formulas. It is not difficult to show that the finite difference formulas are equivalent to the derivatives of interpolating polynomials. For example, a simple way of deriving the formula just shown for the second derivative is to interpolate a quadratic and find its second derivative (which is essentially just the leading coefficient).

This finds the three-point finite difference formula for the second derivative by differentiating the polynomial interpolating the three points $(x_{i-1}, f(x_{i-1}))$, $(x_i, f(x_i))$, and $(x_{i+1}, f(x_{i+1}))$.

```
In[9]:= D[InterpolatingPolynomial[Table[{x_{i+k}, f[x_{i+k}]}, {k, -1, 1}], z], z, z]
```

$$\text{Out[9]}= \frac{2\left(-\frac{-f[x_{-1+i}]+f[x_i]}{-x_{-1+i}+x_i}+\frac{-f[x_i]+f[x_{1+i}]}{-x_i+x_{1+i}}\right)}{-x_{-1+i}+x_{1+i}}$$

In this form of the formula, it is easy to see that it is effectively a difference of the forward and backward first-order derivative approximations. Sometimes it is advantageous to use finite differences in this way, particularly for terms with coefficients inside of derivatives, such as $(a(x)\,u_x)_x$, which commonly appear in PDEs.

Another property made apparent by considering interpolation formulas is that the point at which you get the derivative approximation need not be on the grid. A common use of this is with staggered grids where the derivative may be wanted at the midpoints between grid points.

This generates a fourth-order approximation for the first derivative on a uniform staggered grid, x_i, where the main grid points $x_{i+k/2}$ are at $x_i + h\,k/2$, for odd k.

```
In[10]:= Simplify[
          D[InterpolatingPolynomial[Table[{x_i + k h / 2, f[x_{i+k/2}]}, {k, -3, 3, 2}], z], z] /.
            z → x_i]
```

$$\text{Out[10]}= \frac{f\left[x_{-\frac{3}{2}+i}\right]-27\,f\left[x_{-\frac{1}{2}+i}\right]+27\,f\left[x_{\frac{1}{2}+i}\right]-f\left[x_{\frac{3}{2}+i}\right]}{24\,h}$$

The fourth-order error coefficient for this formula is $\frac{3}{640}\,h^4\,f^{(5)}(x_i)$ versus $\frac{1}{30}\,h^4\,f^{(5)}(x_i)$ for the standard fourth-order formula derived next. Much of the reduced error can be attributed to the reduced stencil size.

This generates a fourth-order approximation for the first derivative at a point on a uniform grid.

```
In[11]:= Simplify[
          D[InterpolatingPolynomial[Table[{x_i + k h, f[x_{i+k}]}, {k, -2, 2, 1}], z], z] /.
            z → x_i]
```

$$\text{Out[11]}= \frac{f[x_{-2+i}]-8\,f[x_{-1+i}]+8\,f[x_{1+i}]-f[x_{2+i}]}{12\,h}$$

In general, a finite difference formula using n points will be exact for functions that are polynomials of degree $n-1$ and have asymptotic order at least $n-m$. On uniform grids, you can expect higher asymptotic order, especially for centered differences.

Using efficient polynomial interpolation techniques is a reasonable way to generate coefficients, but B. Fornberg has developed a fast algorithm for finite difference weight generation [F92], [F98], which is substantially faster.

In [F98], Fornberg presents a one-line *Mathematica* formula for explicit finite differences.

This is the simple formula of Fornberg for generating weights on a uniform grid. Here it has been modified slightly by making it a function definition.

```
In[12]:= UFDWeights[m_, n_, s_] :=
          CoefficientList[Normal[Series[x^s Log[x]^m, {x, 1, n}] / h^m], x]
```

Here m is the order of the derivative, n is the number of grid intervals enclosed in the stencil, and s is the number of grid intervals between the point at which the derivative is approximated and the leftmost edge of the stencil. There is no requirement that s be an integer; noninteger values simply lead to staggered grid approximations. Setting s to be $n/2$ always generates a centered formula.

This uses the Fornberg formula to generate the weights for a staggered fourth-order approximation to the first derivative. This is the same one computed earlier with `InterpolatingPolynomial`.

```
In[13]:= UFDWeights[1, 3, 3 / 2]
```

$$\text{Out[13]}= \left\{\frac{1}{24\,h}, -\frac{9}{8\,h}, \frac{9}{8\,h}, -\frac{1}{24\,h}\right\}$$

A table of some commonly used finite difference formulas follows for reference.

formula	*error term*
$f'(x_i) \simeq \frac{f(x_{i-2})-4\,f(x_{i-1})+3\,f(x_i)}{2\,h}$	$\frac{1}{3}\,h^2\,f^{(3)}$
$f'(x_i) \simeq \frac{f(x_{i+1})-f(x_{i-1})}{2\,h}$	$\frac{1}{6}\,h^2\,f^{(3)}$
$f'(x_i) \simeq \frac{-3\,f(x_i)+4\,f(x_{i+1})-f(x_{i+2})}{2\,h}$	$\frac{1}{3}\,h^2\,f^{(3)}$

$f'(x_i) \simeq \frac{3 f(x_{i-4})-16 f(x_{i-3})+36 f(x_{i-2})-48 f(x_{i-1})+25 f(x_i)}{12 h}$	$\frac{1}{5} h^4 f^{(5)}$
$f'(x_i) \simeq \frac{-f(x_{i-3})+6 f(x_{i-2})-18 f(x_{i-1})+10 f(x_i)+3 f(x_{i+1})}{12 h}$	$\frac{1}{20} h^4 f^{(5)}$
$f'(x_i) \simeq \frac{f(x_{i-2})-8 f(x_{i-1})+8 f(x_{i+1})-f(x_{i+2})}{12 h}$	$\frac{1}{30} h^4 f^{(5)}$
$f'(x_i) \simeq \frac{-3 f(x_{i-1})-10 f(x_i)+18 f(x_{i+1})-6 f(x_{i+2})+f(x_{i+3})}{12 h}$	$\frac{1}{20} h^4 f^{(5)}$
$f'(x_i) \simeq \frac{-25 f(x_i)+48 f(x_{i+1})-36 f(x_{i+2})+16 f(x_{i+3})-3 f(x_{i+4})}{12 h}$	$\frac{1}{5} h^4 f^{(5)}$
$f'(x_i) \simeq \frac{10 f(x_{i-6})-72 f(x_{i-5})+225 f(x_{i-4})-400 f(x_{i-3})+450 f(x_{i-2})-360 f(x_{i-1})+147 f(x_i)}{60 h}$	$\frac{1}{7} h^6 f^{(7)}$
$f'(x_i) \simeq \frac{-2 f(x_{i-5})+15 f(x_{i-4})-50 f(x_{i-3})+100 f(x_{i-2})-150 f(x_{i-1})+77 f(x_i)+10 f(x_{i+1})}{60 h}$	$\frac{1}{42} h^6 f^{(7)}$
$f'(x_i) \simeq \frac{f(x_{i-4})-8 f(x_{i-3})+30 f(x_{i-2})-80 f(x_{i-1})+35 f(x_i)+24 f(x_{i+1})-2 f(x_{i+2})}{60 h}$	$\frac{1}{105} h^6 f^{(7)}$
$f'(x_i) \simeq \frac{-f(x_{i-3})+9 f(x_{i-2})-45 f(x_{i-1})+45 f(x_{i+1})-9 f(x_{i+2})+f(x_{i+3})}{60 h}$	$\frac{1}{140} h^6 f^{(7)}$
$f'(x_i) \simeq \frac{2 f(x_{i-2})-24 f(x_{i-1})-35 f(x_i)+80 f(x_{i+1})-30 f(x_{i+2})+8 f(x_{i+3})-f(x_{i+4})}{60 h}$	$\frac{1}{105} h^6 f^{(7)}$
$f'(x_i) \simeq \frac{-10 f(x_{i-1})-77 f(x_i)+150 f(x_{i+1})-100 f(x_{i+2})+50 f(x_{i+3})-15 f(x_{i+4})+2 f(x_{i+5})}{60 h}$	$\frac{1}{42} h^6 f^{(7)}$
$f'(x_i) \simeq \frac{-147 f(x_i)+360 f(x_{i+1})-450 f(x_{i+2})+400 f(x_{i+3})-225 f(x_{i+4})+72 f(x_{i+5})-10 f(x_{i+6})}{60 h}$	$\frac{1}{7} h^6 f^{(7)}$

Finite difference formulas on uniform grids for the first derivative.

formula	*error term*
$f''(x_i) \simeq \frac{-f(x_{i-3})+4 f(x_{i-2})-5 f(x_{i-1})+2 f(x_i)}{h^2}$	$\frac{11}{12} h^2 f^{(4)}$
$f''(x_i) \simeq \frac{f(x_{i-1})-2 f(x_i)+f(x_{i+1})}{h^2}$	$\frac{1}{12} h^2 f^{(4)}$
$f''(x_i) \simeq \frac{2 f(x_i)-5 f(x_{i+1})+4 f(x_{i+2})-f(x_{i+3})}{h^2}$	$\frac{11}{12} h^2 f^{(4)}$
$f''(x_i) \simeq \frac{-10 f(x_{i-5})+61 f(x_{i-4})-156 f(x_{i-3})+214 f(x_{i-2})-154 f(x_{i-1})+45 f(x_i)}{12 h^2}$	$\frac{137}{180} h^4 f^{(6)}$
$f''(x_i) \simeq \frac{f(x_{i-4})-6 f(x_{i-3})+14 f(x_{i-2})-4 f(x_{i-1})-15 f(x_i)+10 f(x_{i+1})}{12 h^2}$	$\frac{13}{180} h^4 f^{(6)}$
$f''(x_i) \simeq \frac{-f(x_{i-2})+16 f(x_{i-1})-30 f(x_i)+16 f(x_{i+1})-f(x_{i+2})}{12 h^2}$	$\frac{1}{90} h^4 f^{(6)}$

$$f''(x_i) \simeq \frac{10 f(x_{i-1}) - 15 f(x_i) - 4 f(x_{i+1}) + 14 f(x_{i+2}) - 6 f(x_{i+3}) + f(x_{i+4})}{12 h^2} \qquad \frac{13}{180} h^4 f^{(6)}$$

$$f''(x_i) \simeq \frac{45 f(x_i) - 154 f(x_{i+1}) + 214 f(x_{i+2}) - 156 f(x_{i+3}) + 61 f(x_{i+4}) - 10 f(x_{i+5})}{12 h^2} \qquad \frac{137}{180} h^4 f^{(6)}$$

$$f''(x_i) \simeq \frac{1}{180 h^2} (-126 f(x_{i-7}) + 1019 f(x_{i-6}) - 3618 f(x_{i-5}) + 7380 f(x_{i-4}) - 9490 f(x_{i-3}) + 7911 f(x_{i-2}) - 4014 f(x_{i-1}) + 938 f(x_i)) \qquad \frac{363}{560} h^6 f^{(8)}$$

$$f''(x_i) \simeq \frac{1}{180 h^2} (11 f(x_{i-6}) - 90 f(x_{i-5}) + 324 f(x_{i-4}) - 670 f(x_{i-3}) + 855 f(x_{i-2}) - 486 f(x_{i-1}) - 70 f(x_i) + 126 f(x_{i+1})) \qquad \frac{29}{560} h^6 f^{(8)}$$

$$f''(x_i) \simeq \frac{1}{180 h^2} (-2 f(x_{i-5}) + 16 f(x_{i-4}) - 54 f(x_{i-3}) + 85 f(x_{i-2}) + 130 f(x_{i-1}) - 378 f(x_i) + 214 f(x_{i+1}) - 11 f(x_{i+2})) \qquad \frac{47}{5040} h^6 f^{(8)}$$

$$f''(x_i) \simeq \frac{2 f(x_{i-3}) - 27 f(x_{i-2}) + 270 f(x_{i-1}) - 490 f(x_i) + 270 f(x_{i+1}) - 27 f(x_{i+2}) + 2 f(x_{i+3})}{180 h^2} \qquad \frac{1}{560} h^6 f^{(8)}$$

$$f''(x_i) \simeq \frac{1}{180 h^2} (-11 f(x_{i-2}) + 214 f(x_{i-1}) - 378 f(x_i) + 130 f(x_{i+1}) + 85 f(x_{i+2}) - 54 f(x_{i+3}) + 16 f(x_{i+4}) - 2 f(x_{i+5})) \qquad \frac{47}{5040} h^6 f^{(8)}$$

$$f''(x_i) \simeq \frac{1}{180 h^2} (126 f(x_{i-1}) - 70 f(x_i) - 486 f(x_{i+1}) + 855 f(x_{i+2}) - 670 f(x_{i+3}) + 324 f(x_{i+4}) - 90 f(x_{i+5}) + 11 f(x_{i+6})) \qquad \frac{29}{560} h^6 f^{(8)}$$

$$f''(x_i) \simeq \frac{1}{180 h^2} (938 f(x_i) - 4014 f(x_{i+1}) + 7911 f(x_{i+2}) - 9490 f(x_{i+3}) + 7380 f(x_{i+4}) - 3618 f(x_{i+5}) + 1019 f(x_{i+6}) - 126 f(x_{i+7})) \qquad \frac{363}{560} h^6 f^{(8)}$$

Finite difference formulas on uniform grids for the second derivative.

One thing to notice from this table is that the farther the formulas get from centered, the larger the error term coefficient, sometimes by factors of hundreds. For this reason, sometimes where one-sided derivative formulas are required (such as at boundaries), formulas of higher order are used to offset the extra error.

NDSolve`FiniteDifferenceDerivative

Fornberg [F92], [F98] also gives an algorithm that, though not quite so elegant and simple, is more general and, in particular, is applicable to nonuniform grids. It is not difficult to program in *Mathematica*, but to make it as efficient as possible, a new kernel function has been provided as a simpler interface (along with some additional features).

`NDSolve`FiniteDifferenceDerivative[Derivative[m], grid, values]`

approximate the m^{th}-order derivative for the function that takes on values on the grid

`NDSolve`FiniteDifferenceDerivative[Derivative[m1, m2, ..., mn], {grid1, grid2, ..., gridn}, values]`

approximate the partial derivative of order (m_1, m_2, …, m_n) for the function of n variables that takes on values on the tensor product grid defined by the outer product of ($grid_1$, $grid_2$, …, $grid_n$)

`NDSolve`FiniteDifferenceDerivative[Derivative[m1, m2, ..., mn], {grid1, grid2, ..., gridn}]`

compute the finite difference weights needed to approximate the partial derivative of order (m_1, m_2, …, m_n) for the function of n variables on the tensor product grid defined by the outer product of ($grid_1$, $grid_2$, …, $grid_n$); the result is returned as an `NDSolve`FiniteDifferenceDerivativeFunction`, which can be repeatedly applied to values on the grid

Finding finite difference approximations to derivatives.

This defines a uniform grid with points spaced apart by a symbolic distance h.

```
In[14]:= ugrid = h Range[0, 8]
Out[14]= {0, h, 2 h, 3 h, 4 h, 5 h, 6 h, 7 h, 8 h}
```

This gives the first derivative formulas on the grid for a symbolic function `f`.

```
In[15]:= NDSolve`FiniteDifferenceDerivative[Derivative[1], ugrid, Map[f, ugrid]]
```

$$\text{Out[15]}= \left\{-\frac{25\,f[0]}{12\,h}+\frac{4\,f[h]}{h}-\frac{3\,f[2\,h]}{h}+\frac{4\,f[3\,h]}{3\,h}-\frac{f[4\,h]}{4\,h},\right.$$
$$-\frac{f[0]}{4\,h}-\frac{5\,f[h]}{6\,h}+\frac{3\,f[2\,h]}{2\,h}-\frac{f[3\,h]}{2\,h}+\frac{f[4\,h]}{12\,h},\ \frac{f[0]}{12\,h}-\frac{2\,f[h]}{3\,h}+\frac{2\,f[3\,h]}{3\,h}-\frac{f[4\,h]}{12\,h},$$
$$\frac{f[h]}{12\,h}-\frac{2\,f[2\,h]}{3\,h}+\frac{2\,f[4\,h]}{3\,h}-\frac{f[5\,h]}{12\,h},\ \frac{f[2\,h]}{12\,h}-\frac{2\,f[3\,h]}{3\,h}+\frac{2\,f[5\,h]}{3\,h}-\frac{f[6\,h]}{12\,h},$$
$$\frac{f[3\,h]}{12\,h}-\frac{2\,f[4\,h]}{3\,h}+\frac{2\,f[6\,h]}{3\,h}-\frac{f[7\,h]}{12\,h},\ \frac{f[4\,h]}{12\,h}-\frac{2\,f[5\,h]}{3\,h}+\frac{2\,f[7\,h]}{3\,h}-\frac{f[8\,h]}{12\,h},$$
$$\left.-\frac{f[4\,h]}{12\,h}+\frac{f[5\,h]}{2\,h}-\frac{3\,f[6\,h]}{2\,h}+\frac{5\,f[7\,h]}{6\,h}+\frac{f[8\,h]}{4\,h},\ \frac{f[4\,h]}{4\,h}-\frac{4\,f[5\,h]}{3\,h}+\frac{3\,f[6\,h]}{h}-\frac{4\,f[7\,h]}{h}+\frac{25\,f[8\,h]}{12\,h}\right\}$$

The derivatives at the endpoints are computed using one-sided formulas. The formulas shown in the previous example are fourth-order accurate, which is the default. In general, when you

use a symbolic grid and/or data, you get symbolic formulas. This is often useful for doing analysis on the methods; however, for actual numerical grids, it is usually faster and more accurate to give the numerical grid to `NDSolve`FiniteDifferenceDerivative` rather than using the symbolic formulas.

This defines a randomly spaced grid between 0 and 2π.

```
In[16]:= rgrid = Sort[Join[{0, 2 π}, Table[2 π RandomReal[], {10}]]]
Out[16]= {0, 0.94367, 1.005, 1.08873, 1.72052, 1.78776, 2.41574, 2.49119, 2.93248, 4.44508, 6.20621, 2 π}
```

This approximates the derivative of the sine function at each point on the grid.

```
In[17]:= NDSolve`FiniteDifferenceDerivative[Derivative[1], rgrid, Sin[rgrid]]
Out[17]= {0.989891, 0.586852, 0.536072, 0.463601, -0.149152,
 -0.215212, -0.747842, -0.795502, -0.97065, -0.247503, 0.99769, 0.999131}
```

This shows the error in the approximations.

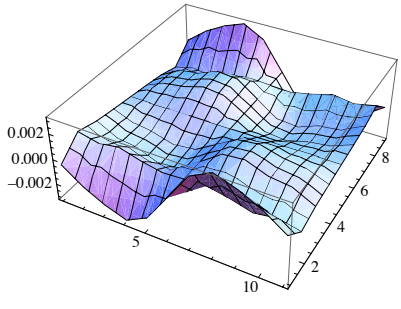
```
In[18]:= % - Cos[rgrid]
Out[18]= {-0.0101091, 0.000031019, -0.0000173088, -0.0000130366, 9.03135×10^-6, 0.0000521639,
 0.0000926836, 0.000336785, 0.00756426, 0.0166339, 0.000651758, -0.000869237}
```

In multiple dimensions, `NDSolve`FiniteDifferenceDerivative` works on tensor product grids, and you only need to specify the grid points for each dimension.

This defines grids *xgrid* and *ygrid* for the x and y direction, gives an approximation for the mixed xy partial derivative of the Gaussian on the tensor product of *xgrid* and *ygrid*, and makes a surface plot of the error.

```
In[19]:= xgrid = Range[0, 8];
ygrid = Range[0, 10];
gaussian[x_, y_] = Exp[-((x - 4)^2 + (y - 5)^2) / 10];
values = Outer[gaussian, xgrid, ygrid];
ListPlot3D[NDSolve`FiniteDifferenceDerivative[{1, 1}, {xgrid, ygrid}, values] -
  Outer[Function[{x, y}, Evaluate[D[gaussian[x, y], x, y]]], xgrid, ygrid]]
```

Out[23]=



Note that the values need to be given in a matrix corresponding to the outer product of the grid coordinates.

`NDSolve`FiniteDifferenceDerivative` does not compute weights for sums of derivatives. This means that for common operators like the Laplacian, you need to combine two approximations.

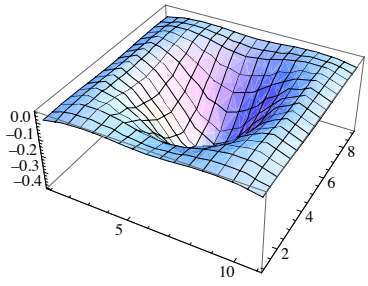
This makes a function that approximates the Laplacian operator on a tensor product grid.

In[24]:=
```
lap[values_, {xgrid_, ygrid_}] :=
 NDSolve`FiniteDifferenceDerivative[{2, 0}, {xgrid, ygrid}, values] +
  NDSolve`FiniteDifferenceDerivative[{0, 2}, {xgrid, ygrid}, values]
```

This uses the function to approximate the Laplacian for the same grid and Gaussian function used in the previous example.

In[25]:=
```
ListPlot3D[lap[values, {xgrid, ygrid}]]
```

Out[25]=



option name	*default value*	
"DifferenceOrder"	4	asymptotic order of the error
PeriodicInterpolation	False	whether to consider the values as those of a periodic function with the period equal to the interval enclosed by the grid

Options for `NDSolve`FiniteDifferenceDerivative`.

This approximates the derivatives for the sine function on the random grid defined earlier, assuming that the function repeats periodically.

In[26]:=
```
NDSolve`FiniteDifferenceDerivative[
 1, rgrid, Sin[rgrid], PeriodicInterpolation → True]
```

Out[26]=
```
{0.99895, 0.586765, 0.536072, 0.463601, -0.149152,
 -0.215212, -0.747842, -0.795502, -0.97065, -0.247503, 0.994585, 0.99895}
```

When using `PeriodicInterpolation -> True`, you can omit the last point in the values since it should always be the same as the first. This feature is useful when solving a PDE with periodic boundary conditions.

This generates second-order finite difference formulas for the first derivative of a symbolic function.

```
In[27]:= NDSolve`FiniteDifferenceDerivative[1,
           {x-1, x0, x1}, {f-1, f0, f1}, "DifferenceOrder" → 2]
```

$$\text{Out[27]}= \left\{\frac{f_1\,(x_{-1}-x_0)}{(-x_{-1}+x_1)\,(-x_0+x_1)}+\frac{f_0\,(-x_{-1}+x_1)}{(-x_{-1}+x_0)\,(-x_0+x_1)}+\frac{f_{-1}\left(-1-\frac{-x_{-1}+x_1}{-x_{-1}+x_0}\right)}{-x_{-1}+x_1},\right.$$
$$\frac{f_1\,(-x_{-1}+x_0)}{(-x_{-1}+x_1)\,(-x_0+x_1)}-\frac{f_{-1}\,(-x_0+x_1)}{(-x_{-1}+x_0)\,(-x_{-1}+x_1)}+\frac{f_0\left(-1+\frac{-x_0+x_1}{-x_{-1}+x_0}\right)}{-x_0+x_1},$$
$$\left.-\frac{f_{-1}\,(x_0-x_1)}{(-x_{-1}+x_0)\,(-x_{-1}+x_1)}-\frac{f_0\,(-x_{-1}+x_1)}{(-x_{-1}+x_0)\,(-x_0+x_1)}+\frac{f_1\,(-x_{-1}+x_0)\left(\frac{-x_{-1}+x_1}{-x_{-1}+x_0}+\frac{-x_0+x_1}{-x_{-1}+x_0}\right)}{(-x_{-1}+x_1)\,(-x_0+x_1)}\right\}$$

Fourth-order differences typically provide a good balance between truncation (approximation) error and roundoff error for machine precision. However, there are some applications where fourth-order differences produce excessive oscillation (Gibb's phenomena), so second-order differences are better. Also, for high-precision, higher-order differences may be appropriate. Even values of "DifferenceOrder" use centered formulas, which typically have smaller error coefficients than noncentered formulas, so even values are recommended when appropriate.

NDSolve`FiniteDifferenceDerivativeFunction

When computing the solution to a PDE, it is common to repeatedly apply the same finite difference approximation to different values on the same grid. A significant savings can be made by storing the necessary weight computations and applying them to the changing data. When you omit the (third) argument with function values in NDSolve`FiniteDifferenceDerivative, the result will be an NDSolve`FiniteDifferenceDerivativeFunction, which is a data object that stores the weight computations in a efficient form for future repeated use.

NDSolve`FiniteDifferenceDerivative[{m_1,m_2,...},{$grid_1$,$grid_2$,...}]	compute the finite difference weights needed to approximate the partial derivative of order (m_1, m_2, ...) for the function of n variables on the tensor product grid defined by the outer product of ($grid_1$, $grid_2$, ...); the result is returned as an NDSolve`FiniteDifferenceDerivativeFunction object
NDSolve`FiniteDifferenceDerivativeFunction[Derivative[m],$data$]	a data object that contains the weights and other data needed to quickly approximate the m^{th}-order derivative of a function; in the standard output form, only the Derivative[m] operator it approximates is shown
NDSolve`FiniteDifferenceDerivativeFunction[$data$][$values$]	approximate the derivative of the function that takes on values on the grid used to determine data

Computing finite difference weights for repeated use.

This defines a uniform grid with 25 points on the unit interval and evaluates the sine function with one period on the grid.

```
In[2]:= n = 24;
        grid = N[Range[0, n] / n];
        values = Sin[2 π grid]
```

```
Out[4]= {0., 0.258819, 0.5, 0.707107, 0.866025, 0.965926, 1., 0.965926, 0.866025,
  0.707107, 0.5, 0.258819, 1.22465×10^-16, -0.258819, -0.5, -0.707107, -0.866025,
  -0.965926, -1., -0.965926, -0.866025, -0.707107, -0.5, -0.258819, -2.44929×10^-16}
```

This defines an NDSolve`FiniteDifferenceDerivativeFunction, which can be repeatedly applied to different values on the grid to approximate the second derivative.

```
In[5]:= fddf = NDSolve`FiniteDifferenceDerivative[Derivative[2], grid]
```

```
Out[5]= NDSolve`FiniteDifferenceDerivativeFunction[Derivative[2], <>]
```

Note that the standard output form is abbreviated and only shows the derivative operators that are approximated.

This computes the approximation to the second derivative of the sine function.

```
In[6]:= fddf[values]
```

```
Out[6]= {0.0720267, -10.2248, -19.7382, -27.914, -34.1875, -38.1312, -39.4764, -38.1312,
  -34.1875, -27.914, -19.7382, -10.2172, 3.39687×10^-13, 10.2172, 19.7382, 27.914,
  34.1875, 38.1312, 39.4764, 38.1312, 34.1875, 27.914, 19.7382, 10.2248, -0.0720267}
```

This function is only applicable for values defined on the particular grid used to construct it. If your problem requires changing the grid, you will need to use `NDSolve`FiniteDifferenceDerivative` to generate weights each time the grid changes. However, when you can use `NDSolve`FiniteDifferenceDerivativeFunction` objects, evaluation will be substantially faster.

This compares timings for computing the Laplacian with the function just defined and with the definition of the previous section. A loop is used to repeat the calculation in each case because it is too fast for the differences to show up with `Timing`.

```
In[9]:= repeats = 10 000;
        {First[Timing[Do[fddf[values], {repeats}]]],
         First[Timing[Do[NDSolve`FiniteDifferenceDerivative[
              Derivative[2], grid, values], {repeats}]]]}
Out[10]= {0.047, 2.25}
```

An `NDSolve`FiniteDifferenceDerivativeFunction` can be used repeatedly in many situations. As a simple example, consider a collocation method for solving the boundary value problem

$$u_{xx} + \sin(x)\, u = \lambda\, u;\ u(0) = u(1) = 0$$

on the unit interval. (This simple method is not necessarily the best way to solve this particular problem, but it is useful as an example.)

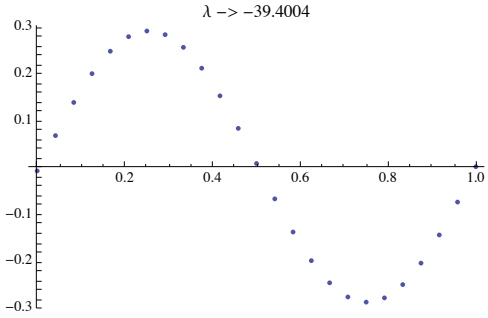
This defines a function that will have all components zero at an approximate solution of the boundary value problem. Using the intermediate vector v and setting its endpoints (parts {1,-1}) to 0 is a fast and simple trick to enforce the boundary conditions. Evaluation is prevented except for numbers λ because this would not work otherwise. (Also, because `Times` is `Listable`, `Sin[2 Pi grid] u` would thread componentwise.)

```
In[11]:= Clear[fun];
         fun[u_, λ_?NumberQ] :=
          Module[{n = Length[u], v =  fddf[u] + (Sin[grid] - λ) u},
           v[[{1, -1}]] = 0.;
           {v, u.u - 1}]
```

This uses `FindRoot` to find an approximate eigenfunction using the constant coefficient case for a starting value and shows a plot of the eigenfunction.

```
In[13]:= s4 = FindRoot[fun[u, λ], {u, values}, {λ, -4 π^2}];
         ListPlot[Transpose[{grid, u /. s4}], PlotLabel → ToString[Last[s4]]]
```

Out[14]=

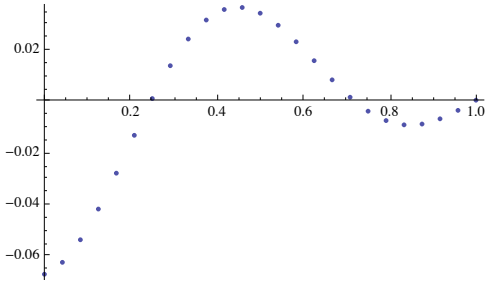


Since the setup for this problem is so simple, it is easy to compare various alternatives. For example, to compare the solution above, which used the default fourth-order differences, to the usual use of second-order differences, all that needs to be changed is the "`DifferenceOrder`".

This solves the boundary value problem using second-order differences and shows a plot of the difference between it and the fourth-order solution.

```
In[39]:= fddf = NDSolve`FiniteDifferenceDerivative[
            Derivative[2], grid, "DifferenceOrder" → 2];
         s2 = FindRoot[fun[u, λ], {u, values}, {λ, -4 π^2}];
         ListPlot[Transpose[{grid, (u /. s4) - (u /. s2)}]]
```

Out[41]=



One way to determine which is the better solution is to study the convergence as the grid is refined. This will be considered to some extent in the section on differentiation matrices below.

