

Analysis of Functions

- Operations that Work on Functions 272
- Function Plotting..... 272
 - Function Plotting Using Dependencies 273
 - Function Plotting Using Controls..... 273
 - Plotting a User-Defined Function..... 274
- Solving Differential Equations 274
 - Terminology 274
 - ODE Inputs 275
 - ODE Outputs..... 275
 - The Derivative Function 276
 - A First-Order Equation 277
 - A System of Coupled First-Order Equations..... 279
 - Optimizing the Derivative Function..... 280
 - Higher Order Equations 280
 - Free-Run Mode..... 282
 - Stiff Systems..... 283
 - Error Monitoring..... 284
 - Solution Methods..... 286
 - Interrupting IntegrateODE..... 286
 - Stopping and Restarting IntegrateODE..... 286
 - Stopping IntegrateODE on a Condition 287
- Integrating a User Function..... 288
- Finding Function Roots..... 290
 - Roots of Polynomials with Real Coefficients..... 290
 - Roots of a 1D Nonlinear Function..... 291
 - Roots of a System of Multidimensional Nonlinear Functions 293
 - Caveats for Multidimensional Root Finding 295
- Finding Minima and Maxima of Functions 295
 - Extreme Points of a 1D Nonlinear Function 296
 - Extrema of Multidimensional Nonlinear Functions..... 298
 - Stopping Tolerances 299
 - Problems with Multidimensional Optimization 299
- References 301

Operations that Work on Functions

Some Igor operations work on functions rather than data in waves. These operations take as input one or more functions that you define in the Procedure window. The result is some calculation based on function values produced when Igor evaluates your function.

Because the operations evaluate a function, they work on continuous data. That is, the functions are not restricted to data values that you provide from measurements. They can be evaluated at any input values. Of course, a computer works with discrete digital numbers, so even a “continuous” function is broken into discrete values. Usually these discrete values are so close together that they are continuous for practical purposes. Occasionally, however, the discrete nature of computer computations causes problems.

The following operations use functions as inputs:

- **IntegrateODE** computes numerical solutions to ordinary differential equations. The differential equations are defined as user functions. The IntegrateODE operation is described under **Solving Differential Equations** on page III-274.
- **FindRoots** computes solutions to $f(x)=a$, where a is a constant (often zero). The input x may represent a vector of x values. A special form of FindRoots computes roots of polynomials. The FindRoots operation is described in the section **Finding Function Roots** on page III-290.
- **Optimize** finds minima or maxima of a function, which may have one or more input variables. The Optimize operation is described in the section **Finding Minima and Maxima of Functions** on page III-295.
- **Integrate1D** integrates a function between two specified limits. Despite its name, it can also be used for integrating in two or more dimensions. See **Integrating a User Function** on page III-288.

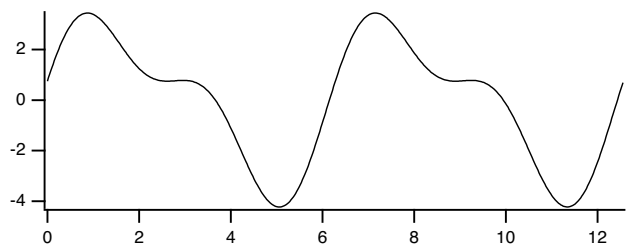
Function Plotting

Function plotting is very easy in Igor, assuming that you understand what a waveform is (see **Waveform Model of Data** on page II-57) and how X scaling works. Here are the steps to plot a function.

1. Decide how many data points you want to plot.
2. Make a wave with that many points.
3. Use the SetScale operation to set the wave’s X scaling. This defines the domain over which you are going to plot the function.
4. Display the wave in a graph.
5. Execute a waveform assignment statement to set the data values of the wave.

Here is an example.

```
Make/O/N=500 wave0
SetScale/I x, 0, 4*PI, wave0      // plot function from x=0 to x=4π
Display wave0
wave0 = 3*sin(x) + 1.5*sin(2*x + PI/6)
```



To evaluate the function over a different domain, you need to reexecute the SetScale command with different parameters. This redefines “x” for the wave. Then you need to reexecute the waveform assignment statement. For example,

```
SetScale/I x, 0, 2*PI, wave0 // plot function from x=0 to x=2π
wave0 = 3*sin(x) + 1.5*sin(2*x + PI/6)
```

Reexecuting commands is easy, using the shortcuts shown in **History Area** on page II-9.

Function Plotting Using Dependencies

If you get tired of reexecuting the waveform assignment statement each time you change the domain, you can use a dependency to cause Igor to automatically reexecute it. To do this, use := instead of =.

```
wave0 := 3*sin(x) + 1.5*sin(2*x + PI/6)
```

See Chapter IV-9, **Dependencies**, for details.

You have made wave0 depend on “X”. The SetScale operation changes the meaning of “X” for the wave. Now when you do a SetScale on wave0, Igor will automatically reexecute the assignment.

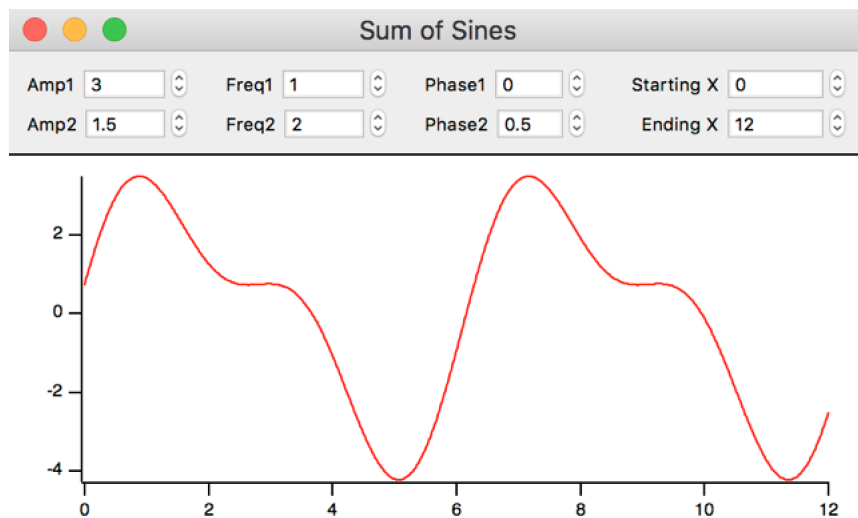
You can take this further by using global variables instead of literal numbers in the right-hand expression. For example:

```
Variable/G amp1=3, amp2=1.5, freq1=1, freq2=2, phase1=0, phase2=PI/6
wave0 := amp1*sin(freq1*x + phase1) + amp2*sin(freq2*x + phase2)
```

Now, wave0 depends on these global variables. If you change them, Igor will automatically reexecute the assignment.

Function Plotting Using Controls

For a slick presentation of function plotting, you can put controls in the graph to set the values of the global variables. When you change the value in the control, the global variable changes, which reexecutes the assignment. This changes the wave, which updates the graph. Here is what the graph would look like.



We’ve added two additional global variables and connected them to the Starting X and Ending X controls. This allows us to set the domain. These controls are both linked to an action procedure that does a SetScale on the wave.

Controls are explained in detail in Chapter III-14, **Controls and Control Panels**.

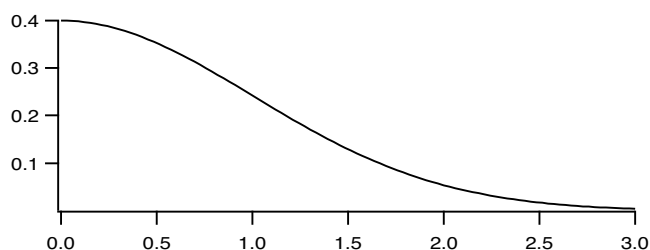
Plotting a User-Defined Function

In the preceding example we used the built-in `sin` function in the right-hand expression. We can also use a user-defined function. Here is an example using a very simple function — the normal probability distribution function.

```
Function NormalProb(x)
  Variable x

  // the constant is 1/sqrt(2*pi) evaluated in double-precision
  return      0.398942280401433*exp(-(0.5*x^2))
End

Make/N=100 wave0; SetScale/I x, 0, 3, wave0; wave0 = NormalProb(x)
Display wave0
```



Note that, although we are using the `NormalProb` function to fill a wave, the `NormalProb` function itself has nothing to do with waves. It merely takes an input and returns a single output. We could also test the `NormalProb` function at a single point by executing

```
Print NormalProb(0)
```

This would print the output of the function in the history area.

It is the act of using the `NormalProb` function in a wave assignment statement that fills the wave with data values. As it executes the wave assignment, Igor calls the `NormalProb` function over and over again, 100 times in this case, passing it a different parameter each time and storing the output from the `NormalProb` function in successive points of the destination wave.

For more information on Wave Assignments, see **Waveform Arithmetic and Assignments** on page II-69. You may also find it helpful to read Chapter IV-1, **Working with Commands**.

`WaveMetrics` provides a procedure package that provides a convenient user interface to graph mathematical expressions. To use it, choose `Analysis`→`Packages`→`Function Grapher`. This displays a graph with controls to create and display a function. Click the `Help` button in the graph to learn how to use it.

Solving Differential Equations

Numerical solutions to initial-value problems involving ordinary differential equations can be calculated using the `IntegrateODE` operation (see page V-392). You provide a user-defined function that implements a system of differential equations. The solution to your differential equations are calculated by marching the solution forward or backward from the initial conditions in a series of steps or increments in the independent variable.

Terminology

Referring to the independent variable and the dependent variables is very cumbersome, so we refer to these as `X` and `Y[i]`. Of course, `X` may represent distance or time or anything else.

A system of differential equations will be written in terms of derivatives of the `Y[i]`s, or $dy[i]/dx$.

ODE Inputs

You provide to IntegrateODE a function to calculate the derivatives or right-hand-sides of your system of differential equations.

