName`*`, 0)`

The WaveInfo function returns a string containing a semicolon-separated list of information about the named wave.

The second parameter is reserved for future use and must be zero.

Details

The string contains six kinds of information. Each group is prefaced by a keyword and colon, and terminated with a semicolon. The keywords are:

Keyword	**Information Following Keyword**
DUNITS	The wave's data units.
FULLSCALE	Three numbers indicating whether the wave has any data full scale information, and the min and max data full scale values. The format of the FULLSCALE description is: `FULLSCALE:`*`validFS`*`,`*`minFS`*`,`*`maxFS`*`;` *validFS* is 1 if *minFS* and *maxFS* have been set via a `SetScale d` command; otherwise it is 0.
LOCK	Reads back the value set by **SetWaveLock**.

Keyword	Information Following Keyword
MODIFIED	1 if the wave has been modified since the experiment was last saved, else 0.
MODTIME	The date and time that the wave was last modified in seconds since January 1, 1904.
NUMTYPE	A number denoting the data type of the wave. For text waves this is 0. For wave reference waves it is 16384. For data folder reference waves it is 256. For numeric waves it is one of the following: 1: Complex, added to one of the following 2: 32-bit (single precision) floating point 4: 64-bit (double precision) floating point 8: 8-bit signed integer 16: 16-bit signed integer 32: 32-bit signed integer 64: Unsigned, added to 8, 16, or 32 if wave is unsigned For example, the number denoting a complex double precision wave is 5 (i.e., 1+4).
PATH	The name of the symbolic path in which the wave file is stored (e.g., PATH:home;) or nothing if there is no path for the wave (PATH:;).
SIZEINBYTES	The total size of the wave in bytes. This includes the wave's header, data, note, dimension labels, and unit strings. This keyword was added in Igor Pro 7.00.
XUNITS	The wave's X units.

Always pass 0 as the second input parameter. In future versions of Igor, this parameter may request other kinds of information to be returned.

A null wave reference returns a zero-length string. This might be encountered, for instance, when using **WaveRefIndexedDFR** in a loop to act on all waves in a data folder, and the loop has incremented beyond the highest valid index.

Examples

```
Make/O wave1;SetScale x,0,1,"dyn",wave1;SetScale y,3,20,"v",wave1
String info = WaveInfo(wave1,0)
Print NumberByKey("NUMTYPE", info)          // Prints 2
Print StringByKey("DUNITS", info)           // Prints "v"
```

See Also

The functions and operations listed under "About Waves" categories in the Command Help tab of the Igor Help Browser; among them are **CreationDate**, **ModDate**, **WaveType**, **note**, and **numpnts**.

NumberByKey and **StringByKey** functions for parsing the returned keyword list.

WaveInfo lacks information about multidimensional waves. Individual functions are provided to return dimension-related information: **DimDelta**, **DimOffset**, **DimSize**, **WaveUnits**, and **GetDimLabel**.

WaveList

`WaveList(`*`matchStr`*`, `*`separatorStr`*`, `*`optionsStr`*`)`

The WaveList function returns a string containing a list of waves selected from the current data folder based on *matchStr* and *optionsStr* parameters. See **Details** for information on listing waves in graphs, and for references to newer, data folder-aware functions.

Details

For a wave name to appear in the output string, it must match *matchStr* and also must fit the requirements of *optionsStr* and it must be in the current data folder. The first character of *separatorStr* is appended to each wave name as the output string is generated.

The name of each wave is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything.

For example:

`"*"`	Matches all wave names in current data folder.
`"xyz"`	Matches wave name xyz only, if xyz is in the current data folder.
`"*xyz"`	Matches wave names which end with xyz and are in the current data folder.
`"xyz*"`	Matches wave names which begin with xyz and are in the current data folder.
`"*xyz*"`	Matches wave names which contain xyz and are in the current data folder.
`"abc*xyz"`	Matches wave names which begin with abc and end with xyz and are in the current data folder.

matchStr may begin with the ! character to return windows that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches wave names which do not end with xyz.

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

optionsStr is used to further qualify the wave.

Use `""` to accept all waves in the current data folder that are permitted by *matchStr*.

Set *optionsStr* to one or more of the following comma-separated keyword-value pairs:

optionsStr	**Selection Criteria**
`"BYTE:0"` *or* `"BYTE:1"`	Waves that are not 8-bit integer (if 0) or only waves that are 8-bit integer (if 1).
`"CMPLX:0"` *or* `"CMPLX:1"`	Waves that are not complex (if 0) or only waves that are complex (if 1).
`"DIMS:`*`numberOfDims`*`"`	All waves in current data folder that have *numberOfDims* dimensions. This is the number of dimensions reported by **WaveDims**.
	Use `"DIMS:0"` for all waves having no points (`numpnts(w)==0`).
	Use "DIMS:1" for graph traces (or one of the X, Y, and Z waves of a contour plot).
	Use "DIMS:2" for false color and indexed color images (see **Indexed Color Details** on page II-312).
	Use "DIMS:3" for direct color images (see **Direct Color Details** on page II-313).
`"DF:0"` *or* `"DF:1"`	Consider waves that are not data folder reference waves (if 0) or only waves that are data folder reference waves (if 1). You can create waves that contain data folder references using the **Make** /DF flag.
`"DP:0"` *or* `"DP:1"`	Waves that are not double precision floating point (if 0) or only waves that are double precision floating point (if 1).
`"INT64:0"` *or* `"INT64:1"`	Consider waves that are not 64-bit integer (if 0) or only waves that are 64-bit integer (if 1). 64-bit integer waves are supported in Igor7 and later.
`"INTEGER:0"` *or* `"INTEGER:1"`	Waves that are not 32-bit integer (if 0) or only waves that are 32-bit integer (if 1).
`"MAXCHUNKS:`*`max`*`"`	Waves having no more than *max* chunks.

optionsStr	**Selection Criteria**
`"MAXCOLS:`*`max`*`"`	Waves having no more than *max* columns.
`"MAXLAYERS:`*`max`*`"`	Waves having no more than *max* layers.
`"MAXROWS:`*`max`*`"`	Waves having no more than *max* rows.
`"MINCHUNKS:`*`min`*`"`	Waves having at least *min* chunks.
`"MINCOLS:`*`min`*`"`	Waves having at least *min* columns.
`"MINLAYERS:`*`min`*`"`	Waves having at least *min* layers.
`"MINROWS:`*`min`*`"`	Waves having at least *min* rows.
`"SP:0"` *or* `"SP:1"`	Waves that are not single precision floating point (if 0) or only waves that are single precision floating point (if 1).
`"TEXT:0"` *or* `"TEXT:1"`	Waves that are not text (if 0) or only waves that are text (if 1).
`"UNSIGNED:0"` *or* `"UNSIGNED:1"`	Waves that are not unsigned integer (if 0) or only waves that are unsigned integer (if 1).
`"WAVE:0"` *or* `"WAVE:1"`	Consider waves that do not contain wave references (if 0) or only waves that contain wave references (if 1). You can create waves that contain wave references using the **Make** /WAVE flag.
`"WIN:"`	All waves in the current data folder that are displayed in the top graph or table.
`"WIN:`*`windowName`*`"`	All waves in the current data folder that are displayed in the named table or graph window or subwindow.
`"WORD:0"` *or* `"WORD:1"`	Waves that are not 16-bit integer (if 0) or only waves that are 16-bit integer (if 1).

You can specify more than one option by separating the options with a comma. See the **Examples**.

Note: Even when *optionsStr* is used to list waves used in a graph or table, the waves must be in the current data folder.

Note: In addition to waves displayed as normal graph traces, WaveList will list matrix waves used with **AppendImage** or NewImage and the X, Y, and Z waves used with **AppendXYZContour**.

Note: Individual contour traces are not listed because they have no corresponding waves. See **Contour Traces** on page II-283.

There are several functions that are more useful for listing waves in graphs and tables.

WaveList with `WIN:`*`windowName`* gives only the names of the waves in the graph or table and does not include the data folder for each wave. If you need to know what data folder the waves are in, use **WaveRefIndexed** to get the wave itself and then if needed use **GetWavesDataFolder** to get the path.

When identifying a subwindow with `WIN:`*`windowName`*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

To list the actual waves used in a graph, or to distinguish two or more instances of the same named wave in a graph, use **TraceNameList**. This function can be used in conjunction with **TraceNameToWaveRef**, and **XWaveRefFromTrace**.

Use **ContourNameList** to list contour plots in a given window and **ContourNameToWaveRef** to access the waves used to generate the contour plot.

To list the contour traces (that is, the contour lines themselves) use **TraceNameList** with the appropriate option.

Use **ImageNameList** to list images in a given window and **ImageNameToWaveRef** to access the waves used to generate the images.

Processing Lists of Waves

Contrary to what you might expect, you can *not* use the output of WaveList directly with operations that have a list of waves as their parameters. See **Processing Lists of Waves** on page IV-187 for ways of dealing with this.

Examples

```
// Returns a list of all waves in the current data folder.
WaveList("*",";","")

// Returns a list of all waves in the current data folder and displayed in the top table or graph.
WaveList("*", ";","WIN:")

// Returns a list of waves in the current data folder whose names
// end in "_bkg" and which are displayed in Graph0 as 1D traces.
WaveList("*_bkg", ";", "WIN:Graph0")

// Returns a list of waves in the current data folder whose names do not
// end in "X" and which are displayed in Graph0 as 1D traces or as one
// of the X, Y, and Z waves of an AppendXYZContour plot.
WaveList("!*X", ";", "WIN:Graph0,DIMS:1")
```

See Also

Chapter II-6, **Multidimensional Waves**.

Execute, **ContourNameList**, **ImageNameList**, **TraceNameList**, and **WaveRefIndexed**.

WaveMax

`WaveMax(`*`waveName`*` [, `*`x1`*`, `*`x2`*`])`

The WaveMax function returns the maximum value in the wave for points between x=*x1* to x=*x2*, inclusive.

Details

If *x1* and *x2* are not specified, they default to -inf and +inf, respectively.

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with `pnt2x(`*`waveName`*`,`*`pointNumber1`*`)`, and a similar expression for *x2*.

If the points nearest to *x1* or *x2* are not within the point range of 0 to numpnts(*waveName*)-1, WaveMax limits them to the nearest of point 0 or point numpnts(*waveName*)-1.

For a floating-point wave, WaveMax runs about three times faster than getting the same information using WaveStats. For an integer wave, WaveMax runs about ten times faster than WaveStats. The advantage may not hold for short waves.

See Also

The **WaveMin** function and **WaveStats** operation.

WaveMeanStdv

`WaveMeanStdv `*`srcWave binSizeWave`*

The WaveMeanStdv operation calculates the standard deviation of the means for the specified bin distribution saving the result in the wave W_MeanStdv.

For each entry in *binSizeWave*, *srcWave* is divided into the specified number of bins. Values in each bin are averaged and then the mean and standard deviation of the averages (among all bins) are calculated. The value of the standard deviation of the bin averages divided by the mean is then stored in W_MeanStdv corresponding to the bin size entry in *binSizeWave*.

All entries in *binSizeWave* must be positive integers.

Details

When the number of points in *srcWave* does not divide evenly into the bin size entry from *binSizeWave*, the last bin will have a smaller number of data points. In order not to skew the results the values corresponding to the last bin will be dropped. If your data set is small compared to the bin size you might want to pad *srcWave* with additional values (e.g., duplicate values from the beginning of the wave).

This operation does not support NaNs. If you get a NaN as an entry in the output wave then there is either a NaN in *srcWave* or something is wrong with the calculation for that entry.

WaveMin

`WaveMin(`*`waveName`*` [, `*`x1`*`, `*`x2`*`])`

The WaveMin function returns the minimum value in the wave for points between x=*x1* to x=*x2*, inclusive.

Details

If *x1* and *x2* are not specified, they default to -inf and +inf, respectively.

The X scaling of the wave is used only to locate the points nearest to x=*x1* and x=*x2*. To use point indexing, replace *x1* with `pnt2x(`*`waveName`*`,`*`pointNumber1`*`)`, and a similar expression for *x2*.

If the points nearest to *x1* or *x2* are not within the point range of 0 to numpnts(*waveName*)-1, WaveMin limits them to the nearest of point 0 or point numpnts(*waveName*)-1.

For a floating-point wave, WaveMin runs about three times faster than getting the same information using WaveStats. For an integer wave, WaveMin runs about ten times faster than WaveStats. The advantage may not hold for short waves.

See Also

The **WaveMax** function and **WaveStats** operation.

WaveName

`WaveName(`*`winNameStr`*`, `*`index`*`, `*`type`*`)`

The WaveName function returns a string containing the name of the *index*th wave of the specified *type* in the named window or subwindow.

Parameters

winNameStr can be `""` to refer to the top graph or table.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

WaveName works on waves displayed in a graph, in a table or on the list of waves in the current data folder. If the window is a table, WaveName returns the column name (e.g., "wave0.d"), rather than the name of the wave itself (e.g., "wave0").