While the most vital information about an `NDSolve`FiniteDifferenceDerivativeFunction` object, the derivative order, is displayed in its output form, sometimes it is useful to extract this and other information from an `NDSolve`FiniteDifferenceDerivativeFunction`, say for use in a program. The structure of the way the data is stored may change between versions of *Mathematica*, so extracting the information by using parts of the expression is not recommended. A better alternative is to use any of the several method functions provided for this purpose.

Let *FDDF* represent an NDSolve`FiniteDifferenceDerivativeFunction[*data*] object.

FDDF@"DerivativeOrder"	get the derivative order that *FDDF* approximates
FDDF@"DifferenceOrder"	get the list with the difference order used for the approximation in each dimension
FDDF@"PeriodicInterpolation"	get the list with elements True or False indicating whether periodic interpolation is used for each dimension
FDDF@"Coordinates"	get the list with the grid coordinates in each dimension
FDDF@"Grid"	form the tensor of the grid points; this is the outer product of the grid coordinates
FDDF@"DifferentiationMatrix"	compute the sparse differentiation matrix *mat* such that *mat*.Flatten[*values*] is equivalent to Flatten[*FDDF*[*values*]]

Method functions for exacting information from an NDSolve`FiniteDifferenceDerivativeFunction[*data*] object.

Any of the method functions that return a list with an element for each of the dimensions can be used with an integer argument *dim*, which will return only the value for that particular dimension such that *FDDF*@*method*[*dim*] = (*FDDF*@*method*)[[*dim*]].

The following examples show how you might use some of these methods.

Here is an NDSolve`FiniteDifferenceDerivativeFunction object created with random grids having between 10 and 16 points in each dimension.

```
In[15]:= fddf = NDSolve`FiniteDifferenceDerivative[Derivative[0, 1, 2],
           Table[Sort[Join[{0., 1.}, Table[RandomReal[], {RandomInteger[{8, 14}]}]]],
            {3}], PeriodicInterpolation → True]
Out[15]= NDSolve`FiniteDifferenceDerivativeFunction[Derivative[0, 1, 2], <>]
```

This shows the dimensions of the outer product grid.

```
In[20]:= Dimensions[tpg = fddf@"Grid"]
Out[20]= {15, 10, 11, 3}
```

Note that the rank of the grid point tensor is one more than the dimensionality of the tensor product. This is because the three coordinates defining each point are in a list themselves. If you have a function that depends on the grid variables, you can use Apply[*f*, *fddf*["Grid"], {*n*}] where *n* = Length[*fddf*["DerivativeOrder"]] is the dimensionality of the space in which you are approximating the derivative.

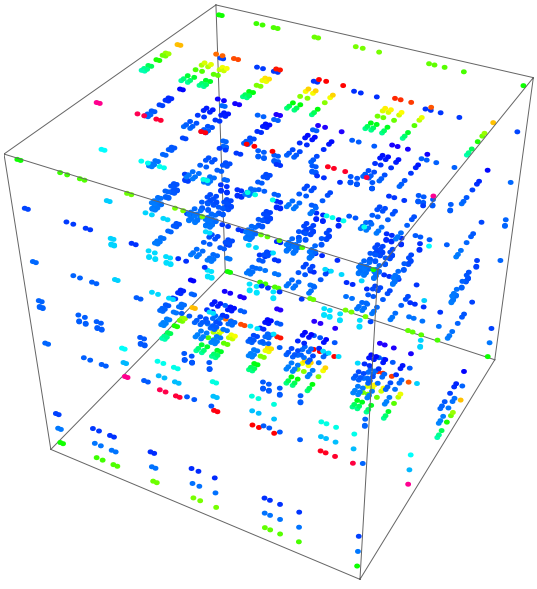
This defines a Gaussian function of 3 variables and applies it to the grid on which the NDSolve`FiniteDifferenceDerivativeFunction is defined.

In[21]:=
```
f = Function[{x, y, z}, Exp[-((x - .5)^2 + (y - .5)^2 + (z - .5)^2)]];
values = Apply[f, fddf@"Grid", {Length[fddf["DerivativeOrder"]]}];
```

This shows a 3-dimensional plot of the grid points colored according to the scaled value of the derivative.

In[23]:=
```
Module[{dvals = fddf[values], maxval, minval},
 maxval = Max[dvals];
 minval = Min[dvals];
 Graphics3D[MapThread[{Hue[(#2 - minval) / (maxval - minval)], Point[#1]} &,
   {fddf["Grid"], fddf[values]}, Length[fddf["DerivativeOrder"]]]]]
```

Out[23]=



For a moderate-sized tensor product grid like the example here, using Apply is reasonably fast. However, as the grid size gets larger, this approach may not be the fastest because Apply can only be used in limited ways with the *Mathematica* compiler and hence, with packed arrays. If you can define your function so you can use Map instead of Apply, you may be able to use a CompiledFunction since Map has greater applicability within the *Mathematica* compiler than does Apply.

This defines a CompiledFunction that uses Map to get the values on the grid. (If the first grid dimension is greater than the system option "MapCompileLength", then you do not need to construct the CompiledFunction since the compilation is done automatically when grid is a packed array.)

```
In[24]:= cf = Compile[{{grid, _Real, 4}},
           Map[Function[{X}, Module[{Xs = X - .5}, Exp[-(Xs.Xs)]]], grid, {3}]]
Out[24]= CompiledFunction[{grid},
          Map[Function[{X}, Module[{Xs = X - 0.5}, e^-Xs.Xs]], grid, {3}], -CompiledCode-]
```

An even better approach, when possible, is to take advantage of listability when your function consists of operations and functions which have the Listable attribute. The trick is to separate the x, y, and z values at each of the points on the tensor product grid. The fastest way to do this is using Transpose[*fddf*["Grid"]], RotateLeft[Range[$n+1$]]], where n = Length[*fddf*["DerivativeOrder"]] is the dimensionality of the space in which you are approximating the derivative. This will return a list of length n, which has the values on the grid for each of the component dimensions separately. With the Listable attribute, functions applied to this will thread over the grid.

This defines a function that takes advantage of the fact that Exp has the Listable attribute to find the values on the grid.

```
In[25]:= fgrid[grid_] :=
          Apply[f, Transpose[grid, RotateLeft[Range[TensorRank[grid]], 1]]]
```

This compares timings for the three methods. The commands are repeated several times to get more accurate timings.

```
In[26]:= Module[
          {repeats = 100, grid = fddf["Grid"], n = Length[fddf["DerivativeOrder"]]},
          {First[Timing[Do[Apply[f, grid, {n}], {repeats}]]],
           First[Timing[Do[cf[grid], {repeats}]]],
           First[Timing[Do[fgrid[grid], {repeats}]]]}]
Out[26]= {1.766, 0.125, 0.047}
```

The example timings show that using the CompiledFunction is typically much faster than using Apply and taking advantage of listability is a little faster yet.

Pseudospectral Derivatives

The maximum value the difference order can take on is determined by the number of points in the grid. If you exceed this, a warning message will be given and the order reduced automatically.

This uses maximal order to approximate the first derivative of the sine function on a random grid.

```
In[50]:= NDSolve`FiniteDifferenceDerivative[1,
          rgrid, Sin[rgrid], "DifferenceOrder" → Length[rgrid]]
```

NDSolve`FiniteDifferenceDerivative::ordred: There are insufficient points in dimension 1 to achieve the requested approximation order. Order will be reduced to 11.

```
Out[50]= {1.00001, 0.586821, 0.536089, 0.463614, -0.149161, -0.215265,
          -0.747934, -0.795838, -0.978214, -0.264155, 0.997089, 0.999941}
```

Using a limiting order is commonly referred to as a *pseudospectral* derivative. A common problem with these is that artificial oscillations (Runge's phenomena) can be extreme. However, there are two instances where this is not the case: a uniform grid with periodic repetition and a grid with points at the zeros of the Chebyshev polynomials, T_n, or Chebyshev-Gauss-Lobatto points [F96a], [QV94]. The computation in both of these cases can be done using a fast Fourier transform, which is efficient and minimizes roundoff error.

`"DifferenceOrder"->`n	use n^{th}-order finite differences to approximate the derivative
`"DifferenceOrder"->Length[`*grid*`]`	use the highest possible order finite differences to approximate the derivative on the grid (not generally recommended)
`"DifferenceOrder"->` `"Pseudospectral"`	use a pseudospectral derivative approximation; only applicable when the grid points are spaced corresponding to the Chebyshev-Gauss-Lobatto points or when the grid is uniform with `PeriodicInterpolation -> True`
`"DifferenceOrder"->{`n_1`,`n_2`,...}`	use difference orders n_1, n_2, ... in dimensions 1, 2, ... respectively

Settings for the `"DifferenceOrder"` option.

This gives a pseudospectral approximation for the second derivative of the sine function on a uniform grid.

```
In[27]:= ugrid = N[2 π Range[0, 10] / 10];
         NDSolve`FiniteDifferenceDerivative[1, ugrid, Sin[ugrid],
          PeriodicInterpolation → True, "DifferenceOrder" -> "Pseudospectral"]
Out[28]= {1., 0.809017, 0.309017, -0.309017, -0.809017, -1., -0.809017, -0.309017, 0.309017, 0.809017, 1.}
```

This computes the error at each point. The approximation is accurate to roundoff because the effective basis for the pseudospectral derivative on a uniform grid for a periodic function are the trigonometric functions.

```
In[29]:= % - Cos[ugrid]
```

```
Out[29]= {6.66134×10^-16, -7.77156×10^-16, 4.996×10^-16, 1.11022×10^-16, -3.33067×10^-16, 4.44089×10^-16,
 -3.33067×10^-16, 3.33067×10^-16, -3.88578×10^-16, -1.11022×10^-16, 6.66134×10^-16}
```

The Chebyshev-Gauss-Lobatto points are the zeros of $(1 - x^2) T_n'(x)$. Using the property $T_n(x) = T_n(\cos(\theta)) == \cos(n\theta)$, these can be shown to be at $x_j = \cos\left(\frac{\pi j}{n}\right)$.

This defines a simple function that generates a grid of n points with leftmost point at $x0$ and interval length L having the spacing of the Chebyshev-Gauss-Lobatto points.

```
In[30]:= CGLGrid[x0_, L_, n_Integer /; n > 1] :=
          1
 x0 + ---- L (1 - Cos[π Range[0, n - 1] / (n - 1)])
          2
```

This computes the pseudospectral derivative for a Gaussian function.

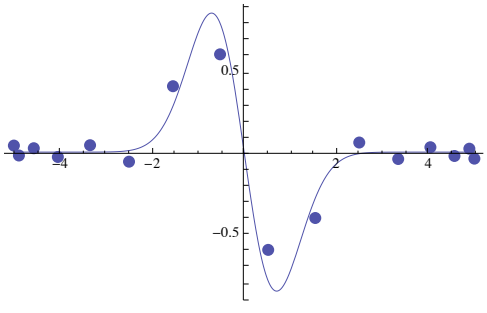
```
In[31]:= cgrid = CGLGrid[-5, 10., 16]; NDSolve`FiniteDifferenceDerivative[
  1, cgrid, Exp[-cgrid^2], "DifferenceOrder" -> "Pseudospectral"]
```

```
Out[31]= {0.0402426, -0.0209922, 0.0239151, -0.0300589, 0.0425553, -0.0590871, 0.40663, 0.60336,
 -0.60336, -0.40663, 0.0590871, -0.0425553, 0.0300589, -0.0239151, 0.0209922, -0.0402426}
```

This shows a plot of the approximation and the exact values.

```
In[32]:= Show[{
   ListPlot[Transpose[{cgrid, %}], PlotStyle → PointSize[0.025]],
   Plot[Evaluate[D[Exp[-x^2], x]], {x, -5, 5}]}, PlotRange → All]
```

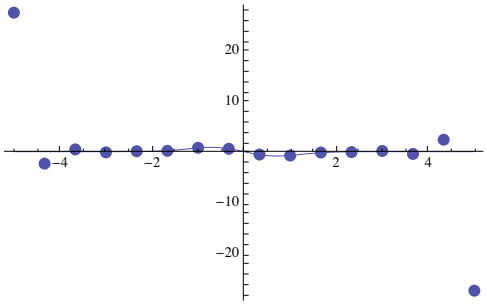
```
Out[32]=
```



This shows a plot of the derivative computed using a uniform grid with the same number of points with maximal difference order.

```
In[35]:= ugrid = -5 + 10. Range[0, 15] / 15;
Show[{
  ListPlot[
   Transpose[{ugrid, NDSolve`FiniteDifferenceDerivative[1, ugrid, Exp[-ugrid^2],
      "DifferenceOrder" → Length[ugrid] - 1]}], PlotStyle → PointSize[0.025]],
  Plot[Evaluate[D[Exp[-x^2], x]], {x, -5, 5}]}, PlotRange → All]
```

Out[36]=

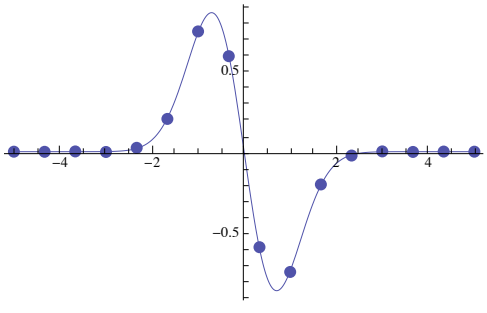


Even though the approximation is somewhat better in the center (because the points are more closely spaced there in the uniform grid), the plot clearly shows the disastrous oscillation typical of overly high-order finite difference approximations. Using the Chebyshev-Gauss-Lobatto spacing has minimized this.

This shows a plot of the pseudospectral derivative approximation computed using a uniform grid with periodic repetition.

```
In[70]:= ugrid = -5 + 10. Range[0, 15] / 15;
Show[ {
  ListPlot[Transpose[{ugrid, NDSolve`FiniteDifferenceDerivative[
      1, ugrid, Exp[-ugrid^2], "DifferenceOrder" → "Pseudospectral",
      PeriodicInterpolation → True]}], PlotStyle → PointSize[0.025]],
  Plot[Evaluate[D[Exp[-x^2], x]], {x, -5, 5}]}, PlotRange → All]
```

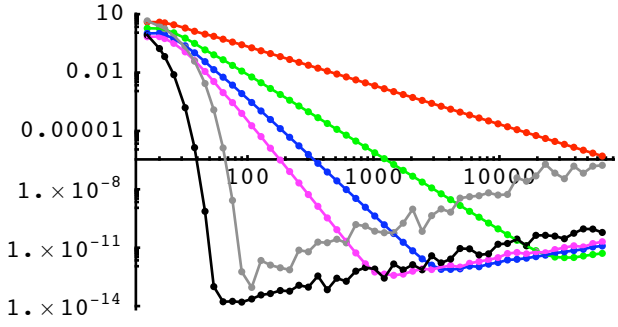
Out[71]=



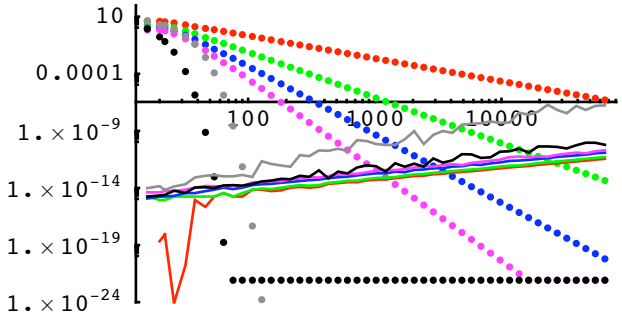
With the assumption of periodicity, the approximation is significantly improved. The accuracy of the periodic pseudospectral approximations is sufficiently high to justify, in some cases, using a larger computational domain to simulate periodicity, say for a pulse like the example. Despite the great accuracy of these approximations, they are not without pitfalls: one of the worst is probably aliasing error, whereby an oscillatory function component with too great a frequency can be misapproximated or disappear entirely.

Accuracy and Convergence of Finite Difference Approximations

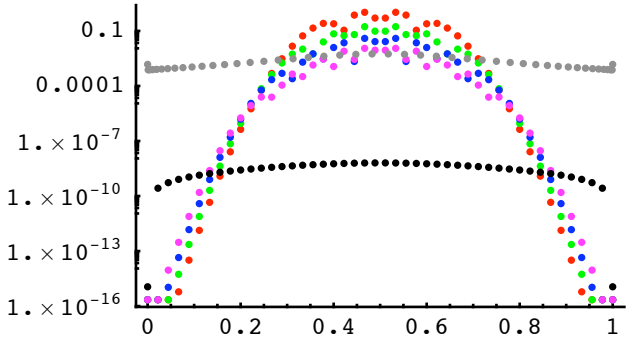
When using finite differences, it is important to keep in mind that the truncation error, or the asymptotic approximation error induced by cutting off the Taylor series approximation, is not the only source of error. There are two other sources of error in applying finite difference formulas; condition error and roundoff error [GMW81]. Roundoff error comes from roundoff in the arithmetic computations required. Condition error comes from magnification of any errors in the function values, typically from the division by a power of the step size, and so grows with decreasing step size. This means that in practice, even though the truncation error approaches zero as h does, the actual error will start growing beyond some point. The following figures demonstrate the typical behavior as h becomes small for a smooth function.



A logarithmic plot of the maximum error for approximating the first derivative of the Gaussian $f(x) = e^{-(15(x-1/2))^2}$ at points on a grid covering the interval $[0, 1]$ as a function of the number of grid points, n, using machine precision. Finite differences of order 2, 4, 6, and 8 on a uniform grid are shown in red, green, blue, and magenta, respectively. Pseudospectral derivatives with uniform (periodic) and Chebyshev spacing are shown in black and gray, respectively.



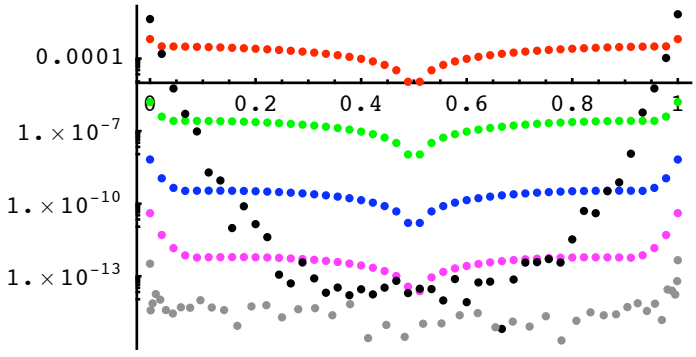
A logarithmic plot of the truncation error (dotted) and the condition and roundoff error (solid line) for approximating the first derivative of the Gaussian $f(x) = e^{-(15\,(x-1/2))^2}$ at points on a grid covering the interval $[0, 1]$ as a function of the number of grid points, n. Finite differences of order 2, 4, 6, and 8 on a uniform grid are shown in red, green, blue, and magenta, respectively. Pseudospectral derivatives with uniform (periodic) and Chebyshev spacing are shown in black and gray, respectively. The truncation error was computed by computing the approximations with very high precision. The roundoff and condition error was estimated by subtracting the machine-precision approximation from the high-precision approximation. The roundoff and condition error tends to increase linearly (because of the $1/h$ factor common to finite difference formulas for the first derivative) and tends to be a little bit higher for higher-order derivatives. The pseudospectral derivatives show more variations because the error of the FFT computations vary with length. Note that the truncation error for the uniform (periodic) pseudospectral derivative does not decrease below about 10^{-22}. This is because, mathematically, the Gaussian is not a periodic function; this error in essence gives the deviation from periodicity.



A semilogarithmic plot of the error for approximating the first derivative of the Gaussian $f(x) = e^{-(x-1/2)^2}$ as a function of x at points on a 45-point grid covering the interval $[0, 1]$. Finite differences of order 2, 4, 6, and 8 on a uniform grid are shown in red, green, blue, and magenta, respectively. Pseudospectral derivatives with uniform (periodic) and Chebyshev spacing are shown in black and gray, respectively. All but the pseudospectral derivative with Chebyshev spacing were computed using uniform spacing $1/45$. It is apparent that the error for the pseudospectral derivatives is not so localized; not surprising since the approximation at any point is based on the values over the whole grid. The error for the finite difference approximations are localized and the magnitude of the errors follows the size of the Gaussian (which is parabolic on a semilogarithmic plot).

From the second plot, it is apparent that there is a size for which the best possible derivative approximation is found; for larger h, the truncation error dominates, and for smaller h, the condition and roundoff error dominate. The optimal h tends to give better approximations for higher-order differences. This is not typically an issue for spatial discretization of PDEs because computing to that level of accuracy would be prohibitively expensive. However, this error balance is a vitally important issue when using low-order differences to approximate, for example, Jacobian matrices. To avoid extra function evaluations, first-order forward differences are usually used, and the error balance is proportional to the square root of unit roundoff, so picking a good value of h is important [GMW81].

The plots showed the situation typical for smooth functions where there were no real boundary effects. If the parameter in the Gaussian is changed so the function is flatter, boundary effects begin to appear.

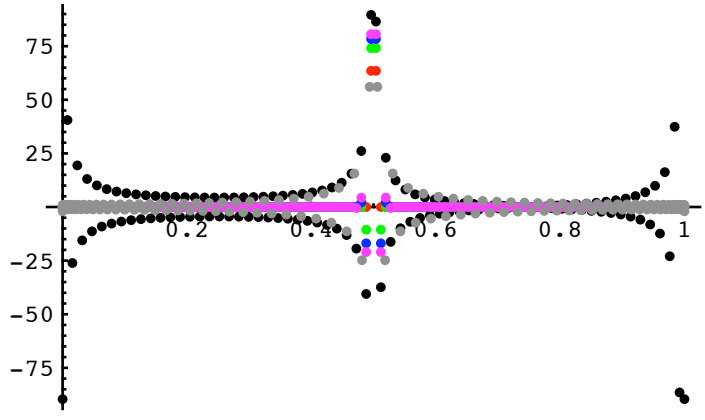


A semilogarithmic plot of the error for approximating the first derivative of the Gaussian $f(x) = e^{-(15\,(x-1/2))^2}$ as a function of x at points on a 45-point grid covering the interval $[0, 1]$. Finite differences of order 2, 4, 6, and 8 on a uniform grid are shown in red, green, blue, and magenta, respectively. Pseudospectral derivatives with uniform (nonperiodic) and Chebyshev spacing are shown in black and gray, respectively. All but the pseudospectral derivative with Chebyshev spacing were computed using uniform spacing $1/45$. The error for the finite difference approximations are localized, and the magnitude of the errors follows the magnitude of the first derivative of the Gaussian. The error near the boundary for the uniform spacing pseudospectral (order-45 polynomial) approximation becomes enormous; as h decreases, this is not bounded. On the other hand, the error for the Chebyshev spacing pseudospectral is more uniform and overall quite small.

From what has so far been shown, it would appear that the higher the order of the approximation, the better. However, there are two additional issues to consider. The higher-order approxi-

mations lead to more expensive function evaluations, and if implicit iteration is needed (as for a stiff problem), then not only is computing the Jacobian more expensive, but the eigenvalues of the matrix also tend to be larger, leading to more stiffness and more difficultly for iterative solvers. This is at an extreme for pseudospectral methods, where the Jacobian has essentially no nonzero entries [F96a]. Of course, these problems are a trade-off for smaller system (and hence matrix) size.

The other issue is associated with discontinuities. Typically, the higher order the polynomial approximation, the worse the approximation. To make matters even worse, for a true discontinuity, the errors magnify as the grid spacing is reduced.



A plot of approximations for the first derivative of the discontinuous unit step function $f(x) = \texttt{UnitStep}\,(x - 1/2)$ as a function of x at points on a 128-point grid covering the interval $[0, 1]$. Finite differences of order 2, 4, 6, and 8 on a uniform grid are shown in red, green, blue, and magenta, respectively. Pseudospectral derivatives with uniform (periodic) and Chebyshev spacing are shown in black and gray, respectively. All but the pseudospectral derivative with Chebyshev spacing were computed using uniform spacing $1/128$. All show oscillatory behavior, but it is apparent that the Chebyshev pseudospectral derivative does better in this regard.

There are numerous alternatives that are used around known discontinuities, such as front tracking. First-order forward differences minimize oscillation, but introduce artificial viscosity terms. One good alternative are the so-called essentially nonoscillatory (ENO) schemes, which have full order away from discontinuities but introduce limits near discontinuities that limit the approximation order and the oscillatory behavior. At this time, ENO schemes are not implemented in `NDSolve`.

In summary, choosing an appropriate difference order depends greatly on the problem structure. The default of 4 was chosen to be generally reasonable for a wide variety of PDEs, but you may want to try other settings for a particular problem to get better results.

Differentiation Matrices

Since differentiation, and naturally finite difference approximation, is a linear operation, an alternative way of expressing the action of a `FiniteDifferenceDerivativeFunction` is with a matrix. A matrix that represents an approximation to the differential operator is referred to as a *differentiation matrix* [F96a]. While differentiation matrices may not always be the optimal way of applying finite difference approximations (particularly in cases where an FFT can be used to reduce complexity and error), they are invaluable as aids for analysis and, sometimes, for use in the linear solvers often needed to solve PDEs.

Let *FDDF* represent an `NDSolve`FiniteDifferenceDerivativeFunction[`*data*`]` object.

FDDF`@"DifferentiationMatrix"`	recast the linear operation of *FDDF* as a matrix that represents the linear operator

Forming a differentiation matrix.

This creates a `FiniteDifferenceDerivativeFunction` object.

```
In[37]:= fdd = NDSolve`FiniteDifferenceDerivative[2, Range[0, 10]]
Out[37]= NDSolve`FiniteDifferenceDerivativeFunction[Derivative[2], <>]
```

This makes a matrix representing the underlying linear operator.

```
In[38]:= smat = fdd["DifferentiationMatrix"]
Out[38]= SparseArray[<59>, {11, 11}]
```

The matrix is given in a sparse form because, in general, differentiation matrices have relatively few nonzero entries.

This converts to a normal dense matrix and displays it using `MatrixForm`.

```
In[39]:= MatrixForm[mat = Normal[smat]]
```

Out[39]//MatrixForm=

$$\begin{pmatrix}
\frac{15}{4} & -\frac{77}{6} & \frac{107}{6} & -13 & \frac{61}{12} & -\frac{5}{6} & 0 & 0 & 0 & 0 & 0 \\
\frac{5}{6} & -\frac{5}{4} & -\frac{1}{3} & \frac{7}{6} & -\frac{1}{2} & \frac{1}{12} & 0 & 0 & 0 & 0 & 0 \\
-\frac{1}{12} & \frac{4}{3} & -\frac{5}{2} & \frac{4}{3} & -\frac{1}{12} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{1}{12} & \frac{4}{3} & -\frac{5}{2} & \frac{4}{3} & -\frac{1}{12} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -\frac{1}{12} & \frac{4}{3} & -\frac{5}{2} & \frac{4}{3} & -\frac{1}{12} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -\frac{1}{12} & \frac{4}{3} & -\frac{5}{2} & \frac{4}{3} & -\frac{1}{12} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -\frac{1}{12} & \frac{4}{3} & -\frac{5}{2} & \frac{4}{3} & -\frac{1}{12} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -\frac{1}{12} & \frac{4}{3} & -\frac{5}{2} & \frac{4}{3} & -\frac{1}{12} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{12} & \frac{4}{3} & -\frac{5}{2} & \frac{4}{3} & -\frac{1}{12} \\
0 & 0 & 0 & 0 & 0 & \frac{1}{12} & -\frac{1}{2} & \frac{7}{6} & -\frac{1}{3} & -\frac{5}{4} & \frac{5}{6} \\
0 & 0 & 0 & 0 & 0 & -\frac{5}{6} & \frac{61}{12} & -13 & \frac{107}{6} & -\frac{77}{6} & \frac{15}{4}
\end{pmatrix}$$

This shows that all three of the representations are roughly equivalent in terms of their action on data.

```
In[40]:= data = Map[Exp[-#^2] &, N[Range[0, 10]]];
         {fdd[data], smat.data, mat.data}
```

```
Out[41]= {{-0.646094, 0.367523, 0.361548, -0.00654414, -0.00136204, -0.0000101341,
    -9.35941×10^-9, -1.15702×10^-12, -1.93287×10^-17, 1.15721×10^-12, -1.15721×10^-11},
   {-0.646094, 0.367523, 0.361548, -0.00654414, -0.00136204, -0.0000101341,
    -9.35941×10^-9, -1.15702×10^-12, -1.93287×10^-17, 1.15721×10^-12, -1.15721×10^-11},
   {-0.646094, 0.367523, 0.361548, -0.00654414, -0.00136204, -0.0000101341,
    -9.35941×10^-9, -1.15702×10^-12, -1.93287×10^-17, 1.15721×10^-12, -1.15721×10^-11}}
```

As mentioned previously, the matrix form is useful for analysis. For example, it can be used in a direct solver or to find the eigenvalues that could, for example, be used for linear stability analysis.

This computes the eigenvalues of the differentiation matrix.

```
In[42]:= Eigenvalues[N[smat]]
```

```
Out[42]= {-4.90697, -3.79232, -2.38895, -1.12435, -0.287414,
   8.12317×10^-6 + 0.0000140698 i, 8.12317×10^-6 - 0.0000140698 i, -0.0000162463,
   -8.45104×10^-6, 4.22552×10^-6 + 7.31779×10^-6 i, 4.22552×10^-6 - 7.31779×10^-6 i}
```

For pseudospectral derivatives, which can be computed using fast Fourier transforms, it may be faster to use the differentiation matrix for small size, but ultimately, on a larger grid, the better complexity and numerical properties of the FFT make this the much better choice.

For multidimensional derivatives, the matrix is formed so that it is operating on the flattened data, the `KroneckerProduct` of the matrices for the one-dimensional derivatives. It is easiest to understand this through an example.

This evaluates a Gaussian function on the grid that is the outer product of grids in the x and y direction.

```
In[4]:= xgrid = N[Range[-2, 2, 1 / 10]];
        ygrid = N[Range[-2, 2, 1 / 8]];
        data = Outer[Exp[-((#1)^2 + (#2)^2)] &, xgrid, ygrid];
```

This defines an `NDSolve`FiniteDifferenceDerivativeFunction` which computes the mixed x-y partial of the function using fourth-order differences.

```
In[7]:= fdd = NDSolve`FiniteDifferenceDerivative[{1, 1}, {xgrid, ygrid}]
Out[7]= NDSolve`FiniteDifferenceDerivativeFunction[Derivative[1, 1], <>]
```

This computes the associated differentiation matrix.