You also provide one output wave for each equation in the system to receive the solution. The solution waves will have a row for each output point you want.

You specify the independent variable either by setting the X scaling of the output waves, by specifying x_0 and deltax using the `/X={x0,deltax}` flag, or by providing an explicit X wave using the `/X=xWave` flag.

For a system of four equations (fourth-order system), if you provide an X wave to specify where you want values, you might have this situation:

Xwave	A	B	C	D
0	1	1	0	0
10	0.98	0.9804	0.0192	0.0003
20	0.962	0.9615	0.0369	0.0015
30	0.943	0.9434	0.0532	0.0033
40	0.926	0.9260	0.0681	0.0057
50	0.909	0.9092	0.0819	0.0087
60	0.893	0.8931	0.0945	0.0122
70	0.878	0.8775	0.1061	0.0163

X wave specifies where to report solutions.
In free-run mode, X wave receives X values for solution rows.

Instead of multiple 1D output waves, you can provide a single 2D output wave:

Xwave	yWave[][0]	yWave[][1]	yWave[][2]	yWave[][3]
0	1	1	0	0
10	0.98	0.98	0.0192	0.000389
20	0.962	0.962	0.0369	0.00152
30	0.943	0.943	0.0532	0.00332
40	0.926	0.926	0.0682	0.00576
50	0.909	0.909	0.0819	0.00876
60	0.893	0.893	0.0946	0.0123
70	0.878	0.878	0.106	0.0163

Igor calculates a solution value for each element of the Y (output) waves.

Before executing IntegrateODE, you must load the initial conditions (the initial $Y[i]$ values) into the first row of the Y waves. Igor then calculates the solution starting from those values. The first solution value is stored in the second element of the Y waves.

If you use the `/R` flag with IntegrateODE to start the integration at a point other than the beginning of the Y wave, the initial conditions must be in the first row specified by the `/R` flag. See **Stopping and Restarting IntegrateODE** on page III-286.

ODE Outputs

The algorithms Igor uses to integrate your ODE systems use adaptive step-size control. That is, the algorithms advance the solution by the largest increment in X that result in errors at least as small as you require. If the solution is changing rapidly, or the solution has some other difficulty, the step sizes may get very small.

Chapter III-10 — Analysis of Functions

IntegrateODE has two schemes for returning solution results to you: you can specify X values where you need solution values, or you can let the solution “free run”.

In the first mode, results are returned to you at values of x corresponding to the X scaling of your Y waves, or at X values that you provide via an X wave or by providing X0 and deltaX and letting Igor calculate the X values. The actual calculation may require X increments smaller than those you ask for. Igor returns results only at the X values you ask for.

In free-run mode, IntegrateODE returns solution values for every step taken by the integration algorithm. In some cases, this may give you extremely small steps. Free-run mode returns to you not only the Y[i] values from the solution, but also values of x[i]. Free-run mode can be useful in that, to some degree, it will return results closely spaced when the solution is changing rapidly and with larger spacing when the solution is changing slowly.

The Derivative Function

You must provide a user-defined function to calculate derivatives corresponding to the equations you want to solve. All equations are solved as systems of first-order equations. Higher-order equations must be transformed to multiple first-order equations (an example is shown later).

The derivative function has this form:

```
Function D(pw, xx, yw, dydx)
    Wave pw          // parameter wave (input)
    Variable xx      // x value at which to calculate derivatives
    Wave yw          // wave containing y[i] (input)
    Wave dydx        // wave to receive dy[i]/dx (output)

    dydx[0] = <expression for one derivative>
    dydx[1] = <expression for next derivative>
    <etc.>

    return 0
End
```

Note the return statement at the end of the function. The function result should normally be 0. If it is 1, IntegrateODE will stop. If the return statement is omitted, the function returns NaN which IntegrateODE treats the same as 0. But it is best to explicitly return 0.

Because the function may produce a large number of outputs, the outputs are returned via a wave in the parameter list.

The parameter wave is simply a wave containing possible adjustable constants. Using a wave for these makes it convenient to change the constants and try a new integration. It also will make it more convenient to do a curve fit to a differential equation. You must create the parameter wave before invoking IntegrateODE. The contents of the parameter wave are of no concern to IntegrateODE and are not touched. In fact, you can change the contents of the parameter wave inside your function and those changes will be permanent.

Other inputs are the value of x at which the derivatives are to be evaluated, and a wave (yw in this example) containing current values of the y[i]’s. The value of X is determined when it calls your function, and the waves yw and dydx are both created and passed to your function when Igor needs new values for the derivatives. Both the input Y wave and the output dydx wave have as many elements as the number of derivative equations in your system of ODEs.

The values in the yw wave correspond to a row of the table in the example above. That is:

Xwave	A	B	C	D
0	1	1	0	0
10	0.98	0.980402	0.019208E	0.000389E
20	0.962	0.961575	0.036908	0.001516E
30	0.943	0.943476	0.053200E	0.003323E
40	0.926	0.926063	0.068182E	0.005755E
50	0.909	0.909297	0.084041E	0.008761E

```
Function D(pw, xx, yw, dydx)
Wave pw // parameter wave (input)
Variable xx // x value at which to calculate derivatives
Wave yw // wave containing y[i] (input)
Wave dydx // wave to receive dy[i]/dx (output)

dydx[0] = <expression for one derivative>
dydx[1] = <expression for next derivative>
<etc.>

return 0
End
```

The wave yw contains the present value, or estimated value, of Y[i] at X =xx. You may need this value to calculate the derivatives.

Your derivative function is called many times during the course of a solution, and it will be called at values of X that do not correspond to X values in the final solution. The reason for this is two-fold: First, the solution method steps from one value of X to another using estimates of the derivatives at several intermediate X values. Second, the spacing between X values that you want may be larger than can be calculated accurately, and Igor may need to find the solution at intermediate values. These intermediate values are not reported to you unless you call the **IntegrateODE** operation (see page V-392) in free-run mode.

Because the derivative function is called at intermediate X values, the yw wave is not the same wave as the Y wave you create and pass to IntegrateODE. Note that one row of your Y wave, or one value from each Y wave, corresponds to the elements of the one-dimensional yw wave that is passed in to your derivative function. While the illustration implies that values from your Y wave are passed to the derivative function, in fact the values in the yw wave passed into the derivative function correspond to whatever Y values the integrator needs at the moment. The correspondence to your Y wave or waves is only conceptual.

You should be aware that, with the exception of the parameter wave (pw above) the waves are not waves that exist in your Igor experiment. Do not try to resize them with InsertPoints/DeletePoints and don't do anything to them with the Redimension operation. The yw wave is input-only; altering it will not change anything. The dydx wave is output-only; the only thing you should do with it is to assign appropriate derivative (right-hand-side) values.

Some examples are presented in the following sections.

A First-Order Equation

Let's say you want a numerical solution to a simple first-order differential equation:

$$\frac{dy}{dx} = -ay$$

First you need to create a function that calculates the derivative. Enter the following in the procedure window:

```
Function FirstOrder(pw, xx, yw, dydx)
Wave pw // pw[0] contains the value of the constant a
Variable xx // not actually used in this example
Wave yw // has just one element- there is just one equation
Wave dydx // has just one element- there is just one equation
```

Chapter III-10 — Analysis of Functions

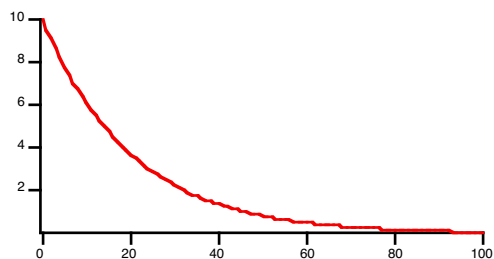
```
// There's only one equation, so only one expression here.
// The constant a in the equation is passed in pw[0]
dydx[0] = -pw[0]*yw[0]

return 0
End
```

Paste the function into the procedure window and then execute these commands:

```
Make/D/O/N=101 YY // wave to receive results
YY[0] = 10 // initial condition- y0=10
Display YY // make a graph
Make/D/O PP={0.05} // set constant a to 0.05
IntegrateODE FirstOrder, PP, YY
```

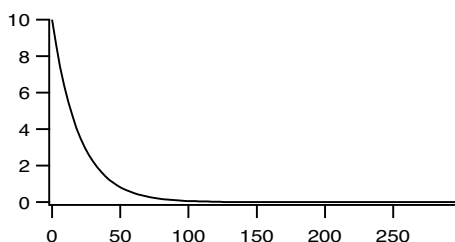
This results in the following graph with the expected exponential decay:



The IntegrateODE command shown in the example is the simplest you can use. It names the derivative function, `FirstOrder`, a parameter wave, `PP`, and a results wave, `YY`.

Because the IntegrateODE command does not explicitly set the X values, the output results are calculated according to the X scaling of the results wave `YY`. You can change the spacing of the X values by changing the X scaling of `YY`:

```
SetScale/P x 0,3,YY // now the results will be at an x interval of 3
IntegrateODE FirstOrder, PP, YY
```



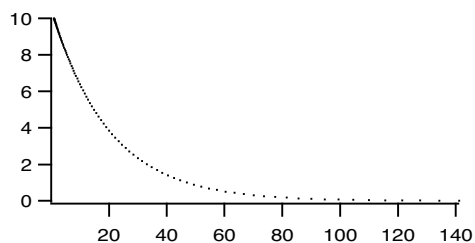
The same thing can be achieved by using your specified `x0` and `deltax` with the `/X` flag:

```
IntegrateODE/X={0,3} FirstOrder, PP, YY
```

We presume that you have your own reasons for using the `/X={x0, deltax}` form. Note that when you do this, it doesn't use the X scaling of your Y wave. If you graph the Y wave the values on the X axis may not match the X values used during the calculation.