For most uses, we recommend that you use **WaveRefIndexed** or **WaveRefIndexedDFR** instead of WaveName. WaveName returns a string containing the wave name only, with no data folder path qualifying it. Thus, you may get erroneous results if the wave referred to in the graph has the same name as a different wave in the current data folder. Likewise, if the named wave resides in a data folder that is not the current data folder, you will not be able to refer to the named wave. Use **WaveRefIndexedDFR** instead.

winNameStr is a string expression containing the name of a graph or table or an empty string (`""`). If the string is empty and *type* is 4 then WaveName works on the list of all waves in the current data folder. If the string is empty and the type parameter is not 4 then WaveName works on the top graph or table.

index starts from zero.

type is a number from 1 to 4. When type is 4 and *winNameStr* is `""`, WaveName works on the list of all waves in the current data folder.

For graph windows, *type* is 1 for y waves, 2 for x waves, 3 for either y or x waves.

For table windows, *type* is 1 for data columns, 2 for index or dimension label columns, 3 for either data or index or dimension label columns.

WaveName returns an empty string (`""`) if there is no wave matching the parameters.

Examples

```
WaveName("",0,4)     // Returns name first wave current data folder.
WaveName("",0,1)     // Returns name of first Y wave in the top graph.
WaveName("Graph0",1,2)     // Returns name of second X wave in Graph0.
WaveName("Table0",1,3)     // Returns name of second column in Table0.
```

WaveRefIndexed

`WaveRefIndexed(`*`winNameStr, index, type`*`)`

The WaveRefIndexed function returns a wave reference to the *index*th wave of the specified *type* in the named window or subwindow.

To iterate through the waves in a data folder, use **WaveRefIndexedDFR** instead of WaveRefIndexed.

Parameters

winNameStr can be `""` to refer to the top graph or table window or the current data folder.

When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

WaveRefIndexed is analogous to WaveName but works better with data folders. We recommend that you use it instead of WaveName.

winNameStr is a string expression containing the name of a graph or table or an empty string (`""`). If the string is empty and *type* is 4 then WaveRefIndexed works on Igor's list of all waves in the current data folder. If the string is empty and the type parameter is not 4 then WaveRefIndexed works on the top graph or table.

index starts from zero.

type is a number from 1 to 4. When type is 4 and *winNameStr* is `""`, WaveRefIndexed works on the list of all waves in the current data folder.

For graph windows, *type* is 1 for y waves, 2 for x waves, 3 for either y or x waves.

For table windows, *type* is 1 for data columns, 2 for index or dimension label columns, 3 for either data or index or dimension label columns.

WaveRefIndexed returns a null reference (see **WaveExists**) if there is no wave matching the parameters.

Examples

```
WaveRefIndexed("",0,1)           // Returns first Y wave in the top graph.
WaveRefIndexed("Graph0",1,2)     // Returns second X wave in Graph0.
WaveRefIndexed("Table0",1,3)     // wave in second column in Table0.
```

See Also

WaveRefIndexedDFR, **NameOfWave**, **GetWavesDataFolder**

For a discussion of wave references, see **Wave Reference Functions** on page IV-186.

WaveRefIndexedDFR

`WaveRefIndexedDFR(`*`dfr, index`*`)`

The WaveRefIndexedDFR function returns a wave reference to the *index*th wave in the specified data folder.

Parameters

dfr is a data folder reference.

index is the zero-based index of the wave you want to access.

Details

WaveRefIndexedDFR returns a null reference (see **WaveExists**) if there is no wave corresponding to index in the specified data folder.

Example

```
// DemoWaveRefIndexedDFR can be called like this:
//  DemoWaveRefIndexedDFR(root:, 0)                 // Work on root
//  DemoWaveRefIndexedDFR(root:SubDataFolder, 0)    // Work on root:SubDataFolder
//  DemoWaveRefIndexedDFR(:, 0)                     // Work on current data folder
Function DemoWaveRefIndexedDFR(dfr, recurse)
    DFREF dfr
    Variable recurse

    Variable index = 0
    do
```

```
        Wave/Z w = WaveRefIndexedDFR(dfr, index)
        if (!WaveExists(w))
           break
        endif
        String path = GetWavesDataFolder(w, 2)
        Print path
        index += 1
    while(1)

    if (recurse)
        Variable numChildDataFolders = CountObjectsDFR(dfr, 4)
        Variable i
        for(i=0; i<numChildDataFolders; i+=1)
           String childDFName = GetIndexedObjNameDFR(dfr, 4, i)
           DFREF childDFR = dfr:$childDFName
           DemoWaveRefIndexedDFR(childDFR, 1)
        endfor
    endif
End
```

See Also

WaveRefIndexed, **NameOfWave**, **GetWavesDataFolder**

For a discussion of wave references, see **Wave Reference Functions** on page IV-186.

WaveRefsEqual

`WaveRefsEqual(`*`w1, w2`*`)`

The WaveRefsEqual function returns the truth the two wave references are the same.

See Also

Wave Reference Functions on page IV-186

WaveRefWaveToList

`WaveRefWaveToList(`*`waveRefWave, option`*`)`

The WaveRefWaveToList function returns a semicolon-separated string list containing data folder paths.

Each element of the returned string list is the full or partial path to the wave referenced by the corresponding element of *waveRefWave*. Entries in *waveRefWave* that are NULL or entries that correspond to free waves result in an empty list element.

The WaveRefWaveToList function was added in Igor Pro 7.00.

Parameters

waveRefWave is a wave reference wave each element of which contains a reference to an existing wave or NULL (0).

option determines if the returned path is a full path or a partial path relative to the current data folder:

option=0:	Full path.
option=1:	Partial path relative to the current data folder.

Other values of *option* are reserved for the future.

Example

```
Function Test()
    SetDataFolder root:
    Make/O/FREE aaa
    Make/O bbb
    Make/O/WAVE/N=3 wr
    Wr[0]=aaa
    // Wr[1] is null by initialization.
    wr[2]=bbb
    Print WaveRefWaveToList(wr,0)
End
```

```
// Executing Test() gives:
  ;;root:bbb;
```

The first empty string corresponds to the free wave 'aaa' and the second empty string corresponds to the null entry in the wave reference wave.

See Also

ListToWaveRefWave, **ListToTextWave**, **Wave References** on page IV-65

WaveStats

`WaveStats [`*`flags`*`]` ***`waveName`***

The WaveStats operation computes several statistics on the named wave.

Flags

/ALPH=*val*	Sets the significance level for the confidence interval of the mean (default *val*=0.05).
/C=*method*	Calculates statistics for complex waves only. Does not affect real waves. You can use *method* in various combinations to process the real, imaginary, magnitude, and phase of the wave. The result is stored in the wave M_WaveStats (see **Details** for format). *method* is defined as follows: *method*=0: Default; ignores the imaginary part of *waveName*. Use /W to also store statistics in M_WaveStats. *method*=1: Calculates statistics for real part of *waveName* and stores it in M_WaveStats. *method*=2: Calculates statistics for imaginary part of *waveName* and stores the result in M_WaveStats. *method*=4: Calculates statistics for magnitude of *waveName*, i.e., `sqrt(real^2 +imag^2)`, and stores the result in M_WaveStats. *method*=8: Calculate statistics for phase of *waveName* using `atan2(imag,real)`. If you use a single *method* the results are stored both in M_WaveStats and in the standard variables (e.g., V_avg, etc.). If you specify *method* as a combination of more than one binary field then the variables reflect the results for the lowest chosen field and all results are stored in the wave M_WaveStats. For example, if you use /C=12, the variables will be set for the statistics of the magnitude and M_WaveStats will contain columns corresponding to the magnitude and to the phase. In this mode V_numInfs will always be zero. **Note**: If you invoke this operation and M_WaveStats already exists in the current data folder, it will be either overwritten or initialized to NaN.
/M=*moment*	Calculates statistical moments. *moment* is defined as follows: *moment*=1: Calculates only lower moments: V_avg, V_npnts, V_numInfs, and V_numNaNs. Use it if you do not need the higher moments. *moment*=2: Default; calculates both lower moments and higher order quantities: V_sdev, V_rms, V_adev, V_skew, and v_kurt.
/Q	Prevents results from being printed in history.

/PCST	Computes the statistics on a per-column basis for a real valued wave of two or more dimensions. The results are saved in the wave M_WaveStats which has the same number of columns, layers and chunks as the input wave and where the rows, designated by dimension labels, contain the standard WaveStats statistics. All the V_ variables are set to NaN. Note that this flag is not compatible with the flags /C, /R, /RMD. The /PCST flag was added in Igor Pro 7.00.
/R=(*startX*,*endX*)	Specifies an X range of the wave to evaluate.
/R=[*startP*,*endP*]	Specifies a point range of the wave to evaluate. If you specify the range as /R=[*startP*] then the end of the range is taken as the end of the wave. If /R is omitted, the entire wave is evaluated.

/RMD=[*firstRow*,*lastRow*][*firstColumn*,*lastColumn*][*firstLayer*,*lastlayer*][*firstChunk*,*lastChunk*]

Designates a contiguous range of data in the source wave to which the operation is to be applied. This flag was added in Igor Pro 7.00.

You can include all higher dimensions by leaving off the corresponding brackets. For example:

```
/RMD=[firstRow,lastRow]
```

includes all available columns, layers and chunks.

You can use empty brackets to include all of a given dimension. For example:

```
/RMD=[][firstColumn,lastColumn]
```

means "all rows from column A to column B".

You can use a * to specify the end of any dimension. For example:

```
/RMD=[firstRow,*]
```

means "from firstRow through the last row".

/W	Stores results in the wave M_WaveStats in addition to the various V_ variables when /C=0.
/Z	No error reporting.
/ZSCR	Computes z scores $z_i = \frac{Y_i - \overline{Y}}{\sigma}$, which are saved in W_ZScores.

Details

WaveStats uses a two-pass algorithm to produce more accurate results than obtained by computing the binomial expansions of the third and fourth order moments.

WaveStats returns the statistics in the automatically created variables:

`V_npnts`	Number of points in range excluding points whose value is NaN or INF.
`V_numNans`	Number of NaNs.
`V_numINFs`	Number of INFs.
`V_avg`	Average of data values.
`V_sum`	Sum of data values.

V_sdev	Standard deviation of data values, $\sigma = \sqrt{\frac{\sum (Y_i - V\_avg)^2}{V\_npnts - 1}}$ “Variance” is V_sdev^2.
V_sem	Standard error of the mean $sem = \frac{\sigma}{\sqrt{V\_npnts}}$
`V_rms`	RMS of Y values $= \sqrt{\frac{1}{V\_npnts} \sum Y_i^2}$
V_adev	Average deviation $= \frac{1}{V\_npnts} \sum_{i=0}^{V\_npnts-1} \left\lvert Y_i - \overline{Y} \right\rvert$
V_skew	Skewness $= \frac{1}{V\_npnts} \sum_{i=0}^{V\_npnts-1} \left(\frac{Y_i - \overline{Y}}{\sigma} \right)^3$
V_kurt	Kurtosis $= \left(\frac{1}{V\_npnts} \sum_{i=0}^{V\_npnts-1} \left(\frac{Y_i - \overline{Y}}{\sigma} \right)^4 \right) - 3$
`V_minloc`	X location of minimum data value.
`V_min`	Minimum data value.
`V_maxloc`	X location of maximum data value.
`V_max`	Maximum data value.
`V_minRowLoc`	Row containing minimum data value.
`V_maxRowLoc`	Row containing maximum data value.
`V_minColLoc`	Column containing minimum data value (2D or higher waves).
`V_maxColLoc`	Column containing maximum data value (2D or higher waves).
`V_minLayerLoc`	Layer containing minimum data value (3D or higher waves).
`V_maxLayerLoc`	Layer containing maximum data value (3D or higher waves).
`V_minChunkLoc`	Chunk containing minimum data value (4D waves only).
`V_maxChunkLoc`	Chunk containing maximum data value (4D waves only).
`V_startRow`	The unscaled index of the first row included in caculating statistics.
`V_endRow`	The unscaled index of the last row included in caculating statistics.
`V_startCol`	The unscaled index of the first column included in calculating statistics. Set only when /RMD is used.
`V_endCol`	The unscaled index of the last column included in calculating statistics. Set only when /RMD is used.

`V_startLayer`	The unscaled index of the first layer included in calculating statistics. Set only when /RMD is used.
`V_endLayer`	The unscaled index of the last layer included in calculating statistics. Set only when /RMD is used.
`V_startChunk`	The unscaled index of the first chunk included in calculating statistics. Set only when /RMD is used.
`V_endChunk`	The unscaled index of the last chunk included in calculating statistics. Set only when /RMD is used.

WaveStats prints the statistics in the history area unless /Q is specified. The various multidimensional min and max location variables will only print to the history area for waves having the appropriate dimensionality.