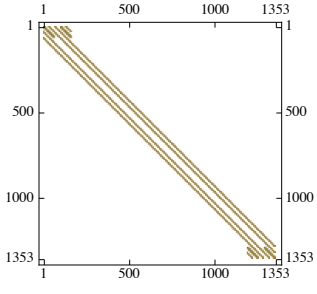
```
In[8]:= dm = fdd["DifferentiationMatrix"]
Out[8]= SparseArray[<22 848>, {1353, 1353}]
```

Note that the differentiation matrix is a 1353×1353 matrix. The number 1353 is the total number of points on the tensor product grid, that, of course, is the product of the number of points on the x and y grids. The differentiation matrix operates on a vector of data which comes from flattening data on the tensor product grid. The matrix is also very sparse; only about one-half of a percent of the entries are nonzero. This is easily seen with a plot of the positions with nonzero values.

Show a plot of the positions with nonzero values for the differentiation matrix.

```
In[9]:= MatrixPlot[Unitize[dm]]
```

Out[9]=



This compares the computation of the mixed x-y partial with the two methods.

```
In[53]:= Max[dm.Flatten[data] - Flatten[fdd[data]]]
Out[53]= 3.60822×10^-15
```

The matrix is the `KroneckerProduct`, or direct matrix product of the 1-dimensional matrices.

Get the 1-dimensional differentiation matrices and form their direct matrix product.

```
In[16]:= fddx = NDSolve`FiniteDifferenceDerivative[{1}, {xgrid}];
         fddy = NDSolve`FiniteDifferenceDerivative[{1}, {ygrid}];
         dmk = KroneckerProduct[fddx@"DifferentiationMatrix",
           fddy@"DifferentiationMatrix"]; dmk == dm
Out[17]= True
```

Using the differentiation matrix results in slightly different values for machine numbers because the order of operations is different which, in turn, leads to different roundoff errors.

The differentation matrix can be advantageous when what is desired is a linear combination of derivatives. For example, the computation of the Laplacian operator can be put into a single matrix.

This makes a function that approximates the Laplacian operator on a the tensor product grid.

```
In[18]:= flap =
          Function[Evaluate[NDSolve`FiniteDifferenceDerivative[{2, 0}, {xgrid, ygrid}][#] +
             NDSolve`FiniteDifferenceDerivative[{0, 2}, {xgrid, ygrid}][#]]]
Out[18]= NDSolve`FiniteDifferenceDerivativeFunction[Derivative[0, 2], <>][#1] +
           NDSolve`FiniteDifferenceDerivativeFunction[Derivative[2, 0], <>][#1] &
```

This computes the differentiation matrices associated with the derivatives in the x and y direction.

```
In[19]:= dmlist = Map[(Head[#]["DifferentiationMatrix"]) &, List @@ First[flap]]
Out[19]= {SparseArray[<6929>, {1353, 1353}], SparseArray[<6897>, {1353, 1353}]}
```

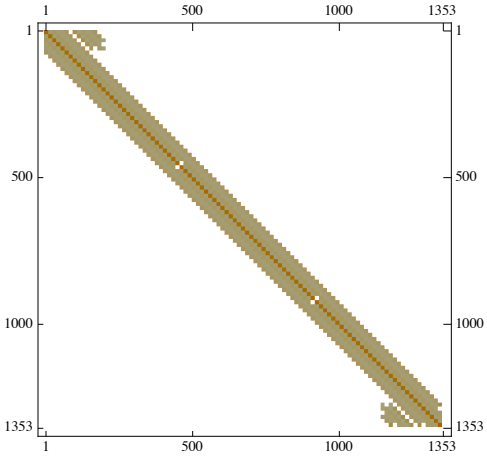
This adds the two sparse matrices together resulting in a single matrix for the Laplacian operator.

```
In[68]:= slap = Total[dmlist]
Out[68]= SparseArray[<12 473>, {1353, 1353}]
```

This shows a plot of the positions with nonzero values for the differentiation matrix.

```
In[69]:= MatrixPlot[Unitize[slap]]
```

Out[69]=



This compares the values and timings for the two different ways of approximating the Laplacian.

```
In[64]:= Block[{repeats = 1000, l1, l2},
          data = Developer`ToPackedArray[data];
          fdata = Flatten[data];
          Map[First, {
            Timing[Do[l1 = flap[data], {repeats}]],
            Timing[Do[l2 = slap.fdata, {repeats}]],
            {Max[Flatten[l1] - l2]}
           }]
         ]
```

Out[64]= $\{0.14, 0.047, 1.39888 \times 10^{-14}\}$

Interpretation of Discretized Dependent Variables

When a dependent variable is given in a monitor (e.g. `StepMonitor`) option or in a method where interpretation of the dependent variable is needed (e.g. `EventLocator` and `Projection`), for ODEs, the interpretation is generally clear: at a particular value of time (or the independent variable), use the value for that component of the solution for the dependent variable.

For PDEs, the interpretation to use is not so obvious. Mathematically speaking, the dependent variable at a particular time is a function of space. This leads to the default interpretation, which is to represent the dependent variable as an approximate function across the spatial domain using an `InterpolatingFunction`.

Another possible interpretation for PDEs is to consider the dependent variable at a particular time as representing the spatially discretized values at that time—that is, discretized both in time and space. You can request that monitors and methods use this fully discretized interpretation by using the `MethodOfLines` option `DiscretizedMonitorVariables -> True`.

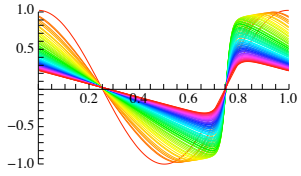
The best way to see the difference between the two interpretations is with an example.

This solves Burgers' equation. The `StepMonitor` is set so that it makes a plot of the solution at the step time of every tenth time step, producing a sequence of curves of gradated color. You can animate the motion by replacing `Show` with `ListAnimate`; note that the motion of the wave in the animation does not reflect actual wave speed since it effectively includes the step size used by `NDSolve`.

In[5]:=
```
curves = Reap[Block[{count = 0}, Timing[
       NDSolve[{D[u[t, x] , t] ==  0.01 D[u[t, x], x, x] + u[t, x] D[u[t, x], x],
         u[0, x] == Cos[2 Pi x], u[t, 0] == u[t, 1]}, u, {t, 0, 1}, {x, 0, 1},
        StepMonitor :> If[Mod[count++, 10] == 0, Sow[Plot[u[t, x], {x, 0, 1},
            PlotRange → {{0, 1}, {-1, 1}}, PlotStyle → Hue[t]]]], Method →
         {"MethodOfLines", "SpatialDiscretization" → {"TensorProductGrid",
            "MinPoints" → 100, "DifferenceOrder" → "Pseudospectral"}}]]]][[2, 1]];
```

In[8]:=
```
Show[curves]
```

Out[6]=



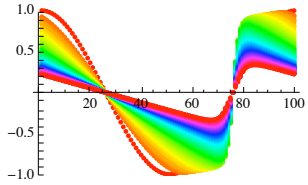
In executing the command above, $u[t, x]$ in the `StepMonitor` is effectively a function of x, so it can be plotted with plot. You could do other operations on it, such as numerical integration.

This solves Burgers' equation. The `StepMonitor` is set so that it makes a list plot of the spatially discretized solution at the step time every tenth step. You can animate the motion by replacing `Show` with `ListAnimate`.

In[10]:=
```
discretecurves =
  Reap[Block[{count = 0}, Timing[NDSolve[{D[u[t, x] , t] ==  0.01 D[u[t, x], x, x] +
        u[t, x] D[u[t, x], x], u[0, x] == Cos[2 Pi x], u[t, 0] == u[t, 1]},
      u, {t, 0, 1}, {x, 0, 1}, StepMonitor :> If[Mod[count++, 10] == 0,
        Sow[ListPlot[u[t, x], PlotRange → {-1, 1}, PlotStyle → Hue[t]]];],
      Method → {"MethodOfLines", "DiscretizedMonitorVariables" → True,
        "SpatialDiscretization" → {"TensorProductGrid", "MinPoints" → 100,
          "DifferenceOrder" → "Pseudospectral"}}]]][[2, 1]];
```

In[11]:=
```
Show[discretecurves]
```

Out[11]=



In this case, $u[t, x]$ is given at each step as a vector with the discretized values of the solution on the spatial grid. Showing the discretization points makes for a more informative monitor in this example since it allows you to see how well the front is resolved as it forms.

The vector of values contains no information about the grid itself; in the example, the plot is made versus the index values, which shows the correct spacing for a uniform grid. Note that when u is interpreted as a function, the grid will be contained in the `InterpolatingFunction` used to represent the spatial solution, so if you need the grid, the easiest way to get it is to extract it from the `InterpolatingFunction`, which represents $u[t, x]$.

Finally note that using the discretized representation is significantly faster. This may be an important issue if you are using the representation in solution method such as `Projection` or `EventLocator`. An example where event detection is used to prevent solutions from going beyond a computational domain is computed much more quickly by using the discretized interpretation.

Boundary Conditions

Often, with PDEs, it is possible to determine a good numerical way to apply boundary conditions for a particular equation and boundary condition. The example given previously in the introduction of "The Numerical Method of Lines" is such a case. However, the problem of finding a general algorithm is much more difficult and is complicated somewhat by the effect that boundary conditions can have on stiffness and overall stability.

Periodic boundary conditions are particularly simple to deal with: periodic interpolation is used for the finite differences. Since pseudospectral approximations are accurate with uniform grids, solutions can often be found quite efficiently.

NDSolve[{eqn_1, eqn_2, …, $u_1[t, x_{min}] == u_1[t, x_{max}]$, $u_2[t, x_{min}] == u_2[t, x_{max}]$, …}, {$u_1[t, x]$, $u_2[t, x]$, …}, {t, t_{min}, t_{max}}, {x, x_{min}, x_{max}}]

solve a system of partial differential equations for function u_1, u_2, … with periodic boundary conditions in the spatial variable x (assuming that t is a temporal variable)

NDSolve[{eqn_1, eqn_2, …, $u_1[t, x_{1\,min}, x_2, \ldots] == u_1[t, x_{1\,max}\ \ x_2, \ldots]$, $u_2[t, x_{1\,min}, x_2, \ldots] == u_2[t, x_{1\,max}\ \ x_2, \ldots]$, …}, {$u_1[t, x_1, x_2, \ldots]$, $u_2[t, x_1, x_2, \ldots]$, …}, {t, t_{min}, t_{max}}, {x, x_{min}, x_{max}}]

solve a system of partial differential equations for function u_1, u_2, … with periodic boundary conditions in the spatial variable x_1 (assuming that t is a temporal variable)

Giving boundary conditions for partial differential equations.

If you are solving for several functions u_1, u_2, … then for any of the functions to have periodic boundary conditions, all of them must (the condition need only be specified for one function). If you are working with more than one spatial dimension, you can have periodic boundary conditions in some independent variable dimensions and not in others.

This solves a generalization of the sine-Gordon equation to two spatial dimensions with periodic boundary conditions using a pseudospectral method. Without the pseudospectral method enabled by the periodicity, the problem could take much longer to solve.

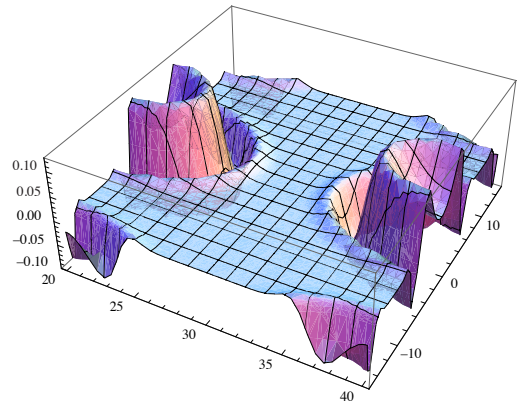
```
In[2]:= sol = NDSolve[{D[u[t, x, y], t, t] ==
      D[u[t, x, y], x, x] + D[u[t, x, y], y, y] - Sin[u[t, x, y]],
     u[0, x, y] == Exp[-(x^2 + y^2)], Derivative[1, 0, 0][u][0, x, y] == 0,
     u[t, -10, y] == u[t, 10, y], u[t, x, -10] == u[t, x, 10]}, u, {t, 0, 6},
    {x, -10, 10}, {y, -10, 10}, Method → {"MethodOfLines", "SpatialDiscretization" →
       {"TensorProductGrid", "DifferenceOrder" -> "Pseudospectral"}}]
Out[2]= {{u → InterpolatingFunction[{{0., 6.}, {-10., 10.}, {-10., 10.}}, <>]}}
```

In the `InterpolatingFunction` object returned as a solution, the ellipses in the notation {…, x_{min}, x_{max}, …} are used to indicate that this dimension repeats periodically

This makes a surface plot of a part of the solution derived from periodic continuation at $t == 6$.

```
In[7]:= Plot3D[First[u[6, x, y] /. sol], {x, 20, 40},
         {y, -15, 15}, PlotRange → All, PlotPoints → 40]
```

Out[7]=



`NDSolve` uses two methods for nonperiodic boundary conditions. Both have their merits and drawbacks. The first method is to differentiate the boundary conditions with respect to the temporal variable and solve for the resulting differential equation(s) at the boundary. The second method is to discretize each boundary condition as it is. This typically results in an algebraic equation for the boundary solution component, so the equations must be solved with a DAE solver. This is controlled with the `DifferentiateBoundaryConditions` option to `MethodOfLines`.

To see how the differentiation method works, consider again the simple example of the method of lines introduction section. In the first formulation, the Dirichlet boundary condition at $x == 0$ was handled by differentiation with respect to t. The Neumann boundary condition was handled using the idea of reflection, which worked fine for a second-order finite difference approximation, but does not generalize quite as easily to higher order (though it can be done easily for this problem by computing the entire reflection). The differentiation method, however, can be used for any order differences on the Neumann boundary condition at $x == 1$. As an example, a solution to the problem will be developed using fourth-order differences.

This is a setting for the number of and spacing between spatial points. It is purposely set small so you can see the resulting equations. You can change it later to improve the accuracy of the approximations.

```
In[8]:= n = 10; h_n = 1 / n;
```

This defines the vector of u_i.

```
In[9]:= U[t_] = Table[u_i[t], {i, 0, n}]
Out[9]= {u0[t], u1[t], u2[t], u3[t], u4[t], u5[t], u6[t], u7[t], u8[t], u9[t], u10[t]}
```

This discretizes the Neumann boundary condition at $x == 1$ in the spatial direction.

```
In[10]:= bc = Last[NDSolve`FiniteDifferenceDerivative[1, h_n Range[0, n], U[t]]] == 0
Out[10]= 5 u6[t]/2 - 40 u7[t]/3 + 30 u8[t] - 40 u9[t] + 125 u10[t]/6 == 0
```

This differentiates the discretized boundary condition with respect to t.

```
In[11]:= bcprime = D[bc, t]
Out[11]= 5/2 u6'[t] - 40/3 u7'[t] + 30 u8'[t] - 40 u9'[t] + 125/6 u10'[t] == 0
```

Technically, it is not necessary that the discretization of the boundary condition be done with the same difference order as the rest of the DE; in fact, since the error terms for the one-sided derivatives are much larger, it may sometimes be desirable to increase the order near the boundaries. `NDSolve` does not do this because it is desirable that the difference order and the `InterpolatingFunction` interpolation order be consistent across the spatial direction.

This is another way of generating the equations using `NDSolve`FiniteDifferenceDerivative`. The first and last will have to be replaced with the appropriate equations from the boundary conditions.

```
In[12]:= eqns = Thread[
           D[U[t], t] == 1/8 NDSolve`FiniteDifferenceDerivative[2, h_n Range[0, n], U[t]]]
```

$$\text{Out[12]}= \left\{u_0'[t] == \frac{1}{8}\left(375\,u_0[t] - \frac{3850\,u_1[t]}{3} + \frac{5350\,u_2[t]}{3} - 1300\,u_3[t] + \frac{1525\,u_4[t]}{3} - \frac{250\,u_5[t]}{3}\right),\right.$$
$$u_1'[t] == \frac{1}{8}\left(\frac{250\,u_0[t]}{3} - 125\,u_1[t] - \frac{100\,u_2[t]}{3} + \frac{350\,u_3[t]}{3} - 50\,u_4[t] + \frac{25\,u_5[t]}{3}\right),$$
$$u_2'[t] == \frac{1}{8}\left(-\frac{25}{3}u_0[t] + \frac{400\,u_1[t]}{3} - 250\,u_2[t] + \frac{400\,u_3[t]}{3} - \frac{25\,u_4[t]}{3}\right),$$
$$u_3'[t] == \frac{1}{8}\left(-\frac{25}{3}u_1[t] + \frac{400\,u_2[t]}{3} - 250\,u_3[t] + \frac{400\,u_4[t]}{3} - \frac{25\,u_5[t]}{3}\right),$$
$$u_4'[t] == \frac{1}{8}\left(-\frac{25}{3}u_2[t] + \frac{400\,u_3[t]}{3} - 250\,u_4[t] + \frac{400\,u_5[t]}{3} - \frac{25\,u_6[t]}{3}\right),$$
$$u_5'[t] == \frac{1}{8}\left(-\frac{25}{3}u_3[t] + \frac{400\,u_4[t]}{3} - 250\,u_5[t] + \frac{400\,u_6[t]}{3} - \frac{25\,u_7[t]}{3}\right),$$
$$u_6'[t] == \frac{1}{8}\left(-\frac{25}{3}u_4[t] + \frac{400\,u_5[t]}{3} - 250\,u_6[t] + \frac{400\,u_7[t]}{3} - \frac{25\,u_8[t]}{3}\right),$$
$$u_7'[t] == \frac{1}{8}\left(-\frac{25}{3}u_5[t] + \frac{400\,u_6[t]}{3} - 250\,u_7[t] + \frac{400\,u_8[t]}{3} - \frac{25\,u_9[t]}{3}\right),$$
$$u_8'[t] == \frac{1}{8}\left(-\frac{25}{3}u_6[t] + \frac{400\,u_7[t]}{3} - 250\,u_8[t] + \frac{400\,u_9[t]}{3} - \frac{25\,u_{10}[t]}{3}\right),$$
$$u_9'[t] == \frac{1}{8}\left(\frac{25\,u_5[t]}{3} - 50\,u_6[t] + \frac{350\,u_7[t]}{3} - \frac{100\,u_8[t]}{3} - 125\,u_9[t] + \frac{250\,u_{10}[t]}{3}\right),$$
$$\left.u_{10}'[t] == \frac{1}{8}\left(-\frac{250}{3}u_5[t] + \frac{1525\,u_6[t]}{3} - 1300\,u_7[t] + \frac{5350\,u_8[t]}{3} - \frac{3850\,u_9[t]}{3} + 375\,u_{10}[t]\right)\right\}$$

Now you can replace the first and last equation with the boundary condition.

```
In[13]:= eqns[[1, 2]] = D[Sin[2 π t], t];
         eqns[[-1]] = bcprime;
         eqns
```

$$\text{Out[15]}= \left\{u_0'[t] == 2\pi\,\text{Cos}[2\pi t],\ u_1'[t] == \frac{1}{8}\left(\frac{250\,u_0[t]}{3} - 125\,u_1[t] - \frac{100\,u_2[t]}{3} + \frac{350\,u_3[t]}{3} - 50\,u_4[t] + \frac{25\,u_5[t]}{3}\right),\right.$$
$$u_2'[t] == \frac{1}{8}\left(-\frac{25}{3}u_0[t] + \frac{400\,u_1[t]}{3} - 250\,u_2[t] + \frac{400\,u_3[t]}{3} - \frac{25\,u_4[t]}{3}\right),$$
$$u_3'[t] == \frac{1}{8}\left(-\frac{25}{3}u_1[t] + \frac{400\,u_2[t]}{3} - 250\,u_3[t] + \frac{400\,u_4[t]}{3} - \frac{25\,u_5[t]}{3}\right),$$
$$u_4'[t] == \frac{1}{8}\left(-\frac{25}{3}u_2[t] + \frac{400\,u_3[t]}{3} - 250\,u_4[t] + \frac{400\,u_5[t]}{3} - \frac{25\,u_6[t]}{3}\right),$$
$$u_5'[t] == \frac{1}{8}\left(-\frac{25}{3}u_3[t] + \frac{400\,u_4[t]}{3} - 250\,u_5[t] + \frac{400\,u_6[t]}{3} - \frac{25\,u_7[t]}{3}\right),$$
$$u_6'[t] == \frac{1}{8}\left(-\frac{25}{3}u_4[t] + \frac{400\,u_5[t]}{3} - 250\,u_6[t] + \frac{400\,u_7[t]}{3} - \frac{25\,u_8[t]}{3}\right),$$

$$u_7'[t] == \frac{1}{8}\left(-\frac{25}{3}u_5[t] + \frac{400\,u_6[t]}{3} - 250\,u_7[t] + \frac{400\,u_8[t]}{3} - \frac{25\,u_9[t]}{3}\right),$$
$$u_8'[t] == \frac{1}{8}\left(-\frac{25}{3}u_6[t] + \frac{400\,u_7[t]}{3} - 250\,u_8[t] + \frac{400\,u_9[t]}{3} - \frac{25\,u_{10}[t]}{3}\right),$$
$$u_9'[t] == \frac{1}{8}\left(\frac{25\,u_5[t]}{3} - 50\,u_6[t] + \frac{350\,u_7[t]}{3} - \frac{100\,u_8[t]}{3} - 125\,u_9[t] + \frac{250\,u_{10}[t]}{3}\right),$$
$$\frac{5}{2}u_6'[t] - \frac{40}{3}u_7'[t] + 30\,u_8'[t] - 40\,u_9'[t] + \frac{125}{6}u_{10}'[t] == 0\Big\}$$

`NDSolve` is capable of solving the system as is for the appropriate derivatives, so it is ready for the ODEs.

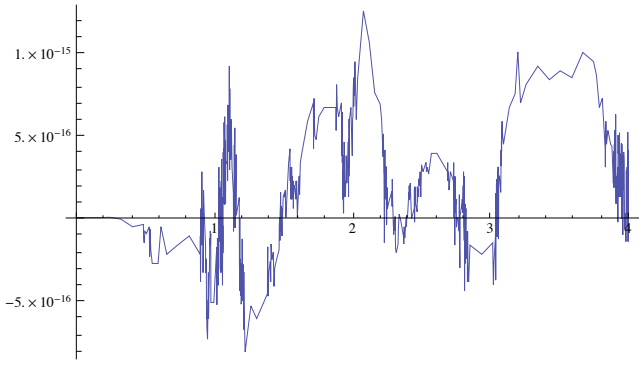
In[16]:=
```
diffsol = NDSolve[{eqns, Thread[U[0] == Table[0, {11}]]}, U[t], {t, 0, 4}]
```

Out[16]=
```
{{u0[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u1[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u2[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u3[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u4[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u5[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u6[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u7[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u8[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u9[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u10[t] → InterpolatingFunction[{{0., 4.}}, <>][t]}}
```

This shows a plot of how well the boundary condition at $x == 1$ was satisfied.

In[17]:=
```
Plot[Evaluate[Apply[Subtract, bc] /. diffsol], {t, 0, 4}]
```

Out[17]=



Treating the boundary conditions as algebraic conditions saves a couple of steps in the processing at the expense of using a DAE solver.

This replaces the first and last equations (from before) with algebraic conditions corresponding to the boundary conditions.

```
In[18]:= eqns[[1]] = u0[t] == Sin[2 π t];
         eqns[[-1]] = bc;
         eqns
```

```
Out[20]= {u0[t] == Sin[2 π t], u1'[t] == 1/8 (250 u0[t]/3 - 125 u1[t] - 100 u2[t]/3 + 350 u3[t]/3 - 50 u4[t] + 25 u5[t]/3),
 u2'[t] == 1/8 (-25/3 u0[t] + 400 u1[t]/3 - 250 u2[t] + 400 u3[t]/3 - 25 u4[t]/3),
 u3'[t] == 1/8 (-25/3 u1[t] + 400 u2[t]/3 - 250 u3[t] + 400 u4[t]/3 - 25 u5[t]/3),
 u4'[t] == 1/8 (-25/3 u2[t] + 400 u3[t]/3 - 250 u4[t] + 400 u5[t]/3 - 25 u6[t]/3),
 u5'[t] == 1/8 (-25/3 u3[t] + 400 u4[t]/3 - 250 u5[t] + 400 u6[t]/3 - 25 u7[t]/3),
 u6'[t] == 1/8 (-25/3 u4[t] + 400 u5[t]/3 - 250 u6[t] + 400 u7[t]/3 - 25 u8[t]/3),
 u7'[t] == 1/8 (-25/3 u5[t] + 400 u6[t]/3 - 250 u7[t] + 400 u8[t]/3 - 25 u9[t]/3),
 u8'[t] == 1/8 (-25/3 u6[t] + 400 u7[t]/3 - 250 u8[t] + 400 u9[t]/3 - 25 u10[t]/3),
 u9'[t] == 1/8 (25 u5[t]/3 - 50 u6[t] + 350 u7[t]/3 - 100 u8[t]/3 - 125 u9[t] + 250 u10[t]/3),
 5 u6[t]/2 - 40 u7[t]/3 + 30 u8[t] - 40 u9[t] + 125 u10[t]/6 == 0}
```

This solves the system of DAEs with `NDSolve`.

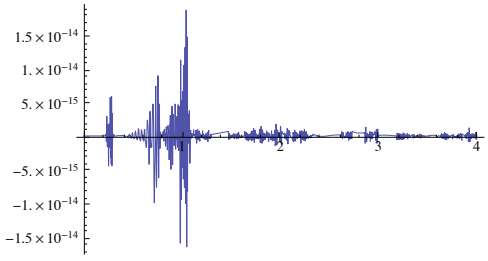
```
In[21]:= daesol = NDSolve[{eqns, Thread[U[0] == Table[0, {11}]]}, U[t], {t, 0, 4}]
```

```
Out[21]= {{u0[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u1[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u2[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u3[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u4[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u5[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u6[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u7[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u8[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u9[t] → InterpolatingFunction[{{0., 4.}}, <>][t],
  u10[t] → InterpolatingFunction[{{0., 4.}}, <>][t]}}
```

This shows how well the boundary condition was satisfied.

```
In[22]:= Plot[Evaluate[Apply[Subtract, bc] /. daesol], {t, 0, 4}, PlotRange → All]
```

Out[22]=

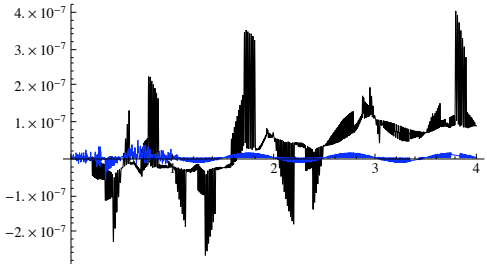


For this example, the boundary condition was satisfied well within tolerances in both cases, but the differentiation method did very slightly better. This is not always true; in some cases, with the differentiation method, the boundary condition can experience cumulative drift since the error control in this case is only local. The Dirichlet boundary condition at $x == 0$ in this example shows some drift.

This makes a plot that compares how well the Dirichlet boundary condition at $x == 0$ was satisfied with the two methods. The solution with the differentiated boundary condition is shown in black.

```
In[23]:= Plot[Evaluate[{u0[t] /. diffsol, u0[t] /. daesol} - Sin[2 π t] ],
          {t, 0, 4}, PlotStyle → {{Black}, {Blue}}, PlotRange → All]
```

Out[23]=



When using `NDSolve`, it is easy to switch between the two methods by using the `DifferentiateBoundaryConditions` option. Remember that when you use `DifferentiateBoundaryConditions -> False`, you are not as free to choose integration methods; the method needs to be a DAE solver.

With systems of PDEs or equations with higher-order derivatives having more complicated boundary conditions, both methods can be made to work in general. When there are multiple boundary conditions at one end, it may be necessary to attach some conditions to interior points. Here is an example of a PDE with two boundary conditions at each end of the spatial interval.

This solves a differential equation with two boundary conditions at each end of the spatial interval. The `StiffnessSwitching` integration method is used to avoid potential problems with stability from the fourth-order derivative.

```
In[25]:= dsol = NDSolve[{D[u[x, t], t, t] == -D[u[x, t], x, x, x, x],
    {u[x, t] == x^2/2 - x^3/3 + x^4/12,
      D[u[x, t], t] == 0} /. t → 0,
    Table[(D[u[x, t], {x, d}] == 0) /. x → b, {b, 0, 1}, {d, 2 b, 2 b + 1}]
   },
   u, {x, 0, 1}, {t, 0, 2}, Method → "StiffnessSwitching", InterpolationOrder → All]
Out[25]= {{u → InterpolatingFunction[{{0., 1.}, {0., 2.}}, <>]}}
```

Understanding the message about spatial error will be addressed in the next section. For now, ignore the message and consider the boundary conditions.

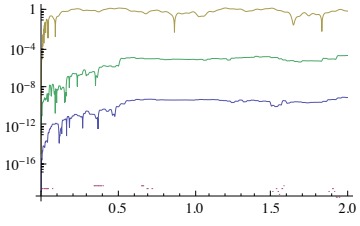
This forms a list of `InterpolatingFunctions` differentiated to the same order as each of the boundary conditions.

```
In[26]:= bct =
  Table[(D[u[x, t], {x, d}] /. x → b) /. First[dsol], {b, 0, 1}, {d, 2 b, 2 b + 1}]
Out[26]= {{InterpolatingFunction[{{0., 1.}, {0., 2.}}, <>][0, t],
    InterpolatingFunction[{{0., 1.}, {0., 2.}}, <>][0, t]},
   {InterpolatingFunction[{{0., 1.}, {0., 2.}}, <>][1, t],
    InterpolatingFunction[{{0., 1.}, {0., 2.}}, <>][1, t]}}
```

This makes a logarithmic plot of how well each of the four boundary conditions is satisfied by the solution computed with `NDSolve` as a function of t.

```
In[27]:= LogPlot[Evaluate[Map[Abs, bct, {2}]], {t, 0, 2}, PlotRange → All]
```

Out[27]=



It is clear that the boundary conditions are satisfied to well within the tolerances allowed by `AccuracyGoal` and `PrecisionGoal` options. It is typical that conditions with higher-order derivatives will not be satisfied as well as those with lower-order derivatives.

Inconsistent Boundary Conditions

It is important that the boundary conditions you specify be consistent with both the initial condition and the PDE. If this is not the case, `NDSolve` will issue a message warning about the inconsistency. When this happens, the solution may not satisfy the boundary conditions, and in the worst cases, instability may appear.