Finally, you don't have to use a constant spacing in X if you provide an X wave. You might want to do this to get closely-spaced values only where the solution changes rapidly. For instance:

```
Make/D/O/N=101 XX // same length as YY
XX = exp(p/20) // X values get farther apart as X increases
Display YY vs XX // make an XY graph
ModifyGraph mode=2 // plot with dots so you can see the points
IntegrateODE/X=XX FirstOrder, PP, YY
```

Note that throughout these examples the initial value of YY has remained at 10.

A System of Coupled First-Order Equations

While many interesting systems are described by simple (possibly nonlinear) first-order equations, more interesting behavior results from systems of coupled equations.

The next example comes from chemical kinetics. Suppose you mix two substances A and B together in a solution and they react to form intermediate phase C. Over time C transforms into final product D:



Here, k_1 , k_2 , and k_3 are rate constants for the reactions. The concentrations of the substances might be given by the following coupled differential equations:

$$\frac{dA}{dt} = \frac{dB}{dt} = -k_1 \cdot A \cdot B + k_2 \cdot C \quad \frac{dC}{dt} = k_1 \cdot A \cdot B - k_2 \cdot C - k_3 \cdot C \quad \frac{dD}{dt} = k_3 \cdot C$$

To solve these equations, first we need a derivative function:

```
Function ChemKinetic(pw, tt, yw, dydt)
  Wave pw          // pw[0] = k1, pw[1] = k2, pw[2] = k3
  Variable tt      // time value at which to calculate derivatives
  Wave yw          // yw[0]-yw[3] containing concentrations of A,B,C,D
  Wave dydt        // wave to receive dA/dt, dB/dt etc. (output)
  dydt[0] = -pw[0]*yw[0]*yw[1] + pw[1]*yw[2]
  dydt[1] = dydt[0] // first two equations are the same
  dydt[2] = pw[0]*yw[0]*yw[1] - pw[1]*yw[2] - pw[2]*yw[2]
  dydt[3] = pw[2]*yw[2]

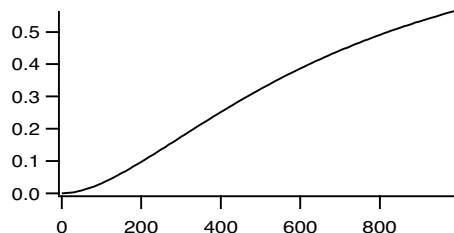
  return 0
End
```

We think that it is easiest to keep track of the results using a single multicolumn Y wave. These commands make a four-column Y wave and use dimension labels to keep track of which column corresponds to which substance:

```
Make/D/O/N=(100,4) ChemKin
SetScale/P x 0,10,ChemKin // calculate concentrations every 10 s
SetDimLabel 1,0,A,ChemKin // set dimension labels to substance names
SetDimLabel 1,1,B,ChemKin // this can be done in a table if you make
SetDimLabel 1,2,C,ChemKin // the table using edit ChemKin.ld
SetDimLabel 1,3,D,ChemKin
ChemKin[0][%A] = 1 // initial conditions: concentration of A
ChemKin[0][%B] = 1 // and B is 1, C and D is 0
ChemKin[0][%C] = 0 // note indexing using dimension labels
ChemKin[0][%D] = 0
Make/D/O KK={0.002,0.0001,0.004} // rate constants
```

Chapter III-10 — Analysis of Functions

```
Display ChemKin[] [%D] // graph concentration of the product
                        // Note graph made with subrange of wave
IntegrateODE/M=1 ChemKinetic, KK, ChemKin
```



Note that the waves `yw` and `dydt` in the derivative function have four elements corresponding to the four equations in the system of ODEs. At a given value of `X` (or `t`) `yw[0]` and `dydt[0]` correspond to the first equation, `yw[1]` and `dydt[1]` to the second, etc.

Note also that we have used the `/M=1` flag to request the Bulirsch-Stoer integration method. For well-behaved systems, it is likely to be the fastest method, taking the largest steps in the solution.

Optimizing the Derivative Function

The Igor compiler does no optimization of your code. Because `IntegrateODE` may call your function thousands (or millions!) of times, efficient code can significantly reduce the time it takes to calculate the solution. For instance, the `ChemKinetic` example function above was written to parallel the chemical equations to make the example clearer. There are three terms that appear multiple times. As written, these terms are calculated again from scratch each time they are encountered. You can save some computation time by precalculating these terms as in the following example:

```
Function ChemKinetic2(pw, tt, yw, dydt)
    Wave pw // pw[0] = k1, pw[1] = k2, pw[2] = k3
    Variable tt // time value at which to calculate derivatives
    Wave yw // yw[0]-yw[3] containing concentrations of A,B,C,D
    Wave dydt // wave to receive dA/dt, dB/dt etc. (output)

    // Calculate common subexpressions
    Variable t1mt2 = pw[0]*yw[0]*yw[1] - pw[1]*yw[2]
    Variable t3 = pw[2]*yw[2]

    dydt[0] = -t1mt2
    dydt[1] = dydt[0] // first two equations are the same
    dydt[2] = t1mt2 - t3
    dydt[3] = t3

    return 0
End
```

These changes reduced the time to compute the solution by about 13 per cent. Your mileage may vary. Larger functions with subexpression repeated many times are prime candidates for this kind of optimization.

Note also that `IntegrateODE` updates the display every time 10 result values are calculated. Screen updates can be very time-consuming, so `IntegrateODE` provides the `/U` flag to control how often the screen is updated. For timing this example we used `/U=1000000` which effectively turned off screen updating.

Higher Order Equations

Not all differential equations (in fact, not many) are expressed as systems of coupled first-order equations, but `IntegrateODE` can only handle such systems. Fortunately, it is always possible to make substitutions to transform an N th-order differential equation into N first-order coupled equations.

You need one equation for each order. Here is the equation for a forced, damped harmonic oscillator (using y for the displacement rather than x to avoid confusion with the independent variable, which is t in this case):

$$\frac{d^2 y}{dt^2} + 2\lambda \frac{dy}{dt} + \omega^2 y = F(t)$$

If we define a new variable v (which happens to be velocity in this case):

$$v = \frac{dy}{dt}$$

Then

$$\frac{d^2 y}{dt^2} = \frac{dv}{dt}$$

Substituting into the original equation gives us two coupled first-order equations:

$$\frac{dv}{dt} = -2\lambda v - \omega^2 y + F(t)$$

$$\frac{dy}{dt} = v$$

Of course, a real implementation of these equations will have to provide something for $F(t)$. A derivative function to implement these equations might look like this:

```
Function Harmonic(pw, tt, yy, dydt)
    Wave pw      // pw[0]=damping, pw[1]=undamped frequency
                // pw[2]=Forcing amplitude, pw[3]=Forcing frequency
    Variable tt
    Wave yy      // yy[0] = velocity, yy[1] = displacement
    Wave dydt

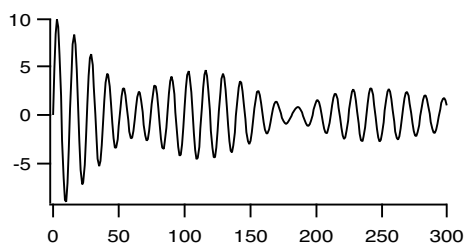
    // simple sinusoidal forcing
    Variable Force = pw[2]*sin(pw[3]*tt)

    dydt[0] = -2*pw[0]*yy[0] - pw[1]*pw[1]*yy[1]+Force
    dydt[1] = yy[0]

    return 0
End
```

And the commands to integrate the equations:

```
Make/D/O/N=(300,2) HarmonicOsc
SetDimLabel 1,0,Velocity,HarmonicOsc
SetDimLabel 1,1,Displacement,HarmonicOsc
HarmonicOsc[0][%Velocity] = 5 // initial velocity
HarmonicOsc[0][%Displacement] = 0 // initial displacement
Make/D/O HarmPW={.01,.5,.1,.45} // damping, freq, forcing amp and freq
Display HarmonicOsc[1][%Displacement]
IntegrateODE Harmonic, HarmPW, HarmonicOsc
```



Free-Run Mode

Most of the examples shown so far use the Y wave's X scaling to set the X values where a solution is desired. In the section **A First-Order Equation** on page III-277, examples are also shown in which the /X flag is used to specify the sequence of X values, either by setting X0 and deltaX or by supplying a wave filled with X values.

These methods have the advantage that you have complete control over the X values where the solution is reported to you. They also are completely deterministic — you know before running IntegrateODE exactly how many points will be calculated and how big your waves need to be.

They also have the potential drawback that you may force IntegrateODE to use smaller X increments than required. If your ODE system is expensive to calculate, this may exact a considerable cost in computation time.

IntegrateODE also offers a “free-run” mode in which the solution is allowed to proceed using whatever X increments are required to achieve the requested accuracy limit. This mode has two possible advantages — it will use the minimum number of solution steps required and it may also produce a higher density of points in areas where the solution changes rapidly (but watch out for stiff systems, see page III-283).

Free-run mode has the disadvantage that in certain cases the solution may require miniscule steps to tip toe through difficult terrain, inundating you with huge numbers of points that you don't really need. You also don't know ahead of time how many points will be required to cover a certain range in X.

To illustrate the use of free-run mode, we will return to the example used in the section **A First-Order Equation** on page III-277. (Make sure the FirstOrder function is compiled in the procedure window.) Because we don't know how many points will be produced, we will make the waves large:

```
Make/D/O/N=1000 FreeRunY      // wave to receive results
FreeRunY = NaN
FreeRunY[0] = 10              // initial condition- y0=10
```

Free-run mode requires that you supply an X wave. Unlike the previous use of an X wave, in free-run mode the X wave is filled by IntegrateODE with the X values at which solution values have been calculated. Like the Y waves, you must provide an initial value in the first row of the X wave. As before, it must have the same number of rows as the Y waves:

```
Make/O/D/N=1000 FreeRunX      // same length as YY
FreeRunX = NaN                // prevent display of extra points
FreeRunX[0] = 0               // initial value of X
```

In free-run mode, only the points that are required are altered. Thus, if you have some preexisting wave contents, they will be seen on a graph. We prevent the resulting confusion by filling the X wave with NaN's (Not a Number, or blanks). Igor graphs do not display points that have NaN values.