The format of the M_WaveStats wave is:

Row	Statistic	Row	Statistic	Row	Statistic	Row	Statistic
0	`numPoints`	9	`minLoc`	18	`maxColLoc`	27	`startCol`
1	`numNaNs`	10	`min`	19	`maxLayerLoc`	28	`endCol`
2	`numInfs`	11	`maxLoc`	20	`maxChunkLoc`	29	`startLayer`
3	`avg`	12	`max`	21	`startRow`	30	`endLayer`
4	`sdev`	13	`minRowLoc`	22	`endRow`	31	`startChunk`
5	`rms`	14	`minColLoc`	23	`sum`	32	`endChunk`
6	`adev`	15	`minLayerLoc`	24	`meanL1`		
7	`skew`	16	`minChunkLoc`	25	`meanL2`		
8	`kurt`	17	`maxRowLoc`	26	`sem`		

`meanL1` and `meanL2` are the confidence intervals for the mean

$$MeanL1 = V\_avg - t_{\alpha,v}\frac{V\_sdev}{\sqrt{V\_npnts}}, \quad \text{and} \quad MeanL2 = V\_avg + t_{\alpha,v}\frac{V\_sdev}{\sqrt{V\_npnts}}$$

where $t_{a,v}$ is the critical value of the Student T distribution for *alpha* significance and degree of freedom $v=V\_npnts-1$.

Use `Edit M_WaveStats.ld` to display the results in a table with dimension labels identifying each of the row statistics.

WaveStats is not entirely multidimensional aware. Even so, much of the information computed by WaveStats is useful. See **Analysis on Multidimensional Waves** on page II-86 for details.

See Also

Chapter III-12, **Statistics** for a function and operation overview.

The **ImageStats** operation for calculating wave statistics for specified regions of interest in 2D matrix waves.

The **WaveMax**, **WaveMin**, **mean**, **median** and **Variance** functions.

WaveTextEncoding

`WaveTextEncoding(`*`wave, element, getEffectiveTextEncoding`*`)`

The WaveTextEncoding function returns the text encoding code for the specified element of a wave. See **Wave Text Encodings** on page III-422 for background information.

This function is used to deal with text encoding issues that sometimes arise in when you load pre-Igor Pro 7 experiments. Most users will have no need to use it.

The WaveTextEncoding function was added in Igor Pro 6.30. The *getEffectiveTextEncoding* parameter was added in Igor Pro 7.00.

Parameters

wave specifies the wave of interest.

element specifies a part of the wave, as follows:

Value	Meaning
1	Wave name
2	Wave units
4	Wave note
8	Wave dimension labels
16	Text wave content

getEffectiveTextEncoding determines if WaveTextEncoding returns a raw text encoding code or an effective text encoding code as explained below.

Details

WaveTextEncoding returns a integer text encoding code. See **Text Encoding Names and Codes** on page III-434 for details.

As explained under **Wave Text Encodings** on page III-422, each of the wave elements has a corresponding text encoding setting. Because the notion of text encoding settings was added in Igor Pro 6.30, waves created by earlier versions have their text encoding settings set to unknown (0).

The text encoding setting stored for a given element is the "raw" text encoding. If it is unknown, then Igor applies some rules when the wave is accessed to determine an "effective" text encoding for the element being accessed. The rules are explained under **Determining the Text Encoding for a Plain Text File** on page III-417.

If *getEffectiveTextEncoding* is non-zero then WaveTextEncoding returns the effective text encoding. If *getEffectiveTextEncoding* is zero it returns the raw text encoding.

See Also

Wave Text Encodings on page III-422, **Text Encoding Names and Codes** on page III-434, **Determining the Text Encoding for a Plain Text File** on page III-417

WaveTransform

`WaveTransform [`*`flags`*`]` ***`keyword srcWave`***

The WaveTransform operation transforms *srcWave* in various ways. If the /O flag is not specified then unless otherwise indicated the output is stored in the wave W_WaveTransform, which will be of the same data type as *srcWave* and saved in the current data folder.

Parameters

keyword is one of the following:

abs	Calculates the absolute value of the entries in *srcWave*. It stores results in W_Abs if *srcWave* is 1D or M_Abs otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.
acos	Calculates the inverse cosine of the entries in *srcWave*. It stores results in W_Acos if *srcWave* is 1D or M_Acos otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.
asin	Calculates the inverse sine of the entries in *srcWave*. It stores results in W_Asin if *srcWave* is 1D or M_Asin otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.

atan	Calculates the inverse tangent of the entries in *srcWave*. It stores results in W_Atan if *srcWave* is 1D or M_Atan otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.
cconjugate	Calculates the complex conjugate of *srcWave*. Stores results in W_CConjugate or M_CConjugate, depending on wave dimensionality, or overwrites *srcWave* if /O is used.
cos	Calculates the cosine of the entries in *srcWave*. It stores results in W_Cos if *srcWave* is 1D or M_Cos otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single or double precision real wave.
crystalToRect	Converts triplet (three column {x,y,z}) waves from nonorthogonal crystallographic coordinates to rectangular cartesian system. The parameters provided in the /P flag are the crystallographic definition of the coordinate system given by {a, b, c, alpha, beta, gamma}. The three angles are assumed to be expressed in radians unless the /D flag is specified. The transformation sets the first component parallel to the vector a and the third component parallel to c*. The output is stored in the current data folder in the wave M_CrystalToRect which has the same data type. If the /O flag is specified, the output overwrites the original data.
flip	Flips the data in *srcWave* about its center. If /O flag is used, *srcWave* is overwritten. Otherwise a new wave is created in the current data folder. The wave is named W_flipped or M_flipped according to the dimensionality of *srcWave*.
index	Fills *srcWave* as in `jack=p`. If /P is specified then `jack=p+`*`param1`*. The /O flag does not apply here.
inverse	Computes `1/srcWave[i]` for each point in *srcWave* and stores it in W_Inverse or M_Inverse depending on the dimensionality of *srcWave*.
inverseIndex	Fills *srcWave* as in `jack=numPnts-1-p`. If /P is specified the `jack=numPnts-1-p+`*`param1`*.
magnitude	Creates a real-valued wave that is the magnitude of *srcWave*. If you do not specify the /O flag, the output is stored in W_Magnitude or M_Magnitude depending on the dimensionality of *srcWave*; the output precision will be the same as *srcWave*.
magsqr	Creates a real-valued wave that is the magnitude squared of *srcWave*. If *srcWave* is a double precision complex wave, the output is also double precision, otherwise the output is a single precision wave. Stores the result in wave W_MagSqr or M_MagSqr, depending on the dimensionality of *srcWave*, or overwrites *srcWave* if /O is used.
max	Calculates the maximum of a point in *srcWave* and a fixed number specified as a single parameter with the /P flag. It stores results in W_max if *srcWave* is 1D or M_max otherwise. It will overwrite *srcWave* when used with the /O flag. See also the min keyword and the example below.
min	Calculates the minimum of a point in *srcWave* and a fixed number specified as a single parameter with the /P flag. It stores results in W_min if *srcWave* is 1D or M_min otherwise. It will overwrite *srcWave* when used with the /O flag. See also the max keyword and the example below.
normalizeArea	Calculates the area under the curve and rescales the wave so that the area is 1. Note that waves with negative areas will be rescaled to positive values. Applies to 1D real-valued waves. It does not affect wave scaling. Stores the result in the wave W_normalizedArea or overwrites *srcWave* if /O is used.
phase	Creates a real-valued wave containing the phase of the complex input wave. If the /O flag is not used, the output is stored in W_Phase or M_Phase depending on the dimensionality of *imageMatrix*. You can also use /P={*norm*} to divide the output wave by the value of *norm*.

rectToCrystal	Converts triplet (three column {x,y,z}) waves from cartesian coordinates to nonorthogonal crystallographic coordinate system. The parameters provided in the /P flag are the crystallographic definition of the coordinate system given by {a, b, c, alpha, beta, gamma}. The three angles are assumed to be expressed in radians unless the /D flag is specified. The transformation assumes the first component parallel to the vector a and the third component parallel to c*. The output is stored in the current data folder in the wave M_RectToCrystal which has the same data type. If the /O flag is specified, the output overwrites the original data.
setConstant	Sets *srcWave* points to a constant value specified by the /V flag. This keyword applies to real, numeric waves only. You can use /R with setConstant to set a subset of a wave. setConstant was added in Igor Pro 7.00.
setZero	Sets all *srcWave* points to zero. setZero was added in Igor Pro 7.00.
sgn	Sets the value to -1 if the entry is negative, 1 otherwise. Stores the results in W_Sgn or overwrites *srcWave* if /O is used. This operation will not work on UNSIGNED waves.
shift	Shifts the position of data in *srcWave* by the specified number of points. Unlike Rotate, WaveTransform discards data points that shift outside existing wave boundaries. After the shift, vacated wave points are set to the specified *fillValue*. The shift and the *fillValue* are specified with the /P flag using the syntax: /P={*numPoints*, *fillValue*}. If you do not provide a fill value, it will be 0 for integer waves and NaN for SP and DP.
sin	Calculates the sine of the entries in srcWave. Stores results in W_Sin if *srcWave* is 1D or M_Sin otherwise. Overwrites *srcWave* when used with the /O flag. *srcWave* must be a real single or double precision floating point wave.
sqrt	Calculates the square root of the entries in *srcWave*. It stores results in W_sqrt if *srcWave* is 1D or M_sqrt otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single- or double-precision real wave.
tan	Calculates the tangent of the entries in *srcWave*. The results are stored in W_tan if *srcWave* is 1D or M_tan otherwise. It will overwrite *srcWave* when used with the /O flag. *srcWave* must be single- or double-precision real wave.
zapINFs	Deletes elements whose value is infinity or -infinity. This is relevant for 1D single-precision and double-precision floating point waves only and does nothing for other types of 1D waves. It is not suitable for multi-dimensional waves and returns an error if *srcWave* is multi-dimensional. Use **MatrixOp** `replace` for multi-dimensional waves.
zapNaNs	Deletes elements whose value is NaN. This is relevant for 1D single-precision and double-precision floating point waves only and does nothing for other types of 1D waves. It is not suitable for multi-dimensional waves and returns an error if *srcWave* is multi-dimensional. Use **MatrixOp** `replaceNaNs` for multi-dimensional waves.

Flags

/D	If present, angles in wave data are interpreted as in degrees. Otherwise they are interpreted as in radians.
/O	Overwrites input wave.
/P={*param1*…}	Specifies parameters as appropriate for the keyword that you are using. The number of parameters and their order depends on the keyword.
/R=[*startRow*,*endRow*][*startCol*,*endCol*][*startLayer*,*endLayer*][*startChunk*,*endChunk*]	Specifies the range of elements to set for the setConstant keyword. You can omit parameters for dimensions that don't exist in *srcWave*. For example, if *srcWave* is 1D, specify just /R=[*startRow*,*endRow*]. /R was added in Igor Pro 7.00.
/V=*value*	Specifies the value to use for the setConstant keyword. /V was added in Igor Pro 7.00.

Examples

```
// Produce output values in the range [-1,1]:
WaveTransform /P={(pi)} phase complexWave

// Faster than myWave=myWave>1 ? 1 : myWave
WaveTransform /P={1}/O min myWave
```

See Also

The **Rotate** operation.

References

Shmueli, U. (Ed.), International Tables for Crystallography, Volume B: 3.3, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

WaveType

`WaveType(`*`waveName`*` [,`*`selector`*` ])`

The WaveType function returns the type of data stored in the wave.

If *selector* = 1, WaveType returns 0 for a null wave, 1 if numeric, 2 if text, 3 if the wave holds data folder references or 4 if the wave holds wave references.

If *selector* = 2, WaveType returns 0 for a null wave, 1 for a normal global wave or 2 for a free wave or a wave that is stored in a free data folder.

If *selector* is omitted or zero, the returned value for non-numeric waves (text waves, wave-reference waves and data folder-reference waves) is 0.

If *selector* is omitted or zero, the returned value for numeric waves is a combination of bit values shown in the following table:

Type	Bit Number	Decimal Value	Hexadecimal Value	
complex	0	1	1	
32-bit float	1	2	2	
64-bit float	2	4	4	
8-bit integer	3	8	8	
16-bit integer	4	16	10	
32-bit integer	5	32	20	
64-bit integer	7	128	80	Requires Igor Pro 7 or later
unsigned	6	64	40	

The unsigned bit is used only with the integer types while the complex bit can be used with any numeric type. Set only one of bits 1-5 or bit 7 as they are mutually exclusive. See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Examples

```
Variable waveIsComplex = WaveType(wave) & 0x01
Variable waveIs32BitFloat = WaveType(wave) & 0x02
Variable waveIs64BitFloat = WaveType(wave) & 0x04
Variable waveIs8BitInteger = WaveType(wave) & 0x08
Variable waveIs16BitInteger = WaveType(wave) & 0x10
Variable waveIs32BitInteger = WaveType(wave) & 0x20
Variable waveIs64BitInteger = WaveType(wave) & 0x80
Variable waveIsUnsigned = WaveType(wave) & 0x40
```

See Also

For concepts related to selector = 1 or 2, see **Free Waves** on page IV-84, **Wave Reference Waves** on page IV-71 and **Data Folder Reference Waves** on page IV-76.