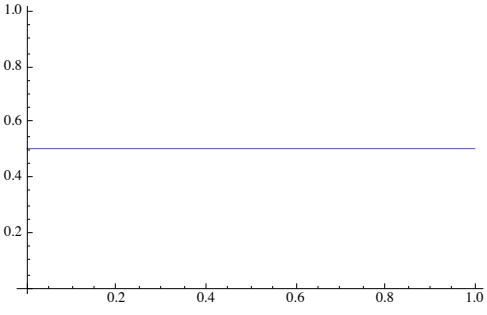
In this example for the heat equation, the boundary condition at $x == 0$ is clearly inconsistent with the initial condition.

```
In[2]:= sol = NDSolve[{D[u[t, x], t] == D[u[t, x], x, x],
    u[t, 0] == 1, u[t, 1] == 0, u[0, x] == .5}, u, {t, 0, 1}, {x, 0, 1}]

NDSolve::ibcinc: Warning: Boundary and initial conditions are inconsistent. >>

Out[2]= {{u → InterpolatingFunction[{{0., 1.}, {0., 1.}}, <>]}}
```

This shows a plot of the solution at $x == 0$ as a function of t. The boundary condition $u(t, 0) = 1$ is clearly not satisfied.

```
In[3]:= Plot[Evaluate[First[u[t, 0] /. sol]], {t, 0, 1}]
```

Out[3]=



The reason the boundary condition is not satisfied is because once it is differentiated, it becomes $u_t(t, 0) = 0$, so the solution will be whatever constant value comes from the initial condition.

When the boundary conditions are not differentiated, the DAE solver in effect modifies the initial conditions so that the boundary condition is satisfied.

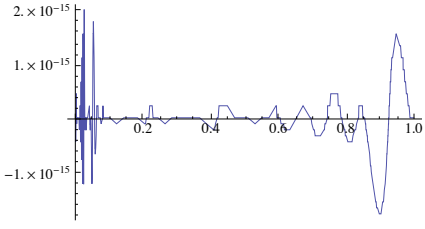
In[4]:=
```
daesol = NDSolve[{D[u[t, x], t] == D[u[t, x], x, x],
    u[t, 0] == 1, u[t, 1] == 0, u[0, x] == 0}, u, {t, 0, 1}, {x, 0, 1},
  Method → {"MethodOfLines", "DifferentiateBoundaryConditions" → False}];
Plot[First[u[t, 0] /. daesol] - 1, {t, 0, 1}, PlotRange → All]
```

NDSolve::ibcinc: Warning: Boundary and initial conditions are inconsistent. ≫

NDSolve::ivcon: The given initial conditions were not consistent with the differential–algebraic equations. NDSolve will attempt to correct the values. ≫

NDSolve::ivres: NDSolve has computed initial values that give a zero residual for the differential–algebraic system, but some components are different from those specified. If you need those to be satisfied, it is recommended that you give initial conditions for all dependent variables and derivatives of them.

Out[5]=

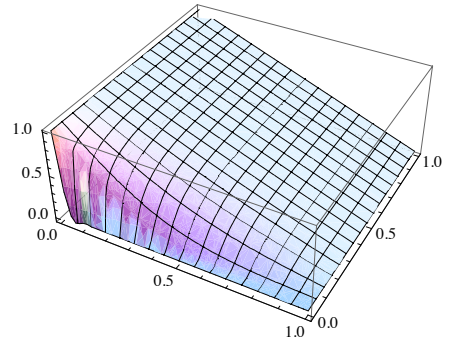


It is not always the case that the DAE solver will find good initial conditions that lead to an effectively correct solution like this. A better way to handle this problem is to give an initial condition that is consistent with the boundary conditions, even if it is discontinuous. In this case the unit step function does what is needed.

This uses a discontinuous initial condition to match the boundary condition, giving a solution correct to the resolution of the spatial discretization.

In[6]:=
```
usol = NDSolve[{D[u[t, x], t] == D[u[t, x], x, x], u[t, 0] == 1,
    u[t, 1] == 0, u[0, x] == UnitStep[-x]}, u, {t, 0, 1}, {x, 0, 1}];
Plot3D[Evaluate[First[u[t, x] /. usol]], {x, 0, 1}, {t, 0, 1}]
```

Out[7]=



In general, with discontinuous initial conditions, spatial error estimates cannot be satisfied, since they are predicated on smoothness so, in general, it is best to choose how well you want to model the effect of the discontinuity by either giving a smooth function which approximates the discontinuity or by specifying explicitly the number of points to use in the spatial discretization. More detail on spatial error estimates and discretization is given in "Spatial Error Estimates".

A more subtle inconsistency arises when the temporal variable has higher-order derivatives and boundary conditions may be differentiated more than once.

Consider the wave equation

$$u_{tt} = u_{xx} \qquad \begin{array}{ll} \text{with initial conditions} & u(0, x) = \sin(x) \quad u_t(0, x) = 0 \\ \text{and boundary conditions} & u(t, 0) = 0 \quad u_x(t, 0) = e^t \end{array}$$

The initial condition $\sin(x)$ satisfies the boundary conditions, so you might be surprised that `NDSolve` issues the `NDSolve::ibcinc` message.

In this example, the boundary and initial conditions appear to be consistent at first glance, but actually have inconsistencies which show up under differentiation.

```
In[8]:= isol = NDSolve[
          {D[u[t, x], t, t] == D[u[t, x], x, x], u[0, x] == Sin[x], (D[u[t, x], t] /. t → 0) == 0,
           u[t, 0] == 0, (D[u[t, x], x] /. x → 0) == Exp[t]}, u, {t, 0, 1}, {x, 0, 2 π}]

        NDSolve::ibcinc: Warning: Boundary and initial conditions are inconsistent. ≫

Out[8]= {{u → InterpolatingFunction[{{0., 1.}, {0., 6.28319}}, <>]}}
```

The inconsistency appears when you differentiate the second initial condition with respect to x, giving $u_{tx}(x, 0) = 0$, and differentiate the second boundary condition with respect to t, giving $u_{xt}(0, t) = e^t$. These two are inconsistent at $x = t = 0$.

Occasionally, `NDSolve` will issue the `NDSolve::ibcinc` message warning about inconsistent boundary conditions when they are actually consistent. This happens due to discretization error in approximating Neumann boundary conditions or any boundary condition that involves a spatial derivative. The reason this happens is that spatial error estimates (see "Spatial Error Estimates") used to determine how many points to discretize with are based on the PDE and the initial condition, but not the boundary conditions. The one-sided finite difference formulas that are used to approximate the boundary conditions also have larger error than a centered formula of the same order, leading to additional discretization error at the boundary. Typically this is not a problem, but it is possible to construct examples where it does occur.

In this example, because of discretization error, `NDSolve` incorrectly warns about inconsistent boundary conditions.

In[9]:=
```
sol = NDSolve[{D[u[x, t], t] == D[u[x, t], x, x], u[x, 0] == 1 - Sin[4 * Pi * x] / (4 * Pi),
    u[0, t] == 1, u[1, t] + Derivative[1, 0][u][1, t] == 0}, u, {x, 0, 1}, {t, 0, 1}]
```

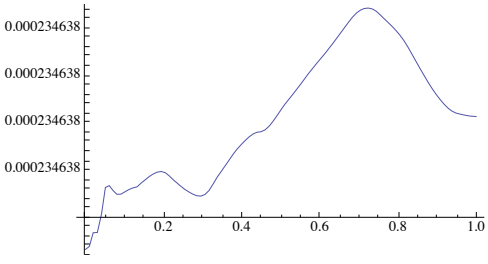
```
NDSolve::ibcinc: Warning: Boundary and initial conditions are inconsistent. ≫
```

Out[9]=
```
{{u → InterpolatingFunction[{{0., 1.}, {0., 1.}}, <>]}}
```

A plot of the boundary condition shows that the error, while not large, is outside of the default tolerances.

In[10]:=
```
Plot[First[u[1, t] + Derivative[1, 0][u][1, t] /. sol], {t, 0, 1}]
```

Out[10]=

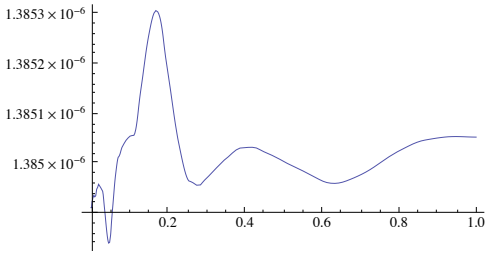


When the boundary conditions are consistent, a way to correct this error is to specify that `NDSolve` use a finer spatial discretization.

With a finer spatial discretization, there is no message and the boundary condition is satisfied better.

In[13]:=
```
fsol =
  NDSolve[{D[u[x, t], t] == D[u[x, t], x, x], u[x, 0] == 1 - Sin[4 * Pi * x] / (4 * Pi),
    u[0, t] == 1, u[1, t] + Derivative[1, 0][u][1, t] == 0},
   u, {x, 0, 1}, {t, 0, 1}, Method → {"MethodOfLines",
     "SpatialDiscretization" → {"TensorProductGrid", "MinPoints" → 100}}];
Plot[First[u[1, t] + Derivative[1, 0][u][1, t] /. fsol], {t, 0, 1}, PlotRange → All]
```

Out[14]=



Spatial Error Estimates

Overview

When NDSolve solves a PDE, unless you have specified the spatial grid for it to use, by giving it explicitly or by giving equal values for the MinPoints and MaxPoints options, NDSolve needs to make a spatial error estimate.

Ideally, the spatial error estimates would be monitored over time and the spatial mesh updated according to the evolution of the solution. The problem of grid adaptivity is difficult enough for a specific type of PDE and certainly has not been solved in any general way. Furthermore, techniques such as local refinement can be problematic with the method of lines since changing the number of mesh points requires a complete restart of the ODE methods. There are moving mesh techniques that appear promising for this approach, but at this point, NDSolve uses a static grid. The grid to use is determined by an a priori error estimate based on the initial condition. An a posteriori check is done at the end of the temporal interval for reasonable consistency and a warning message is given if that fails. This can, of course, be fooled, but in practice it provides a reasonable compromise. The most common cause of failure is when initial conditions have little variation, so the estimates are essentially meaningless. In this case, you may need to choose some appropriate grid settings yourself.

Load a package that will be used for extraction of data from InterpolatingFunction objects.

```
In[1]:= Needs["DifferentialEquations`InterpolatingFunctionAnatomy`"]
```

A priori Error Estimates

When NDSolve solves a PDE using the method of lines, a decision has to be made on an appropriate spatial grid. NDSolve does this using an error estimate based on the initial condition (thus, a priori).

It is easiest to show how this works in the context of an example. For illustrative purposes, consider the sine-Gordon equation in one dimension with periodic boundary conditions.

This solves the sine-Gordon equation with a Gaussian initial condition.

```
In[5]:= ndsol =
         NDSolve[{D[u[x, t], t, t] == D[u[x, t], x, x] - Sin[u[x, t]], u[x, 0] == Exp[-(x^2)],
           Derivative[0, 1][u][x, 0] == 0, u[-5, t] == u[5, t]}, u, {x, -5, 5}, {t, 0, 5}]
Out[5]= {{u → InterpolatingFunction[{{-5., 5.}, {0., 5.}}, <>]}}
```

This gives the number of spatial and temporal points used, respectively.

```
In[6]:= Map[Length, InterpolatingFunctionCoordinates[First[u /. ndsol]]]
Out[6]= {97, 15}
```

The temporal points are chosen adaptively by the ODE method based on local error control. NDSolve used 97 (98 including the periodic endpoint) spatial points. This choice will be illustrated through the steps that follow.

In the equation processing phase of NDSolve, one of the first things that happen is that equations with second- (or higher-) order temporal derivatives are replaced with systems with only first-order temporal derivatives.

This is a first-order system equivalent to the sine-Gordon equation earlier.

```
In[7]:= {D[u[x, t], t] == v[x, t], D[v[x, t], t] == D[u[x, t], x, x] + -Sin[u[x, t]]}
Out[7]= {u^(0,1)[x, t] == v[x, t], v^(0,1)[x, t] == -Sin[u[x, t]] + u^(2,0)[x, t]}
```

The next stage is to solve for the temporal derivatives.

This is the solution for the temporal derivatives, with the right-hand side of the equations in normal (ODE) form.

```
In[8]:= rhs = {D[u[x, t], t], D[v[x, t], t]} /. Solve[%, {D[u[x, t], t], D[v[x, t], t]}]
Out[8]= {{v[x, t], -Sin[u[x, t]] + u^(2,0)[x, t]}}
```

Now the problem is to choose a uniform grid that will approximate the derivative to within local error tolerances as specified by AccuracyGoal and PrecisionGoal. For this illustration, use the default "DifferenceOrder" (4) and the default AccuracyGoal and PrecisionGoal (both 4 for PDEs). The methods used to integrate the system of ODEs that results from discretization base their own error estimates on the assumption of sufficiently accurate function values. The estimates here have the goal of finding a spatial grid for which (at least with the initial condition) the spatial error is somewhat balanced with the local temporal error.

This sets variables to reflect the default settings for "DifferenceOrder", AccuracyGoal, and PrecisionGoal.

```
In[9]:= p = 4;
        atol = 1.*^-4;
        rtol = 1.*^-4;
```

The error estimate is based on Richardson extrapolation. If you know that the error is $O(h^p)$ and you have two approximations y_1 and y_2 at different values, h_1 and h_2 of h, then you can, in effect, extrapolate to the limit $h \to 0$ to get an error estimate

$$y_1 - y_2 = \left(c\, h_1^p + y\right) - \left(c\, h_2^p + y\right) = c\, h_1^p \left(1 - \left(\frac{h_2}{h_1}\right)^p\right)$$

so the error in y_1 is estimated to be

$$\|y_1 - y\| \cong c\, h_1^p = \frac{\|y_1 - y_2\|}{\left(1 - \left(\frac{h_2}{h_1}\right)^p\right)} \tag{1}$$

Here y_1 and y_2 are vectors of different length and y is a function, so you need to choose an appropriate norm. If you choose $h_1 = 2\, h_2$, then you can simply use a scaled norm on the components common to both vectors, which is all of y_1 and every other point of y_2. This is a good choice because it does not require any interpolation between grids.

For a given interval on which you want to set up a uniform grid, you can define a function $h(n) = L/n$, where L is the length of the interval such that the grid is $\{x_0, x_1, x_1, \ldots, x_n\}$, where $x_j == x_0 + j\, h(n)$.

This defines functions that return the step size h and a list of grid points as a function of n for the sine-Gordon equation.

```
In[12]:= Clear[h, grid];
         h[n_] := 10/n;
         grid[n_] := N[-5 + Range[0, n] * h[n]];
```

For a given grid, the equation can be discretized using finite differences. This is easily done using `NDSolve`FiniteDifferenceDerivative`.

This defines a symbolic discretization of the right-hand side of the sine-Gordon equation as a function of a grid. It returns a function of u and v, which gives the approximate values for u_t and v_t in a list. (Note that in principle this works for any grid, uniform or not, though in the following, only uniform grids will be used.)

```
In[15]:= sdrhs[grid_] := Block[{app, u, v},
           app = rhs /.
             Derivative[i_, 0][var : (u | v)][x, t] :> NDSolve`FiniteDifferenceDerivative[
                 i, grid, "DifferenceOrder" → p, PeriodicInterpolation → True][var];
           app = app /. (var : (u | v))[x, t] :> var;
           Function[{u, v}, Evaluate[app]]]
```

For a given step size and grid, you can also discretize the initial conditions for u and v.

This defines a function that discretizes the initial conditions for u and v. The last grid point is dropped because, by periodic continuation, it is considered the same as the first.

```
In[16]:= dinit[n_] := Transpose[Map[Function[{x}, {Exp[-x^2], 0}], Drop[grid[n], -1]]]
```

The quantity of interest is the approximation of the right-hand side for a particular value of n with this initial condition.

This defines a function that returns a vector consisting of the approximation of the right-hand side of the equation for the initial condition for a given step size and grid. The vector is flattened to make subsequent analysis of it simpler.

```
In[17]:= rhsinit[n_] := Flatten[Apply[sdrhs[grid[n]], dinit[n]]]
```

Starting with a particular value of n, you can obtain the error estimate by generating the right hand side for n and $2n$ points.

This gives an example of the right-hand side approximation vector for a grid with 10 points.

```
In[18]:= rhsinit[10]
```

```
Out[18]= {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -0.0000202683, -0.00136216, -0.00666755,
 0.343233, 0.0477511, -2.36351, 0.0477511, 0.343233, -0.00666755, -0.00136216}
```

This gives an example of the right-hand side approximation vector for a grid with 20 points.

```
In[19]:= rhsinit[20]
```

```
Out[19]= {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -5.80538×10^-8,
 -1.01297×10^-6, -0.0000168453, -0.0000373357, 0.00285852, 0.0419719, 0.248286,
 0.640267, 0.337863, -1.48981, -2.77952, -1.48981, 0.337863, 0.640267,
 0.248286, 0.0419719, 0.00285852, -0.0000373357, -0.0000168453, -1.01297×10^-6}
```

As mentioned earlier, every other point on the grid with $2n$ points lies on the grid with n points. Thus, for simplicity, you can use a norm that only compares points common to both grids. Because the goal is to ultimately satisfy absolute and relative tolerance criteria, it is appropriate to use a scaled norm. In addition to taking into account the size of the right-hand side for the scaling, it is also important to include the size of the corresponding components of u and v on the grid since error in the right-hand side is ultimately included in u and v.

This defines a norm function for the difference in the approximation of the right-hand side.

```
In[20]:= dnorm[rhsn_, rhs2n_, uv_] := Module[{rhs2 = Take[rhs2n, {1, -1, 2}]},
    NDSolve`ScaledVectorNorm[Infinity, {rtol, atol}][
     rhsn - rhs2, Internal`MaxAbs[rhs2, uv]]] /;
   ((Length[rhs2n] == 2 Length[rhsn]) && (Length[rhsn] == Length[uv]))
```

This applies the norm function to the two approximations found.

```
In[21]:= dnorm[rhsinit[10], rhsinit[20], Flatten[dinit[10]]]
Out[21]= 2168.47
```

To get the error estimate form the distance, according to the Richardson extrapolation formula (3), this simply needs to be divided by $(1-(h_2/h_1)^p) = (1-2^{-p})$.

This computes the error estimate for $n == 10$. Since this is based on a scaled norm, the tolerance criteria are satisfied if the result is less than 1.

```
In[22]:= % / (1 - 2^-p)
Out[22]= 2313.04
```

This makes a function that combines the earlier functions to give an error estimate as a function of n.

```
In[23]:= errest[n_] := dnorm[rhsinit[n], rhsinit[2 n], Flatten[dinit[n]]] / (1 - 2^-p)
```

The goal is to find the minimum value of n, such that the error estimate is less than or equal to 1 (since it is based on a scaled norm). In principle, it would be possible to use a root-finding algorithm on this, but since n can only be an integer, this would be overkill and adjustments would have to be made to the stopping conditions. An easier solution is simply to use the simple Richardson extrapolation formula to predict what value of n would be appropriate and repeat the prediction process until the appropriate n is found.

The condition to satisfy is

$$c\, h_{\text{opt}}^p = 1$$

and you have estimated that

$$c\, h(n)^p \simeq \text{errest}(n)$$

so you can project that

$$h_{\text{opt}} \simeq h(n) \left(\frac{1}{\text{errest}(n)} \right)^{1/p}$$

or in terms of n, which is proportional to the reciprocal of h,

$$n_{\text{opt}} \simeq \left\lfloor n\, \text{errest}(n)^{1/p} \right\rfloor$$

This computes the predicted optimal value of n based on the error estimate for $n == 10$ computed earlier.

```
In[24]:= Ceiling[10 errest[10]^(1/p)]
Out[24]= 70
```

This computes the error estimate for the new value of n.

```
In[25]:= errest[%]
Out[25]= 3.75253
```

Often the case that a prediction based on a very coarse grid will be inadequate. A coarse grid may completely miss some solution features that contribute to the error on a finer grid. Also, the error estimate is based on an asymptotic formula, so for coarse spacings, the estimate itself may not be very good, even when all the solution features are resolved to some extent.

In practice, it can be fairly expensive to compute these error estimates. Also, it is not necessary to find the very optimal n, but one that satisfies the error estimate. Remember, everything can change as the PDE evolves, so it is simply not worth a lot of extra effort to find an optimal spacing for just the initial time. A simple solution is to include an extra factor greater than 1 in the prediction formula for the new n. Even with an extra factor, it may still take a few iterations to get to an acceptable value. It does, however, typically make the process faster.

This defines a function that gives a predicted value for the number of grid points, which should satisfy the error estimate.

```
In[26]:= pred[n_] := Ceiling[1.05 n errest[n]^(1/p)]
```

This iterates the predictions until a value is found that satisfies the error estimate.

```
In[27]:= NestWhileList[pred, 10, (errest[#] > 1) &]
Out[27]= {10, 73, 100}
```

It is important to note that this process must have a limiting value since it may not be possible to satisfy the error tolerances, for example, with a discontinuous initial function. In `NDSolve`, the `MaxSteps` option provides the limit; for spatial discretization, this defaults to a total of 10000 across all spatial dimensions.

Pseudospectral derivatives cannot use this error estimate since they have an exponential rather than a polynomial convergence. An estimate can be made based on the formula used earlier in

the limit `p -> Infinity`. What this amounts to is considering the result on the finer grid to be exact and basing the error estimate on the difference since $1-2^{-p}$ approaches 1. A better approach is to use the fact that on a given grid with n points, the pseudospectral method is $O(h^n)$. When comparing for two grids, it is appropriate to use the smaller n for p. This provides an imperfect, but adequate estimate for the purpose of determining grid size.

This modifies the error estimation function so that it will work with pseudospectral derivatives.

```
In[28]:= errest[n_] :=
          dnorm[rhsinit[n], rhsinit[2 n], Flatten[dinit[n]]] / (1 - 2^-If[p === "Pseudospectral",n,p])
```

The prediction formula can be modified to use n instead of p in a similar way.

This modifies the function predicting an appropriate value of n to work with pseudospectral derivatives. This formulation does not try to pick an efficient FFT length.

```
In[29]:= pred[n_] := Ceiling[1.05 n errest[n]^(1/If[p === "Pseudospectral",n,p])]
```

When finalizing the choice of n for a pseudospectral method, an additional consideration is to choose a value that not only satisfies the tolerance conditions, but is also an efficient length for computing FFTs. In *Mathematica*, an efficient FFT does not require a power of two length since the `Fourier` command has a prime factor algorithm built in.

Typically, the difference order has a profound effect on the number of points required to satisfy the error estimate.

This makes a table of the number of points required to satisfy the a priori error estimate as a function of the difference order.

```
In[30]:= TableForm[Map[Block[{p = #}, {p, NestWhile[pred, 10, (errest[#] > 1) &]}] &,
           {2, 4, 6, 8, "Pseudospectral"}],
          TableHeadings → {{}, {"DifferenceOrder", "Number of points"}}]
```

Out[30]//TableForm=

DifferenceOrder	Number of points
2	804
4	100
6	53
8	37
Pseudospectral	24

A table of the number of points required as a function of difference order goes a long way toward explaining why the default setting for the method of lines is `"DifferenceOrder" -> 4`: the improvement from 2 to 4 is usually most dramatic and in the default tolerance range, fourth-order differences do not tend to produce large roundoff errors, which can be the case with higher orders. Pseudospectral differences are often a good choice, particularly with periodic boundary conditions, but they are not a good choice for the default because they lead to full Jacobian matrices, which can be expensive to generate and solve if needed for stiff equations.

For nonperiodic grids, the error estimate is done using only interior points. The reason is that the error coefficients for the derivatives near the boundary are different. This may miss features that are near the boundary, but the main idea is to keep the estimate simple and inexpensive since the evolution of the PDE may change everything anyway.

For multiple spatial dimensions, the determination is made one dimension at a time. Since better resolution in one dimension may change the requirements for another, the process is repeated in reverse order to improve the choice.

A posteriori Error Estimates

When the solution of a PDE is computed with NDSolve, a final step is to do a spatial error estimate on the evolved solution and issue a warning message if this is excessively large.

These error estimates are done in a manner similar to the a priori estimates described previously. The only real difference is that, instead of using grids with n and $2n$ points to estimate the error, grids with $n/2$ and n points are used. This is because, while there is no way to generate the values on a grid of $2n$ points without using interpolation, which would introduce its own errors, values are readily available on a grid of $n/2$ points simply by taking every other value. This is easily done in the Richardson extrapolation formula by using $h_2 == 2h_1$, which gives

$$\|y_1 - y\| \cong \frac{\|y_1 - y_2\|}{(2^p - 1)}$$

This defines a function (based on functions defined in the previous section) that can compute an error estimate on the solution of the sine-Gordon equation from solutions for u and v expressed as vectors. The function has been defined to be a function of the grid since this is applied to a grid already constructed. (Note, as defined here, this only works for grids of even length. It is not difficult to handle odd length, but it makes the function somewhat more complicated.)

```
In[31]:= posterrest[{uvec_, vvec_}, grid_] := Module[{
           huvec = Take[uvec, {1, -1, 2}],
           hvvec = Take[vvec, {1, -1, 2}],
           hgrid = Take[grid, {1, -1, 2}]},
          dnorm[Flatten[sdrhs[hgrid][huvec, hvvec]],
            Flatten[sdrhs[grid][uvec, vvec]], Flatten[{huvec, hvvec}]] /
           (2^If[p === "Pseudospectral", Length[grid]/2, p] - 1)]
```

This solves the sine-Gordon equation with a Gaussian initial condition.

```
In[41]:= ndsol = First[NDSolve[{D[u[x, t], t, t] == D[u[x, t], x, x] + -Sin[u[x, t]],
             u[x, 0] == Exp[-(x^2)], Derivative[0, 1][u][x, 0] == 0, u[-5, t] == u[5, t]},
           u, {x, -5, 5}, {t, 0, 5}, InterpolationOrder → All]]
Out[41]= {u → InterpolatingFunction[{{-5., 5.}, {0., 5.}}, <>]}
```

This is the grid used in the spatial direction that is the first set of coordinates used in the `InterpolatingFunction`. A grid with the last point dropped is used to obtain the values because of periodic continuation.

```
In[42]:= ndgrid = InterpolatingFunctionCoordinates[u /. ndsol][[1]]
         pgrid = Drop[ndgrid, -1];
```

```
Out[42]= {-5., -4.89583, -4.79167, -4.6875, -4.58333, -4.47917, -4.375, -4.27083, -4.16667, -4.0625,
 -3.95833, -3.85417, -3.75, -3.64583, -3.54167, -3.4375, -3.33333, -3.22917, -3.125,
 -3.02083, -2.91667, -2.8125, -2.70833, -2.60417, -2.5, -2.39583, -2.29167, -2.1875,
 -2.08333, -1.97917, -1.875, -1.77083, -1.66667, -1.5625, -1.45833, -1.35417, -1.25,
 -1.14583, -1.04167, -0.9375, -0.833333, -0.729167, -0.625, -0.520833, -0.416667,
 -0.3125, -0.208333, -0.104167, 0., 0.104167, 0.208333, 0.3125, 0.416667, 0.520833, 0.625,
 0.729167, 0.833333, 0.9375, 1.04167, 1.14583, 1.25, 1.35417, 1.45833, 1.5625, 1.66667,
 1.77083, 1.875, 1.97917, 2.08333, 2.1875, 2.29167, 2.39583, 2.5, 2.60417, 2.70833, 2.8125,
 2.91667, 3.02083, 3.125, 3.22917, 3.33333, 3.4375, 3.54167, 3.64583, 3.75, 3.85417,
 3.95833, 4.0625, 4.16667, 4.27083, 4.375, 4.47917, 4.58333, 4.6875, 4.79167, 4.89583, 5.}
```

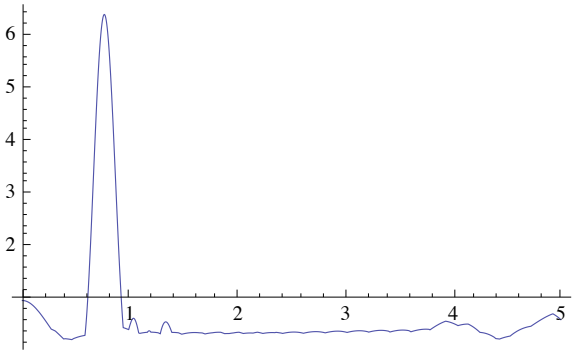
This makes a function that gives the a posteriori error estimate at a particular numerical value of t.

```
In[44]:= peet[t_?NumberQ] :=
          posterrest[{u[pgrid, t], Derivative[0, 1][u][pgrid, t]} /. ndsol, ndgrid]
```

This makes a plot of the a posteriori error estimate as a function of t.

```
In[45]:= Plot[peet[t], {t, 0, 5}, PlotRange → All]
```

```
Out[45]=
```



The large amount of local variation seen in this function is typical. For that reason, `NDSolve` does not warn about excessive error unless this estimate gets above 10 (rather than the value of 1, which is used to choose the grid based on initial conditions). The extra factor of 10 is further justified by the fact that the a posteriori error estimate is less accurate than the a priori one. Thus, when `NDSolve` issues a warning message based on the a posteriori error estimate, it is usually because new solution features have appeared or because there is instability in the solution process.

This is an example with the same initial condition used in the earlier examples, but where NDSolve gives a warning message based on the a posteriori error estimate.

```
In[46]:= bsol = First[NDSolve[{D[u[x, t], t] == 0.01 D[u[x, t], x, x] - u[x, t] D[u[x, t], x],
      u[x, 0] == e^-x^2, u[-5, t] == u[5, t]}, u, {x, -5, 5}, {t, 0, 4}]]
```

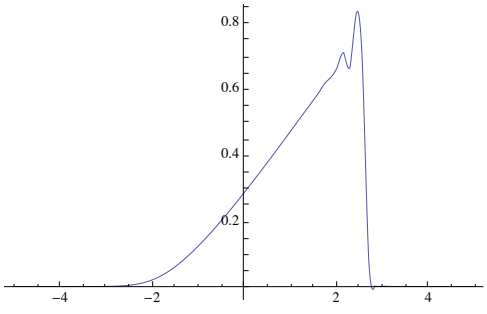
```
NDSolve::eerr :
  Warning: Scaled local spatial error estimate of 272.7279341590405` at t = 4.` in the direction
    of independent variable x is much greater than prescribed error tolerance.
    Grid spacing with 75 points may be too large to achieve the desired accuracy
    or precision.  A singularity may have formed or you may want to specify a
    smaller grid spacing using the MaxStepSize or MinPoints method options. ≫
```

```
Out[46]= {u → InterpolatingFunction[{{-5., 5.}, {0., 4.}}, <>]}
```

This shows a plot of the solution at $t == 4$. It is apparent that the warning message is appropriate because the oscillations near the peak are not physical.

```
In[47]:= Plot[u[x, 4] /. bsol, {x, -5, 5}, PlotRange → All]
```

Out[47]=



When the NDSolve::eerr message does show up, it may be necessary for you to use options to control the grid selection process since it is likely that the default settings did not find an accurate solution.