Make a graph:

```
Display FreeRunY vs FreeRunX  // make an XY graph
ModifyGraph mode=3, marker=19 // plot with dots to show the points
```

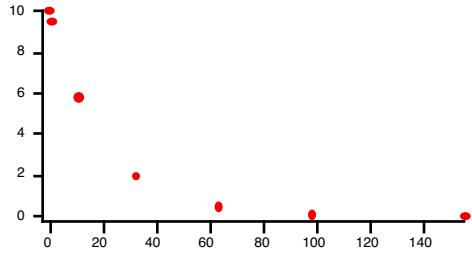
Make the parameter wave and set the value of the equation's lone coefficient:

```
Make/D/O PP={0.05}           // set constant a to 0.05
```

And finally do the integration in free-run mode. The /XRUN flag specifies a suggested first step size and the maximum X value. When the solution passes the maximum X value (100 in this case) or when your waves are filled, IntegrateODE will stop.

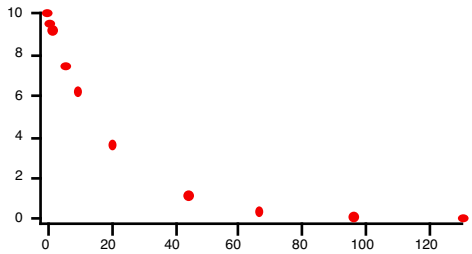
```
FreeRunX = NaN;FreeRunX[0] = 0
IntegrateODE/M=1/X=FreeRunX/XRUN={1,100} FirstOrder, PP, FreeRunY
```

In the earlier example, we (rather arbitrarily) chose 100 steps to make a reasonably smooth plot. In this case, it took 6 steps to cover the same X range, and the steps are closest together at the beginning where the exponential decay is most rapid:



Asking for more accuracy will cause smaller steps to be taken (9 when we executed the following command):

```
FreeRunX = NaN;FreeRunX[0] = 0
IntegrateODE/M=1/X=FreeRunX/XRUN={1,100}/E=1e-14 FirstOrder, PP, FreeRunY
```



After IntegrateODE has finished, you can use Redimension and the V_ODETotalSteps variable to adjust the size of the waves to just the points actually calculated:

```
Redimension/N=(V_ODETotalSteps+1) FreeRunY, FreeRunX
```

Note that we added 1 to V_ODETotalSteps to account for the initial value in row zero.

Stiff Systems

Some systems of differential equations involve components having very different time (or decay) constants. This can create what is called a “stiff” system; even though the short time constant decays rapidly and contributes negligibly to the solution after a very short time, ordinary solution methods (/M = 0, 1, and 2) are unstable because of the presence of the short time-constant component. IntegrateODE offers the Backward Differentiation Formula method (BDF, flag /M=3) to handle stiff systems.

A rather artificial example is the system (see “Numerical Recipes in C”, edition 2, page 734; see **References** on page III-301)

$$\begin{aligned} du/dt &= 998u + 1998v \\ dv/dt &= -999u - 1999v \end{aligned}$$

Here is the derivative function that implements this system:

```
Function StiffODE(pw, tt, yy, dydt)
    Wave pw          // not actually used because the coefficients
                    // are hard-coded to give a stiff system
    Variable tt
```

```
Wave yy
Wave dydt

dydt[0] = 998*yy[0] + 1998*yy[1]
dydt[1] = -999*yy[0] - 1999*yy[1]

return 0
End
```

Commands to set up the wave required and to make a suitable graph:

```
Make/D/O/N=(3000,2) StiffSystemY
Make/O/N=0 dummy // dummy coefficient wave
StiffSystemY = 0
StiffSystemY[0][0] = 1 // initial condition for u component
make/D/O/N=3000 StiffSystemX
Display StiffSystemY[][0] vs StiffSystemX
AppendToGraph/L=vComponentAxis StiffSystemY[][1] vs StiffSystemX

// make a nice-looking graph with dots to show where the solution points are
ModifyGraph axisEnab(left)={0,0.48},axisEnab(vComponentAxis)={0.52,1}
DelayUpdate
ModifyGraph freePos(vComponentAxis)={0,kwFraction}
ModifyGraph mode=2,lsize=2,rgb=(0,0,65535)
```

These commands solve this system using the Bulirsch-Stoer method using free run mode to minimize the number of solution steps computed:

```
StiffSystemX = nan // hide unused solution points
StiffSystemX[0] = 0 // initial X value
IntegrateODE/M=1/X=StiffSystemX/XRUN={1e-6, 2} StiffODE, dummy, StiffSystemY
Print "Required ",V_ODETotalSteps, " steps to solve using Bulirsch-Stoer"
```

which results in this message in the history area:

```
Required 401 steps to solve using Bulirsch-Stoer
```

These commands solve this system using the BDF method:

```
StiffSystemX = nan // hide unused solution points
StiffSystemX[0] = 0 // initial X value
IntegrateODE/M=3/X=StiffSystemX/XRUN={1e-6, 2} StiffODE, dummy, StiffSystemY
Print "Required ",V_ODETotalSteps, " steps to solve using BDF"
```

This results in this message in the history area:

```
Required 133 steps to solve using BDF
```

The difference between 401 steps and 133 is significant! Be aware, however, that the BDF method is not the most efficient for nonstiff problems.

Error Monitoring

To achieve the fastest possible solution to your differential equations, Igor uses algorithms with adaptive step sizing. As each step is calculated, an estimate of the truncation error is also calculated and compared to a criterion that you specify. If the error is too large, a smaller step size is used. If the error is small compared to what you asked for, a larger step size is used for the next step.

Igor monitors the errors by scaling the error by some (hopefully meaningful) number and comparing to an error level.

The Runge-Kutta and Bulirsch-Stoers methods (IntegrateODE flag /M=0 or /M=1) estimates the errors for each of your differential equations and the largest is used for the adjustments:

$$\text{Max}\left(\frac{\text{Error}_i}{\text{Scale}_i}\right) < \text{eps}$$

The Adams-Moulton and BDF methods (IntegrateODE flag /M=2 or /M=3) estimate the errors and use the root mean square of the error vector:

$$\left[\frac{1}{N} \sum \left(\frac{Error_i}{Scale_i} \right)^2 \right]^{1/2} < eps .$$

Igor sets *eps* to 10⁻⁶ by default. If you want a different error level, use the /E=*eps* flag to set a different value of *eps*. Using the harmonic oscillator example, we now set a more relaxed error criterion than the default:

```
IntegrateODE/E=1e-3 Harmonic, HarmPW, HarmonicOsc
```

The error scaling can be composed of several parts, each optional:

$$Scale_i = h \cdot (C_i + y_i + dy_i/dx)$$

By default Igor uses constant scaling, setting *h*=1 and *C_i*=1, and does not use the *y_i* and *dy_i/dx* terms making *Scale_i*=1. In that case, *eps* represents an absolute error level: the error in the calculated values should be less than *eps*. An absolute error specification is often acceptable, but it may not be appropriate if the output values are of very different magnitudes.

You can provide your own customized values for *C_i* using the /S=*scaleWave* flag. You must first create a wave having one point for each differential equation. Fill it with your desired scaling values, and add the /S flag to the IntegrateODE operation:

```
Make/O/D errScale={1,5}
IntegrateODE/S=errScale Harmonic, HarmPW, HarmonicOsc
```

Typically, the constant values should be selected to be near the maximum values for each component of your system of equations.

Finally, you can control what Igor includes in *Scale_i* using the /F=*errMethod* flag. The argument to /F is a bitwise value with a bit for each component of the equation above:

<i>errMethod</i>	What It Does
1	Add a constant <i>C_i</i> from <i>scaleWave</i> (or 1's if no <i>scaleWave</i>).
2	Add the current value of <i>y_i's</i> , the calculated result.
4	Add the current value of the derivatives, <i>dy_i/dx</i> .
8	Multiply by <i>h</i> , the current step size

Use *errMethod* = 2 if you want the errors to be a fraction of the current value of Y. That might be appropriate for solutions that asymptotically approach zero when you need smaller errors as the solution approaches zero.

The Scale numbers can never equal zero, and usually it isn't appropriate for *Scale_i* to get very small. Thus, it isn't usually a good idea to use *errMethod* = 2 with solutions that pass through zero. A good way to avoid this problem can be to add the values of the derivatives (*errMethod* = (2+4)), or to add a small constant:

```
Make/D errScale=1e-6
IntegrateODE/S=errScale/F=(2+1) ...
```

Finally, in some cases you need the much more stringent requirement that the errors be less than some global value. Since the solutions are the result of adding up myriad sequential solutions, any truncation error has the potential to add up catastrophically if the errors happen to be all of the same sign. If you are using Runge-Kutta and Bulirsch-Stoers methods (IntegrateODE flag /M=0 or /M=1), you can achieve global error limits by setting bit 3 of *errMethod* (/F=8) to multiply the error by the current step size (*h* in the equation above). If you are using Adams-Moulton and BDF methods (IntegrateODE flag /M=2 or /M=3) bit 3 does nothing; in that case, a conservative value of *eps* would be needed.

Higher accuracy will make the solvers use smaller steps, requiring more computation time. The trade-off for smaller step size is computation time. If you get too greedy, the step size can get so small that the X incre-

ments are smaller than the computer's digital resolution. If this happens Igor will stop the calculation and complain.

Solution Methods

Igor makes four solution methods available, Runge-Kutta-Fehlberg, Bulirsch-Stoers with Richardson extrapolation, Adams-Moulton and Backward Differentiation Formula.