WaveUnits

`WaveUnits(`*`waveName, dimNumber`*`)`

The WaveUnits function returns a string containing the units for the given dimension.

Use *dimNumber*=0 for rows, 1 for columns, 2 for layers, and 3 for chunks. Use -1 to get the data units. If the wave is just 1D, *dimNumber*=0 returns X units and 1 returns data units. This behavior is just like the WaveMetrics procedure WaveUnits found in the WaveMetrics Procedures folder in previous versions of Igor Pro.

See Also

DimDelta, **DimOffset**, **DimSize**, **SetScale**

wfprintf

`wfprintf `*`refNumOrStr, formatStr`*` [`*`flags`*`] `*`waveName`*` [, `*`waveName`*`]…`

The wfprintf operation is like the **printf** operation except that it prints the contents of the named waves to a file whose file reference number is in *refNum*.

The **Save** operation also outputs wave data to a text file. Use Save unless you need the added flexibility provided by wfprintf.

Parameters

refNumOrStr is a numeric expression, a string variable or an SVAR pointing to a global string variable.

If a numeric expression, then it is a file reference number returned by the **Open** operation or an expression that evaluates to 1.

If *refNumOrStr* is 1, Igor prints to the history area instead of to a file.

If *refNumOrStr* is the name of a string variable, the wave contents are “printed” to the named string variable. *refNumOrStr* can also be the name of an SVAR to print to a global string:

```
SVAR sv = root:globalString
wfprintf sv, "", wave0
```

refNumOrStr can not be an element of a text wave.

The value of each named wave is printed to the file according to the conversion specified in *formatStr*.

formatStr contains one numeric conversion specification per column. See **printf**. If *formatStr* is `""`, wfprintf uses a default format which gives tab-delimited columns. *formatStr* is limited to 800 bytes.

Flags

Note: /R must follow the *formatStr* parameter directly without an intervening comma.

/R=(*startX*,*endX*)	Specifies an X range in the wave(s) to print.
/R=[*startP*,*endP*]	Specifies a point range in the wave(s) to print.

Details

As of Igor7, wfprintf supports 1D and 2D waves. Previously it supported 1D waves only.

The number of conversion characters in *formatStr* must exactly match the number of wave columns in all input waves. With real waves, the total number of columns is limited to 100. With complex waves, the real column and imaginary column each count as a column and the total number of columns is limited to 200.

The only conversion characters allowed are fFeEgdouxXcs (the floating point, integer and string conversion characters). You cannot use an asterisk to specify field width or precision. If any of these restrictions is intolerable, you can use fprintf in a loop.

With integer conversion characters d, o, u, x, and X, applied to floating point waves, wfprintf truncates the fractional part.

Examples

```
Function Example1()
   Make/O/N=10 wave0=sin(p*pi/10)                // test numeric wave
   Make/O/N=10/T textWave= "row "+num2istr(p)     // test text wave
   Variable refNum
   Open/P=home refNum as "output.txt"// open file for write
   wfprintf refNum, "%s = %g\r"/R=[0,5], textWave, wave0    // print 6 values each
   Close refNum
End
```

The resulting output.txt file contains:

```
row 0 = 0
row 1 = 0.309017
row 2 = 0.587785
row 3 = 0.809017
row 4 = 0.951057
row 5 = 1
```

```
Function/S NumericWaveToStringList(w)
   Wave w                        // numeric wave (if text, use /T here and %s below)
   String list
   wfprintf list, "%g;" w        // semicolon-separated list
   return list
End
```

```
Print NumericWaveToStringList(wave0)
  0;0.309017;0.587785;0.809017;0.951057;1;0.951057;0.809017;0.587785;0.309017;
```

See Also

The **printf** operation for complete format and parameter descriptions and **Creating Formatted Text** on page IV-244. The **Open** operation about *refNum* and for another way of writing wave files.

The **Save** operation.

WhichListItem

WhichListItem(*itemStr*, *listStr* [, *listSepStr* [, *startIndex* [, *matchCase*]]])

The WhichListItem function returns the index of the first item of *listStr* that matches *itemStr*. *listStr* should contain items separated by the *listSepStr* character, such as "abc;def;". If the item is not found in the list, -1 is returned.

Use WhichListItem to locate an item in a string containing a list of items separated by a single character, such as those returned by functions like **TraceNameList** or **AnnotationList**, or a line from a delimited text file.

listSepStr, *startIndex*, and *matchCase* are optional; their defaults are ";", 0, and 1 respectively.

Details

WhichListItem differs from **FindListItem** in that WhichListItem returns a list index, while FindListItem returns a character offset into a string.

listStr is searched for *itemStr* bound by *listSepStr* on the left and right.

listStr is treated as if it ends with a *listSepStr* even if it doesn't.

Searches for *listSepStr* are always case-sensitive. The comparison of *itemStr* to the contents of *listStr* is usually case-sensitive. Setting the optional *matchCase* parameter to 0 makes the comparison case insensitive.

If *itemStr* is not found, if *listStr* is `""`, or if *startIndex* is not within the range of 0 to ItemsInList(*listStr*)-1, then -1 is returned.

In Igor6, only the first byte of *listSepStr* was used. In Igor7 and later, all bytes are used.

Items can be empty. In `"abc;def;;ghi"`, the third item, whose zero-based index is 2, is empty. In `";def;;ghi;"` the first and third items, whose zero-based indices are 0 and 2, are empty.

If *startIndex* is specified, then *listSepStr* must also be specified. If *matchCase* is specified, *startIndex* and *listSepStr* must be specified.

Examples

```
Print WhichListItem("wave0", "wave0;wave1;")          // prints 0
Print WhichListItem("c", "a;b;")                      // prints -1
Print WhichListItem("", "a;;b;")                      // prints 1
Print WhichListItem("c", "a,b,c,x,c", ",")            // prints 2
Print WhichListItem("c", "a,b,c,x,c", ",", 3)         // prints 4
Print WhichListItem("C", "x;c;C;")                    // prints 2
Print WhichListItem("C", "x;c;C;", ";", 0, 0)         // prints 1
```

See Also

The **AddListItem**, **FindListItem**, **FunctionList**, **ItemsInList**, **RemoveListItem**, **RemoveFromList**, **StringFromList**, **StringList**, **TraceNameList**, **VariableList**, and **WaveList** functions.

WignerTransform

`WignerTransform [/Z][/WIDE=`*`wSize`*`][/GAUS=`*`gaussianWidth`*`][/DEST=`*`destWave`*`]` ***`srcWave`***

The WignerTransform operation computes the Wigner transformation of a 1D signal in *srcWave*, which is the name of a real or complex wave. The result of the WignerTransform is stored in *destWave* or in the wave M_Wigner in the current data folder.

Flags

/DEST=*destWave*	Creates by default a real wave reference for the destination wave in a user function. See **Automatic Creation of WAVE References** on page IV-66 for details.
/GAUS=*gWidth*	Computes the Gaussian Wigner Transform, which is a convolution of the Wigner Transform with a two-dimensional Gaussian (in the two parameters of the transform). The computation of the transform simplifies significantly when the product of the widths of the two Gaussians is unity (minimum uncertainty ellipse). *gWidth* uses the same units as the *srcWave* scaling.
/WIDE=*wSize*	Computes Wigner Transform and sets the transform width to *wSize*. This is the default transformation with *wSize* set to the size of *srcWave*.
/Z	No error reporting.

Details

The Wigner transform maps a time signal *U*(*t*) into a 2D time-frequency representation:

$$W(t,\nu) = \int_{-\infty}^{\infty} U\left(t+\frac{x}{2}\right)U^*\left(t-\frac{x}{2}\right)e^{-2\pi i x\nu}\,dx.$$

The computation of the Wigner transform evaluates the offset product

$$U\left(t+\frac{x}{2}\right)U^*\left(t-\frac{x}{2}\right)$$

over a finite window and then Fourier transforms the result. The offset product can be evaluated over a finite window width, which can vary from a few elements of the input wave to the full length of the wave. You can control the width of this window using the /WIDE flag. If you do not specify the output destination, WignerTransform saves the results in the wave M_Wigner in the current data folder.

Although the Wigner transform is real, the output will be complex when *srcWave* is complex. By inspecting the complex wave you can gain some insight into the numerical stability of the algorithm. The X-scaling of the output wave is identical to the scaling of *srcWave*. The Y-scaling of the input wave is taken from the Fourier Transform of the offset product, which in turn is determined by the X-scaling of *srcWave*. Specifically, if `dx=DimDelta(`*`srcWave`*`,0)` and *srcWave* has N points then `dy=DimDelta(M_Wigner,1)=1/(dx*N)`. WignerTransform does not set the units of the output wave.

The Ambiguity Function is related to the Wigner Transform by a Fourier Transform, and is defined by

$$A(\tau,\nu)=\int_{-\infty}^{\infty}U\left(t+\frac{\tau}{2}\right)U^*\left(t-\frac{\tau}{2}\right)e^{-2\pi it\nu}dt.$$

Convolving the Wigner Transform with a 2D Gaussian leads to what is sometimes called the Gaussian Wigner Transform or GWT. Formally the GWT is given by the equation:

$$GWT\left(t,\nu;\delta_t,\delta_\nu\right)=\frac{1}{\delta_t\delta_\nu}\iint dt'd\nu'W(t',\nu')\exp\left\{-2\pi\left[\left(\frac{t-t'}{\delta_t}\right)^2+\left(\frac{\nu-\nu'}{\delta_\nu}\right)^2\right]\right\}.$$

Computationally this equation simplifies if the respective widths of the two Gaussians satisfy the minimum uncertainty condition $\delta_t*\delta_\nu=1$. The /GAUS flag calculates the Gaussian Wigner Transform using your specified width, δ_t, and it selects a δ_ν such that it satisfies the minimum uncertainty condition.

See Also

CWT, **FFT**, and **WaveTransform** operations.

For further discussion and examples see **Wigner Transform** on page III-250.

References

Wigner, E. P., On the quantum correction for thermo-dynamic equilibrium, *Physics Review*, *40*, 749-759, 1932.

Bartelt, H.O., K.-H. Brenner, and A.W. Lohman, The Wigner distribution function and its optical production, *Optics Communications*, *32*, 32-38, 1980.

Window

`Window` ***`macroName`*****`(`**`[`***`parameters`***`]`**`)`** `[:`***`macro type`***`]`

The Window keyword introduces a macro that recreates a graph, table, layout, or control panel window. The macro appears in the appropriate submenu of the Windows menu. Window macros are automatically created when you close a graph, table, layout, control panel, or XOP target window. You should use **Macro**, **Proc**, or **Function** instead of Window for your own window macros. Otherwise, it works the same as **Macro**.

See Also

The **Macro**, **Proc**, and **Function** keywords. **Data Folders and Window Recreation Macros** on page II-103 for details.

Macro Syntax on page IV-110 for further information.

WindowFunction

WindowFunction [**/FFT**[=*f*] **/DEST=*destWave***] ***windowKind*, *srcWave***

The WindowFunction operation multiplies a one-dimensional (real or complex) *srcWave* by the named window function.

By default the result overwrites *srcWave*.

Parameters

srcWave	A one-dimensional wave of any numerical type. See **ImageWindow** for windowing two-dimensional data.
windowKind	Specifies the windowing function. Choices for *windowKind* are: Bartlett, Blackman367, Blackman361, Blackman492, Blackman474, Cos1, Cos2, Cos3, Cos4, Hamming, Hanning, KaiserBessel20, KaiserBessel25, KaiserBessel30, Parzen, Poisson2, Poisson3, Poisson4, and Riemann. See **FFT** for window equations and details. The equations assume that /FFT=1.

Flags

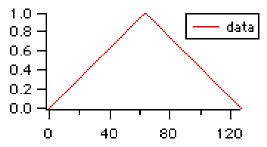
/DEST=*destWave*	Creates or overwrites *destWave* with the result of the multiplication of *srcWave* and the window function. When used in a function, the WindowFunction operation by default creates a real wave reference for the destination wave. See **Automatic Creation of WAVE References** on page IV-66 for details.
/FFT [=1]	The window interval is `0…N=numpnts(`*`srcWave`*`)`. This sets the first value of *srcWave* to zero, but not the last value. This is appropriate for windowing data in preparation for Fourier Transforms, and is the same algorithm used by **FFT**. The window interval is `0…N=numpnts(`*`srcWave`*`)-1` if /FFT is missing or /FFT=0. This sets the first and last value of *srcWave* to 0. This is the (only) algorithm that the Hanning operation uses.