Controlling the Spatial Grid Selection

The NDSolve implementation of the method of lines has several ways to control the selection of the spatial grid.

option name	*default value*	
AccuracyGoal	Automatic	the number of digits of absolute tolerance for determining grid spacing
PrecisionGoal	Automatic	the number of digits of relative tolerance for determining grid spacing
"DifferenceOrder"	Automatic	the order of finite difference approximation to use for spatial discretization
Coordinates	Automatic	the list of coordinates for each spatial dimension $\{\{x_1, x_2, \ldots\}, \{y_1, y_2, \ldots\}, \ldots\}$ for independent variable dimensions $x, y, \ldots$; this overrides the settings for all the options following in this list
MinPoints	Automatic	the minimum number of points to be used for each dimension in the grid; for Automatic, value will be determined by the minimum number of points needed to compute an error estimate for the given difference order
MaxPoints	Automatic	the maximum number of points to be used in the grid
StartingPoints	Automatic	the number of points to begin the process of grid refinement using the a priori error estimates
MinStepSize	Automatic	the minimum grid spacing to use
MaxStepSize	Automatic	the maximum grid spacing to use
StartingStepSize	Automatic	the grid spacing to use to begin the process of grid refinement using the a priori error estimates

Tensor product grid options for the method of lines.

All the options for tensor product grid discretization can be given as a list with length equal to the number of spatial dimensions, in which case the parameter for each spatial dimension is determined by the corresponding component of the list.

With the exception of pseudospectral methods on nonperiodic problems, discretization is done with uniform grids, so when solving a problem on interval length L, there is a direct correspondence between the Points and StepSize options:

$$\begin{aligned} \texttt{MaxPoints} \to n &\iff \texttt{MaxStepSize} \to L/n \\ \texttt{MinPoints} \to n &\iff \texttt{MinStepSize} \to L/n \\ \texttt{StartingPoints} \to n &\iff \texttt{StartingStepSize} \to L/n \end{aligned}$$

The StepSize options are effectively converted to the equivalent Points values. They are simply provided for convenience since sometimes it is more natural to relate problem specification to step size rather then the number of discretization points. When values other then Automatic are specified for both the Points and corresponding StepSize option, generally, the more stringent restriction is used.

In the previous section an example was shown where the solution was not resolved sufficiently because the solution steepened as it evolved. The examples that follow will show some different ways of modifying the grid parameters so that the near shock is better resolved.

One way to avoid the oscillations that showed up in the solution as the profile steepened is to make sure that you use sufficient points to resolve the profile at its steepest. In the one-hump solution of Burgers' equation,

$$u_t + u\,u_x = \nu\,u_{xx}$$

it can be shown [W76] that the width of the shock profile is proportional to ν as $\nu \to 0$. More than 95% of the change is included in a layer of width $10\,\nu$. Thus, if you pick a maximum step size of half the profile width, there will always be a point somewhere in the steep part of the profile, and there is a hope of resolving it without significant oscillation.

This computes the solution to Burgers' equation, such that there are sufficient points to resolve the shock profile.

```
In[48]:= ν = 0.01;
         bsol2 = First[NDSolve[
             {D[u[x, t], t] == ν D[u[x, t], x, x] - u[x, t] D[u[x, t], x], u[x, 0] == ℯ^-x^2,
              u[-5, t] == u[5, t]}, u, {x, -5, 5}, {t, 0, 4}, Method → {"MethodOfLines",
               "SpatialDiscretization" → {"TensorProductGrid", "MaxStepSize" → 10 ν / 2}}]]

        NDSolve::eerr :
          Warning: Scaled local spatial error estimate of 82.77168552068868` at t = 4.` in the direction
              of independent variable x is much greater than prescribed error tolerance.
              Grid spacing with 201 points may be too large to achieve the desired accuracy
              or precision.  A singularity may have formed or you may want to specify a
              smaller grid spacing using the MaxStepSize or MinPoints method options. ≫

Out[49]= {u → InterpolatingFunction[{{-5., 5.}, {0., 4.}}, <>]}
```

Note that resolving the profile alone is by no means sufficient to meet the default tolerances of NDSolve, which requires an accuracy of 10^{-4}. However, once you have sufficient point to resolve the basic profile, it is not unreasonable to project based on the a posteriori error estimate shown in the NDSolve::eerr message (with an extra 10% since, after all, it is just a projection).

This computes the solution to Burgers' equation with the maximum step size chosen so that it should be small enough to meet the default error tolerances based on a projection from the error of the previous calculation.

```
In[50]:= ν = 0.01;
bsol3 = First[NDSolve[{D[u[x, t], t] == ν D[u[x, t], x, x] - u[x, t] D[u[x, t], x],
     u[x, 0] == ℯ^-x^2, u[-5, t] == u[5, t]}, u, {x, -5, 5},
    {t, 0, 4}, Method → {"MethodOfLines", "SpatialDiscretization" →
       {"TensorProductGrid", "MaxStepSize" → (10 ν / 2) / ((1.1) 85^(1/4))}}]]
Out[51]= {u → InterpolatingFunction[{{-5., 5.}, {0., 4.}}, <>]}
```

To compare solutions like this, it is useful to look at a plot of the solution only at the spatial grid points. Because the grid points are stored as a part of the `InterpolatingFunction`, it is fairly simple to define a function that does this.

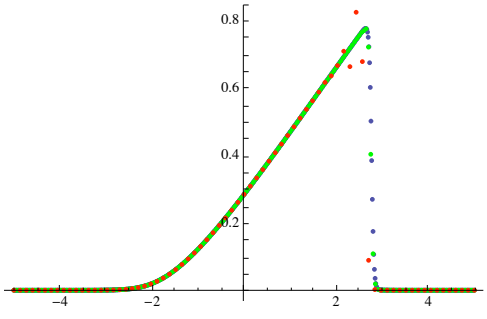
This defines a function that plots a solution only at the spatial grid points at a time t.

```
In[52]:= GridPointPlot[{u → if_InterpolatingFunction}, t_, opts___] :=
 Module[{grid = InterpolatingFunctionCoordinates[if][[1]]},
  ListPlot[Transpose[{grid, if[grid, t]}], opts]]
```

This makes a plot comparing the three solutions found at $t = 4$.

```
In[53]:= Show[Block[{ t = 4}, {
    GridPointPlot[bsol3, 4],
    GridPointPlot[bsol2, 4, PlotStyle → Hue[1 / 3]],
    GridPointPlot[bsol, 4, PlotStyle → Hue[1]]
   }
 ], PlotRange → All]
```

Out[53]=



In this example, the left-hand side of the domain really does not need so many points. The points need to be clustered where the steep profile evolves, so it might make sense to consider explicitly specifying a grid that has more points where the profile appears.

This solves Burgers' equation on a specified grid that has most of its points to the right of $x = 1$.

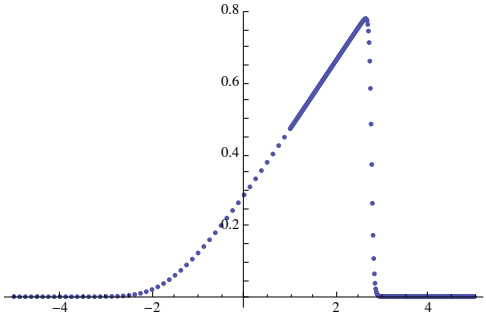
In[54]:=
```
mygrid = Join[-5. + 10 Range[0, 48] / 80, 1. + Range[1, 4 × 70] / 70];
ν = 0.01;
bsolg = First[NDSolve[
   {D[u[x, t], t] == ν D[u[x, t], x, x] - u[x, t] D[u[x, t], x], u[x, 0] == ℯ^-x^2,
    u[-5, t] == u[5, t]}, u, {x, -5, 5}, {t, 0, 4}, Method → {"MethodOfLines",
     "SpatialDiscretization" → {"TensorProductGrid", "Coordinates" → {mygrid}}}]]
```

Out[56]=
```
{u → InterpolatingFunction[{{-5., 5.}, {0., 4.}}, <>]}
```

This makes a plot of the values of the solution at the assigned spatial grid points.

In[57]:=
```
GridPointPlot[bsolg, 4]
```

Out[57]=



Many of the same principles apply to multiple spatial dimensions. Burgers' equation in two dimensions with anisotropy provides a good example.

This solves a variant of Burgers' equation in 2 dimensions with different velocities in the x and y directions.

In[58]:=
```
ν = 0.075;
sol1 =
 First[NDSolve[{D[u[t, x, y], t] == ν (D[u[t, x, y], x, x] + D[u[t, x, y], y, y]) -
      u[t, x, y] (2 D[u[t, x, y], x] - D[u[t, x, y], y]),
    u[0, x, y] == Exp[-(x^2 + y^2)], u[t, -4, y] == u[t, 4, y],
    u[t, x, -4] == u[t, x, 4]}, u, {t, 0, 2}, {x, -4, 4}, {y, -4, 4}]]
```

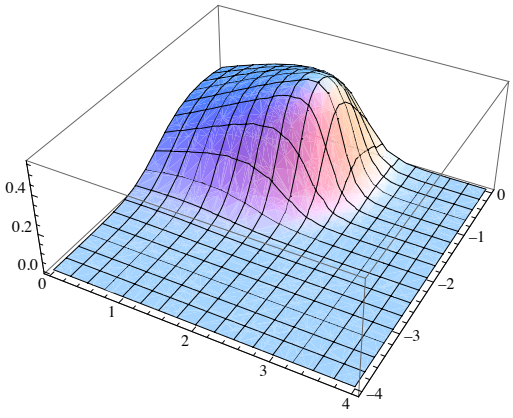
```
NDSolve::eerr :
  Warning: Scaled local spatial error estimate of 29.72177327883787` at t = 2.` in the direction
     of independent variable x is much greater than prescribed error tolerance.
     Grid spacing with 69 points may be too large to achieve the desired accuracy
     or precision.  A singularity may have formed or you may want to specify a
     smaller grid spacing using the MaxStepSize or MinPoints method options.
```

Out[59]=
```
{u → InterpolatingFunction[{{0., 2.}, {-4., 4.}, {-4., 4.}}, <>]}
```

This shows a surface plot of the leading edge of the solution at $t = 2$.

In[60]:=
```
Plot3D[u[2, x, y] /. sol1, {x, 0, 4}, {y, -4, 0}, PlotRange → All]
```

Out[60]=



Similar to the one-dimensional case, the leading edge steepens. Since the viscosity term (ν) is larger, the steepening is not quite so extreme, and this default solution actually resolves the front reasonably well. Therefore it should be possible to project from the error estimate to meet the default tolerances. A simple scaling argument indicates that the profile width in the x direction will be narrower than in the y direction by a factor of $\sqrt{2}$. Thus, it makes sense that the step sizes in the y direction can be larger than those in the x direction by this factor, or, correspondingly, that the minimum number of points can be a factor of $1/\sqrt{2}$ less.

This solves the 2-dimensional variant of Burgers' equation with appropriate step size restrictions in the x and y direction projected from the a posteriori error estimate of the previous computation, which was done with 69 points in the x direction.

In[61]:=
```
ν = 0.075;
sol2 =
 First[NDSolve[{D[u[t, x, y], t] == ν (D[u[t, x, y], x, x] + D[u[t, x, y], y, y]) -
       u[t, x, y] (2 D[u[t, x, y], x] - D[u[t, x, y], y]), u[0, x, y] == Exp[-(x^2 + y^2)],
    u[t, -4, y] == u[t, 4, y], u[t, x, -4] == u[t, x, 4]}, u, {t, 0, 2},
   {x, -4, 4}, {y, -4, 4}, Method → {"MethodOfLines", "SpatialDiscretization" →
      {"TensorProductGrid", "MinPoints" → Ceiling[{1, 1/Sqrt[2]} 69 31^(1/4)]}}]]
```

Out[62]=
```
{u → InterpolatingFunction[{{0., 2.}, {-4., 4.}, {-4., 4.}}, <>]}
```

This solution takes a substantial amount of time to compute, which is not surprising since the solution involves solving a system of more than 18000 ODEs. In many cases, particularly in more than one spatial dimension, the default tolerances may be unrealistic to achieve, so you

may have to reduce them by using AccuracyGoal and/or PrecisionGoal appropriately. Sometimes, especially with the coarser grids that come with less stringent tolerances, the systems are not stiff and it is possible to use explicit methods, that avoid the numerical linear algebra, which can be problematic, especially for higher-dimensional problems. For this example, using Method -> ExplicitRungeKutta gets the solution in about half the time.

Any of the other grid options can be specified as a list giving the values for each dimension. When only a single value is given, it is used for all the spatial dimensions. The two exceptions to this are MaxPoints, where a single value is taken to be the total number of grid points in the outer product, and Coordinates, where a grid must be specified explicitly for each dimension.

This chooses parts of the grid from the previous solutions so that it is more closely spaced where the front is steeper.

In[63]:=
```
ν = 0.075;
xgrid = Join[Select[Part[u /. sol1, 3, 2], Negative],
   {0.}, Select[Part[u /. sol2, 3, 2], Positive]];
ygrid = Join[Select[Part[u /. sol2, 3, 3], Negative], {0.},
  Select[Part[u /. sol1, 3, 3], Positive]]; sol3 =
 First[NDSolve[{D[u[t, x, y], t] == ν (D[u[t, x, y], x, x] + D[u[t, x, y], y, y]) -
       u[t, x, y] (2 D[u[t, x, y], x] - D[u[t, x, y], y]), u[0, x, y] == Exp[-(x^2 + y^2)],
    u[t, -4, y] == u[t, 4, y], u[t, x, -4] == u[t, x, 4]}, u, {t, 0, 2},
   {x, -4, 4}, {y, -4, 4}, Method → {"MethodOfLines", "SpatialDiscretization" →
      {"TensorProductGrid", "Coordinates" → {xgrid, ygrid}}}]]
```

Out[65]=
```
{u → InterpolatingFunction[{{0., 2.}, {-4., 4.}, {-4., 4.}}, <>]}
```

It is important to keep in mind that the a posteriori spatial error estimates are simply estimates of the local error in computing spatial derivatives and may not reflect the actual accumulated spatial error for a given solution. One way to get an estimate on the actual spatial error is to compute the solution to very stringent tolerances in time for different spatial grids. To show how this works, consider again the simpler one-dimensional Burgers' equation.

This computes a list of solutions using $\{33, 65, \ldots, 4097\}$ spatial grid points to compute the solution to Burgers' equation for difference orders 2, 4, 6, and pseudospectral. The temporal accuracy and precision tolerances are set very high so that essentially all of the error comes from the spatial discretization. Note that by specifying $\{t, 4, 4\}$ in NDSolve, only the solution at $t = 4$ is saved. Without this precaution, some of the solutions for the finer grids (which take many more time steps) could exhaust available memory. Even so, the list of solutions takes a substantial amount of time to compute.

```
In[66]:= ν = 0.01;
         solutions = Map[Table[
               n = 2^i + 1;
               u /.
                First[NDSolve[{D[u[x, t], t] == ν D[u[x, t], x, x] - u[x, t] D[u[x, t], x],
                    u[x, 0] == Exp[-x^2], u[-5, t] == u[5, t]}, u, {x, -5, 5}, {t, 4, 4},
                  AccuracyGoal → 10, PrecisionGoal → 10, MaxSteps → Infinity,
                  Method → {"MethodOfLines", "SpatialDiscretization" →
                     {"TensorProductGrid", "DifferenceOrder" → #, AccuracyGoal → 0,
                      PrecisionGoal → 0, "MaxPoints" → n, "MinPoints" → n}}]],
               {i, 5, If[NumberQ[#], 12, 11]}
              ] &, {2, 4, 6, "Pseudospectral"}];
```

Given two solutions, a comparison needs to be done between the two. To keep out any sources of error except for that in the solutions themselves, it is best to use the data that is interpolated to make the `InterpolatingFunction`. This can be done by using points common to the two solutions.

This defines a function to estimate error by comparing two different solutions at the points common to both. The arguments *coarse* and *fine* should be the solutions on the coarser and finer grids, respectively. This works for the solutions generated earlier with grid spacing varying by powers of 2.

```
In[68]:= Clear[errfun];
         errfun[t_, coarse_InterpolatingFunction, fine_InterpolatingFunction] :=
          Module[{cgrid = InterpolatingFunctionCoordinates[coarse][[1]], c, f},
           c = coarse[cgrid, t];
           f = fine[cgrid, t];
           Norm[f - c, ∞] / Length[cgrid]]
```

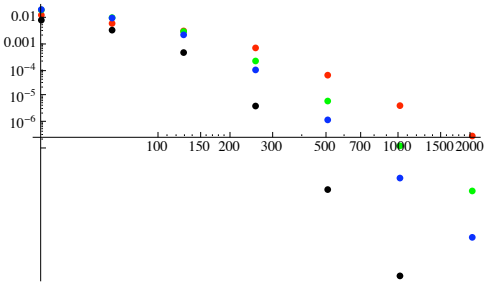
To get an indication of the general trend in error (in cases of instability, solutions do not converge, so this does not assume that), you can compare the difference of successive pairs of solutions.

This defines a function that will plot a sequence of error estimates for the successive solutions found for a given difference order and uses it to make a logarithmic plot of the estimated error as a function of the number of grid points.

```
In[69]:= Clear[errplot];
         errplot[t_, sols : {_InterpolatingFunction ..}, opts___] :=
          Module[{errs, lens},
           errs = MapThread[errfun[t, ##] &, Transpose[Partition[sols, 2, 1]]];
           lens = Map[Length, Drop[sols[[All, 3, 1]], -1]];
           ListLogLogPlot[Transpose[{lens, errs}], opts]]
```

```
In[71]:= colors = {RGBColor[1, 0, 0], RGBColor[0, 1, 0],
           RGBColor[0, 0, 1], RGBColor[0, 0, 0]}; Show[Block[{c = -1 / 3},
           MapThread[errplot[4, #1, PlotStyle → {PointSize[0.015], #2}] &,
            {solutions, colors}]], PlotRange → All]
```

Out[71]=



A logarithmic plot of the maximum spatial error in approximating the solution of Burgers' equation at $t = 4$ as a function of the number of grid points. Finite differences of order 2, 4, and 6 on a uniform grid are shown in red, green, and blue, respectively. Pseudospectral derivatives with uniform (periodic) spacing are shown in black.

The upper-left part of the plot are the grids where the profile is not adequately resolved, so differences are simply of magnitude order 1 (it would be a lot worse if there was instability). However, once there are a sufficient number of points to resolve the profile without oscillation, convergence becomes quite rapid. Not surprisingly, the slope of the logarithmic line is –4, which corresponds to the difference order `NDSolve` uses by default. If your grid is fine enough to be in the asymptotically converging part, a simpler error estimate could be effected by using Richardson extrapolation as in the previous two sections, but on the overall solution rather than the local error. On the other hand, computing more values and viewing a plot gives a better indication of whether you are in the asymptotic regime or not.

It is fairly clear from the plot that the best solution computed is the pseudospectral one with 2049 points (the one with more points was not computed because its spatial accuracy far exceeds the temporal tolerances that were set). This solution can, in effect, be treated almost as an exact solution, at least up to error tolerances of 10^{-9}or so.

To get a perspective of how best to solve the problem, it is useful to do the following: for each solution found that was at least a reasonable approximation, recompute it with the temporal accuracy tolerance set to be comparable to the possible spatial accuracy of the solution and plot the resulting accuracy as a function of solution time. The following (somewhat complicated) commands do this.

This identifies the "best" solution that will be used, in effect, as an exact solution in the computations that follow. It is dropped from the list of solutions to compare it to since the comparison would be meaningless.

```
In[72]:= best = Last[Last[solutions]];
         solutions[[-1]] = Drop[solutions[[-1]], -1];
```

This defines a function that, given a difference order, `do`, and a solution, `sol`, computed with that difference order, recomputes it with local temporal tolerance slightly more stringent than the actual spatial accuracy achieved if that accuracy is sufficient. The function output is a list of {number of grid points, difference order, time to compute in seconds, actual error of the recomputed solution}.

```
In[74]:= TimeAccuracy[do_][sol_] := Block[{tol, ag, n, solt, Second = 1},
           tol = errfun[4, sol, best];
           ag = -Log[10., tol];
           If[ag < 2,
            $Failed,
            n = Length[sol[[3, 1]]];
            secs = First[Timing[solt = First[
                  u /. NDSolve[{D[u[x, t], t] == ν D[u[x, t], x, x] - u[x, t] D[u[x, t], x],
                      u[x, 0] == Exp[-x^2], u[-5, t] == u[5, t]}, u, {x, -5, 5}, {t, 4, 4},
                    AccuracyGoal → ag + 1, PrecisionGoal → Infinity, MaxSteps → Infinity,
                    Method → {"MethodOfLines", "SpatialDiscretization" →
                        {"TensorProductGrid", "DifferenceOrder" → do, AccuracyGoal → 0,
                         PrecisionGoal → 0, "MaxPoints" → n, "MinPoints" → n}}]]]];
            {n, do, secs, errfun[4, solt, best]}
           ]
          ]
```

This applies the function to each of the previously computed solutions. (With the appropriate difference order!)

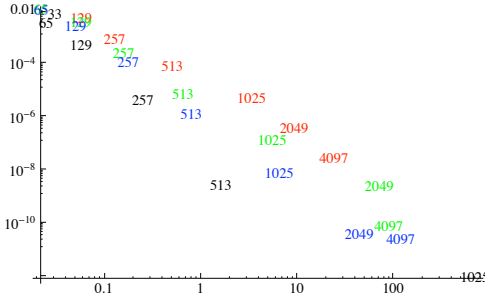
```
In[75]:= results =
          MapThread[Map[TimeAccuracy[#1], #2] &, {{2, 4, 6, "Pseudospectral"}, solutions}]
```

```
Out[75]= {{$Failed, $Failed, {129, 2, 0.06, 0.00432122}, {257, 2, 0.12, 0.000724265},
           {513, 2, 0.671, 0.0000661853}, {1025, 2, 1.903, 4.44696×10^-6},
           {2049, 2, 5.879, 3.10464×10^-7}, {4097, 2, 17.235, 2.4643×10^-8}},
          {$Failed, {65, 4, 0.02, 0.00979942}, {129, 4, 0.1, 0.00300281}, {257, 4, 0.161, 0.000213248},
           {513, 4, 1.742, 6.02345×10^-6}, {1025, 4, 5.438, 1.13695×10^-7},
           {2049, 4, 43.793, 2.10218×10^-9}, {4097, 4, 63.551, 6.48318×10^-11}},
          {$Failed, {65, 6, 0.03, 0.00853295}, {129, 6, 0.14, 0.00212781},
           {257, 6, 0.37, 0.0000935051}, {513, 6, 1.392, 1.1052×10^-6}, {1025, 6, 7.14, 6.38732×10^-9},
           {2049, 6, 35.121, 3.22349×10^-11}, {4097, 6, 89.809, 2.15934×10^-11}},
          {{33, Pseudospectral, 0.02, 0.00610004}, {65, Pseudospectral, 0.03, 0.00287949},
           {129, Pseudospectral, 0.08, 0.000417946}, {257, Pseudospectral, 0.22, 3.72935×10^-6},
           {513, Pseudospectral, 2.063, 2.28232×10^-9}, {1025, Pseudospectral, 544.974, 8.81844×10^-13}}}
```

This removes the cases that were not recomputed and makes a logarithmic plot of accuracy as a function of computation time.

```
In[76]:= fres = Map[DeleteCases[#, $Failed] &, results];
         ListLogLogPlot[fres[[All, All, {3, 4}]],
          PlotRange → All, PlotStyle → White, Epilog → MapThread[
            Function[{c, d}, {c, Apply[Text[ToString[#1], Log[{#3, #4}]] &, d, 1]}],
            {{Red, Green, Blue, Black}, fres}]]
```

Out[76]=



A logarithmic plot of the error in approximating the solution of Burgers' equation at $t = 4$ as a function of the computation time. Each point shown indicates the number of spatial grid points used to compute the solution. Finite differences of order 2, 4, and 6 on a uniform grid are shown in red, blue, and green, respectively. Pseudospectral derivatives with uniform (periodic) spacing are shown in black. Note that the cost of the pseudospectral method jumps dramatically from 513 to 1025. This is because the method has switched to the stiff solver, which is very expensive with the dense Jacobian produced by the discretization.

The resulting graph demonstrates quite forcefully that, when they work, as in this case, periodic pseudospectral approximations are incredibly efficient. Otherwise, up to a point, the higher the difference order, the better the approximation will generally be. These are all features of smooth problems, which this particular instance of Burgers' equation is. However, the higher-order solutions would generally be quite poor if you went toward the limit $\nu \to 0$.

One final point to note is that the above graph was computed using the `Automatic` method for the temporal direction. This uses LSODA, which switches between a stiff and nonstiff method depending on how the solution evolves. For the coarser grids, strictly explicit methods are typically a bit faster, and, except for the pseudospectral case, the implicit BDF methods are faster for the finer grids. A variety of alternative ODE methods are available in `NDSolve`.

Error at the Boundaries

The a priori error estimates are computed in the interior of the computational region because the differences used there all have consistent error terms that can be used to effectively estimate the number of points to use. Including the boundaries in the estimates would complicate the process beyond what is justified for such an a priori estimate. Typically, this approach is successful in keeping the error under reasonable control. However, there are a few cases which can lead to difficulties.

Occasionally it may occur that because the error terms are larger for the one-sided derivatives used at the boundary, `NDSolve` will detect an inconsistency between boundary and initial conditions, which is an artifact of the discretization error.

This solves the one-dimensional heat equation with the left end held at constant temperature and the right end radiating into free space.

```
In[2]:= solution = First[NDSolve[{∂t u[x, t] == ∂x,x u[x, t], u[x, 0] == 1 - Sin[4 π x]/(4 π),
           u[0, t] == 1, u[1, t] + u^(1,0)[1, t] == 0}, u, {x, 0, 1}, {t, 0, 1}]]

NDSolve::ibcinc: Warning: Boundary and initial conditions are inconsistent.

Out[2]= {u → InterpolatingFunction[{{0., 1.}, {0., 1.}}, <>]}
```

The `NDSolve:ibcinc` message is issued, in this case, completely to the larger discretization error at the right boundary. For this particular example, the extra error is not a problem because it gets damped out due to the nature of the equation. However, it is possible to eliminate the message by using just a few more spatial points.

This computes the solution to the same equation as above, but using a minimum of 50 points in the x direction.

```
In[3]:= solution =
          First[NDSolve[{∂t u[x, t] == ∂x,x u[x, t], u[x, 0] == 1 - Sin[4 π x]/(4 π), u[0, t] == 1,
            u[1, t] + u^(1,0)[1, t] == 0}, u, {x, 0, 1}, {t, 0, 1}, Method → {"MethodOfLines",
              "SpatialDiscretization" → {"TensorProductGrid", MinPoints → 50}}]]

Out[3]= {u → InterpolatingFunction[{{0., 1.}, {0., 1.}}, <>]}
```

One other case where error problems at the boundary can affect the discretization unexpectedly is when periodic boundary conditions are given with a function that is not truly periodic, so that an unintended discontinuity is introduced into the computation.

This begins the computation of the solution to the sine-Gordon equation with a Gaussian initial condition and periodic boundary conditions. The NDSolve command is wrapped with TimeConstrained since solving the given problem can take a very long time and a large amount of system memory.

In[4]:=
```
L = 1;
TimeConstrained[
 sol1 = First[NDSolve[{D[u[t, x], t, t] == D[u[t, x], x, x] - Sin[u[t, x]],
     u[0, x] == Exp[-x^2], Derivative[1, 0][u][0, x] == 0, u[t, -1] == u[t, 1]},
    u, {t, 0, 1}, {x, -1, 1}, Method → StiffnessSwitching]], 10]
```

NDSolve::mxsst: Using maximum number of grid points 10000 allowed by the MaxPoints or MinStepSize options for independent variable x.

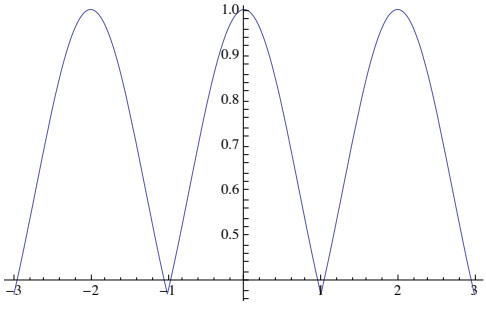
Out[5]= $Aborted

The problem here is that the initial condition is effectively discontinuous when the periodic continuation is taken into account.

This shows a plot of the initial condition over the extent of three full periods.

In[6]:=
```
Plot[Exp[-(Mod[x + 1, 2] - 1)^2], {x, -3, 3}]
```

Out[6]=



Since there is always a large derivative error at the cusps, NDSolve is forced to use the maximum number of points in an attempt to satisfy the a priori error bound. To make matters worse, the extreme change makes solving the resulting ODEs more difficult, leading to a very long solution time which uses a lot of memory.

If the discontinuity is really intended, you will typically want to specify a number of points or spacing for the spatial grid that will be sufficient to handle the aspects of the discontinuity you are interested in. To model discontinuities with high accuracy will typically take specialized methods that are beyond the scope of the general methods that NDSolve provides.

On the other hand, if the discontinuity was unintended, say in this example by simply choosing a computational domain that was too small, it can usually be fixed easily enough by extending the domain or by adding in terms to smooth things between periods.

This solves the sine-Gordon problem on a computational domain large enough so that the discontinuity in the initial condition is negligible compared to the error allowed by the default tolerances.

```
In[7]:= L = 10;
        Timing[sol2 = First[NDSolve[{D[u[t, x], t, t] == D[u[t, x], x, x] - Sin[u[t, x]],
              u[0, x] == Exp[-x^2], Derivative[1, 0][u][0, x] == 0,
              u[t, -L] == u[t, L]}, u, {t, 0, 1}, {x, -L, L}]]]
Out[8]= {0.031, {u → InterpolatingFunction[{{0., 1.}, {-10., 10.}}, <>]}}
```

Numerical Solution of Boundary Value Problems

"Shooting" Method

The shooting method works by considering the boundary conditions as a multivariate function of initial conditions at some point, reducing the boundary value problem to finding the initial conditions that give a root. The advantage of the shooting method is that it takes advantage of the speed and adaptivity of methods for initial value problems. The disadvantage of the method is that it is not as robust as finite difference or collocation methods: some initial value problems with growing modes are inherently unstable even though the BVP itself may be quite well posed and stable.