Runge-Kutta-Fehlberg is a robust method capable of surviving solutions or derivatives that aren't smooth, or even have discontinuous derivatives. Bulirsch-Stoers, for well-behaved systems, will take larger steps than Runge-Kutta-Fehlberg, so it may be considerably faster. Step size for a given problem is larger so you get greater accuracy. Of course, if you ask for values closer together than the achievable step size, you get no advantage from this.

Details of these methods can be found in the second edition of *Numerical Recipes* (see **References** on page III-301).

The Adams-Moulton and Backward Differentiation Formula (BDF) methods are adapted from the CVODE package developed at Lawrence Livermore National Laboratory. In our very limited experience, for well-behaved nonstiff systems the Bulirsch-Stoers method is much more efficient than either the Runge-Kutta-Fehlberg method or the Adams-Moulton method, in that it requires significantly fewer steps for a given problem.

As shown above, stiff systems benefit greatly by the use of the BDF method. However, for nonstiff methods, it is not as efficient as the other methods.

See **IntegrateODE** on page V-392 for references on these methods.

Interrupting IntegrateODE

Numerical solutions to differential equations can require considerable computation and, therefore, time. If you find that a solution is taking too long you can abort the operation by clicking the Abort button in the status bar. You may need to press and hold to make sure IntegrateODE notices.

When you abort an integration, IntegrateODE returns whatever results have been calculated. If those results are useful, you can restart the calculation from that point, using the last calculated result row as the initial conditions. Use the `/R=(startX)` flag to specify where you want to start.

For Igor programmers, the `V_ODEStepCompleted` variable will be set to the last result. It is probably a good idea to restart a step or two before that:

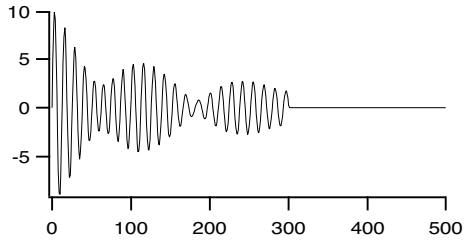
```
IntegrateODE/R=(V_ODEStepCompleted-1) ...
```

Stopping and Restarting IntegrateODE

Any result can be used as initial conditions for a new solution. Thus, you can use the `/R` flag to calculate just a part of the solution, then finish later using the `/R` flag to pick up where you left off. For instance, using the harmonic oscillator example:

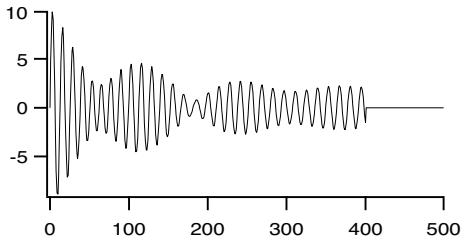
```
Make/D/O/N=(500,2) HarmonicOsc = 0
SetDimLabel 1,0,Velocity,HarmonicOsc
SetDimLabel 1,1,Displacement,HarmonicOsc
HarmonicOsc[0][%Velocity] = 5           // initial velocity
HarmonicOsc[0][%Displacement] = 0     // initial displacement
Make/D/O HarmPW={.01,.5,.1,.45}      // damping, freq, forcing amp and freq
Display HarmonicOsc[] [%Displacement]
```

```
IntegrateODE/M=1/R=[,300] Harmonic, HarmPW, HarmonicOsc
```

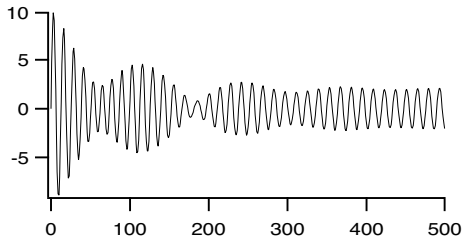
The calculation has been done for points 0-300. Note the comma in `/R=[,300]`, which sets 300 as the end point, not the start point. Now you can restart at 300 and continue to the 400th point:

```
IntegrateODE/M=1/R=[300,400] Harmonic, HarmPW, HarmonicOsc
```



or finish the entire 500 points. Perhaps you need to start from an earlier point:

```
IntegrateODE/M=1/R=[350] Harmonic, HarmPW, HarmonicOsc
```



Stopping IntegrateODE on a Condition

Sometimes it is useful to be able to stop the calculation based on output values from the integration, rather than stopping when a certain value of the independent variable is reached. For instance, a common way to simulate a neuron firing is to solve the relevant system of equations until the output reaches a certain value. At that point, the solution should be stopped and the initial conditions reset to values appropriate to the triggered condition. Then the calculation can be re-started from that point.

The ability to stop and re-start the calculation is a general solution to the problem of discontinuities in the system you are solving. Integrate the system up to the point of the discontinuity, stop and re-start using a derivative function that reflects the system after the discontinuity.

There are two ways to stop the integration depending on the solution values.

The first way is to use the `/STOP={stopWave, mode}` flag, supplying a *stopWave* containing stopping conditions. *stopWave* must have one column for each equation in your system. Each column can specify stopping

Chapter III-10 — Analysis of Functions

on a value of the solution for the equation corresponding to the column, or stopping on a value of the derivative corresponding to that equation, or both. Each row has different significance:

Row	Meaning
0	Stop flag for solution value 0: Ignore condition on solution for this equation 1: Stop when solution value is greater than the value in row 1 -1: Stop when solution value is less than the value in row 1
1	Value of solution at which to stop
2	Stop flag for derivative 0: Ignore condition on derivative for this equation 1: Stop when derivative value is greater than the value in row 3 -1: Stop when derivative value is less than the value in row 3
3	Value of derivative at which to stop

In the chemical kinetics example above (see **A System of Coupled First-Order Equations** on page III-279) the system has four equations so you need a stop wave with four columns. This wave:

Row	ChemKin_Stop[][0]	ChemKin_Stop[][1]	ChemKin_Stop[][2]	ChemKin_Stop[][3]
0	0	0	-1	1
1	0	0	0.15	0.4
2	0	0	-1	0
3	0	0	0	0

will stop the integration when the concentration of species C (column 2) is less than 0.15, or when the concentration of species D (column 3) is greater than 0.4, or when the derivative of the concentration of species C is less than zero.

When you have multiple stopping criteria, as in this example, you can specify either OR stopping mode or AND stopping mode using the `mode` parameter of the `/STOP` flag. If `mode` is 0, OR mode is applied — any of the conditions with a non-zero flag will stop the integration. If `mode` is 1, AND mode is applied — all conditions with a non-zero flag must be satisfied in order for the integration to be stopped.

The second way to stop the integration is by returning a value of 1 from the derivative function. You can apply any condition you like in the function so it is possible to make much more complex stopping conditions this way than using the `/STOP` flag. However, the derivative function is called for a many intermediate points during a single step, some of which aren't necessarily even on the eventual solution trajectory. That means that you could be applying your stopping criterion to values that are not meaningful to the final solution. That may be particularly true at a time when the internal step size is contracting — the derivative function may be called for points beyond the eventual solution point as the solver tries a step size that doesn't succeed.

Integrating a User Function

You can use the `Integrate1D` function to numerically integrate a user function. For example, if you want to evaluate the integral

$$I(a, b) = \int_a^b \exp[-x^3 \sin(2\pi/x^2)] dx,$$

you need to start by defining the user function

```
Function userFunc(v)  
  Variable v
```

```
    return exp(-v^3*sin(2*pi/v^2))
End
```

The Integrate1D function supports three integration methods: Trapezoidal, Romberg and Gaussian Quadrature.

```
Printf "%.10f\r" Integrate1D(userfunc,0.1,0.5,0)      // default trapezoidal
0.3990547412
Printf "%.10f\r" Integrate1D(userfunc,0.1,0.5,1)      // Romberg
0.3996269165
Printf "%.10f\r" Integrate1D(userfunc,0.1,0.5,2,100) // Gaussian Q.
0.3990546953
```

For comparison, you can also execute:

```
Make/O/N=1000 tmp
Setscale/I x,.1,.5,"" tmp
tmp=userfunc(x)
Printf "%.10f\r" area(tmp,-inf,inf)
0.3990545084
```

Integrate1D can also handle complex valued functions. For example, if you want to evaluate the integral

$$I(a, b) = \int_a^b \exp\{ix \sin x\} dx'$$

a possible user function could be

```
Function/C cUserFunc(v)
    Variable v
    Variable/C arg=cmplx(0,v*sin(v))
    return exp(arg)
End
```

Note that the user function is declared with a /C flag and that Integrate1D must be assigned to a complex number in order for it to accept a complex user function.

```
Variable/C complexResult=Integrate1D(cUserFunc,0.1,0.2,1)
print complexResult
(0.0999693,0.00232272)
```

Also note that if you just try to print the result without using a complex variable as shown above, you need to use the /C flag with print:

```
Print/C Integrate1D(cUserFunc,0.1,0.2,1)
```

in order to force the function to integrate a complex valued expression.

You can also evaluate multidimensional integrals with the help of Integrate1D. The trick is in recognizing the fact that the user function can itself return a 1D integral of another function which in turn can return a 1D integral of a third function and so on. Here is an example of integrating a 2D function: $f(x,y) = 2x + 3y + xy$.

```
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)
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
  return Integrate1D(userFunction1,globalXmin,globalXmax,1)
End
```

Finding Function Roots

The **FindRoots** operation finds roots or zeros of a nonlinear function, a system of nonlinear functions, or of a polynomial with real coefficients.

Here we discuss how the operation works, and give some examples. The discussion falls naturally into three sections:

- Polynomial roots
- Roots of 1D nonlinear functions
- Roots of systems of multidimensional nonlinear functions

Igor's **FindRoots** operation finds function zeroes. Naturally, you can find other solutions as well. If you have a function $f(x)$ and you want to find the X values that result in $f(x) = 1$, you would find roots of the function $g(x) = f(x) - 1$. The **FindRoots** operation provides the **/Z** flag to make this more convenient.