Details

A “window function” alters the input data by decreasing values near the start and end of the data smoothly towards zero, so that when the FFT of the data is computed the effects of nonintegral-periodic signals are diminished. This improves the ability of the FFT to distinguish among closely-spaced frequencies. Each window function has advantages and disadvantages, usually trading off rejection of “leakage” against the ability to discriminate adjacent frequencies. For more details, see the **References**.

WindowFunction stores the window function’s normalization value (the average squared window value) in V_value. This is the value you would get from WaveStats’s `V_rms*V_rms` for a wave of *srcWave*’s length whose values were all equal to 1:

```
Make/O data = 1
WindowFunction Bartlet, data      // Bartlet allowed as synonym for Bartlett
Print V_value                 // Prints 0.330709, mean of squared window values
```



```
WaveStats/Q data
Print V_rms*V_rms              // Prints 0.330709
```

See Also

FFT, **ImageWindow**, **DPSS**

References

For more information about the use of window functions see:

Harris, F.J., On the use of windows for harmonic analysis with the discrete Fourier Transform, *Proc, IEEE, 66*, 51-83, 1978.

Wikipedia entry: <`http://en.wikipedia.org/wiki/Window_function`>.

WinList

`WinList(`*`matchStr`*`, `*`separatorStr`*`, `*`optionsStr`*`)`

The WinList function returns a string containing a list of windows selected based on the *matchStr* and *optionsStr* parameters.

Details

For a window name to appear in the output string, it must match *matchStr* and also must fit the requirements of *optionsStr*. The first character of *separatorStr* is appended to each window name as the output string is generated.

The name of each window is compared to *matchStr*, which is some combination of normal characters and the asterisk wildcard character that matches anything. For example:

"*"	Matches all window names
"xyz"	Matches window name xyz only
"*xyz"	Matches window names which end with xyz
"xyz*"	Matches window names which begin with xyz
"*xyz*"	Matches window names which contain xyz
"abc*xyz"	Matches window names which begin with abc and end with xyz

matchStr may begin with the ! character to return windows that do not match the rest of *matchStr*. For example:

"!*xyz"	Matches window names which *do not* end with xyz

The ! character is considered to be a normal character if it appears anywhere else, but there is no practical use for it except as the first character of *matchStr*.

optionsStr is used to further qualify the window. The acceptable values for *optionsStr* are:

`""`	Consider all windows.
`"WIN:"`	The target window.
`"WIN:`*`windowTypes`*`"`	Consider windows that match *windowTypes*.
`"INCLUDE:`*`includeTypes`*`"`	Consider procedure windows that match *includeTypes*. Using INCLUDE: implies WIN:128.
`"INDEPENDENTMODULE:1"`	Consider procedure windows that are part of any independent module as well as those that are not. Matching windows names are actually the window titles followed by `" [<independent module name>]"`. Using INDEPENDENTMODULE: implies WIN:128.
`"INDEPENDENTMODULE:0"`	Consider procedure windows only if they are not part of any independent module. Matching windows names are actually the window titles, which for an external file includes the file extension, such as "WMMenus.ipf". Using INDEPENDENTMODULE: implies WIN:128.
"FLT:1"	Return only panels that were created with NewPanel/FLT=1. Specifying "FLT" also implies "WIN:64". Omit FLT or use "FLT:0" to return windows that do not float (and most do not).

"FLT:2"	Return only panels that were created with NewPanel/FLT=2. Specifying "FLT" also implies "WIN:64".
"VISIBLE:1"	Return only visible windows (ignore hidden windows).

windowTypes is a literal number. The window name goes into the output string only if it passes the match test and its type is compatible with *windowTypes*. *windowTypes* is a bitwise parameter:

1:	Graphs
2:	Tables
4:	Layouts
16:	Notebooks
64:	Panels
128:	Procedure windows
512:	Help windows
4096:	XOP target windows
16384:	Camera windows in Igor Pro 7.00 or later
65536:	Gizmo windows in Igor Pro 7.00 or later

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Procedure windows and help windows don't have names. WinList returns the window title instead.

includeTypes is also a literal number. The window name goes into the output string only if it passes the match test and its type is compatible with *includeTypes*. *includeTypes* is one of:

1:	Procedure windows that are not #included.
2:	Procedure windows included by `#include "`*`someFileName`*`"`.
4:	Procedure windows included by `#include <`*`someFileName`*`>`.

or a bitwise combination of the above for more than one type of inclusion.

You can combine the `WIN`, `INCLUDE` and `INDEPENDENTMODULE` options by separating them with a comma.

When the INDEPENDENTMODULE option is used, the title of any procedure window that is part of an independent module will be followed by `" [<independent module name>]"`.

For example, if a procedure file contains:

```
#pragma IndependentModule=myIndependentModule
#include <Axis Utilities>
```

A call to WinList like this:

```
String list = WinList("* [myIndependentModule]", ";", "INDEPENDENTMODULE:1")
```

will store "Axis Utilities.ipf [myIndependentModule];" in the `list` string, along with any other procedure windows that are part of that independent module.

When the INDEPENDENTMODULE option is omitted, the returned procedure window titles do not include any independent module name suffix, and the procedure files "visible" to WinList depend on the setting of `SetIgorOption independentModuleDev` (which must be done after opening the experiment):

`SetIgorOption independentModuleDev=0`	Consider procedure windows only if they are not part of any independent module and if they are not hidden (using `#pragma hide`, for example).
`SetIgorOption independentModuleDev=1`	Consider all procedure windows including those in independent modules or hidden.

Examples

Command	Returned List
`WinList("*",";","")`	All existing non-floating windows.
`WinList("*", ";","WIN:3")`	All graph and table windows.
`WinList("Result_*", ";", "WIN:1")`	Graphs whose names start with “Result_”.
`WinList("*", ";","WIN:64,FLT:1,FLT:2")`	All floating panel windows.
`WinList("*", ";","INCLUDE:6")`	All #included procedure windows.
`WinList("*", ";","WIN:1,INCLUDE:6")`	All graphs and #included procedure windows.

See Also

Independent Modules on page IV-224. The **ChildWindowList** and **WinType** functions.

WinName

`WinName(`*`index, windowTypes`*`[, `*`visibleWindowsOnly`*`[, `*`floatKind`*`]])`

The WinName function returns a string containing the name of the *index*th window of the specified *type*, or an empty string (`""`) if no window fits the parameters.

If the optional *visibleWindowsOnly* parameter is nonzero, only visible windows are considered. Otherwise both visible and hidden windows are considered.

If the optional *floatKind* parameter is 1, only floating windows created with NewPanel/FLT=1 are considered. If *floatKind* is 2, only NewPanel/FLT=2 windows are considered. *windowTypes* must contain at least 64 (panels).

If *floatKind* is omitted or is 0 only non-floating ("normal") windows are considered.

Procedure windows don’t have names. WinName returns the procedure window title instead.

Details

index starts from zero, and returns the top-most window matching the parameters.

The window names are ordered in window-stacking order, as returned by WinList.

`DoWindow/B` moves the window to the back and changes the index needed to retrieve its name to the greatest index that returns any name.

Hiding or showing a window (with `SetWindow hide=1` or `Notebook visible=0` or by manual means) does not affect the index associated with the window.

windowTypes is a bitwise parameter:

1:	Graphs.
2:	Tables.
4:	Layouts.
16:	Notebooks.
64:	Panels.
128:	Procedure windows.
4096:	XOP target windows.
16384:	Camera windows in Igor Pro 7.00 or later
65536:	Gizmo windows in Igor Pro 7.00 or later

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Examples

```
Print WinName(0,1)          // Prints the name of the top graph.
Print WinName(0,3)          // Prints the name of the top graph or table.
```

```
String win=WinName(0,1)    // The name of the top visible graph.
SetWindow $win hide=1      // Hide the graph (it may already be hidden).
Print WinName(0,1)         // Prints the name of the now-hidden graph.
Print WinName(0,1,1)       // Prints the name of the top visible graph.
Print WinName(0,64,1,1)    // Name of the top visible NewPanel/FLT=1 window.
```

See Also

WinList, **DoWindow** (/F and /B flags), **SetWindow** (hide keyword), **Notebook (Miscellaneous)** (visible keyword), **NewPanel** (/FLT flag).

WinRecreation

WinRecreation(*winStr*, *options*)

The WinRecreation function returns a string containing the window recreation macro (or style macro) for the named window.

Parameters

winStr is the name of a graph, table, page layout, panel, notebook, Gizmo, camera, or XOP target window or the title of a procedure window or help file. If *winStr* is "" and options is 0 or 1, information for the top graph, table, page layout, panel, notebook, or XOP target window is returned.

As of Igor Pro 7.00, *winStr* may be a subwindow path. The returned recreation macro is generated as if the subwindow were extracted from its host as a standalone window. See **Subwindow Syntax** on page III-87 for details on forming the subwindow path.

The meaning of *options* depends on the type of window as described in the following sections.

Target Window Details

Target windows include graphs, tables, page layouts, panels, notebooks, and XOP target windows.

If *options* is 0, WinRecreation returns the window recreation macro.

If *options* is 1, WinRecreation returns the style macro or an empty string if the window does not support style macros.

Graphs Details

If *options* is 2, WinRecreation returns a recreation macro in which all occurrences of wave names are replaced with an ID number having the form ##<number>## (for instance, ##25##). These ID numbers can be found easily using the **strsearch** function. This is intended for applications that need to alter the recreation macro by replacing wave names with something else, usually other wave names. The ID numbers are the same as those returned by the **GetWindow** operation with the wavelist keyword.

Graphs and Panels Details

If *options* is 4, WinRecreation returns the window recreation macro without the default behavior of causing the graph to revert to “normal” mode (as if the GraphNormal operation had been called). This allows the use of WinRecreation when a graph or panel is in drawing tools mode without exiting that mode. For windows other than graphs or panels, this is equivalent to an *options* value of 0.

Notebooks Details

If *options* is -1, WinRecreation returns the same text that the Generate Commands menu item would generate with the Selected paragraphs radio button selected and all the checkboxes selected (includes text commands).

If *options* is 0, WinRecreation returns the same text that the Generate Commands menu item would generate with the Entire document radio button selected and all the checkboxes *except* “Generate text commands” selected).

If *options* is 1, WinRecreation returns the same text that the Generate Commands menu item would generate with the Entire document radio button selected and all the checkboxes selected (includes text commands).

Regardless of the value of *options* the text returned by WinRecreation for notebook always ends with 5 lines of file-related information formatted as comments:

```
// File Name: MyNotebook.txt
// Path: "Macintosh HD:Desktop Folder:"
// Symbolic Path: home
// Selection Start: paragraph 100, position 31
// Selection End: paragraph 100, position 31
```

Help Windows Details

WinRecreation returns the same 5 lines of file-related information as described above for notebooks.

Set *options* to -3 to ensure that *winStr* is interpreted as a help window title (help windows have only titles, not window names).

Procedures Details

WinRecreation returns the same 5 lines of file-related information as described above for notebooks.

Set *options* to -2 to ensure that *winStr* is interpreted as a procedure window title (procedure windows have only titles, not window names).

If `SetIgorOption IndependentModuleDev=1` is in effect, *winStr* can also be a procedure window title followed by a space and, in brackets, an independent module name. In such cases WinRecreation returns text from or information about the specified procedure file which is part of that independent module. (See **Independent Modules** on page IV-224 for independent module details.)

For example, in an experiment containing:

```
#pragma IndependentModule=myIM
#include <Axis Utilities>
```

code like this:

```
String text=WinRecreation("Axis Utilities.ipf [myIM]",-2)
```

will return the file-related information for the Axis Utilities.ipf procedure window, which is normally a hidden part of the myIM independent module.

To get the text content of a procedure window, use the **ProcedureText** function.

Examples

```
WinRecreation("Graph0",0)     // Returns recreation macro for Graph0.
WinRecreation("",1)                     // Style macro for top window.
String win= WinName(0,16,1)                // top visible notebook
String str= WinRecreation(str,-1)          // Selected Text commands
Variable line= itemsInList(str,"\r")-5     // First file info line
Print StringFromList(line, str,"\r")       // Print File Name:
Print StringFromList(line+1, str,"\r")     // Print Path:
Print StringFromList(line+2, str,"\r")     // Print Symbolic Path:
Print StringFromList(line+3, str,"\r")     // Selection Start:
Print StringFromList(line+4, str,"\r")     // Selection End:
```

See Also

Saving a Window as a Recreation Macro on page II-42.

WinType

`WinType(`*`winNameStr`*`)`

The WinType function returns a value indicating the type of the named window.

Details

winNameStr is a string or string expression containing the name of a window or subwindow, or `""` to signify the target window. When identifying a subwindow with *winNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

WinType returns the following values:
0: No window by that name.

1:	Graph
2:	Table
3:	Layout
5:	Notebook
7:	Panel
13:	XOP target window

15:	Camera window in Igor Pro 7.00 or later
17:	Gizmo window in Igor Pro 7.00 or later

Because command and procedure windows do not have *names* (they only have *titles*), WinType can not even be asked about those windows.