Consider the BVP system

$$X'(t) = F(t, X(t)); G(X(t_1), X(t_2), \ldots, X(t_n)) = 0, t_1 < t_2 < \ldots < t_n$$

The shooting method looks for initial conditions $X(t_0) = c$ so that $G = 0$. Since you are varying the initial conditions, it makes sense to think of $X = X_c$ as a function of them, so shooting can be thought of as finding c such that with

$$X_c'(t) = F(t, X_c(t)); X_c(t_0) = c$$

$$G(X_c(t_1), X_c(t_2), \ldots, X_c(t_n)) = 0$$

After setting up the function for G, the problem is effectively passed to `FindRoot` to find the initial conditions c giving the root. The default method is to use Newton's method, which

involves computing the Jacobian. While the Jacobian can be computed using finite differences, the sensitivity of solutions of an IVP to its initial conditions may be too much to get reasonably accurate derivative values, so it is advantageous to compute the Jacobian as a solution to ODEs.

Linearization and Newton's Method

Linear problems can be described by

$$X_c'(t) = J(t)\, X_c(t) + F_0(t); X_c(t_0) = c$$

$$G(X_c(t_1), X_c(t_2), \ldots, X_c(t_n)) = B_0 + B_1\, X_c(t_1) + B_2\, X_c(t_2) + \ldots B_n\, X_c(t_n)$$

Where $J(t)$ is a matrix and $F_0(t)$ is a vector both possibly depending on t, B_0 is a constant vector, and $B_1, B_2, \ldots, B_n$ are constant matrices.

Let $Y = \frac{\partial X_c(t)}{\partial c}$. Then, differentiating both the IVP and boundary conditions with respect to c gives

$$Y'(t) = J(t)\, Y(t)); Y(t_0) = I$$

$$\frac{\partial G}{\partial c} = B_1\, Y(t_1) + B_2\, Y(t_2) + \ldots B_n\, Y(t_n) = 0$$

Since G is linear, when thought of as a function of c, you have $G(c) = G(c_0) + \left(\frac{\partial G}{\partial c}\right)(c - c_0)$, so the value of c for which $G(c) = 0$ satisfies

$$c = c_0 + \left(\frac{\partial G}{\partial c}\right)^{-1} G(c_0)$$

for any particular initial condition c_0.

For nonlinear problems, let $J(t)$ be the Jacobian for the nonlinear ODE system, and let B_i be the Jacobian of the i^{th} boundary condition. Then computation of $\frac{\partial G}{\partial c}$ for the linearized system gives the Jacobian for the nonlinear system for a particular initial condition, leading to a Newton iteration,

$$c_{n+1} = c_n + \left(\frac{\partial G}{\partial c}(c_n)\right)^{-1} G(c_n)$$

"StartingInitialConditions"

For boundary value problems, there is no guarantee of uniqueness as there is in the initial value problem case. "Shooting" will find only one solution. Just as you can affect the particular solution FindRoot gets for a system of nonlinear algebraic equations by changing the starting values, you can change the solution that "Shooting" finds by giving different initial conditions to start the iterations from.

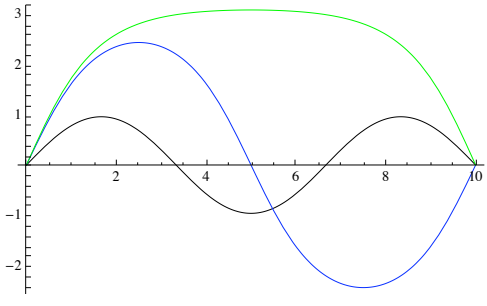
"StartingInitialConditions" is an option of the "Shooting" method that allows you to specify the values and position of the initial conditions to start the shooting process from.

The shooting method by default starts with zero initial conditions so that if there is a zero solution, it will be returned.

This computes a very simple solution to the boundary value problem
$x'' + \sin(x) == 0$ with $x(0) = x(1) = 0$.

In[105]:=
```
sols =
  Map[First[NDSolve[{x''[t] + Sin[x[t]] == 0, x[0] == x[10] == 0}, x, t, Method →
        {"Shooting", "StartingInitialConditions" → {x[0] == 0, x'[0] == #}}]] &,
    {1.5, 1.75, 2}];
Plot[Evaluate[x[t] /. sols], {t, 0, 10}, PlotStyle → {Black, Blue, Green}]
```

Out[106]=



By default, "Shooting" starts from the left side of the interval and shoots forward in time. There are cases where is it advantageous to go backwards, or even from a point somewhere in the middle of the interval.

Consider the linear boundary value problem

$$x'''(t) - 2\lambda x''(t) - \lambda^2 x'(t) + 2\lambda^3 x(t) = \left(\lambda^2 + \pi^2\right)(2\lambda\cos(\pi t) + \pi\sin(\pi t))$$

$$x(0) = 1 + \frac{1 + e^{-2\lambda} + e^{-\lambda}}{2 + e^{-\lambda}},\ x(1) = 0,\ x'(1) = \frac{3\lambda - e^{-\lambda}\lambda}{2 + e^{-\lambda}}$$

that has a solution

$$x(t) = \frac{e^{\lambda(t-1)} + e^{2\lambda(t-1)} + e^{-\lambda t}}{2 + e^{-\lambda}} + \cos(\pi t)$$

For moderate values of λ, the initial value problem starting at $t = 0$ becomes unstable because of the growing $e^{\lambda(t-1)}$ and $e^{2\lambda(t-1)}$ terms. Similarly, starting at $t = 1$, instability arises from the $e^{-\lambda t}$ term, though this is not as large as the term in the forward direction. Beyond some value of λ, shooting will not be able to get a good solution because the sensitivity in either direction will be too great. However, up to that point, choosing a point in the interval that balances the growth in the two directions will give the best solution.

This gives the equation, boundary conditions, and exact solution as *Mathematica* input.

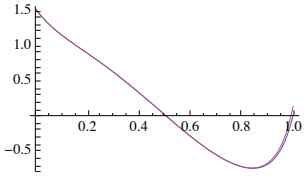
```
In[107]:= eqn =
            x'''[t] - 2 λ x''[t] - λ^2 x'[t] + 2 λ^3 x[t] == (λ^2 + π^2) (2 λ Cos[π t] + π Sin[π t]);
          bcs = {x[0] == 1 + (1 + E^(-2 λ) + E^-λ)/(2 + E^-λ), x[1] == 0, x'[1] == (3 λ - E^-λ λ)/(2 + E^-λ)};
          xsol[t_] = (E^(λ (t-1)) + E^(2 λ (t-1)) + E^(-λ t))/(2 + E^-λ) + Cos[π t];
```

This solves the system with $\lambda = 10$ shooting from the default $t = 0$.

```
In[110]:= Block[{λ = 10},
           sol = First[NDSolve[{eqn, bcs}, x, t]];
           Plot[{xsol[t], x[t] /. sol}, {t, 0, 1}]]
```

```
NDSolve::bvluc:
  The equations derived from the boundary conditions are numerically ill-conditioned. The boundary
    conditions may not be sufficient to uniquely define a solution. The
    computed solution may match the boundary conditions poorly.

NDSolve::berr:
  There are significant errors {-1.11022×10^-16, 6.95123×10^-6, 0.000139029} in the boundary
    value residuals.  Returning the best solution found.
```

Out[110]=



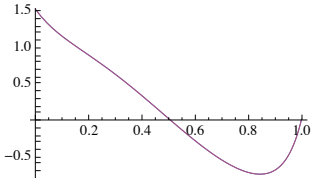
Shooting from $t = 0$, the "Shooting" method gives warnings about an ill-conditioned matrix, and further that the boundary conditions are not satisfied as well as they should be. This is because a small error at $t = 0$ is amplified by $e^{20} \simeq 4 \times 10^8$. Since the reciprocal of this is of the same order

of magnitude as the local truncation error, visible errors as those seen in the plot are not surprising. In the reverse direction, the magnification will be much less: $e^{10} \simeq 2 \times 10^4$, so the solution should be much better.

This computes the solution using shooting from $t = 1$.

```
In[111]:= Block[{λ = 10},
           sol = First[NDSolve[{eqn, bcs}, x, t,
              Method → {"Shooting", "StartingInitialConditions" →
                 {x[1] == 0, x'[1] == (3 λ - e^-λ λ)/(2 + e^-λ), x''[1] == 0}}]];
           Plot[{xsol[t], x[t] /. sol}, {t, 0, 1}]]
```

Out[111]=

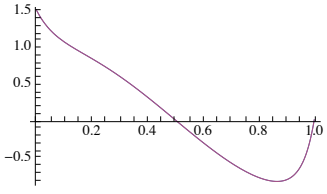


A good point to choose is actually one that will balance the sensitivity in each direction, which is about at $t = 2/3$. With this, the error with $\lambda = 15$ will still be under reasonable control.

This computes the solution for $\lambda = 15$ shooting from $t = 2/3$.

```
In[112]:= Block[{λ = 15},
           sol = First[NDSolve[{eqn, bcs}, x, t,
              Method → {"Shooting", "StartingInitialConditions" →
                 {x[2 / 3] == 0, x'[2 / 3] == 0, x''[2 / 3] == 0}}]];
           Plot[{xsol[t], x[t] /. sol}, {t, 0, 1}]]
```

Out[112]=



Option summary

option name	default value	
"StartingInitialConditions"	Automatic	the initial conditions to initiate the shooting method from
"ImplicitSolver"	Automatic	the method to use for solving the implicit equation defined by the boundary conditions; this should be an acceptable value for the Method option of FindRoot
"MaxIterations"	Automatic	how many iterations to use for the implicit solver method
"Method"	Automatic	the method to use for integrating the system of ODEs

"Shooting" method options.

"Chasing" Method

The method of chasing came from a manuscript of Gel'fand and Lokutsiyevskii first published in English in [BZ65] and further described in [Na79]. The idea is to establish a set of auxiliary problems that can be solved to find initial conditions at one of the boundaries. Once the initial conditions are determined, the usual methods for solving initial value problems can be applied. The chasing method is, in effect, a shooting method that uses the linearity of the problem to good advantage.

Consider the linear ODE

$$X'(t) == A(t)\,X(t) + A_0(t) \tag{2}$$

where $X(t) = (x_1(t), x_2(t), \ldots, x_n(t))$, $A(t)$ is the coefficient matrix, and $A_0(t)$ is the inhomogeneous coefficient vector, with n linear boundary conditions

$$B_i.X(t_i) == b_{i0},\ i = 1, 2, \cdots, n \tag{3}$$

where $B_i = (b_{i1}, b_{i2}, \cdots, b_{in})$ is a coefficient vector. From this, construct the augmented homogenous system

$$\overline{X}'(t) = \overline{A}(t)\,\overline{X}(t), \qquad \overline{B}_i.\overline{X}(t_i) = 0 \tag{4}$$

where

$$\overline{X}(t) = \begin{pmatrix} 1 \\ x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{pmatrix}, \quad \overline{A}(t) = \begin{pmatrix} a_{01}(t) & a_{11}(t) & a_{12}(t) & \cdots & a_{1n}(t) \\ a_{02}(t) & a_{21}(t) & a_{22}(t) & \cdots & a_{2n}(t) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0n}(t) & a_{n1}(t) & a_{n2}(t) & \cdots & a_{nn}(t) \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}, \text{ and } \quad \overline{B}_i = \begin{pmatrix} b_{i0} \\ b_{i1} \\ b_{i2} \\ \vdots \\ b_{in} \end{pmatrix}$$

The chasing method amounts to finding a vector function $\Phi_i(t)$ such that $\Phi_i(t_i) = \overline{B_i}$ and $\Phi_i(t)\,\overline{X}(t) = 0$. Once the function $\Phi_i(t)$ is known, if there is a full set of boundary conditions, solving

$$\begin{pmatrix} \Phi_1(t_0) \\ \Phi_2(t_0) \\ \vdots \\ \Phi_n(t_0) \end{pmatrix} \overline{X}(t_0) = 0 \tag{5}$$

can be used to determine initial conditions $(x_1(t_0), x_2(t_0), \cdots, x_n(t_0))$ that can be used with the usual initial value problem solvers. Note that the solution to system (3) is nontrivial because the first component of X is always 1.

Thus, solving the boundary value problem is reduced to solving the auxiliary problems for the $\Phi_i(t)$. Differentiating the equation for $\Phi_i(t)$ gives

$$\Phi_i(t)\,\overline{X}'(t) + \overline{X}(t)\,\Phi_i'(t) = 0$$

Substituting the differential equation,

$$\overline{A}(t)\,\overline{X}(t)\,\Phi_i(t) + \overline{X}(t)\,\Phi_i'(t) = 0$$

and transposing

$$\overline{X}(t)\left(\Phi_i'(t) + \overline{A}^T(t)\,\Phi_i(t)\right) = 0$$

Since this should hold for all solutions $\overline{X}$, you have the initial value problem for Φ_i,

$$\Phi_i'(t) + \overline{A}^T(t)\,\Phi_i(t) = 0 \quad \text{with initial condition} \quad \Phi_i(t_i) = B_i \tag{6}$$

Given t_0 where you want to have solutions to all of the boundary value problems, *Mathematica* just uses `NDSolve` to solve the auxiliary problems for $\Phi_1, \Phi_2, \ldots, \Phi_m$ by integrating them to t_0. The

results are then combined into the matrix of (3) that is solved for $X(t_0)$ to obtain the initial value problem that `NDSolve` integrates to give the returned solution.

This variant of the method is further described in and used by the *MathSource* package [R98], which also allows you to solve linear eigenvalue problems.

There is an alternative, nonlinear way to set up the auxiliary problems that is closer to the original method proposed by Gel'fand and Lokutsiyevskii. Assume that the boundary conditions are linearly independent (if not, then the problem is insufficiently specified). Then in each B_i, there is at least one nonzero component. Without loss of generality, assume that $b_{ij} \neq 0$. Now solve for Φ_{ij} in terms of the other components of Φ_i, $\Phi_{ij} = \tilde{B}_i.\tilde{\Phi}_i$, where $\tilde{\Phi}_i = \left(1, \Phi_{i1}, \cdots, \Phi_{ij-1}, \cdots, \Phi_{ij+1}, \cdots, \Phi_{in}\right)$ and $\tilde{B}_i = (b_{i0,}\ b_{i1}, \cdots, b_{ij-1}, \cdots, b_{ij+1}, \cdots, b_{in})\big/ -b_{ij}$. Using (5) and replacing Φ_{ij}, and thinking of $x_n(t)$ in terms of the other components of $x(t)$ you get the nonlinear equation

$$\tilde{\Phi}_i{}'(t) = -\tilde{A}^T[t]\,\tilde{\Phi}_i(t) + \left(A_j.\tilde{\Phi}_i(t)\right)\tilde{\Phi}_i(t)$$

where $\tilde{A}$ is A with the j^{th}column removed and A_j is the j^{th} column of A. The nonlinear method can be more numerically stable than the linear method, but it has the disadvantage that integration along the real line may lead to singularities. This problem can be eliminated by integrating on a contour in the complex plane. However, the integration in the complex plane typically has more numerical error than a simple integration along the real line, so in practice, the nonlinear method does not typically give results better than the linear method. For this reason, and because it is also generally faster, the default for *Mathematica* is to use the linear method.

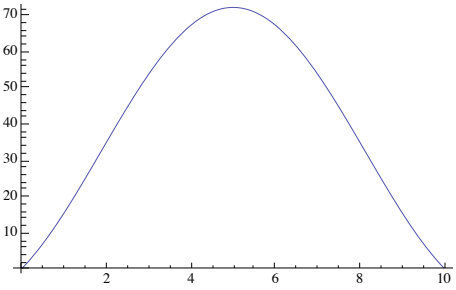
This solves a two-point boundary value problem for a second-order equation.

```
In[113]:= nsol1 = NDSolve[{y''[t] + y[t] / 4 == 8, y[0] == 0, y[10] == 0}, y, {t, 0, 10}]
Out[113]= {{y → InterpolatingFunction[{{0., 10.}}, <>]}}
```

This shows a plot of the solution.

```
In[114]:= Plot[First[y[t] /. nsol1], {t, 0, 10}]
```

Out[114]=



The solver can solve multipoint boundary value problems of linear systems of equations. (Note that each boundary equation must be at one specific value of *t*.)

```
In[115]:= bconds = {
            x[0] + x'[0] + y[0] + y'[0] == 1,
            x[1] + 2 x'[1] + 3 y[1] + 4 y'[1] == 5,
            y[2] + 2 y'[2] == 4,
            x[3] - x'[3] == 7};
          nsol2 = NDSolve[{
            x''[t] + x[t] + y[t] == t, y''[t] + y[t] == Cos[t],
            bconds},
           {x, y},
           {t, 0, 4}]
```

```
Out[116]= {{x → InterpolatingFunction[{{0., 4.}}, <>], y → InterpolatingFunction[{{0., 4.}}, <>]}}
```

In general, you cannot expect the boundary value equations to be satisfied to the close tolerance of `Equal`.

This checks to see if the boundary conditions are "satisfied".

```
In[117]:= bconds /. First[nsol2]
```

```
Out[117]= {True, False, False, False}
```

They are typically only be satisfied at most tolerances that come from the `AccuracyGoal` and `PrecisionGoal` options of `NDSolve`. Usually, the actual accuracy and precision is less because these are used for local, not global, error control.

This checks the residual error at each of the boundary conditions.

```
In[118]:= Apply[Subtract, bconds, 1] /. First[nsol2]
```

```
Out[118]= {0., -2.5751×10^-7, -4.13357×10^-8, -2.95508×10^-8}
```

When you give `NDSolve` a problem that has no solution, numerical error may make it appear to be a solvable problem. Typically, `NDSolve` will issue a warning message.

This is a boundary value problem that has no solution.

```
In[125]:= NDSolve[{x''[t] + x[t] == 0, x[0] == 1, x[Pi] == 0},
           x, {t, 0, Pi}, Method → "Chasing"]
```

```
NDSolve::bvluc:
  The equations derived from the boundary conditions are numerically ill-conditioned. The boundary
    conditions may not be sufficient to uniquely define a solution. The
    computed solution may match the boundary conditions poorly.
```

```
Out[125]= {{x → InterpolatingFunction[{{0., 3.14159}}, <>]}}
```

In this case, it is not able to integrate over the entire interval because of nonexistence.

Another situation in which the equations can be ill-conditioned is when the boundary conditions do not give a unique solution.

Here is a boundary value problem that does not have a unique solution. Its general solution is shown as computed symbolically with `DSolve`.

```
In[120]:= dsol =
           First[x /. DSolve[{x''[t] + x[t] == t, x'[0] == 1, x[Pi / 2] == Pi / 2}, x, t]]
```

```
DSolve::bvsing:
  Unable to resolve some of the arbitrary constants in the general solution using the given boundary
    conditions. It is possible that some of the conditions
    have been specified at a singular point for the equation.
```

```
Out[120]= Function[{t}, t + C[1] Cos[t]]
```

`NDSolve` issues a warning message because the matrix to solve for the initial conditions is singular, but has a solution.

```
In[122]:= onesol = First[x /. NDSolve[{x''[t] + x[t] == t, x'[0] == 1, x[Pi / 2] == Pi / 2},
             x, {t, 0, Pi / 2}, Method → "Chasing"]]
```

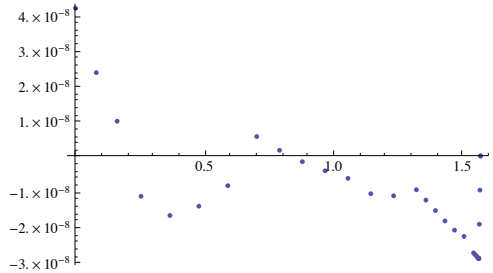
```
NDSolve::bvluc:
  The equations derived from the boundary conditions are numerically ill-conditioned. The boundary
    conditions may not be sufficient to uniquely define a solution. The
    computed solution may match the boundary conditions poorly.
```

```
Out[122]= InterpolatingFunction[{{0., 1.5708}}, <>]
```

You can identify which solution it found by fitting it to the interpolating points. This makes a plot of the error relative to the actual best fit solution.

In[126]:=
```
ip = onesol@"Coordinates"[1];
points = Transpose[{ip, onesol[ip]}];
model = dsol[t] /. C[1] → α;
fit = FindFit[points, model, α, t];
ListPlot[Transpose[{ip, onesol[ip] - ((model /. fit) /. t → ip)}]]
```

Out[130]=



Typically the default values *Mathematica* uses work fine, but you can control the chasing method by giving `NDSolve` the option `Method -> {"Chasing", `*chasing options*`}`. The possible *chasing options* are shown in the following table.

option name	*default value*	
`Method`	`Automatic`	the numerical method to use for computing the initial value problems generated by the chasing algorithm
`"ExtraPrecision"`	`0`	number of digits of extra precision to use for solving the auxiliary initial value problems
`"ChasingType"`	`"LinearChasing"`	the type of chasing to use, which can be either `"LinearChasing"` or `"NonlinearChasing"`

Options for the `"Chasing"` method of `NDSolve`.

The method "ChasingType" -> "NonlinearChasing" itself has two options.

option name	*default value*	
"ContourType"	Ellipse	the shape of contour to use when integration in the complex plane is required, which can either be "Ellipse" or "Rectangle"
"ContourRatio"	1/10	the ratio of the width to the length of the contour; typically a smaller number gives more accurate results, but yields more numerical difficulty in solving the equations

Options for the "NonlinearChasing" option of the "Chasing" method.

These options, especially "ExtraPrecision" can be useful in cases where there is a strong sensitivity to computed initial conditions.

Here is a boundary value problem with a simple solution computed symbolically using DSolve.

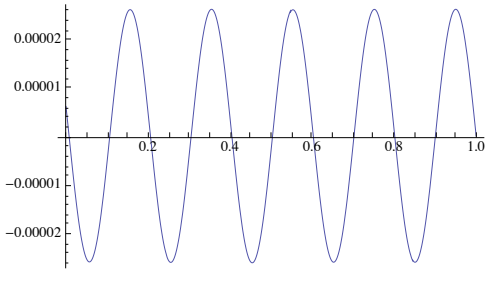
```
In[131]:= bvp = {x''[t] + 1000 x[t] == 0, x[0] == 0, x[1] == 1};
          dsol = First[x /. DSolve[bvp, x, t]]
```

```
Out[132]= Function[{t}, Csc[10 Sqrt[10]] Sin[10 Sqrt[10] t]]
```

This shows the error in the solution computed using the chasing method in NDSolve.

```
In[133]:= sol = First[x /. NDSolve[{x''[t] + 1000 x[t] == 0, x[0] == 0, x[1] == 1},
                x, {t, 0, 1}, Method → "Chasing"]];
          Plot[sol[t] - dsol[t], {t, 0, 1}]
```

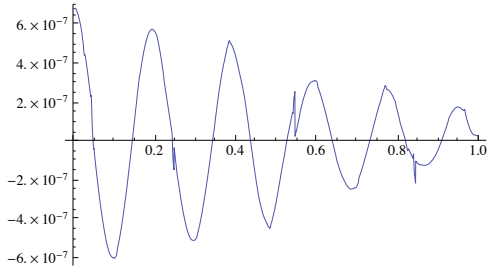
Out[134]=



Using extra precision to solve for the initial conditions reduces the error substantially.

In[135]:=
```
sol = First[x /. NDSolve[{x''[t] + 1000 x[t] == 0, x[0] == 0, x[1] == 1},
      x, {t, 0, 1}, Method → {"Chasing", "ExtraPrecision" → 10}]];
Plot[sol[t] - dsol[t], {t, 0, 1}]
```

Out[136]=

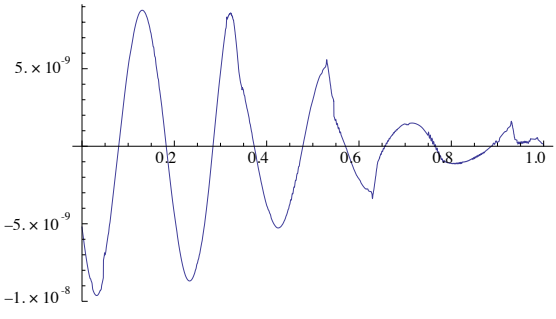


Increasing the extra precision beyond this really will not help because a significant part of the error results from computing the solution once the initial conditions are found. To reduce this, you need to give more stringent `AccuracyGoal` and `PrecisionGoal` options to `NDSolve`.

This uses extra precision to compute the initial conditions along with more stringent settings for the `AccuracyGoal` and `PrecisionGoal` options.

In[137]:=
```
sol = First[x /. NDSolve[{x''[t] + 1000 x[t] == 0, x[0] == 0, x[1] == 1},
      x, {t, 0, 1}, Method → {"Chasing", "ExtraPrecision" → 10},
      AccuracyGoal → 10, PrecisionGoal → 10]];
Plot[sol[t] - dsol[t], {t, 0, 1}]
```

Out[138]=



Boundary Value Problems with Parameters

In many of the applications where boundary value problems arise, there may be undetermined parameters, such as eigenvalues, in the problem itself that may be a part of the desired solution. By introducing the parameters as dependent variables, the problem can often be written as a boundary value problem in standard form.

For example, the flow in a channel can be modeled by

$$f''' - R\left((f')^2 - ff''\right) + \mathrm{Ra} = 0$$

$$f(0) = f'(0) = 0,\ f(1) = 1,\ f'(1) = 0$$

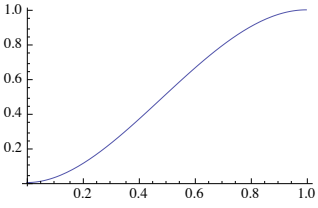
where R (the Reynolds number) is given, but a is to be determined.

To find the solution f and the value of a, just add the equation $a' = 0$.

This solves the flow problem with $R = 1$ for f and a, plots the solution f and returns the value of a.

In[1]:=
```
Block[{R = 1},
 sol = NDSolve[{f'''[t] - R ((f'[t])^2 - f[t] f''[t]) + R a[t] == 0, a'[t] == 0,
    f[0] == f'[0] == f'[1] == 0, f[1] == 1}, {f, a}, t];
 Column[{Plot[f[t] /. First[sol], {t, 0, 1}],
   a[0] /. First[sol]}]]
```

Out[1]=




```
14.3659
```

Numerical Solution of Differential-Algebraic Equations

Introduction

In general, a system of ordinary differential equations (ODEs) can be expressed in the normal form,

$$x' = f(t, x).$$

The derivatives of the dependent variables x are expressed explicitly in terms of the independent variable t and the dependent variables x. As long as the function f has sufficient continuity, a unique solution can always be found for an initial value problem where the values of the dependent variables are given at a specific value of the independent variable.

With differential-algebraic equations (DAEs), the derivatives are not, in general, expressed explicitly. In fact, derivatives of some of the dependent variables typically do not appear in the equations. The general form of a system of DAEs is

$$F(t, x, x') = 0, \tag{7}$$

where the Jacobian with respect to x', $\partial F/\partial x'$ may be singular.

A system of DAEs can be converted to a system of ODEs by differentiating it with respect to the independent variable t. The *index* of a DAE is effectively the number of times you need to differentiate the DAEs to get a system of ODEs. Even though the differentiation is possible, it is not generally used as a computational technique because properties of the original DAEs are often lost in numerical simulations of the differentiated equations.

Thus, numerical methods for DAEs are designed to work with the general form of a system of DAEs. The methods in `NDSolve` are designed to generally solve index-1 DAEs, but may work for higher index problems as well.

This tutorial will show numerous examples that illustrate some of the differences between solving DAEs and ODEs.

This loads packages that will be used in the examples and turns off a message.

```
In[10]:= Needs["DifferentialEquations`InterpolatingFunctionAnatomy`"];
```

The specification of initial conditions is quite different for DAEs than for ODEs. For ODEs, as already mentioned, a set of initial conditions uniquely determines a solution. For DAEs, the situation is not nearly so simple; it may even be difficult to find initial conditions that satisfy the equations at all. To better understand this issue, consider the following example [AP98].

Here is a system of DAEs with three equations, but only one differential term.

```
In[11]:= DAE = ( x1'[t] == x3[t]
                 x2[t] (1 - x2[t]) == 0
                 x1[t] x2[t] + x3[t] (1 - x2[t]) == t );
```

The initial conditions are clearly not free; the second equation requires that $x_2[t_0]$ be either 0 or 1.

This solves the system of DAEs starting with a specified initial condition for the derivative of x_1.

```
In[12]:= sol = NDSolve[{DAE, x1'[0] == 1}, {x1, x2, x3}, {t, 0, 1}]

Out[12]= {{x1 → InterpolatingFunction[{{0., 1.}}, <>],
           x2 → InterpolatingFunction[{{0., 1.}}, <>], x3 → InterpolatingFunction[{{0., 1.}}, <>]}}
```

To get this solution, NDSolve first searches for initial conditions that satisfy the equations, using a combination of Solve and a procedure much like FindRoot. Once consistent initial conditions are found, the DAE is solved using the IDA method.

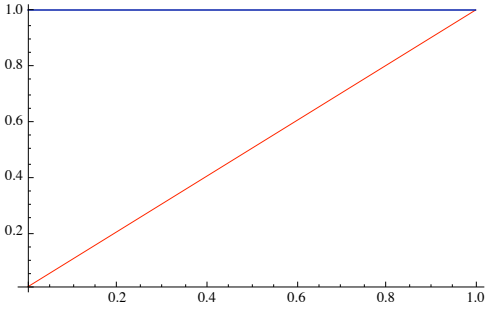
This shows the initial conditions found by NDSolve.

```
In[13]:= {{x1'[0]}, {x1[0], x2[0], x3[0]}} /. First[sol]
Out[13]= {{1.}, {0., 1., 1.}}
```

This shows a plot of the solution. The solution $x_2[0]$ is obscured by the solution $x_3[0]$, which has the same constant value of 1.

```
In[15]:= Plot[Evaluate[{x1[t], x2[t], x3[t]} /. First[sol]], {t, 0, 1},
          PlotStyle → {Red, Black, Blue}]
```

Out[15]=



However, there may not be a solution from all initial conditions that satisfy the equations.