A related problem is to find places in a curve defined by data points where the data pass through zero or another value. In this case, you don't have an analytical expression of the function. For this, use either the **FindLevel** operation (see page V-209) or the **FindLevels** operation (see page V-210); applications of these operations are discussed under **Level Detection** on page III-256.

Roots of Polynomials with Real Coefficients

The **FindRoots** operation can find all the complex roots of a polynomial with real coefficients. As an example, we will find the roots of

$$x^4 - 3.75x^2 - 1.25x + 1.5$$

We just happen to know that this polynomial can be factored as $(x+1)(x-2)(x+1.5)(x-0.5)$ so we already know what the roots are. But let's use Igor to do the work.

First, we need to make a wave with the polynomial coefficients. The wave must have $N+1$ points, where N is the degree of the polynomial. Point zero is the coefficient for the constant term, the last point is the coefficient for the highest-order term:

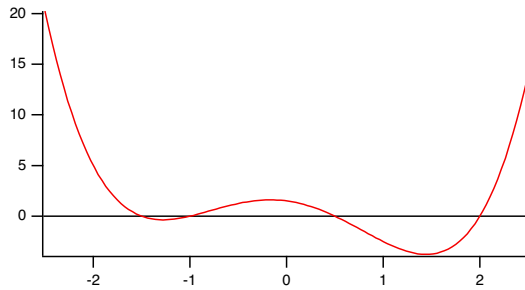
```
Make/D/O PolyCoefs = {1.5, -1.25, -3.75, 0, 1}
```

This wave can be used with the **poly** function to generate polynomial values. For instance:

```
Make/D/O PWave // a wave with 128 points
SetScale/I x -2.5,2.5,PWave // give it an X range of (-2.5, 2.5)
PWave = Poly(PolyCoefs, x) // fill it with polynomial values

Display PWave // and make a graph of it
ModifyGraph zero(left)=1 // add a zero line to show the roots
```

These commands make the following graph:



Now use FindRoots to find the roots:

```
FindRoots/P=PolyCoefs // roots are now in W_polyRoots
Print W_polyRoots
```

This prints:

```
W_polyRoots[0] = {cplx(0.5,0), cplx(-1,0), cplx(-1.5,0), cplx(2,0)}
```

Note that the imaginary part of the roots are zero, because this polynomial was constructed from real factors. In general, this won't be the case.

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).

Roots of a 1D Nonlinear Function

Unlike the case with polynomials, there is no general method for finding all the roots of a nonlinear function. Igor searches for a root of the function using Brent's method, and, depending on circumstances will find one or two roots in one shot.

You must write a user-defined function to define the function whose roots you want to find. Igor calls your function with values of X in the process of searching for a root. The format of the function is as follows:

```
Function myFunc(w,x)
    Wave w
    Variable x

    return <an arithmetic expression>
End
```

The wave w is a coefficient wave — it specifies constant coefficients that you may need to include in the function. It provides a convenient way to alter the coefficients so that you can find roots of members of a function family without having to edit your function code every time. Igor does not alter the values in w .

As an example we will find roots of the function $y = a + b \cdot \text{sinc}(c \cdot (x - x_0))$. Here is a user-defined function to implement this:

```
Function mySinc(w, x)
    Wave w
    Variable x

    return w[0] + w[1] * sinc(w[2] * (x - w[3]))
End
```

Enter this code into the Procedure window and then close the window.

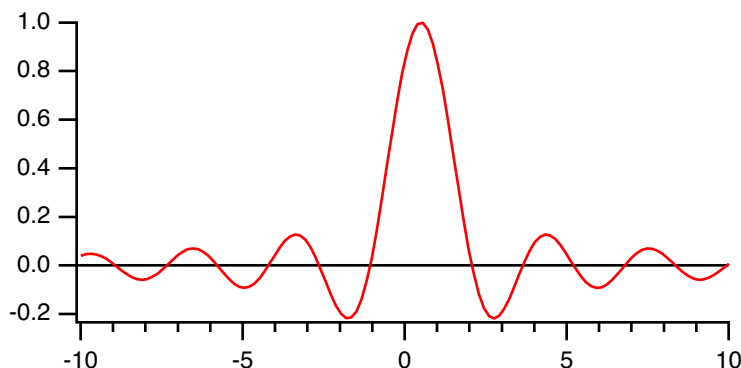
Make a graph of the function:

```
Make/D/O SincCoefs={0, 1, 2, .5} // a sinc function offset by 0.5
Make/D/O SincWave // a wave with 128 points
```

Chapter III-10 — Analysis of Functions

```
SetScale/I x -10,10,SincWave // give it an X range of (-2, 2)
SincWave = mySinc(SincCoefs, x) // fill it with function values

Display SincWave // and make a graph of it
ModifyGraph zero(left)=1 // add a zero line to show the roots
ModifyGraph minor(bottom)=1 // add minor ticks to the graph
```



Now we're ready to find roots.

The algorithm for finding roots requires that the roots first be bracketed, that is, you need to know two X values that are on either side of the root (that is, that give Y values of opposite sign). Making a graph of the function as we did here is a good way to figure out bracketing values. For instance, inspection of the graph above shows that there is a root between $x=1$ and $x=3$. The FindRoots command line to find a root in this range is

```
FindRoots/L=1/H=3 mySinc, SincCoefs
```

The /L flag sets the lower bracket and the /H flag sets the upper bracket. Igor then finds the root between these values, and prints the results in the history:

```
Possible root found at 2.0708
Y value there is -1.46076e-09
```

Igor reports to you both the root (in this case 2.0708) and the Y value at the root, which should be very close to zero (in this case it is -1.46×10^{-9}). Some pathological functions can fool Igor into thinking it has found a root when it hasn't. The Y value at the supposed root should identify if this has happened.

The bracketing values don't actually have to be at points of opposite sign. If the bracket encloses an extreme point, Igor will find it and then find two roots. Thus, you might use this command:

```
FindRoots/L=0/H=5 mySinc, SincCoefs
```

The Y values at $X=0$ and $X=5$ are both positive. Igor finds the minimum point at about $X=2.7$ and then uses that with the original bracketing values as the starting points for finding two roots. The result, printed in the history:

```
Looking for two roots...
```

```
Results for first root:
Possible root found at 2.0708
Y value there is -2.99484e-11
```

```
Results for second root:
Possible root found at 3.64159
Y value there is 3.43031e-11
```

Finally, it isn't always necessary to provide bracketing values. If the /L and /H flags are absent, Igor assigns them the values 0 and 1. In this case, there is no root between $X=0$ and $X=1$, and there is no extreme point. So Igor searches outward in a series of expanding jumps looking for values that will bracket a root (that is, for X values having Y values of opposite sign). Thus, in this case, the following simple command works:

```
FindRoots mySinc, SincCoefs
```

Igor finds the first of the roots we found previously:

```
Possible root found at 2.0708
Y value there is 2.748e-13
```

You may have noticed by now that FindRoots is reporting Y values that are merely small instead of zero. It isn't usually possible to find exact roots with a computer. And asking for very high accuracy requires more iterations of the search algorithm. If function evaluation is time-consuming and you don't need much accuracy, you may not want to find the root with high accuracy. Consequently, you can use the /T flag to alter the acceptable accuracy:

```
FindRoots/T=1e-3 mySinc, SincCoefs // low accuracy
Possible root found at 2.07088
  Y value there is -5.5659e-05
Print/D V_root, V_YatRoot
  2.07088376061435 -5.56589998578327e-05
```

```
FindRoots/T=1e-15 mySinc, SincCoefs // high accuracy
Possible root found at 2.0708
  Y value there is 0
Print/D V_root, V_YatRoot
  2.0707963267949 0
```

This also illustrates another point: the results of FindRoots are stored in variables. We used these variables in this case to print the results to higher accuracy than the six digits used by the report printed by FindRoots.

Roots of a System of Multidimensional Nonlinear Functions

Finding roots of a system of multidimensional nonlinear functions works very similarly to finding roots of a 1D nonlinear function. You provide user-defined functions that define your functions. These functions have nearly the same form as a 1D function, but they have a parameter for each independent variable. For instance, if you are going to find roots of a pair of 2D functions, the functions will look like this:

```
Function myFunc1(w, x1, x2)
  Wave w
  Variable x1, x2

  return <an arithmetic expression>
End
```

```
Function myFunc2(w, x1, x2)
  Wave w
  Variable x1, x2

  return <an arithmetic expression>
End
```

These function look just like the 1D function mySinc we wrote in the previous section, but they have two input X variables, one for each dimension. The number of functions must match the number of dimensions.

We will use the functions

```
f1= w[0]*sin((x-3)/w[1])*cos(y/w[2])
and
f2 = w[0]*cos(x/w[1])*tan((y+5)/w[2])
```

Enter this code into your procedure window:

```
Function myf1(w, xx, yy)
  Wave w
  Variable xx,yy
```

Chapter III-10 — Analysis of Functions

```
    return w[0]*sin(xx/w[1])*cos(yy/w[2])
End

Function myf2(w, xx, yy)
    Wave w
    Variable xx,yy

    return w[0]*cos(xx/w[1])*tan(yy/w[2])
End
```

Before starting, let's make a contour plot to see what we're up against. Here are some commands to make a convenient one:

```
Make/D/O params2D={1,5,4}           // nice set of parameters for both f1 and f2
Make/D/O/N=(50,50) f1Wave, f2Wave  // matrix waves for contouring
SetScale/I x -20,20,f1Wave, f2Wave // nice range of X values
SetScale/I y -20,20,f1Wave, f2Wave // and Y values
f1Wave = myf1(params2D, x, y)       // fill f1Wave with values from f1(x,y)
f2Wave = myf2(params2D, x, y)       // fill f2Wave with values from f2(x,y)
Display /W=(5,42,399,396)           // graph window for contour plot
AppendMatrixContour f1Wave
AppendMatrixContour f2Wave
ModifyContour f2Wave labels=0        // suppress contour labels
ModifyContour f1Wave labels=0
ModifyContour f1Wave rgbLines=(65535,0,0) // make f1 red
ModifyContour f2Wave rgbLines=(0,0,65535) // make f2 blue
ModifyGraph lsize('f2Wave=0')=2     // make zero contours heavy
ModifyGraph lsize('f1Wave=0')=2
```

Places where the zero contours for the two functions cross are the roots we are looking for. In the contour plot you can see several, for instance the points (0,0) and (7.8, 6.4) are approximate roots.