See Also

The **WinName**, **ChildWindowList**, and **WinList** functions.

WMAxisHookStruct

See **NewFreeAxis** for further explanation of WMAxisHookStruct.

```
Structure WMAxisHookStruct
   char win[200]       // Host window or subwindow name
   char axName[32]     // Name of axis
   char mastName[32]   // Name of controlling axis or ""
   char units[50]      // Axis units.
   Variable min        // Current axis range minimum value
   Variable max        // Current axis range maximum value
EndStructure
```

WMBackgroundStruct

See **CtrlNamedBackground**, **Background Tasks** on page IV-298, and **Preemptive Background Task** on page IV-314 for further explanation of WMBackgroundStruct.

```
Structure WMBackgroundStruct
   char name[32]          // Background task name
   UInt32 curRunTicks     // Tick count when task was called
   Int32 started          // TRUE when CtrlNamedBackground start is issued
   UInt32 nextRunTicks    // Precomputed value for next run
                          // but user functions may change this
EndStructure
```

WMButtonAction

This structure is passed to action procedures for button controls created using the **Button** operation.

```
Structure WMButtonAction
   char ctrlName[32]      // Control name
   char win[200]          // Host window or subwindow name
   STRUCT Rect winRect    // Local coordinates of host window
   STRUCT Rect ctrlRect   // Enclosing rectangle of the control
   STRUCT Point mouseLoc  // Mouse location
   Int32 eventCode        // See details below
   Int32 eventMod         // See Control Structure eventMod Field on page III-390
   String userData        // Primary unnamed user data.
   Int32 blockReentry     // See Control Structure blockReentry Field on page III-390
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMButtonAction eventCode Field

Action functions should respond only to documented eventCode values. Other event codes may be added in the future.

The event code passed to the button action procedure has the following meaning:

`eventCode`	Event
-1	Control being killed
1	Mouse down
2	Mouse up
3	Mouse up outside control
4	Mouse moved

eventCode	**Event**
5	Mouse enter
6	Mouse leave
7	Mouse dragged while outside the control

Events 2 and 3 happen only after event 1.

Events 4, 5, and 6 happen only when the mouse is over the control but happen regardless of the mouse button state.

Event 7 happens only when the mouse is pressed inside the control and then dragged outside.

WMCheckboxAction

This structure is passed to action procedures for checkbox controls created using the **CheckBox** operation.

```
Structure WMCheckboxAction
   char ctrlName[32]       // Control name
   char win[200]           // Host window or subwindow name
   STRUCT Rect winRect     // Local coordinates of host window
   STRUCT Rect ctrlRect    // Enclosing rectangle of the control
   STRUCT Point mouseLoc   // Mouse location
   Int32 eventCode         // See details below
   Int32 eventMod          // See Control Structure eventMod Field on page III-390
   String userData         // Primary unnamed user data
   Int32 blockReentry      // See Control Structure blockReentry Field on page III-390
   Int32 checked           // Checkbox state
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMCheckboxAction eventCode Field

Action functions should respond only to documented eventCode values. Other event codes may be added in the future.

The event code passed to the checkbox action procedure has the following meaning:

eventCode	**Event**
-1	Control being killed
2	Mouse up

WMCustomControlAction

This structure is passed to action procedures for custom controls created using the **CustomControl** operation.

```
Structure WMCustomControlAction
   char ctrlName[32]       // Control name
   char win[200]           // Host window or subwindow name
   STRUCT Rect winRect     // Local coordinates of host window
   STRUCT Rect ctrlRect    // Enclosing rectangle of the control
   STRUCT Point mouseLoc   // Mouse location
   Int32 eventCode         // See details below
   Int32 eventMod          // See Control Structure eventMod Field on page III-390
   String userData         // Primary unnamed user data
   Int32 blockReentry      // See Control Structure blockReentry Field on page III-390
   Int32 missedEvents      // TRUE when events occurred but the user
                           // function was not available for action
   Int32 mode              // General purpose

   // Used only when eventCode==kCCE_frame
   Int32 curFrame          // Input and output

   // Used when eventCode is kCCE_mousemoved, kCCE_mouseenter or kCCE_mouseleave
   Int32 needAction        // See below for details

   // These fields are valid only with value=varName
   Int32 isVariable        // TRUE if varName is a variable
   Int32 isWave            // TRUE if varName referenced a wave
   Int32 isString          // TRUE if varName is a String type
```

```
    NVAR nVal               // Valid if isVariable and not isString
    SVAR sVal               // Valid if isVariable and isString
    WAVE nWave              // Valid if isWave and not isString
    WAVE/T sWave            // Valid if isWave and not isString
    Int32 rowIndex          // If isWave, this is the row index
                            // unless rowLabel is not empty
    char rowLabel[32]       // Wave row label

    // These fields are valid only when eventCode==kCCE_char
    Int32 kbChar            // Keyboard key character code
    Int32 specialKeyCode    // See Keyboard Events on page IV-281 - Added in Igor Pro 7
    char keyText[16]        // UTF-8 string representing key struck - Added in Igor Pro 7
    Int32 kbMods            // Keyboard key modifiers bit field. See details below.
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMCustomControl eventMod Field

When determining the state of the eventCode member in the WMCustomControlAction structure, the various values you use are listed in the following table. You can define the kCCE symbolic constants by adding this to your procedure file:

```
#include <CustomControl Definitions>
```

Event Code	Description
`kCCE_mousedown = 1`	Mouse down in control.
`kCCE_mouseup = 2`	Mouse up in control.
`kCCE_mouseup_out = 3`	Mouse up outside control.
`kCCE_mousemoved = 4`	Mouse moved (happens only when mouse is over the control).
`kCCE_enter = 5`	Mouse entered control.
`kCCE_leave = 6`	Mouse left control.
`kCCE_mouseDraggedOutside = 7`	The mouse moved while it was outside the control. This event is delivered only after the mouse is pressed inside the control and dragged outside. While the mouse is inside the control, kCCE_mousemoved is delivered whether the mouse button is up or down.
`kCCE_draw = 10`	Time to draw custom content.
`kCCE_mode = 11`	Sent when executing `CustomControl` *`name`*`, mode=`*m*.
`kCCE_frame = 12`	Sent before drawing a subframe of a custom picture.
`kCCE_dispose = 13`	Sent as the control is killed.
`kCCE_modernize = 14`	Sent when dependency (variable or wave set by value=*varName* parameter) fires. It will also get draw events, which probably don't need a response.
`kCCE_tab = 15`	Sent when user tabs into the control. If you want keystrokes (`kCCE_char`), then set `needAction`.
`kCCE_char = 16`	Sent on keyboard events. Stores the keyboard character in `kbChar` and modifiers bit field is stored in `kbMods`. Sets `needAction` if key event was used and requires a redraw.
`kCCE_drawOSBM = 17`	Called after drawing *pict* from picture parameter into an offscreen bitmap. You can draw custom content here.
`kCCE_idle = 18`	Idle event typically used to blink insertion points etc. Set `needAction` to force the control to redraw. Sent only when the host window is topmost.

WMCustomControl needAction Field

The meaning of needAction depends on the event.

Events kCCE_mousemoved, kCCE_enter, kCCE_leave, and kCCE_mouseDraggedOutside set needAction to TRUE to force redraw, which is normally not done for these events.

Events kCCE_tab and kCCE_mousedown set needAction to TRUE to request keyboard focus (and get kCCE_char events).

Event kCCE_idle sets needAction to TRUE to request redraw.

WMCustomControl kbMods Field

Bit 0:	Command (*Macintosh*)
Bit 1:	Shift
Bit 2:	Alpha Lock. Not supported in Igor7 or later.
Bit 3:	Option (*Macintosh*) or Alt (*Windows*)
Bit 4:	Control (*Macintosh*) or Windows key (*Windows*).

WMDrawUserShapeStruct

See **DrawUserShape** for further explanation of WMDrawUserShapeStruct.

```
Structure WMDrawUserShapeStruct
   char action[32]        // Input: Specifies what action is requested.

   SInt32 options         // Input: Value from /MO flag.
                          // Output: When action is getInfo, set bits as follows:
                          // Set bit 0 if the shape should behave like a simple line.
                          //     When resizing end-points, you will get live updates.
                          // Set bit 1 if the shape is to act like a button;
                          //     You will get mouse down in normal operate mode.
                          // Set bit 2 to get roll-over action.
                          //     You will get hitTest action and
                          //     if 1 is returned, the mouse will be captured.

   SInt32 operateMode     // Input: If 0, the shape is being edited;
                          // if 1, normal operate mode
                          // (only if options bit 1 or 2 was set during getInfo).

   PointF mouseLoc        // Input: The location of the mouse in normalized coordinates.

   SInt32 doSetCursor     // Output: If action is hitTest, set true
                          // to use the following cursor number.
                          // Also used for mouseMoved in rollover mode.

   SInt32 cursorCode      // Output: If action is hitTest and doSetCursor is set,
                          // then set this to the desired Igor cursor number.

   double x0,y0,x1,y1     // Input: Coordinates of the enclosing rectangle of the shape.

   RectF objectR          // Input: Coordinates of the enclosing rectangle of the shape
                          // in device units.

   char winName[MAX_HostChildSpec+1] // Input: Full path to host subwindow

   // Information about the coordinate system
   Rect drawRect          // Draw rect in device coordinates
   Rect plotRect          // In a graph, this is the plot area
   char xcName[MAX_OBJ_NAME+1]   // Name of X coordinate system, may be axis name
   char ycName[MAX_OBJ_NAME+1]   // Name of Y coordinate system, may be axis name

   double angle           // Input: Rotation angle, use when displaying text
   String textString      // Input: Use or ignore; special output for "getInfo"
   String privateString   // Input and output: Maintained by Igor
                          // but defined by user function;
                          // may be binary; special output for "getInfo"
EndStructure
```

WMFitInfoStruct

See **The `WMFitInfoStruct` Structure** on page III-231 for further explanation of WMFitInfoStruct.

```
Structure WMFitInfoStruct
   char IterStarted       // Nonzero on the first call of an iteration
   char DoingDestWave     // Nonzero when called to evaluate autodest wave
   char StopNow           // Fit function sets this to nonzero to
                          // indicate that a problem has occurred
                          // and fitting should stop
   Int32 IterNumber       // Number of iterations completed
   Int32 ParamPerturbed   // See The WMFitInfoStruct Structure on page III-231
EndStructure
```

WMGizmoHookStruct

See **Gizmo Named Hook Functions** on page II-384 for further explanation of WMGizmoHookStruct.

```
Structure WMGizmoHookStruct
   Int32 version
   char winName[32]
   char eventName[32]
   Int32 width
   Int32 height
   Int32 mouseX
   Int32 mouseY
   Variable xmin
   Variable xmax
   Variable ymin
   Variable ymax
   Variable zmin
   Variable zmax
   Variable eulerA
   Variable eulerB
   Variable eulerC
   Variable wheelDx
   Variable wheelDy
EndStructure
```

WMListboxAction

This structure is passed to action procedures for listbox controls created using the **ListBox** operation.

```
Structure WMListboxAction
   char ctrlName[32]       // Control name
   char win[200]           // Host window or subwindow name
   STRUCT Rect winRect     // Local coordinates of host window
   STRUCT Rect ctrlRect    // Enclosing rectangle of the control
   STRUCT Point mouseLoc   // Mouse location
   Int32 eventCode         // See details below
   Int32 eventMod          // See Control Structure eventMod Field on page III-390
   String userData         // Primary unnamed user data
   Int32 blockReentry      // See Control Structure blockReentry Field on page III-390
   Int32 eventCode2        // Obsolete
   Int32 row               // Selection row. See ListBox for details.
   Int32 col               // Selection column. See ListBox for details.
   WAVE/T listWave         // List wave specified by ListBox command
   WAVE selWave            // Selection wave specified by ListBox command
   WAVE colorWave          // Color wave specified by ListBox command
   WAVE/T titleWave        // Title wave specified by ListBox command
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMListboxAction eventCode Field

Action functions should respond only to documented eventCode values. Other event codes may be added in the future.

The event code passed to the listbox action procedure has the following meaning:

eventCode	Meaning
-1	Control being killed.
1	Mouse down.
2	Mouse up.

eventCode	Meaning
3	Double click.
4	Cell selection (mouse or arrow keys).
5	Cell selection plus Shift key.
6	Begin edit.
7	Finish edit.
8	Vertical scroll. See **Scroll Event Warnings** on page V-434.
9	Horizontal scroll by user or by the hScroll=*h* keyword.
10	Top row set by row=*r* or first column set by col=*c* keywords.
11	Column divider resized.
12	Keystroke, character code is place in row field. See **Note on Keystroke Event** on page V-435.
13	Checkbox was clicked. This event is sent after selWave is updated.