This tries to find a solution with $x_2[0]$ starting from steady state with derivative 0.

```
In[16]:= sols = NDSolve[{DAE, x1'[0] == 0}, {x1, x2, x3}, {t, 0, 1}]

NDSolve::nderr: Error test failure at t == 0.`; unable to continue.

Out[16]= {{x1 → InterpolatingFunction[{{0., 0.}}, <>],
  x2 → InterpolatingFunction[{{0., 0.}}, <>], x3 → InterpolatingFunction[{{0., 0.}}, <>]}}
```

This shows the initial conditions found by NDSolve.

```
In[17]:= {{x1'[0]}, {x1[0], x2[0], x3[0]}} /. First[sols]
Out[17]= {{0.}, {0., 1., 0.}}
```

If you look at the equations with x_2 set to 1, you can see why it is not possible to advance beyond $t == 1$.

Substitute $x_2[t] = 1$ into the equations.

```
In[18]:= DAE /. x2[t] → 1
Out[18]= {{x1'[t] == x3[t]}, {True}, {x1[t] == t}}
```

The middle equation effectively drops out. If you differentiate the last equation with $x_2[t] = 1$, you get the condition $x_1'[t] = 1$, but then the first equation is inconsistent with the value of $x_3[t] = 0$ in the initial conditions.

It turns out that the only solution with $x_2[t] = 1$ is $\{x_2[t] = t, x_2[t] = 1, x_3[t] = 1\}$, and along this solution, the system has index 2.

The other set of solutions for the problem is when $x_2[t] = 0$. You can find these by specifying that as an initial condition.

This finds a solution with $x_2[t] = 0$. It is also necessary to specify a value for $x_1[0]$ because it is a differential variable.

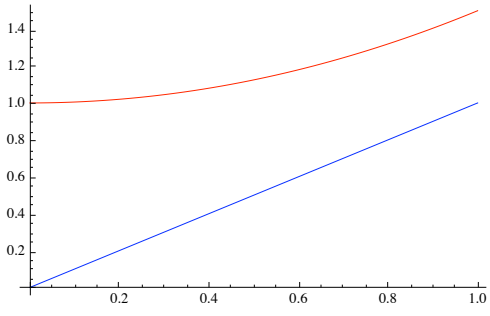
```
In[19]:= sol0 = NDSolve[{DAE, x1[0] == 1, x2[0] == 0}, {x1, x2, x3}, {t, 0, 1}]
```

```
Out[19]= {{x1 → InterpolatingFunction[{{0., 1.}}, <>],
   x2 → InterpolatingFunction[{{0., 1.}}, <>], x3 → InterpolatingFunction[{{0., 1.}}, <>]}}
```

This shows a plot of the nonzero components of the solution.

```
In[21]:= Plot[Evaluate[{x1[t], x3[t]} /. First[sol0]], {t, 0, 1},
  PlotStyle → {Red, Blue}]
```

Out[21]=



In general, you must specify initial conditions for the differential variables because typically there is a parametrized general solution. For this problem with $x_2[t] = 0$, the general solution is $\{x_1[t] = x_1[0] + t^2/2, x_2[t] = 0, x_3[t] == t\}$, so it is necessary to give $x_1[0]$ to determine the solution.

`NDSolve` cannot always find initial conditions consistent with the equations because sometimes this is a difficult problem. "Often the most difficult part of solving a DAE system in applications is to determine a consistent set of initial conditions with which to start the computation". [BCP89]

NDSolve fails to find a consistent initial condition.

```
In[22]:= NDSolve[{DAE, x1[0] == 1}, {x1, x2, x3}, {t, 0, 1}]

NDSolve::icfail:
  Unable to find initial conditions that satisfy the residual function within specified tolerances.
    Try giving initial conditions for both values and derivatives of the functions.

Out[22]= {}
```

If NDSolve fails to find consistent initial conditions, you can use FindRoot with a good starting value or some other procedure to obtain consistent initial conditions and supply them. If you know values close to a good starting guess, NDSolve uses these values to start its search, which may help. You may specify values of the dependent variables and their derivatives.

With index-1 systems of DAEs, it is often possible to differentiate and use an ODE solver to get the solution.

Here is the Robertson chemical kinetics problem. Because of the large and small rate constants, the problem is quite stiff.

```
In[23]:= kinetics =
           {y1'[t] == -1/25 y1[t] + 10^4 y2[t] y3[t], y2'[t] == 1/25 y1[t] - 3 × 10^7 y2[t]^2};
         balance = y1[t] + y2[t] + y3[t] == 1;
         start = {y1[0] == 1, y2[0] == 0, y3[0] == 0};
```

This solves the Robertson kinetics problem as an ODE by differentiating the balance equation.

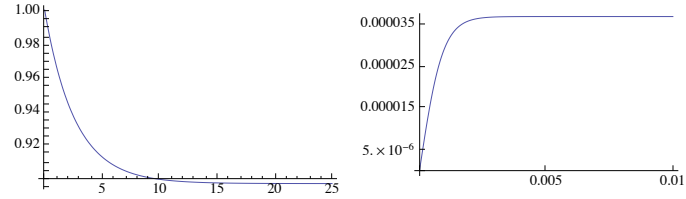
```
In[26]:= odesol =
          First[NDSolve[{kinetics, D[balance, t], start}, {y1, y2, y3}, {t, 0, 40000}]]

Out[26]= {y1 → InterpolatingFunction[{{0., 40000.}}, <>],
          y2 → InterpolatingFunction[{{0., 40000.}}, <>], y3 → InterpolatingFunction[{{0., 40000.}}, <>]}
```

The stiffness of the problem is supported by y_1 and y_2 having their main variation on two completely different time scales.

This shows the solutions y_1 and y_2.

In[27]:=
```
GraphicsRow[{
  Plot[y₁[t] /. odesol, {t, 0, 25}, PlotRange → All, ImageSize → 200],
  Plot[y₂[t] /. odesol, {t, 0, 0.01}, PlotRange → All, ImageSize → 200,
   Ticks -> {{0.0, 0.005, 0.01}, {0.0, 0.000005, 0.000015, 0.000025, 0.000035}}]
 }]
```

Out[27]=



This solves the Robertson kinetics problem as a DAE.

In[33]:=
```
daesol = First[NDSolve[{kinetics, balance, start}, {y₁, y₂, y₃}, {t, 0, 40 000}]]
```

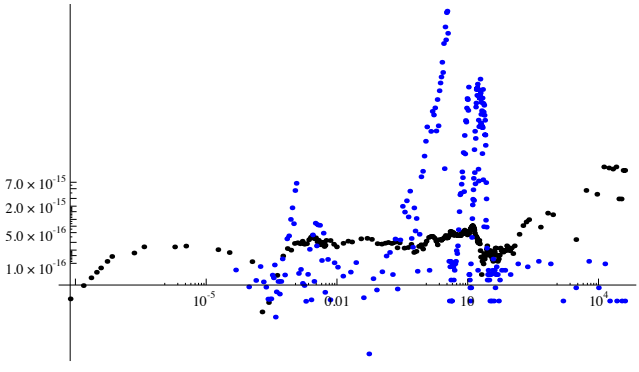
Out[33]=
```
{y₁ → InterpolatingFunction[{{0., 40 000.}}, <>],
 y₂ → InterpolatingFunction[{{0., 40 000.}}, <>], y₃ → InterpolatingFunction[{{0., 40 000.}}, <>]}
```

The solutions for a given component will appear quite close, but comparing the chemical balance constraint shows a difference between them.

Here is a graph of the error in the balance equation for the ODE and DAE solutions, shown in black and blue respectively. A log-log scale is used because of the large variation in t and the magnitude of the error.

In[34]:=
```
berr[t_] = Abs[Apply[Subtract, balance]];
gode = First[InterpolatingFunctionCoordinates[y₁ /. odesol]];
gdae = First[InterpolatingFunctionCoordinates[y₁ /. daesol]];
Show[{
  ListLogLogPlot[Transpose[{gode, berr[gode] /. odesol}], PlotStyle → Black],
  ListLogLogPlot[
   Transpose[{gdae, berr[gdae] /. daesol}], PlotStyle → RGBColor[0, 0, 1]]
 }, ImageSize → 400, PlotRange → All]
```

Out[37]=



In this case, both solutions satisfied the balance equations well beyond expected tolerances. Note that even though the error in the balance equation was greater at some points for the DAE solution, over the long term, the DAE solution is brought back to better satisfy the constraint once the range of quick variation is passed.

You may want to solve some DAEs of the form

$$x'(t) = f(t, x(t))$$
$$g(t, x(t)) = 0,$$

such that the solution of the differential equation is required to satisfy a particular constraint. `NDSolve` cannot handle such DAEs directly because the index is too high and `NDSolve` expects the number of equations to be the same as the number of dependent variables. `NDSolve` does, however, have a `Projection` method that will often solve the problem.

A very simple example of such a constrained system is a nonlinear oscillator modeling the motion of a pendulum.

This defines the equation, invariant constraint, and starting condition for a simulation of the motion of a pendulum.

```
In[55]:= equation = x''[t] + Sin[x[t]] == 0;
         invariant = x'[t]^2 - 2 Cos[x[t]];
         start = {x[0] == 1, x'[0] == 0};
```

Note that the differential equation is effectively the derivative of the invariant, so one way to solve the equation is to use the invariant.

This solves for the motion of a pendulum using the invariant equation. The `SolveDelayed` option tells `NDSolve` not to symbolically solve the quadratic equation for x', but instead to solve the system as a DAE.

```
In[58]:= isol = First[
           NDSolve[{invariant == -2 Cos[1], start}, x, {t, 0, 1000}, SolveDelayed → True]]
Out[58]= {x → InterpolatingFunction[{{0., 1000.}}, <>]}
```

However, this solution may not be quite what you expect: the invariant equation has the solution $x[t]$ == constant when it starts with $x'[t] == 0$. In fact it does not have unique solutions from this starting point. This is because if you do actually solve for x', the function does not satisfy the continuity requirements for uniqueness.

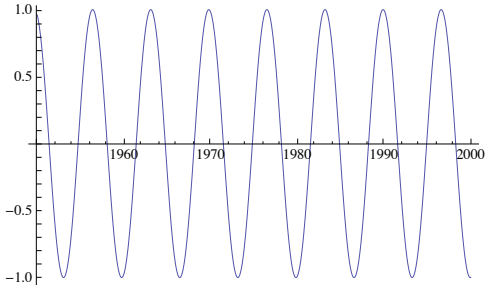
This solves for the motion of a pendulum using only the differential equation. The method "ExplicitRungeKutta" is used because it can also be a submethod of the projection method.

```
In[59]:= dsol =
          First[NDSolve[{equation, start}, x, {t, 0, 2000}, Method → "ExplicitRungeKutta"]]
Out[59]= {x → InterpolatingFunction[{{0., 2000.}}, <>]}
```

This shows the solution plotted over the last several periods.

```
In[60]:= Plot[x[t] /. dsol, {t, 1950, 2000}]
```

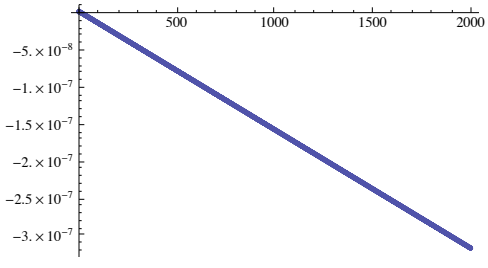
Out[60]=



This shows a plot of the invariant at the ends of the time steps NDSolve took.

```
In[61]:= ts = First[InterpolatingFunctionCoordinates[x /. dsol]];
         ListPlot[Transpose[{ts, invariant + 2 Cos[1] /. dsol /. t → ts}], PlotRange → All]
```

Out[62]=



The error in the invariant is not large, but it does show a steady and consistent drift. Eventually, it could be large enough to affect the fidelity of the solution.

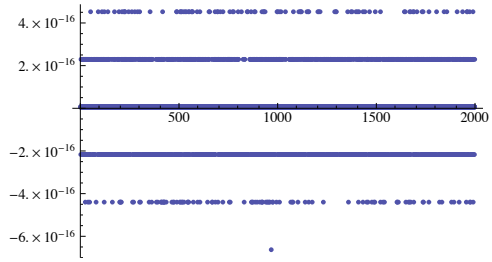
This solves for the motion of the pendulum, constraining the motion at each step to lie on the invariant.

```
In[63]:= psol = First[NDSolve[{equation, start}, x, {t, 0, 2000},
            Method → {Projection, Method → "ExplicitRungeKutta", Invariants → invariant}]]
Out[63]= {x → InterpolatingFunction[{{0., 2000.}}, <>]}
```

This shows a plot of the invariant at the ends of the time steps NDSolve took with the projection method.

```
In[64]:= ts = First[InterpolatingFunctionCoordinates[x /. psol]];
         ListPlot[Transpose[{ts, invariant + 2 Cos[1] /. psol /. t → ts}], PlotRange → All]
```

Out[65]=



IDA Method for NDSolve

The IDA package is part of the SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers) developed at the Center for Applied Scientific Computing of Lawrence Livermore National Laboratory. As described in the IDA user guide [HT99], "IDA is a general purpose solver for the initial value problem for systems of differential-algebraic equations (DAEs). The name IDA stands for Implicit Differential-Algebraic solver. IDA is based on DASPK ..." DASPK [BHP94], [BHP98] is a Fortran code for solving large-scale differential-algebraic systems.

In *Mathematica*, an interface has been provided to the IDA package so that rather than needing to write a function in C for evaluating the residual and compiling the program, *Mathematica* generates the function automatically from the equations you input to NDSolve.

IDA solves the system (1) with Backward Differentiation Formula (BDF) methods of orders 1 through 5, implemented using a variable-step form. The BDF of order k is at time $t_n = t_{n-1} + h_n$ is given by the formula

$$\sum_{i=1}^{k} a_{n,i}\, x_{n-i} = h_n\, x_n'.$$

The coefficients $a_{n,i}$ depend on the order k and past step sizes. Applying the BDF to the DAE (1) gives a system of nonlinear equations to solve:

$$F\left(t_n,\, x_n,\, \frac{1}{h_n}\sum_{i=1}^{k} a_{n,i}\, x_{n-i}\right) = 0.$$

The solution of the system is achieved by Newton-type methods, typically using an approximation to the Jacobian

$$J = \frac{\partial F}{\partial x} + c_n \frac{\partial F}{\partial x'}, \quad \text{where} \quad c_n = \frac{\alpha_{n,0}}{h_n}. \tag{8}$$

"Its [IDAs] most notable feature is that, in the solution of the underlying nonlinear system at each time step, it offers a choice of Newton/direct methods or an Inexact Newton/Krylov (iterative) method." [HT99] In *Mathematica*, you can access these solvers using method options or use the default *Mathematica* `LinearSolve`, which switches automatically to direct sparse solvers for large problems.

At each step of the solution, IDA computes an estimate E_n of the local truncation error and the step size and order are chosen so that the weighted norm `Norm[`E_n / w_n`]` is less than 1. The j^{th}component, $w_{n,j}$, of w_n is given by

$$w_{n,j} = \frac{1}{10^{-prec}\left|x_{n,j}\right| + 10^{-acc}}.$$

The values $prec$ and acc are taken from the `NDSolve` settings for the `PrecisionGoal ->` $prec$ and `AccuracyGoal ->` acc.

Because IDA provides a great deal of flexibility, particularly in the way nonlinear equations are solved, there are a number of method options which allow you to control how this is done. You can use the method options to IDA by giving `NDSolve` the option `Method -> {IDA,` *ida method options*`}`.

The options for the IDA method are associated with the symbol `IDA` in the `NDSolve`` context.

```
In[1]:= Options[NDSolve`IDA]
Out[1]= {MaxDifferenceOrder → 5, ImplicitSolver → Newton}
```

IDA method option name	*default value*	
"ImplicitSolver"	"Newton"	how to solve the implicit equations
"MaxDifferenceOrder"	5	the maximum order BDF to use

`IDA` method options.

When strict accuracy of intermediate values computed with the `InterpolatingFunction` object returned from `NDSolve` is important, you will want to use the `NDSolve` method option setting `InterpolationOrder -> All` that uses interpolation based on the order of the method, sometimes called dense output, to represent the solution between times steps. By default `NDSolve` stores a minimal amount of data to represent the solution well enough for graphical purposes. Keeping the amount of data small saves on both memory and time for more complicated solutions.

As an example which highlights the difference between minimal output and full method interpolation order, consider tracking a quantity, $f(t) = x(t)^2 + y(t)^2$ derived from the solution of a simple linear equation where the exact solution can be computed using `DSolve`.

This defines the function f giving the quantity as a function of time based on solutions $x[t]$ and $y[t]$.

```
In[2]:= f[t_] := x[t]^2 + y[t]^2;
```

This defines the linear equations along with initial conditions.

```
In[3]:= eqns = {x'[t] == x[t] - 2 y[t], y'[t] == x[t] + y[t]};
        ics = {x[0] == 1, y[0] == 1};
```

The exact value of f as a function of time can be computed symbolically using `DSolve`.

```
In[4]:= f_exact[t_] = First[f[t] /. DSolve[{eqns, ics}, {x, y}, t]]
```

$$\text{Out[4]}= e^{2t}\left(\text{Cos}\left[\sqrt{2}\ t\right] - \sqrt{2}\ \text{Sin}\left[\sqrt{2}\ t\right]\right)^2 + \frac{1}{4}e^{2t}\left(2\,\text{Cos}\left[\sqrt{2}\ t\right] + \sqrt{2}\ \text{Sin}\left[\sqrt{2}\ t\right]\right)^2$$

The exact solution will be compared with solutions computed with and without dense output.

A simple way to track the quantity is to create a function which derives it from the numerical solution of the differential equation.

```
In[5]:= f1[t_] = First[f[t] /. NDSolve[{eqns, ics}, {x, y}, {t, 0, 1}]]
```

```
Out[5]= InterpolatingFunction[{{0., 1.}}, <>][t]^2 + InterpolatingFunction[{{0., 1.}}, <>][t]^2
```

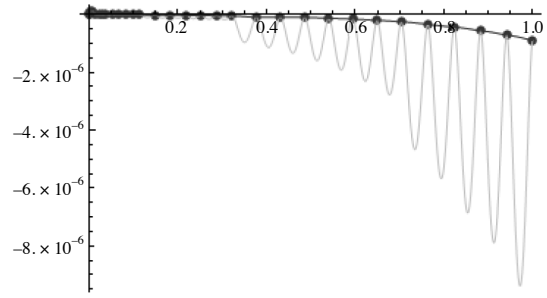
It can also be computed with dense output.

```
In[6]:= f1_dense[t_] =
          First[f[t] /. NDSolve[{eqns, ics}, {x, y}, {t, 0, 1}, InterpolationOrder → All]]
```

```
Out[6]= InterpolatingFunction[{{0., 1.}}, <>][t]^2 + InterpolatingFunction[{{0., 1.}}, <>][t]^2
```

This plot shows the error in the two computed solutions. The computed solution at the time steps are indicated by black dots. The default output error is shown in gray and the dense output error in black.

```
In[7]:= Needs["DifferentialEquations`InterpolatingFunctionAnatomy`"];
        t1 = Cases[f1[t], (if_InterpolatingFunction)[t] →
             InterpolatingFunctionCoordinates[if], Infinity][[1, 1]];
        pode = Show[Block[{$DisplayFunction = Identity},
            {ListPlot[Transpose[{t1, f_exact[t1] - f1[t1]}], PlotStyle → PointSize[.02]],
             Plot[f_exact[t] - f1[t], {t, 0, 1}, PlotStyle → RGBColor[.8, .8, .8]],
             Plot[f_exact[t] - f1_dense[t], {t, 0, 1}]}], PlotRange → All]
```



Out[7]=

From the plot, it is quite apparent that when the time steps get large, the default solution output has much larger error between time steps. The dense output solution represents the solution computed by the solver even between time steps. Keep in mind, however, that the dense output solution takes up much more space.

This compares the sizes of the default and dense output solutions.

```
In[8]:= ByteCount /@ {f1[t], f1_dense[t]}
Out[8]= {3560, 17 648}
```

When the quantity you want to derive from the solution is complicated, you can ensure that it is locally kept within tolerances by giving it as an algebraic quantity, forcing the solver to keep its error in control.

This adds a dependent variable with an algebraic equation that sets the dependent variable equal to the quantity of interest.

```
In[9]:= f2[t_] = First[g[t] /. NDSolve[{eqns, ics, g[t] == f[t]}, {x, y, g}, {t, 0, 1}]]
Out[9]= InterpolatingFunction[{{0., 1.}}, <>][t]
```

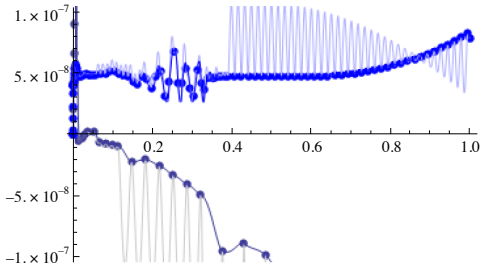
This computes the same solution with dense output.

```
In[10]:= f2_dense[t_] = First[g[t] /. NDSolve[{eqns, ics, g[t] == f[t]},
             {x, y, g}, {t, 0, 1}, InterpolationOrder → All]]
Out[10]= InterpolatingFunction[{{0., 1.}}, <>][t]
```

This makes a plot comparing the error for all four solutions. The time steps for IDA are shown as blue points and the dense output from IDA is shown in blue with the default output shown in light blue.

In[11]:=
```
t2 = InterpolatingFunctionCoordinates[Head[f2[t]]][[1]];
Show[{pode, ListPlot[Transpose[{t2, fexact[t2] - f2[t2]}],
    PlotStyle → {RGBColor[0, 0, 1], PointSize[0.02]}],
  Plot[fexact[t] - f2[t], {t, 0, 1}, PlotStyle → RGBColor[.7, .7, 1]],
  Plot[fexact[t] - f2dense[t], {t, 0, 1}, PlotStyle → RGBColor[0, 0, 1]]},
 PlotRange → {{0, 1}, 1*^-7 {-1, 1}}]
```

Out[11]=



You can see from the plot that the error is somewhat smaller when the quantity is computed algebraically along with the solution.

The remainder of this documentation will focus on suboptions of the two possible settings for the "ImplicitSolver" option, which can be "Newton" or "GMRES". With "Newton", the Jacobian or an approximation to it is computed and the linear system is solved to find the Newton step. On the other hand, "GMRES" is an iterative method and, rather than computing the entire Jacobian, a directional derivative is computed for each iterative step.

The "Newton" method has one method option, "LinearSolveMethod", which you can use to tell *Mathematica* how to solve the linear equations required to find the Newton step. There are several possible values for this option.

Automatic	this is the default, automatically switch between using the *Mathematica* LinearSolve and Band methods depending on the band width of the Jacobian; for systems with larger band width, this will automatically switch to a direct sparse solver for large systems with sparse Jacobians
"Band"	use the IDA band method (see the IDA user manual for more information)
"Dense"	use the IDA dense method (see the IDA user manual for more information)

Possible settings for the "LinearSolveMethod" option.

The "GMRES" method may be substantially faster, but is typically quite a bit more tricky to use because to really be effective typically requires a preconditioner, which can be supplied via a method option. There are also some other method options that control the Krylov subspace process. To use these, refer to the IDA user guide [HT99].

GMRES method option name	*default value*	
"Preconditioner"	Automatic	a *Mathematica* function that returns another function that preconditions
"OrthogonalizationType"	"ModifiedGramSchmidt"	this can also be "ClassicalGramSchmidt" (see variable gstype in the IDA user guide)
"MaxKrylovSubspaceDimension"	Automatic	maximum susbspace dimension (see variable maxl in the IDA user guide)
"MaxKrylovRestarts"	Automatic	maximum number of restarts (see variable maxrs in the IDA user guide)

"GMRES" method options.

As an example problem, consider a two-dimensional Burgers' equation.

$$u_t = \nu\left(u_{xx} + u_{yy}\right) - \frac{1}{2}\left(\left(u^2\right)_x + \left(u^2\right)_y\right)$$

This can typically be solved with an ordinary differential equation solver, but in this example two things are achieved by using the DAE solver. First, boundary conditions are enforced as algebraic conditions. Second, NDSolve is forced to use conservative differencing by using an algebraic term. For comparison, a known exact solution will be used for initial and boundary conditions.

This defines a function that satisfies Burger's equation.

```
In[12]:= Bsol[t_, x_, y_] = 1 / (1 + Exp[(x + y - t) / (2 ν)]);
```

This defines initial and boundary conditions for the unit square consistent with the exact solution.

```
In[13]:= ic = u[0, x, y] == Bsol[0, x, y];
         bc = {
            u[t, 0, y] == Bsol[t, 0, y], u[t, 1, y] == Bsol[t, 1, y],
            u[t, x, 0] == Bsol[t, x, 0], u[t, x, 1] == Bsol[t, x, 1]};
```

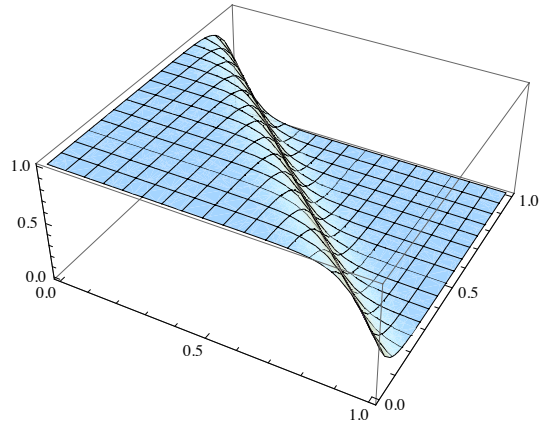
This defines the differential equation.

```
In[14]:= de = D[u[t, x, y], t] == ν (D[u[t, x, y], x, x] + D[u[t, x, y], y, y]) -
              u[t, x, y] (D[u[t, x, y], x] + D[u[t, x, y], y]);
```

This sets the diffusion constant ν to a value for which we can find a solution in a reasonable amount of time and shows a plot of the solution at $t == 1$.

In[15]:=
```
ν = 0.025;
Plot3D[Bsol[1, x, y], {x, 0, 1}, {y, 0, 1}]
```

Out[15]=



You can see from the plot that with $\nu = 0.025$, there is a fairly steep front. This moves with constant speed.

This solves the problem using the default settings for `NDSolve` and the IDA method with the exception of the "`DifferentiateBoundaryConditions`" option for "`MethodOfLines`", which causes `NDSolve` to treat the boundary conditions as algebraic.

In[16]:=
```
Timing[sol = NDSolve[{de, ic, bc}, u, {t, 0, 1}, {x, 0, 1}, {y, 0, 1},
    Method → {"MethodOfLines", "DifferentiateBoundaryConditions" → False}]]
```

Out[16]=
```
{2.233, {{u → InterpolatingFunction[{{0., 1.}, {0., 1.}, {0., 1.}}, <>]}}}
```

Since there is an exact solution to compare to, the overall solution accuracy can be compared as well.

This defines a function that finds the maximum deviation between the exact and computed solutions at the grid points at all of the time steps.

In[17]:=
```
errfun[sol_] := Module[{ifun = First[u /. sol], grid, exvals, gvals},
  grid = InterpolatingFunctionGrid[ifun];
  gvals = InterpolatingFunctionValuesOnGrid[ifun];
  exvals =
   Apply[Bsol, Transpose[grid, RotateLeft[Range[ArrayDepth[grid]], 1]]];
  Max[Abs[exvals - gvals]]]
```

This computes the maximal error for the computed solution.

In[18]:=
```
errfun[sol]
```

Out[18]=
```
0.000749446
```

In the following, a comparison will be made with different settings for the options of the IDA method. To emphasize the option settings, a function will be defined to time the computation of the solution and give the maximal error.

This defines a function for comparing different IDA option settings.

```
In[19]:= TimeSolution[idaopts___] := Module[{time, err, steps},
           time =
            First[Timing[sol = NDSolve[{de, ic, bc}, u, {t, 0, 1}, {x, 0, 1}, {y, 0, 1},
                 Method → {"MethodOfLines", "DifferentiateBoundaryConditions" → False,
                   Method → {IDA, idaopts}}]]];
           err = errfun[sol];
           steps =
            Length[First[InterpolatingFunctionCoordinates[First[u /. sol]]]] "Steps";
           {time, err, steps}]
```

No options use the previous defaults.

```
In[20]:= TimeSolution[]
Out[20]= {2.184, 0.000749446, 88 Steps}
```

This uses the "Band" method.

```
In[21]:= TimeSolution["ImplicitSolver" → {"Newton", "LinearSolveMethod" → "Band"}]
Out[21]= {8.543, 0.000749497, 88 Steps}
```

The "Band" method is not very effective because the bandwidth of the Jacobian is relatively large, partly because of the fourth-order derivatives used and partly because the one-sided stencils used near the boundary add width at the top and bottom. You can specify the bandwidth explicitly.

This uses the "Band" method with the width set to include the stencil of the differences in only one direction.

```
In[22]:= TimeSolution[
          "ImplicitSolver" → {"Newton", "LinearSolveMethod" → {"Band", "BandWidth" → 3}}]
Out[22]= {7.441, 0.000937962, 311 Steps}
```

While the solution time was smaller, notice that the error is slightly greater and the total number of time steps is a lot greater. If the problem was more stiff, the iterations likely would not have converged because it was missing information from the other direction. Ideally, the bandwidth should not eliminate information from an entire dimension.

This computes the grids used in the X and Y directions and shows the number of points used.

```
In[23]:= {X, Y} = InterpolatingFunctionCoordinates[First[u /. sol]][[{2, 3}]];
         {nx, ny} = {Length[X], Length[Y]}
Out[23]= {51, 51}
```

This uses the "Band" method with the width set to include at least part of the stencil in both directions.

```
In[24]:= TimeSolution[
          "ImplicitSolver" → {"Newton", "LinearSolveMethod" → {"Band", "BandWidth" → 51}}]
Out[24]= {2.273, 0.00085973, 88 Steps}
```

With the more appropriate setting of the bandwidth, the solution is still slightly slower than in the default case. The "Band" method can sometimes be effective on two-dimensional problems, but is usually most effective on one-dimensional problems.

This computes the solution using the "GMRES" implicit solver without a preconditioner.

```
In[25]:= TimeSolution["ImplicitSolver" → "GMRES"]
Out[25]= {26.137, 0.00435431, 672 Steps}
```

This is incredibly slow! Using the "GMRES" method without a preconditioner is not recommended for this very reason. However, finding a good preconditioner is not usually trivial. For this example, a diagonal preconditioner will be used.