The algorithm that searches for roots needs a starting point. You can specify this in the FindRoots command with the /X flag, or if you don't use /X, Igor starts by default at the origin, $X_n = 0$. You must also specify both functions and a coefficient wave for each function. In this case we will use the same coefficient wave for each. The functions and coefficient waves are specified in pairs. Since we are looking for roots of two 2D functions, we have two function-wave pairs:

```
FindRoots myf1,params2D, myf2,params2D
```

Igor finds a root at the origin, and prints the results. The X,Y coordinates of the root are stored in the wave W_Root:

```
    Root found after 4 function evaluations.
W_Root={0,0}
    Function values at root:
W_YatRoot={0,0}
```

The wave W_YatRoot holds the values of each of the functions evaluated at the root.

If that's not the root you want to find, use /X to specify a different starting point:

```
    FindRoots/X={7.7,6.3} myf1,params2D, myf2,params2D
    Root found after 47 function evaluations.
W_Root={-1.10261e-14,12.5664}
    Function values at root:
W_YatRoot={2.20522e-15,-1.89882e-15}
```

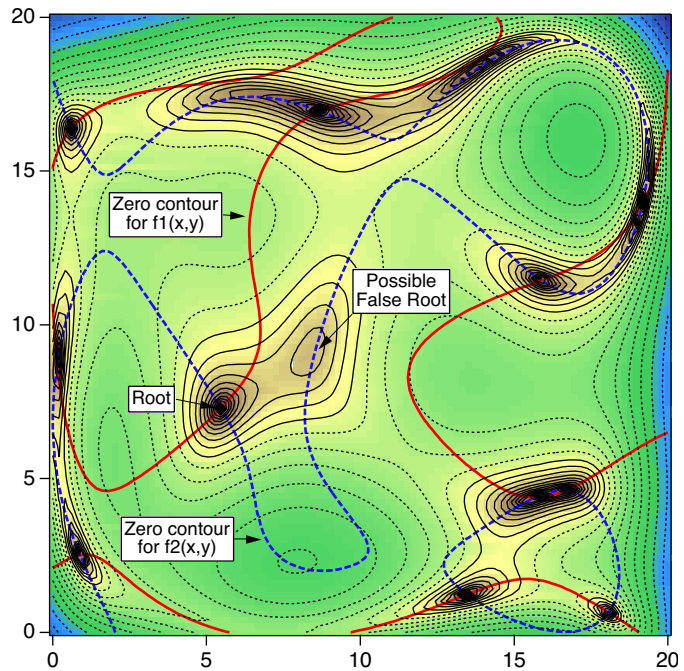

Caveats for Multidimensional Root Finding

Finding roots of multidimensional nonlinear functions is not straightforward. There is no general, foolproof way to do it. The method Igor uses is to search for minima in the sum of the squares of the functions. Since the squared values must be positive, the only places where this sum can be zero is at points where all the functions are zero at the same time. That point is a root, and it is also a minimum in the summed squares of the functions.

To find the zero points, Igor searches for local minima by travelling downhill from the starting point. Unfortunately, a local minimum doesn't have to be a root, it just has to be someplace where the sum of squares of the functions is less than surrounding points.

The adjacent graph shows how this can happen.

The two heavy lines are the zero contours for two functions (they happen to be fifth-order 2D polynomials). Where these zero contours cross are the roots for the system of the two functions.



The thin lines are contours of $f_1(x,y)^2 + f_2(x,y)^2$, with dotted lines for high values; minima are surrounded by thin, solid contours. You can see that every intersection between the heavy zero contours is surrounded by thin contours showing that these are minima in the sum of the squared functions. One such point is labeled "Root".

There is at least one point, labelled "False Root", where there is a minimum but the zero contours don't cross. That is not a root, but FindRoots may find it anyway. For instance, a real root:

```
FindRoots /x={3,6} MyPoly2d, nn1coefs, MyPoly2d, nn2coefs
Root found after 11 function evaluations.
W_Root={5.4623,7.28975}
Function values at root:
W_YatRoot={-4.15845e-13,1.08297e-12}
```

This point is the point marked "Root". However:

```
FindRoots/x={9,10} MyPoly2d, nn1coefs, MyPoly2d, nn2coefs
Root found after 52 function evaluations.
W_Root={8.38701,9.10129}
Function values at root:
W_YatRoot={0.0686792,0.0129881}
```

You can see from the values in $W_YatRoot$ that this is not a root. This point is marked "False root" on the figure above.

The polynomials used in this example have too many coefficients to be conveniently shown here. To see this example and others in action, try out the demo experiment. It is called "MD Root Finder Demo" and you will find it in your Igor Pro 7 folder, in the Examples:Analysis: folder.

Finding Minima and Maxima of Functions

The **Optimize** operation finds extreme values (maxima or minima) of a nonlinear function.

Chapter III-10 — Analysis of Functions

Here we discuss how the operation works, and give some examples. The discussion falls naturally into two sections:

- Extrema of 1D nonlinear functions
- Extrema of multidimensional nonlinear functions

A related problem is to find peaks or troughs in a curve defined by data points. In this case, you don't have an analytical expression of the function. To do this with one dimensional data, use the **FindPeak** operation (see page V-212).

Extreme Points of a 1D Nonlinear Function

The Optimize operation finds local maxima or minima of functions. That is, if a function has some X value where the nearby Y values are all higher than at that X value, it is deemed to be a minimum. Finding the point where a functions value is lower or higher than any other point anywhere is a much more difficult problem that is not addressed by the Optimize operation.

You must write a user-defined function to define the function for which the extreme points are calculated. Igor will call your function with values of X in the process of searching for a root. The format of the function is as follows:

```
Function myFunc(w,x)
    Wave w
    Variable x

    return f(x)          // an expression...
End
```

The wave w is a coefficient wave — it specifies constant coefficients that you may need to include in the function. It provides a convenient way to alter the coefficients so that you can find extreme points of members of a function family without having to edit your function code every time. Igor does not alter the values in w.

Although the coefficient wave must be present in the Function declaration, it does not have to be referenced in the function body. This may save computation time, arriving at the solution faster. You will have to create a dummy wave to list in the FindRoots command.

As an example we will find extreme points of the equation

$$y = a+b*\text{sinc}(c*(x-x_0))$$

A suitable user-defined function might look like this:

```
Function mySinc(w, x)
    Wave w
    Variable x

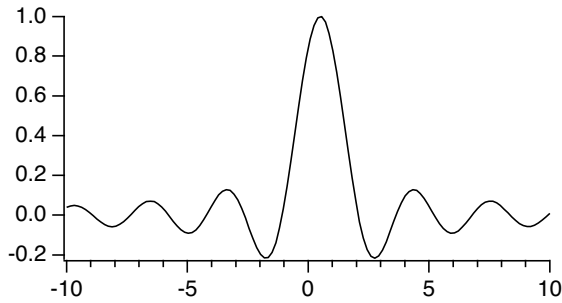
    return w[0]+w[1]*sinc(w[2]*(x-w[3]))
End
```

Enter this code into the Procedure window and then close the window.

Make a graph of the function:

```
Make/D/O SincCoefs={0, 1, 2, .5} // a sinc function offset by 0.5
Make/D/O SincWave                // a wave with 128 points
SetScale/I x -10,10,SincWave     // give it an X range of [-10, 10]
SincWave = mySinc(SincCoefs, x) // fill it with function values

Display SincWave                // and make a graph of it
ModifyGraph minor(bottom)=1     // add minor ticks to the graph
```



Now we're ready to find extreme points.

The algorithm for finding extreme points requires that the extreme points first be bracketed, that is, you need to know two X values that are on either side of the extreme point (that is, two points that have a lower or higher point between). Making a graph of the function as we did here is a good way to figure out bracketing values. For instance, inspection of the graph above shows that there is a minimum between $x=1$ and $x=4$. The Optimize command line to find a minimum in this range is

```
Optimize/L=1/H=4 mySinc, SincCoefs
```

The /L flag sets the lower bracket and the /H flag sets the upper bracket. Igor then finds the minimum between these values, and prints the results in the history:

```
Optimize probably found a minimum. Optimize stopped because
the Optimize operation found a minimum within the specified tolerance.
Current best solution: 2.7467
Function value at solution: -0.217234
 13 iterations, 14 function calls
V_minloc = 2.7467, V_min = -0.217234, V_OptNumIters = 13, V_OptNumFunctionCalls = 14
```

Igor reports to you both the X value that minimizes the function (in this case 2.7467) and the Y value at the minimum.

The bracketing values don't necessarily have to bracket the solution. Igor first tries to find the desired extremum between the bracketing values. If it fails, the bracketing interval is expanded searching for a suitable bracketing interval. If you don't use /L and /H, Igor sets the bracketing interval to [0,1]. In the case of the mySinc function, that doesn't include a minimum. Here is what happens in that case:

```
Optimize mySinc, SincCoefs
Optimize probably found a minimum. Optimize stopped because
the Optimize operation found a minimum within the specified tolerance.
Current best solution: 2.74671
Function value at solution: -0.217234
 16 iterations, 47 function calls
V_minloc = 2.74671, V_min = -0.217234, V_OptNumIters = 16, V_OptNumFunctionCalls = 47
```

Note that Igor found the same minimum that it found before.