WMListboxAction row and col Fields

The row field is the row number of selection in interior or -1 if in title area.

The col field is the column number of the selection.

The meanings of row and col are different for eventCodes 8 through 11:

Code	row	col
8	top visible row	horiz shift in pixels.
9	top visible row	horiz shift (user scroll).
9	-1	horiz shift (hScroll keyword).
10	top visible row	-1 (row keyword).
10	-1	first visible col (col keyword).
11	column shift	column resized by user.

If eventCode is 11, row is the horizontal shift in pixels of the column col that was resized, not the total horizontal shift of the list as reported in V_horizScroll by **ControlInfo**. If row is negative, the divider was moved to the left. col=0 corresponds to adjusting the divider on the right side of the first column. Use ControlInfo to get a list of all column widths.

WMMarkerHookStruct

See **Custom Marker Hook Functions** on page IV-289 for further explanation of WMMarkerHookStruct.

```
Structure WMMarkerHookStruct
   Int32 usage                // 0= normal draw, 1= legend draw
   Int32 marker               // Marker number minus start
   float x, y                 // Location of desired center of marker
   float size                 // Half width/height of marker
   Int32 opaque               // 1 if marker should be opaque
   float penThick             // Stroke width
   STRUCT RGBColor mrkRGB     // Fill color
   STRUCT RGBColor eraseRGB   // Background color
   STRUCT RGBColor penRGB     // Stroke color
   WAVE ywave                 // Trace's y wave
   double ywIndex             // Point number on ywave where marker is being drawn
EndStructure
```

WMPopupAction

This structure is passed to action procedures for popup menu controls created using the **PopupMenu** operation.

```
Structure WMPopupAction
   char ctrlName[32]       // Control name
   char win[200]           // Host window or subwindow name
   STRUCT Rect winRect     // Local coordinates of host window
   STRUCT Rect ctrlRect    // Enclosing rectangle of the control
   STRUCT Point mouseLoc   // Mouse location
   Int32 eventCode         // See details below
   Int32 eventMod          // See Control Structure eventMod Field on page III-390
   String userData         // Primary unnamed user data
   Int32 blockReentry      // See Control Structure blockReentry Field on page III-390
   Int32 popNum            // Item number currently selected (1-based)
   char popStr[MAXCMDLEN]  // Contents of current popup item
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMPopupAction eventCode Field

Action functions should respond only to documented eventCode values. Other event codes may be added in the future.

The event code passed to the pop-up menu action procedure has the following meaning:

eventCode	**Event**
-1	Control being killed
2	Mouse up

WMSetVariableAction

This structure is passed to action procedures for SetVariable controls created using the **SetVariable** operation.

```
Structure WMSetVariableAction
   char ctrlName[32]       // Control name
   char win[200]           // Host window or subwindow name
   STRUCT Rect winRect     // Local coordinates of host window
   STRUCT Rect ctrlRect    // Enclosing rectangle of the control
   STRUCT Point mouseLoc   // Mouse location
   Int32 eventCode         // See details below
   Int32 eventMod          // See Control Structure eventMod Field on page III-390
   String userData         // Primary unnamed user data
   Int32 blockReentry      // See Control Structure blockReentry Field on page III-390
   Int32 isStr             // TRUE for a string variable
   Variable dval           // Numeric value of variable
   char sval[MAXCMDLEN]    // Value of variable as a string
   char vName[MAX_OBJ_NAME+2 + (MAXDIMS * (MAX_OBJ_NAME+5)) + 1]
   WAVE svWave             // Valid if using wave
   Int32 rowIndex                    // Row index for a wave if rowLabel is empty
   char rowLabel[MAX_OBJ_NAME+1]     // Wave row dimension label
   Int32 colIndex                    // Column index for a wave if colLabel is empty
   char colLabel[MAX_OBJ_NAME+1]     // Wave column dimension label
   Int32 layerIndex                  // Layer index for a wave if layerLabel is empty
   char layerLabel[MAX_OBJ_NAME+1]   // Wave layer label
   Int32 chunkIndex                  // Chunk index for a wave if chunkLabel is empty
   char chunkLabel[MAX_OBJ_NAME+1]   // Wave chunk label
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMSetVariableAction eventCode Field

Action functions should respond only to documented eventCode values. Other event codes may be added in the future.

The event code passed to the SetVariable action procedure has the following meaning:

eventCode	Meaning
-1	Control being killed
1	Mouse up
2	Enter key
3	Live update
4	Mouse scroll wheel up
5	Mouse scroll wheel down
6	Value changed by dependency update
7	Begin edit (Igor7 or later)
8	End edit (Igor7 or later)

Code -1 is never sent to an old-style (non-structure parameter) action procedure.

Codes 4 and 5 are sent only for string SetVariables or numeric SetVariables whose increment setting is zero.

For numeric SetVariables whose increment is non-zero, the mouse scroll wheel acts like a mouse click on the increment or decrement arrows.

Code 6 is by default sent to only structure-based action procedures.

Use SetIgorOption EnableSVE6=0 to disable sending this event at all and EnableSVE6=2 to send the event to both structure-based and old-style SetVariable action procedures. The default for EnableSVE6 is =1.

WMSliderAction

This structure is passed to action procedures for slider controls created using the **Slider** operation.

```
Structure WMSliderAction
   char ctrlName[32]       // Control name
   char win[200]           // Host window or subwindow name
   STRUCT Rect winRect     // Local coordinates of host window
   STRUCT Rect ctrlRect    // Enclosing rectangle of the control
   STRUCT Point mouseLoc   // Mouse location
   Int32 eventCode         // See details below
   Int32 eventMod          // See Control Structure eventMod Field on page III-390
   String userData         // Primary unnamed user data
   Int32 blockReentry      // See Control Structure blockReentry Field on page III-390
   Variable curval         // Value of slider
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMSliderAction eventCode Field

Action functions should respond only to documented eventCode values. Other event codes may be added in the future.

The event code passed to the slider action procedure is a bitwise value with the following meaning:

eventCode	Meaning
Bit 0:	Value set
Bit 1:	Mouse down
Bit 2:	Mouse up
Bit 3:	Mouse moved

If eventCode is -1, the control is being killed.

WMTabControlAction

This structure is passed to action procedures for tab controls created using the **TabControl** operation.

```
Structure WMTabControlAction
   char ctrlName[32]      // Control name
   char win[200]          // Host window or subwindow name
   STRUCT Rect winRect    // Local coordinates of host window
   STRUCT Rect ctrlRect   // Enclosing rectangle of the control
   STRUCT Point mouseLoc  // Mouse location
   Int32 eventCode        // See details below
   Int32 eventMod         // See Control Structure eventMod Field on page III-390
   String userData        // Primary unnamed user data
   Int32 blockReentry     // See Control Structure blockReentry Field on page III-390
   Int32 tab              // Tab number
EndStructure
```

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

WMTabControlAction eventCode Field

Action functions should respond only to documented eventCode values. Other event codes may be added in the future.

The event code passed to tab control action procedure has the following meaning:

eventCode	Event
-1	Control being killed
2	Mouse up

WMWinHookStruct

See **Named Window Hook Functions** on page IV-277 for further explanation of WMWinHookStruct.

```
Structure WMWinHookStruct
   char winName[200]      // Host window or subwindow name
   STRUCT Rect winRect    // Local coordinates of the affected (sub)window
   STRUCT Point mouseLoc  // Mouse location
   Variable ticks         // Tick count when event happened
   Int32 eventCode        // See Named Window Hook Events on page IV-277
   char eventName[32]     // See Named Window Hook Events on page IV-277
   Int32 eventMod         // See Control Structure eventMod Field on page III-390
   char menuName[256]     // Name of the menu item as for SetIgorMenuMode
   char menuItem[256]     // Text of the menu item as for SetIgorMenuMode
   char traceName[32]     // See Named Window Hook Functions on page IV-277
   char cursorName[2]     // Cursor name A through J
   Variable pointNumber   // See Named Window Hook Functions on page IV-277
   Variable yPointNumber  // See Named Window Hook Functions
   Int32 isFree           // 1 if the cursor is not attached to anything
   Int32 keycode          // ASCII value of key struck
   Int32 specialKeyCode   // See Keyboard Events on page IV-281 - Igor Pro 7 or later
   char keyText[16]       // UTF-8 string representing key struck - Igor Pro 7 or later
   char oldWinName[32]    // Simple name of the window or subwindow
   Int32 doSetCursor      // Set to 1 to change cursor to cursorCode
   Int32 cursorCode       // See Setting the Mouse Cursor on page IV-282
   Variable wheelDx       // Vertical lines to scroll
   Variable wheelDy       // Horizontal lines to scroll
EndStructure
```

wnoise

wnoise(*shape*, *scale*)

The wnoise function returns a pseudo-random value from the two-parameter Weibull distribution characterized by the *shape* and *scale,* the respective *gamma* and *alpha* parameters. The two-parameter Weibull probability distribution function is

$$f(x;\alpha,\gamma)=\frac{\gamma}{\alpha}x^{\gamma-1}\exp\left[-\frac{1}{\alpha}x^{\gamma}\right]\qquad \begin{array}{l} x\geq 0 \\ \alpha>0 \\ \gamma>0 \end{array}$$

The mean of the Weibull distribution is

$$\alpha^{\frac{1}{\gamma}}\Gamma\left(1+\frac{1}{\gamma}\right),$$

and the variance is

$$\alpha^{\frac{2}{\gamma}}\Gamma\left(1+\frac{2}{\gamma}\right)-\alpha^{\frac{2}{\gamma}}\left[\Gamma\left(1+\frac{1}{\gamma}\right)\right]^{2}.$$

Note that this definition of the PDF uses different scaling than the one used in StatsWeibullPDF. To match the scaling of StatsWeibullPDF multiply the result from Wnoise by the factor scale^(1-1/shape).

The random number generator initializes using the system clock when Igor Pro starts. This almost guarantees that you will never repeat a sequence. For repeatable "random" numbers, use **SetRandomSeed**. The algorithm uses the Mersenne Twister random number generator.

See Also

The **SetRandomSeed** operation.

Noise Functions on page III-344.

Chapter III-12, **Statistics** for a function and operation overview.

x

`x`

The x function returns the scaled row index for the current point of the destination wave in a wave assignment statement. This is the same as the X value if the destination wave is a vector (1D wave).

Details

Outside of a wave assignment statement, x acts like a normal variable. That is, you can assign a value to it and use it in an expression.

See Also

The **p** function and **Waveform Arithmetic and Assignments** on page II-69.

x2pnt

`x2pnt(`*`waveName, x1`*`)`

The x2pnt function returns the integer point number on the wave whose X value is closest to *x1*.

For higher dimensions, use **ScaleToIndex**.

See Also

DimDelta, **DimOffset**, **pnt2x**, **ScaleToIndex**

For an explanation of waves and X scaling, see **Changing Dimension and Data Scaling** on page II-63.

xcsr

`xcsr(`*`cursorName`*` [, `*`graphNameStr`*`])`

The xcsr function returns the X value of the point which the named cursor (A through J) is on in the top or named graph.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

The result is derived from the wave that the cursor is on, not from the X axis of the graph. If the wave is displayed as an XY pair, the X axis and the wave's X scaling will usually be different.

See Also

The **hcsr**, **pcsr**, **qcsr**, **vcsr**, and **zcsr** functions.

Programming With Cursors on page II-249.

XLLoadWave

`XLLoadWave [`*`flags`*`] [`*`fileNameStr`*`]`

The XLLoadWave operation loads data from the named Excel .xls or .xlsx file into waves.

Parameters

The file to be loaded is specified by *fileNameStr* and /P=*pathName* where *pathName* is the name of an Igor symbolic path. *fileNameStr* can be a full path to the file, in which case /P is not needed, a partial path relative to the folder associated with *pathName*, or the name of a file in the folder associated with *pathName*. If XLLoadWave can not determine the location of the file from *fileNameStr* and *pathName*, it displays a dialog allowing you to specify the file.

If you use a full or partial path for *fileNameStr*, see **Path Separators** on page III-401 for details on forming the path.

If *fileNameStr* is omitted or is "", or if the /I flag is used, XLLoadWave presents an Open File dialog from which you can choose the file to load.

Flags

/A	Automatically assigns arbitrary wave names using "wave" as the base name. Skips names already in use.
/A=*baseName*	Same as /A but it automatically assigns wave names of the form *baseName*0, *baseName*1.
/C=*columnType*	XLLoadWave will use the Deduce from Row method of determining the type of the Excel file columns, using the row specified by *columnType* to deduce column types. See **Deduce from row** on page II-141.
/COLT=*columnTypeStr*	*columnTypeStr* specifies how XLLoadWave should treat each column. For example, "1T3N" means 1 text column followed by 3 numeric columns. See **Determining Wave Types** on page V-964.
/D	Creates double-precision floating point waves. If omitted, XLLoadWave creates single-precision floating point waves.

/F=f	New programming should use the /T flag instead of the /D, /L and /F flags. *f* specifies the data format of the file: *f*=1: Signed integer (8, 16, 32 bits allowed) *f*=2: Creates double-precision waves *f*=3: Floating point (default, 32, 64 bits allowed)
/I	Forces XLLoadWave to display an Open File dialog even if the file is fully specified via /P and *fileNameStr*.
/J=*infoMode*	If infoMode is 1, 2 or 3, XLLoadWave does not load the file but instead returns information about the worksheets within the workbook via the string variable S_value. See **Getting Information About the Excel File** on page V-965.
/K=*k*	Discards waves with fewer than *k* points. For historical reasons, *k* defaults to 2.
/N	Same as /A except that, instead of choosing names that are not in use, it overwrites existing waves.
/N=*baseName*	Same as /N except that it automatically assigns wave names of the form *baseName0*, *baseName1*.
/NAME=*nameList*	*nameList* is a semicolon-separated list of wave names to be used for the loaded waves. See **Wave Names** on page V-963 for details.
/O	Overwrites existing waves in case of a name conflict.
/P=*pathName*	Specifies the folder to look in for *fileNameStr*. *pathName* is the name of an existing symbolic path.
/Q	Suppresses the normal messages in the history area.
/R=(*cell1*,*cell2*)	Restricts loading to the specified cells, e.g. /R=(A3,D21). Row and column numbers start from 1. The /R flag supports an optional extra parameter that should be used only in very rare cases. XLLoadWave reads the range of defined cells from the file itself and clips *cell1* and *cell2* to that range. In very rare cases the file does not accurately identify the range of defined cells so the clipping prevents loading cells that exist in the file. In this rare case, use /R=(*cell1*, *cell2*,1). The last parameter tells XLLoadWave to skip the clipping. If you specify incorrect values for cell1 or *cell2* you may get errors or garbage results.
/S=*sheetNameStr*	Specifies which worksheet to load from a workbook file. If you omit /S=*sheetNameStr*, or if *sheetNameStr* is "", XLLoadWave loads the first worksheet in the workbook.
/T	Automatically creates a table of loaded waves.
/V=*v*	Controls the handling of blanks at the end of a column. *v*=0: XLLoadWave leaves blanks at the end of a column in the Igor wave. *v*=1: XLLoadWave removes blanks at the end of a column from the Igor wave. If the column has fewer than two remaining points, it is not loaded into a wave. This is the default mode that is used if you omit /V.
/W=*w*	*w* specifies the row in which XLLoadWave will look for wave names. The first row is row number 1.

Wave Names

The names of the loaded waves are determined by the /A, /N, /W and /NAME flags. If all of the flags are omitted, default names, like ColumnA and ColumnB, are used.

If /W=w is present, names are loaded from row *w* of the worksheet and then converted to standard Igor names by replacing spaces and punctuation characters with underscores.

If /NAME=*nameList* is present, the wave names come from *nameList*, a semicolon-separated list of names. For example:

