```
	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:

$$Max\left(\frac{Error_i}{Scale_i}\right) < eps$$

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:

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