The mySinc function makes it easy to find bracketing values because of the oscillatory nature of the function. Other functions may be more difficult if they contain just one extreme point, or if they have local extreme points but are unbounded elsewhere. Even in an easy case like mySinc, you can't be sure which extreme point Igor will find, so it is always better to supply a good bracket if you possibly can.

You may wish to find maximum points instead of minima. Use the /A flag to specify this:

```
Optimize/A/L=0/H=2 mySinc, SincCoefs
Optimize probably found a maximum. Optimize stopped because
the Optimize operation found a maximum within the specified tolerance.
Current best solution: 0.499999
Function value at solution: 1
 16 iterations, 17 function calls
V_maxloc = 0.499999, V_max = 1, V_OptNumIters = 16, V_OptNumFunctionCalls = 17
```

Chapter III-10 — Analysis of Functions

The results of the Optimize operation are stored in variables. Note that the report that Optimize prints in the history includes only six digits of the values. You can print the results to greater precision using the printf operation and the variables:

```
Printf "The max is at %.15g. The Y value there is %.15g\r", V_maxloc, V_max
```

yields this in the history:

```
The max is at 0.499999480759779. The Y value there is 0.99999999999982
```

Extrema of Multidimensional Nonlinear Functions

Finding extreme points of multidimensional nonlinear functions works very similarly to finding extreme points of a 1D nonlinear function. You provide a user-defined function having almost the same format as for 1D functions. A 2D function will look like this:

```
Function myFunc1(w, x1, x2)
    Wave w
    Variable x1, x2

    return f1(x1, x2)      // an expression...
End
```

This function looks just like the 1D function mySinc we wrote in the previous section, but it has two input X variables, one for each dimension.

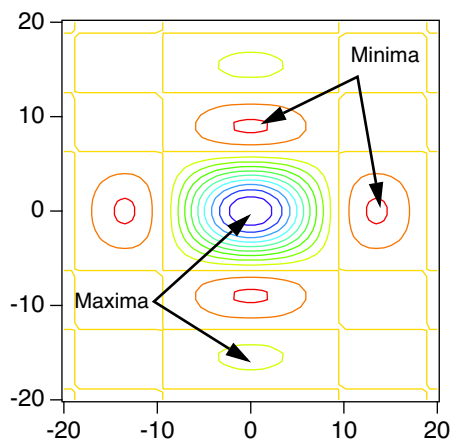
We will make a 2D function based on the sinc function. Copy this code into your procedure window:

```
Function Sinc2D(w, xx, yy)
    Wave w
    Variable xx,yy

    return w[0]*sinc(xx/w[1])*sinc(yy/w[2])
End
```

Before starting, let's make a contour plot to see what we're up against. Here are some commands to make a convenient one:

```
Make/D/O params2D={1,3,2}      // nice set of parameters
Make/D/O/N=(50,50) Sinc2DWave // matrix wave for contouring
SetScale/I x -20,20,Sinc2DWave // nice range of X values for these functions
SetScale/I y -20,20,Sinc2DWave // and Y values
Sinc2DWave = Sinc2D(params2D, x, y) // fill f1Wave with values from f1(x,y)
Display /W=(5,42,399,396)     // graph window for contour plot
AppendMatrixContour Sinc2DWave
ModifyContour Sinc2DWave labels=0 // suppress contour labels to reduce clutter
```



The algorithm that searches for extreme points needs a starting guess which you provide using the /X flag. Here is a command to find a minimum point:

```
Optimize/X={1,5} Sinc2D,params2D
```

This command results in the following report in the history:

```
Optimize probably found a minimum. Optimize stopped because
the gradient is nearly zero.
Current best solution:
  W_Extremum={-0.000147116,8.98677}
Function gradient:
  W_OptGradient={-4.65661e-08,1.11923e-08}
Function value at solution: -0.217234
11 iterations, 36 function calls
V_min = -0.217234, V_OptTermCode = 1, V_OptNumIters = 11, V_OptNumFunctionCalls = 36
```

Note that the minimum is reported via a wave called `W_extremum`. Had you specified a wave using the /X flag, that wave would have been used instead. Another wave, `W_OptGradient`, receives the function gradient at the solution point.

There are quite a few options available to modify the workings of the Optimize operation. See **Optimize** operation on page V-613 for details, including references to reading material.

Stopping Tolerances

The Optimize operation stops refining the solution when certain criteria are met. The most desirable result is that it stops because the function gradient is very close to zero, since that is (almost) diagnostic of an extreme point. The algorithm also will take very small steps near an extreme point, so this is also a stopping criterion. You can set the values for the stopping criteria using the /T={*gradTol*, *stepTol*} flag.

Optimize stops when these conditions are met:

$$\max_{1 \leq i \leq n} \left\{ |g_i| \cdot \frac{\max(|x_i|, \text{typ}X_i)}{\max(|f|, \text{funcSize})} \right\} \leq \text{gradTol}$$

or

$$\max_{1 \leq i \leq n} \left\{ \frac{|\Delta x_i|}{\max(x_i, \text{typ}X_i)} \right\} \leq \text{stepTol}$$

Note that these conditions use values of the gradient (g_i) and step size (Δx_i) that are scaled by a measure of the magnitude of values encountered in the problem. In these equations, x_i is the value of a component of the solution and f is the value of the function at the solution; $\text{typ}X_i$ is a “typical” value of the X component that is set by you using the /R flag and f is a typical function value magnitude which you set using the /Y flag. The values of $\text{typ}X_i$ and f are one if the /R and /F flags are not present.

The default values for *gradTol* and *stepTol* are $\{8.53618 \times 10^{-6}, 7.28664 \times 10^{-11}\}$. These are the values recommended by Dennis and Schnabel (see the references in **Optimize** operation on page V-613) for well-behaved functions when the function values have full double precision resolution. These values are $(6.022 \times 10^{-16})^{1/3}$ and $(6.022 \times 10^{-16})^{2/3}$ as suggested by Dennis and Schnabel (see the references in **Optimize** operation on page V-613), where 6.022×10^{-16} is the smallest double precision floating point number that, when added to 1, is different from 1. Usually the default is pretty good.

Due to floating point truncation errors, it is possible to set *gradTol* and *stepTol* to values that can never be achieved. In that case you may get a message about “no solution was found that is better than the last iteration”.

Problems with Multidimensional Optimization

Finding minima of multidimensional functions is by no means foolproof. The methods used by the Optimize operation are “globally convergent” which means that under suitable circumstances Optimize will be able to find some extreme point from just about any starting guess.

Chapter III-10 — Analysis of Functions

If the gradient of your function is zero, or very nearly so at the starting guess, Optimize has no information on which way to go to find an extreme point. Note that the Sinc2D function has a maximum exactly at (0,0). Here is what happens if you try to find a minimum starting at the origin:

```
Optimize/X={0,0} Sinc2D,params2D
==== The Optimize operation failed to find a minimum. ====
Optimize stopped because
The function gradient at your starting guess is too near zero, suggesting that
it is a critical point.
A different starting guess usually solves this problem.
Current best solution:
  W_Extremum={0,0}
Function gradient:
  W_OptGradient={0,0}
Function value at solution: 1
0 iterations, 3 function calls
V_min = 1, V_OptTermCode = 6, V_OptNumIters = 0, V_OptNumFunctionCalls = 3
```

In this example the function gradient is zero at the origin because there is a function maximum there. A gradient of zero could also be a minimum or a saddle point.

The algorithms used by the Optimize operation assume that your function is smooth, that is, that the first and second derivatives are continuous. Optimize may work with functions that violate this assumption, but it is not guaranteed.

Although Optimize tends to look downhill to the nearest minimum (or uphill to the nearest maximum), it is not guaranteed to find any particular minimum, especially if your starting guess is near a point where the gradient is small. Sometimes using a different method (*/M={stepMethod, hessianMethod}*) will result in a different answer. You can limit the maximum step size to keep progress more or less local (*/S=maxStep*). If you set the maximum step size too small, however, Optimize may stop early because the maximum step size is exceeded too many times. Here is an example using the Sinc2D function. If the starting guess is near the origin, the gradient is small and the solution shoots off into the hinterlands (only a portion of the history report is shown):

```
Optimize/X={1,1} Sinc2D,params2D
  W_Extremum={-61.1192,298.438}
Function gradient:
  W_OptGradient={-1.04095e-08,-1.19703e-08}
```

Use */S* to limit the step size, and find a minimum nearer to the starting guess:

```
Optimize/X={1,1}/S=10 Sinc2D,params2D
  W_Extremum={-0.00014911,8.9868}
Function gradient:
  W_OptGradient={9.31323e-09,5.92775e-08}
```

But if the maximum step is too small, it doesn't work:

```
Optimize/X={1,1}/S=1 Sinc2D,params2D
==== The Optimize operation failed to find a minimum. ====
Optimize stopped because
the maximum step size was exceeded in five consecutive iterations.
This can happen if the function is unbounded (there is no minimum),
or the function approaches the minimum asymptotically. It may also be that the
maximum step size (1) is too small.
```

Another way to get a solution near by is to set the initial trust region to a small value. This works if you select double dogleg or More Hebdon as the step selection method. It does not apply to the default line search method. Here is an example (note that the double dogleg method is selected using */M={1,0}*):

```
Optimize/X={1,1}/M={1,0}/F=1 Sinc2D,params2D
  W_Extremum={-0.000619834,8.9872}
Function gradient:
  W_OptGradient={1.09896e-07,2.9017e-08}
```

References

The IntegrateODE operation is based on routines in *Numerical Recipes*, and are used by permission:

Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C*, 2nd ed., 994 pp., Cambridge University Press, New York, 1992.

The Adams-Moulton and BDF methods 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.

The Optimize operation uses Brent's method for univariate functions. *Numerical Recipes* has an excellent discussion in section 10.2 of this method (but we didn't use their code).

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.

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..