```
/NAME="StartTime;UnitA;UnitB;"
```

The names in *nameList* can be standard or liberal names. For example, this specifies names two standard names and one liberal name which contains a space:

```
/NAME="Signal;Ambient Temp;Response;"
```

If a name in the list is _skip_, the corresponding Excel column is skipped. For example, this would load the first and third columns and skip the second:

```
/NAME="Signal;_skip_;Response;"
```

If a name in the list is empty, the name used for the corresponding wave is as it would be if /NAME were omitted. This can be used to skip columns while taking wave names from the spreadsheet for loaded columns. In this example, the names of the first and third waves would be determined by row 1 of the spreadsheet while the second column would be skipped:

```
/W=1 /NAME=";_skip_;;"
```

The /N flag instructs Igor to automatically name new waves "wave", or *baseName* if /N=*baseName* is used, plus a number. The number starts from zero and increments by one for each wave loaded from the file. If the resulting name conflicts with an existing wave, the existing wave is overwritten.

The /A flag is like /N except that it skips names already in use.

/NAME overrides /W. /A or /N overrides both /NAME and /W.

No matter how the wave names are generated, if there is a name conflict and overwrite is off (/O is omitted), a unique name is generated. See **XLLoadWave and Wave Names** on page II-142 for further details.

Determining Wave Types

The /C or /COLT flag tells XLLoadWave how to decide what kind of wave, numeric, text, or date/time, to make for each Excel column.

Using /C=*columnType* causes XLLoadWave to use the Deduce from Row method of determining the type of the Excel file columns. *columnType* is the Excel row number that XLLoadWave should use to make the deduction.

Using /COLT=*columnTypeStr* causes XLLoadWave treat the columns based on the *columnTypeStr* parameter. If *columnTypeStr* is "N", XLLoadWave uses the Treat all Columns as Numeric method. If *columnTypeStr* is "T", XLLoadWave uses the Treat all Columns as Text method. If *columnTypeStr* is "D", XLLoadWave uses the Treat all Columns as Date method.

For any other value of columnTypeStr , XLLoadWave uses the Use Column Type String method. For example, "1T5N" tells XLLoadWave to create a text wave for the first column and numeric waves for the next 5 or more columns.

If you omit /C and /COLT, XLLoadWave uses the Treat all Columns as Numeric method.

See **What XLLoadWave Loads** on page II-140 for further details.

Output Variables

XLLoadWave sets the followin output variables:

`V_flag`	Number of waves loaded.
`S_fileName`	Name of the file being loaded.
`S_path`	File system path to the folder containing the file.
`S_waveNames`	Semicolon-separated list of the names of loaded waves.
`S_worksheetName`	Name of the loaded worksheet within the workbook file.
`S_value`	Set only if you use the /J flag. See **Getting Information About the Excel File** below.

S_path uses Macintosh path syntax (e.g., “`hd:FolderA:FolderB:`”), even on Windows. It includes a trailing colon.

When XLLoadWave presents an Open File dialog and the user cancels, V_flag is set to 0 and S_fileName is set to `""`.

Getting Information About the Excel File

The /J flag allows you to get information about an Excel file without actually loading it.

If *infoMode* is 1, XLLoadWave does not load the file but instead returns a semicolon-separated list of the names of the worksheets within the workbook via the string variable S_value.

If *infoMode* is 2, XLLoadWave does not load the file but instead returns information about the first worksheet or the worksheet specified by /S via the string variable S_value. The format of the returned information is:

```
NAME:<worksheet name>;FIRSTROW:<first row>;FIRSTCOL:<first col>;LASTROW:<last
    row>;LASTCOL:<last col>;
```

<first row> and <last row> are 1-based row numbers. <first col> and <last col> are 1-based column numbers; 1 refers to Column A. These refer to the defined rows and columns in the worksheet even if some or all cells are blank. If <last col> is zero, this means that there are no defined cells in the worksheet.

If *infoMode* is 3, XLLoadWave does not load the file but instead returns information about the first worksheet or the worksheet specified by /S via the string variable S_value. The format of the returned information is:

```
NAME:<worksheet name>;FIRST:<first cell>;LAST:<last cell>;
```

<first cell> and <last cell> are expressed in standard Excel notation (A1, B24, etc.). These refer to the defined rows and columns in the worksheet even if some or all cells are blank. If <last cell> is "@0", this means that there are no defined cells in the worksheet.

Use the **StringByKey**, **NumberByKey** functions to extract the information from S_value. If you use these functions, your code won't break if we later add a keyword/value pair to the returned information.

Examples

Old versions of Excel came with a number of sample files. One of them was called “Instrument Data”. The following procedure loads an area of this file, makes a table and then makes a graph of the loaded waves.

This example assumes that you have the "Instrument Data.xls" file and a symbolic path named Science that points to the folder containing the file.

```
Function InstrumentData()
    // Load Instrument Data file from the Scientific Analysis folder
    XLLoadWave/O/T/R=(C9,M27)/W=8/C=9/P=Science "Instrument Data.xls"

    // Make graph.
    Display M1, M2, M3 vs X_Time
    Label bottom, "Time"; Label left, "Mass"
    ModifyGraph dateInfo(bottom)={1,0,0}
End
```

See also **Loading Excel Data Into a 2D Wave** on page II-143.

XWaveName

XWaveName(*graphNameStr*, *traceNameStr*)

The XWaveName function returns a string containing the name of the wave supplying the X coordinates for the named trace in the named graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

traceNameStr is the name of the trace in question.

Details

XWaveName returns an empty string (`""`) if the trace is not plotted versus an X wave.

For most uses, we recommend that you use **XWaveRefFromTrace** instead of WaveName. XWaveName returns a string containing the wave name only, with no data folder path qualifying it. Thus, you may get erroneous results if the X wave referred to in the graph has the same name as a different wave in the current data folder. Likewise, if the named wave resides in a folder that is not the current data folder, you will not be able to refer to the named wave.

graphNameStr and *traceNameStr* are strings, *not* names.

Examples

```
Display ywave vs xwave              // XY graph
Print XWaveName("","ywave")         // prints xwave
```

See also

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

XWaveRefFromTrace

`XWaveRefFromTrace(`*`graphNameStr, traceNameStr`*`)`

The XWaveRefFromTrace function returns a wave reference to the wave supplying the X coordinates against which the named trace is displayed in the named graph window or subwindow.

Parameters

graphNameStr can be `""` to refer to the top graph window.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Details

XWaveRefFromTrace returns a null reference (see **WaveExists**) if the wave is not plotted versus an X wave.

graphNameStr and *traceNameStr* are strings, not names.

Examples

```
Display ywave vs xwave                   // XY graph
Print XWaveRefFromTrace("","ywave")[50] // prints value of xwave at point 50
```

See Also

For other commands related to waves and traces: **WaveRefIndexed**, **TraceNameToWaveRef**, **TraceNameList**, **CsrWaveRef**, and **CsrXWaveRef**.

For a description of traces: **ModifyGraph**.

For a discussion of contour traces see **Contour Traces** on page II-283.

For commands referencing other waves in a graph: **ImageNameList**, **ImageNameToWaveRef**, **ContourNameList**, and **ContourNameToWaveRef**.

For a discussion of wave references, see **Wave Reference Functions** on page IV-186.

See Also

Trace Names on page II-216, **Programming With Trace Names** on page IV-81.

y

`y`

The y function returns the Y value for the current column of the destination wave when used in a multidimensional wave assignment statement. Y is the scaled column index whereas **q** is the column index itself.

Details

Unlike **x**, outside of a wave assignment statement, y does not act like a normal variable.

See Also

x, **z**, and **t** functions for other dimensions.

p, **q**, **r**, and **s** functions for the scaled indices.

z

`z`

The z function returns the Z value for the current layer of the destination wave when used in a multidimensional wave assignment statement. z is the scaled layer index whereas **r** is the layer index itself.

Details

Unlike **x**, outside of a wave assignment statement, z does not act like a normal variable.

See Also

x, **y**, and **t** functions for other dimensions.

p, **q**, **r**, and **s** functions for the scaled indices.

zcsr

`zcsr(`*`cursorName`*` [, `*`graphNameStr`*`])`

The zcsr function returns a Z value when the specified cursor is on a contour, image, or waterfall plot. Otherwise, it returns NaN.

Parameters

cursorName identifies the cursor, which can be cursor A through J.

graphNameStr specifies the graph window or subwindow.

When identifying a subwindow with *graphNameStr*, see **Subwindow Syntax** on page III-87 for details on forming the window hierarchy.

Examples

```
Print zcsr(A)              // not zcsr("A")
Print zcsr(A,"Graph0")     // specifies the graph
```

See Also

The **hcsr**, **pcsr**, **qcsr**, **vcsr**, and **xcsr** functions.

Programming With Cursors on page II-249.

zeta

`zeta(`*`a`*`, `*`b`*` [, `*`terms`*` ])`

The zeta function returns the Hurwitz Zeta function for real or complex arguments *a* and *b*

$$\zeta(a,b)=\sum_{k=0}^{\infty}\frac{1}{(k+b)^a},$$

$$\Re(a)>1,$$

$$b\neq 0,-1,-2,...$$

The Riemann zeta function is the special case:

$$\zeta(a)=\zeta(a,1).$$

The zeta function was added in Igor Pro 7.00.

Parameters

The *terms* parameter defaults to 40. In practice evaluation may terminate before the specified number of terms when convergence is achieved.

References

Olver, Frank W. J.; Lozier, Daniel W.; Boisvert, Ronald F.; Clark, Charles W., eds., "NIST Handbook of Mathematical Functions", 607 pp., Cambridge University Press, 2010.

See Also

Dilogarithm

ZernikeR

`ZernikeR(n,m,r)`

The ZernikeR function returns the Zernike radial polynomials of degree *n* that contains no power of *r* that is less than *m*. Here *m* is even or odd according to whether *n* is even or odd, and *r* is in the range 0 to 1.

Note that the full circle polynomials are complex. For any angle *t* (theta), they are given by: `ZernikeR(n,m,r)*exp(imt)`.