The setting of the "Preconditioner" option should be a function f, which accepts four arguments that will be given to it by NDSolve such that $f[t, x, x', c]$ returns another function that will apply the preconditioner to the residual vector. (See IDA user guide [HT99] for details on how the preconditioner is used.) The arguments t, x, x', c are the current time, solution vector, solution derivative vector, and the constant c in formula (2) above. For example, if you can determine a procedure that would generate an appropriate preconditioner matrix P as a function of these arguments, you could use

```
"Preconditioner" -> Function[{t, x, xp, c}, LinearSolve[P[t, x, xp, c]]]
```

to produce a LinearSolveFunction object which will effectively invert the preconditioner matrix P. Typically, for each time the preconditioner function is set up, it is applied to the residual vector several times, so using some sort of factorization such as is contained in a LinearSolveFunction is a good idea.

For the diagonal case, the inverse can be effected simply by using the reciprocal. The most difficult part of setting up a diagonal preconditioner is keeping in mind that values on the boundary should not be affected by it.

This finds the diagonal elements of the differentiation matrix for computing the preconditioner.

```
In[26]:= DM = NDSolve`FiniteDifferenceDerivative[{2, 0}, {X, Y}]@"DifferentiationMatrix" +
            NDSolve`FiniteDifferenceDerivative[{0, 2}, {X, Y}]@"DifferentiationMatrix";
          Short[diag = Tr[DM, List]]
Out[26]//Short= {18 750., 6250., 3125., 3125., <<2593>>, 3125., 3125., 6250., 18 750.}
```

This gets the positions where elements at the boundary that satisfy a simple algebraic condition are in the flattened solution vector.

```
In[27]:= bound = SparseArray[
            {{i_, 1} → 1., {i_, ny} → 1., {1, i_} → 1., {nx, i_} → 1.}, {nx, ny}, 0.];
          Short[pos = Drop[ArrayRules[Flatten[bound]], -1][[All, 1, 1]]]
Out[27]//Short= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, <<180>>, 2592,
                 2593, 2594, 2595, 2596, 2597, 2598, 2599, 2600, 2601}
```

This defines the function that sets up the function called to get the effective inverse of the preconditioner. For the diagonal case, the inverse is done simply by taking the reciprocal.

```
In[28]:= pfun[t_, x_, xp_, c_] := Module[{d, dd},
            d = 1. / (c - v diag);
            d[[pos]] = 1.;
            Function[# dd] /. dd → d]
```

This uses the preconditioned "GMRES" method to compute the solution.

```
In[29]:= TimeSolution["ImplicitSolver" → {"GMRES", "Preconditioner" → pfun}]
Out[29]= {1.161, 0.000716006, 88 Steps}
```

Thus, even with a crude preconditioner, the "GMRES" method computes the solution faster than the using the direct sparse solvers.

For PDE discretizations with higher-order temporal derivatives or systems of PDEs, you may need to look at the corresponding `NDSolve`StateData` object to determine how the variables are ordered so that you can get the structural form of the preconditioner correctly.

Delay Differential Equations

A delay differential equation is a differential equation where the time derivatives at the current time depend on the solution and possibly its derivatives at previous times:

$$\begin{cases} X'(t) = F(t, X(t), X(t-\tau_1), \cdots, X(t-\tau_n), X'(t-\sigma_1), \cdots, X(t-\sigma_m); & t \geq t_0 \\ X(t) = \phi(t); & t \leq t_0 \end{cases}$$

Instead of a simple initial condition, an initial history function $\phi(t)$ needs to be specified. The quantities $\tau_i \geq 0,\ i = 1, \ldots, n$ and $\sigma_i \geq 0,\ i = 1, \ldots, k$ are called the delays or time lags. The delays may be constants, functions $\tau(t)$ and $\sigma(t)$ of t (time dependent delays), or functions $\tau(t, X(t))$ and $\sigma(t, X(t))$ (state dependent delays). Delay equations with delays σ of the derivatives are referred to as neutral delay differential equations (NDDEs).

The equation processing code in `NDSolve` has been designed so that you can input a delay differential equation in essentially mathematical notation.

`x[t-τ]`	dependent variable x with delay τ
`x[t /; t≤t0]==ϕ`	specification of initial history function as expression ϕ for t less than t_0

Inputting delays and initial history.

Currently, the implementation for DDEs in `NDSolve` only supports constant delays.

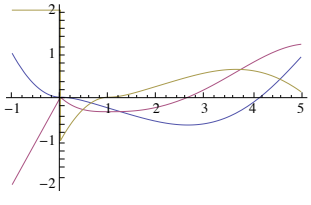
Solve a second order delay differential equation.

```
In[1]:= sol = NDSolve[{x''[t] + x[t - 1] == 0, x[t /; t ≤ 0] == t^2}, x, {t, -1, 5}]
Out[1]= {{x → InterpolatingFunction[{{-1., 5.}}, <>]}}
```

Plot the solution and its first two derivatives.

```
In[2]:= Plot[Evaluate[{x[t], x'[t], x''[t]} /. First[sol]], {t, -1, 5}, PlotRange → All]
```

Out[2]=

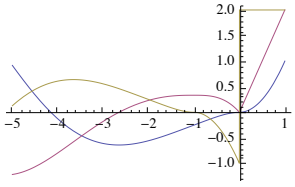


For simplicity, this documentation is written assuming that integration always proceeds from smaller to larger t. However, `NDSolve` supports integration in the other direction if the initial history function is given for value above t_0 and the delays are negative.

Solve a second order delay differential equation in the direction of negative t.

In[3]:=
```
nsol = NDSolve[{x''[t] + x[t + 1] == 0, x[t /; t ≥ 0] == t^2}, x, {t, -5, 1}];
Plot[Evaluate[{x[t], x'[t], x''[t]} /. First[nsol]], {t, -5, 1}, PlotRange → All]
```

Out[3]=



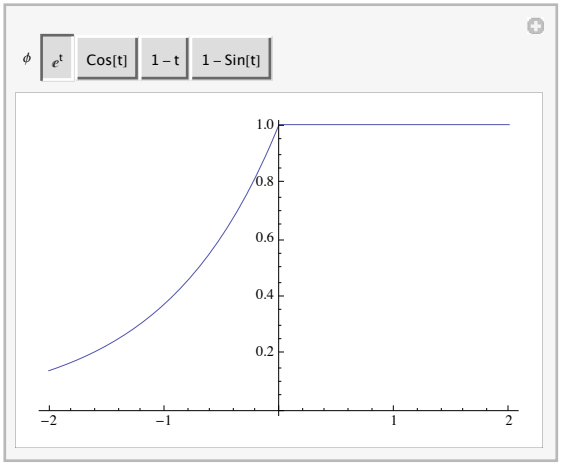
Comparison and Contrast with ODEs

While DDEs look a lot like ODEs the theory for them is quite a bit more complicated and there are some surprising differences with ODEs. This section will show a few examples of the differences.

Look at the solutions of $x'(t) = x(t-1)\,(x(t)-1)$ for different initial history functions.

In[47]:=
```
Manipulate[
 Module[
  {sol = NDSolve[{x'[t] == x[t - 1] (1 - x[t]), x[t /; t ≤ 0] == ϕ}, x, {t, -2, 2}]},
  Plot[Evaluate[x[t] /. First[sol]], {t, -2, 2}]],
 {ϕ, {Exp[t], Cos[t], 1 - t, 1 - Sin[t]}}]
```

Out[47]=

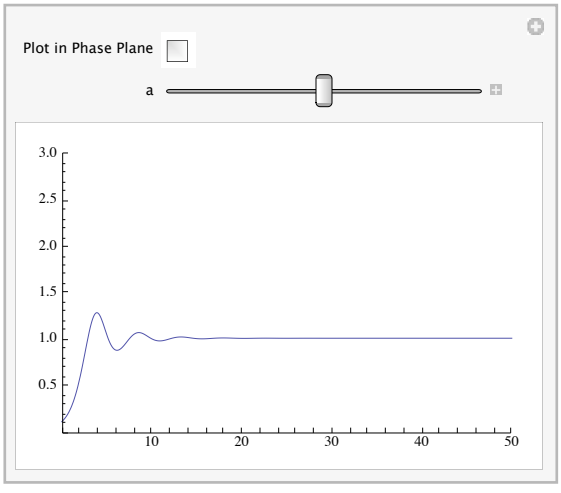


As long as the initial function satisfies $\phi(0) = 1$, the solution for $t > 0$ is always 1. [Z06] With ODEs, you could always integrate backwards in time from a solution to obtain the initial condition.

Investigate at the solutions of $x'(t) = a\, x(t)\,(1 - x(t-1))$ for different values of the parameter a.

In[1]:=
```
Manipulate[
 Module[{T = 50, sol, x, t}, sol = First[x /. NDSolve[
       {x'[t] == a x[t] (1 - x[t - 1]), x[t /; t ≤ 0] == 0.1}, x, {t, 0, T}]];
  If[pp, ParametricPlot[{sol[t], sol[t - 1]}, {t, 1, T},
    PlotRange → {{0, 3}, {0, 3}}],
   Plot[sol[t], {t, 0, T}, PlotRange → {{0, 50}, {0, 3}}]]],
 {{pp, False, "Plot in Phase Plane"}, {False, True}}, {{a, 1}, 0, 2}]
```

Out[1]=



For $a < \frac{1}{e}$, the solutions are monotonic, for $\frac{1}{e} \le a \le \frac{\pi}{2}$ the solutions oscillate. and for $a > \frac{\pi}{2}$ the solutions approach a limit cycle. Of course, for the scalar ODE, solutions are monotonic independent of a.

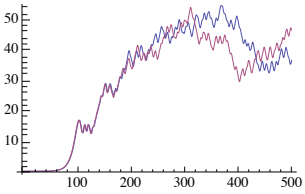
Solve the Ikeda delay differential equation, $x'(t) = \sin(x(t - 2\pi))$ for two nearby constant initial functions.

In[88]:=
```
sol1 =
  First[NDSolve[{x'[t] == Sin[x[t - 20]], x[t /; t ≤ 0] == .0001}, x, {t, 0, 500}]];
sol2 = First[NDSolve[{x'[t] == Sin[x[t - 20]], x[t /; t ≤ 0] == .00011},
    x, {t, 0, 500}]];
```

Plot the solutions.

```
In[90]:= Plot[Evaluate[x[t] /. {sol1, sol2}], {t, 0, 500}]
```

Out[90]=

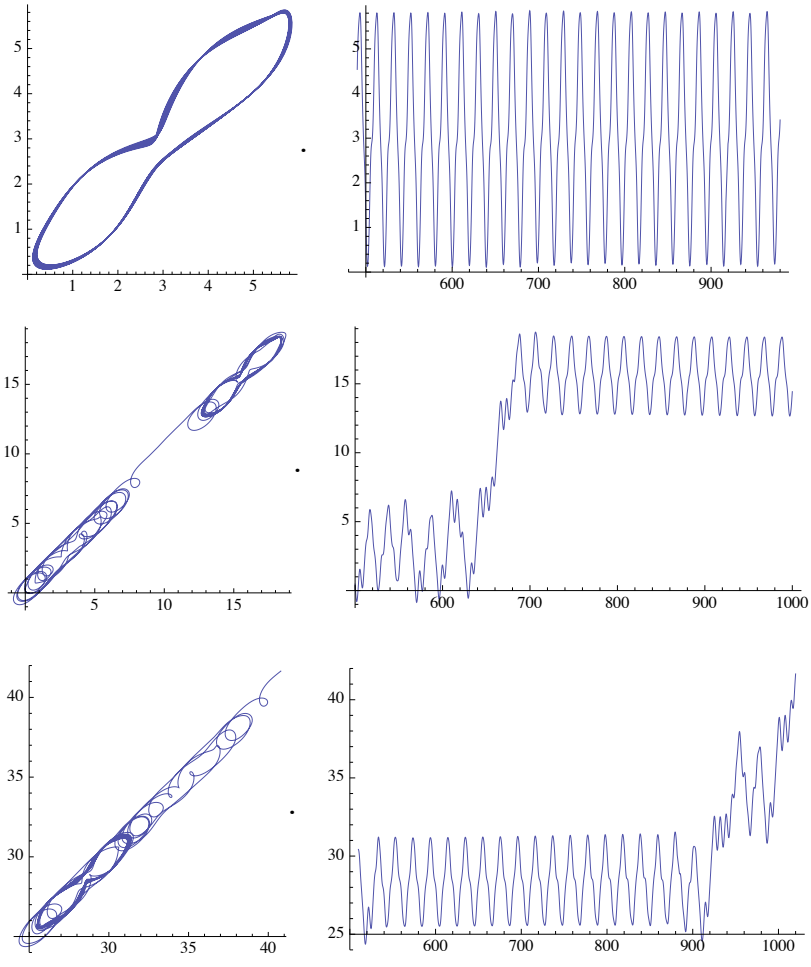


This simple scalar delay differential equation has chaotic solutions and the motion shown above looks very much like Brownian motion. [S07] As the delay τ is increased beyond $\tau = \pi/2$ a limit cycle appears, followed eventually by a period doubling cascade leading to chaos before $\tau = 5$.

Compare solutions for τ=4.9, 5.0, and 5.1

In[104]:=
```
Grid[Table[sol = First[NDSolve[{x'[t] == Sin[x[t - τ]], x[t /; t ≤ 0] == .1},
      x, {t, 100 τ, 200 τ}, MaxSteps → Infinity]];
  {ParametricPlot[Evaluate[{x[t - 1], x[t]} /. sol], {t, 101 τ, 200 τ}].
    Plot[Evaluate[x[t] /. sol], {t, 100 τ, 200 τ}]}, {τ, 4.9, 5.1, .1}]]
```

Out[104]=



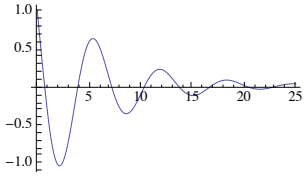
Stability is much more complicated for delay equations as well. It is well known that the linear ODE test equation $x'(t) = \lambda x(t)$ has asymptotically stable solutions if $\mathrm{Re}(\lambda) < 0$ and is unstable if $\mathrm{Re}(\lambda) > 0$.

The closest corresponding DDE is $x'(t) = \lambda\, x(t) + \mu\, x(t-1)$. Even if you consider just real λ and μ the situation is no longer so clear cut. Shown below are some plots of solutions indicating this.

The solution is stable with $\lambda = \frac{1}{2}$ and $\mu = -1$

In[110]:=
```
Block[{λ = 1 / 2, μ = -1, T = 25}, Plot[
  Evaluate[First[x[t] /. NDSolve[{x'[t] == λ x[t] + μ x[t - 1], x[t /; t ≤ 0] == 1 - t},
      x, {t, 0, T}]]], {t, 0, T}, PlotRange → All]]
```

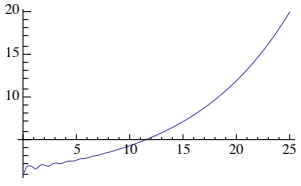
Out[110]=



The solution is unstable with $\lambda = -\frac{7}{2}$ and $\mu = 4$

In[111]:=
```
Block[{λ = -7 / 2, μ = 4, T = 25}, Plot[
  Evaluate[First[x[t] /. NDSolve[{x'[t] == λ x[t] + μ x[t - 1], x[t /; t ≤ 0] == 1 - t},
      x, {t, 0, T}]]], {t, 0, T}, PlotRange → All]]
```

Out[111]=

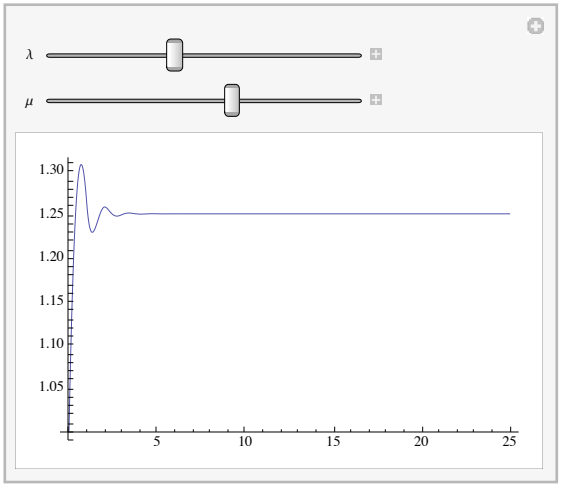


So the solution can be stable with $\lambda > 0$ and unstable with $\lambda < 0$ depending on the value of μ. A `Manipulate` is set up below so that you can investigate the λ-μ plane.

Investigate by varying λ and μ

In[113]:=
```
Manipulate[Module[{T = 25, x, t}, Plot[Evaluate[First[x[t] /.
      NDSolve[{x'[t] == λ x[t] + μ x[t - 1], x[t /; t ≤ 0] == 1 - t}, x, {t, 0, T}]]],
  {t, 0, T}, PlotRange → All]], {λ, -5, 5}, {μ, -5, 5}]
```

Out[113]=



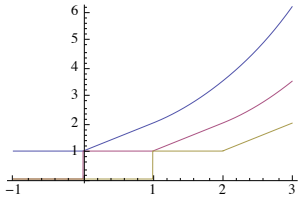
Propagation and Smoothing of Discontinuities

The way discontinuities are propagated by the delays is an important feature of DDEs and has a profound effect on numerical methods for solving them.

Solve $x'(t) = x(t-1)$ with $x(t) = 1$ for $t \le 0$.

```
In[3]:= sol = First[NDSolve[{x'[t] == x[t - 1], x[t /; t ≤ 0] == 1}, x, {t, -1, 3}]]
Out[3]= {x → InterpolatingFunction[{{-1., 3.}}, <>]}

In[4]:= Plot[Evaluate[{x[t], x'[t], x''[t]} /. sol], {t, -1, 3}]
```

Out[4]=



In the example above, $x(t)$ is continuous, but there is a jump discontinuity in $x'(t)$ at $t = 0$ since approaching from the left the value is 0, given by the derivative of the initial history function $x'(t) = \phi'(t) = 0$ while approaching from the right the value is given by the DDE, giving $x'(t) = x(t-1) = \phi(t-1) = 1$.

Near $t=1$, we have by the continuity of x at 0 $\lim_{t\to 1^-} x'(t) = \lim_{t\to 1^-} x(t-1) = \lim_{z\to 0^-} x(z) = \lim_{z\to 0^+} x(z) = \lim_{t\to 1^+} x'(t)$ and so $x'(t)$is continuous at $t = 1$.

Differentiating the equation, we can conclude that $(x')'(t) = x'(t-1)$ so $(x')'(t)$ has a jump discontinuity at $t = 1$. Using essentially the same argument as above, we can conclude that at $t = 2$ the second derivative is continuous.

Similarly, $x^{(k)}(t)$ is continuous at $t = k$ or, in other words, at $t = k$, $x(t)$ is k times differentiable. This is referred to as smoothing and holds generally for non-neutral delay equations. In some cases the smoothing can be faster than one order per interval.[Z06]

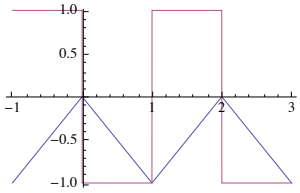
For neutral delay equations the situation is quite different.

Solve $x'(t) = x'(t-1)$ with $x(t) = -t$ for $t \le 0$.

```
In[10]:= sol = First[NDSolve[{x'[t] == -x'[t - 1], x[t /; t ≤ 0] == t}, x, {t, -1, 3}]]
Out[10]= {x → InterpolatingFunction[{{-1., 3.}}, <>]}
```

```
In[11]:= Plot[Evaluate[{x[t], x'[t]} /. sol], {t, -1, 3}]
```

Out[11]=



It is easy to see that the solution is piecewise with $x[t]$ continuous. However,

$$x'(t) = \begin{cases} -1 & 0 < \mathrm{mod}(t, 2) < 1 \\ 1 & 1 < \mathrm{mod}(t, 2) < 2 \end{cases}$$

which has a discontinuity at every non negative integer.

In general, there is no smoothing of discontinuities for neutral DDEs.

The propagation of discontinuities is very important from the standpoint of numerical solvers. If the possible discontinuity points are ignored, then the order of the solver will be reduced. If a discontinuity point is known a more accurate solution can be found by integrating just up to the discontinuity point and then restarting the method just past the point with the new function values. This way, the integration method is used on smooth parts of the solution leading to better accuracy and fewer rejected steps. From any given discontinuity points, future discontinuity points can be determined from the delays and detected by treating them as events to be located.

When there are multiple delays, the propagation of discontinuities can become quite complicated.

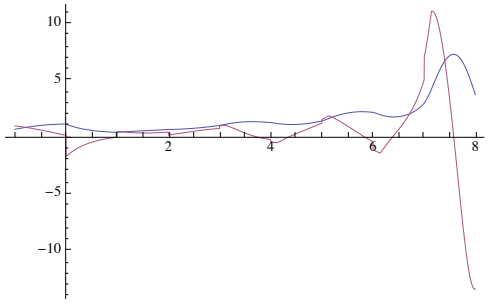
Solve a neutral delay differential equation with two delays.

```
In[109]:= sol =
           NDSolve[{x'[t] == x[t] (x[t - Pi] - x'[t - 1]), x[t /; t ≤ 0] == Cos[t]}, x, {t, -1, 8}]
Out[109]= {{x → InterpolatingFunction[{{-1., 8.}}, <>]}}
```

Plot the solution.

```
In[110]:= Plot[Evaluate[{x[t], x'[t]} /. First[sol]], {t, -1, 8}, PlotRange → All]
```

Out[110]=

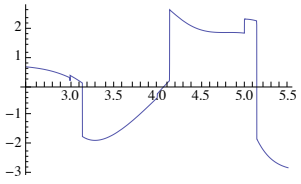


It is clear from the plot that there is a discontinuity at each non negative integer as would be expected from the neutral delay $\sigma = 1$. However, looking at the second and third derivative, it is clear that there are also discontinuities associated with points like $t = \pi$, $1 + \pi$, $2 + \pi$ propagated from the jump discontinuities in $x'(t)$.

Plot the second derivative

```
In[111]:= Plot[Evaluate[x''[t] /. First[sol]], {t, 2.5, 5.5}, PlotRange → All]
```

Out[111]=



In fact, there is a whole tree of discontinuities that are propagated forward in time. A way of determining and displaying the discontinuity tree for a solution interval is shown in the subsection below.

Discontinuity Tree

Define a command that gives the graph for the propagated discontinuities for a DDE with the given delays

```
In[112]:= DiscontinuityTree[t0_, Tend_, delays_] :=
           Module[{dt, next, ord},
            ord[t_] := Infinity;
            ord[t0] = 0;
            next[b_, order_, del_] := Map[dt[b, #, order , del] &, del];
            dt[t_ , {d_, nq_}, order_, del_] := Module[{b = t + d},
              If[b ≤ Tend,
               o  = order + Boole[! nq];
               ord[b] = Min[ord[b], o];
               Sow[{t → b, d}];
               next[b, o, del]]];
            rules = Reap[next[t0, 0, delays]][[2, 1]];
            rules = Tally[rules][[All, 1]];
            f[x_?NumericQ] := {x, ord[x]};
            f[a_ → b_] := f[a] → f[b];
            rules[[All, 1]] = Map[f, rules[[All, 1]]];
            rules]
```

Get the discontinuity tree for the example above up to $t = 8$.

```
In[113]:= tree = Tally[DiscontinuityTree[0, 8, {{1, True}, {π, False}}]][[All, 1]]
```

```
Out[113]= {{{0, 0} → {1, 0}, 1}, {{1, 0} → {2, 0}, 1}, {{2, 0} → {3, 0}, 1}, {{3, 0} → {4, 0}, 1},
           {{4, 0} → {5, 0}, 1}, {{5, 0} → {6, 0}, 1}, {{6, 0} → {7, 0}, 1}, {{7, 0} → {8, 0}, 1},
           {{4, 0} → {4 + π, 1}, π}, {{3, 0} → {3 + π, 1}, π}, {{3 + π, 1} → {4 + π, 1}, 1},
           {{2, 0} → {2 + π, 1}, π}, {{2 + π, 1} → {3 + π, 1}, 1}, {{1, 0} → {1 + π, 1}, π},
           {{1 + π, 1} → {2 + π, 1}, 1}, {{1 + π, 1} → {1 + 2 π, 2}, π}, {{0, 0} → {π, 1}, π},
           {{π, 1} → {1 + π, 1}, 1}, {{π, 1} → {2 π, 2}, π}, {{2 π, 2} → {1 + 2 π, 2}, 1}}
```

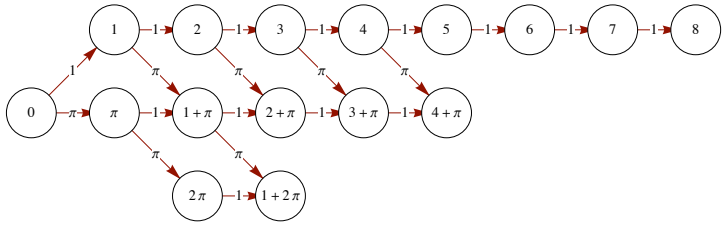
Define a command that shows a plot of $x^{(k)}(t)$ and $x^{(k+1)}(t)$ for a discontinuity of order k.

```
In[116]:= ShowDiscontinuity[{dt_, o_}, ifun_, Δ_] :=
           Quiet[
            Plot[Evaluate[{Derivative[o][ifun][t], Derivative[o + 1][ifun][t]}], {t, dt - Δ,
              dt + Δ}, Exclusions → {dt}, ExclusionsStyle → Red, Frame → True, FrameLabel →
              {None, None, {Derivative[o][x][t], Derivative[o + 1][x][t]}, None}]]
```

Plot as a layered graph, showing the discontinuity plot as a tooltip for each discontinuity.

```
In[117]:= LayeredGraphPlot[tree, Left, VertexLabeling → True, VertexRenderingFunction →
            Function[Tooltip[{White, EdgeForm[Black], Disk[#, .3], Black, Text[#2[[1]], #1]},
              ShowDiscontinuity[#2, First[x /. sol], 1]]]]
```

Out[117]=



Storing History Data

Once the solution has advanced beyond the first discontinuity point, some of the delayed values that need to be computed are outside of the domain of the initial history function and the computed solution needs to be used to get the values, typically be interpolating between steps previously taken. For the DDE solution to be accurate it is essential that the interpolation be as accurate as the method. This is achieved by using dense output for the ODE integration method (the output you get if you use the option `InterpolationOrder -> All` in `NDSolve`). `NDSolve` has a general algorithm for obtaining dense output from most methods, so you can use just about any method as the integrator. Some methods, including the default for DDEs have their own way of getting dense output which is usually more efficient than the general method. Methods that are low enough order, such as `"ExplicitRungeKutta"` with `"DifferenceOrder" -> 3` can just use a cubic Hermite polynomial as the dense output so there is essentially no extra cost in keeping the history.

Since the history data is accessed frequently, it needs to have a quick look up mechanism to determine which step to interpolate within. In `NDSolve`, this is done with a binary search mechanism and the search time is negligible compared with the cost of actual function evaluation.

The data for each successful step is saved before attempting the next step and is saved in a data structure that can repeatedly be expanded efficiently. When `NDSolve` produces the solution, it simply takes this data and restructures it into an `InterpolatingFunction` object, so DDE solutions are always returned with dense output.

The Method of Steps

For constant delays, it is possible to get the entire set of discontinuities as fixed time. The idea of the method of steps is to simply integrate the smooth function over these intervals and restart on the next interval, being sure to reevaluate the function from the right. As long as the intervals do not get too small, the method works quite well in practice.

The method currently implemented for `NDSolve` is based on the method of steps.

Symbolic method of steps

This section defines a symbolic method of steps that illustrates how the method works. Note that to keep the code simpler and more to the point, it does not do any real argument checking. Also, the data structure and look up for the history is not done in an efficient way, but for symbolic solutions this is a minor issue.

Use `DSolve` to integrate over an interval where the solution is smooth.

In[16]:=
```
IntegrateSmooth[rhs_, history_, delayvars_, pfun_, dvars_, {t_, t0_, t1_}] :=
  Module[{delayvals, dvt, tau, hrule, dvrule, dvrules, oderhs, ode, init, sol},
   dvt[tau_] = Map[#[tau] &, dvars];
   hrule[pos_] :=
    Thread[dvars -> Map[Function[Evaluate[{t}], #] &, history[[pos]]]];
   dvrule[(dv_)[z_]] := Module[{delay, pos},
     delay = t - z;
     pos = pfun[t0 - delay];
     dv[z] → (dv[z] /. hrule[pos])];
   dvrules = Map[dvrule, delayvars];
   oderhs = rhs /. dvrules;
   ode = Thread[D[dvt[t], t] == oderhs];
   init = Thread[dvt[t0] == (dvt[t0] /. hrule[-1])];
   sol = DSolve[{ode, init}, dvars, t];
   If[Head[sol] === DSolve || Length[sol] == 0,
    Message[DDESteps::stuck, ode, init];
    Throw[$Failed]];
   dvt[t] /. First[sol]
  ];
DDESteps::stuck =
  "DSolve was not able to find a solution for `1` with initial conditions `2`.";
```

Define a method of steps function that returns `Piecewise` functions.

In[21]:=
```
DDESteps[rhsin_, phin_, dvarsin_, {t_, tinit_, tend_}] :=
  Module[{rhs = Listify[rhsin], phi = Listify[phin], dvars = Listify[dvarsin],
    history, delayvars, delays, dtree, intervals, p, pfun, next, pieces, hfuns},
   history = {phi};
   delayvars = Cases[rhs, (v : ((dv_[z_] | Derivative[1][dv_][z_]) /;
          (MemberQ[dvars, dv] && UnsameQ[z, t]))) → {v, t - z}, Infinity];
   {delayvars, delays} = Map[Union, Transpose[delayvars]];
   dtree = DiscontinuityTree[tinit, tend, Map[{#, True} &, delays]];
   dtree = Union[Flatten[{tinit, tend, dtree[[All, 1, 2, 1]]}]];
   dtree = SortBy[dtree, N];
   intervals = Partition[dtree, 2, 1];
   p = 2;
   pfun =
    Join[{{1, t < tinit}}, Apply[Function[{p++, #1 ≤ t < #2}], intervals, {1}]];
   pfun = Function[Evaluate[{t}], Evaluate[Piecewise[pfun, p]]];
   Catch[Do[
     next = IntegrateSmooth[rhs,
       history, delayvars, pfun, dvars, Prepend[interval, t]];
     history = Append[history, next],
     {interval, intervals}]];
   pieces =
    Flatten[{t < tinit, Apply[(#1 ≤ t < #2) &, Drop[intervals, -1], {1}],
      Apply[(#1 ≤ t ≤ #2) &, Last[intervals]]}];
   pieces = Take[pieces, Length[history]];
   hfuns = Map[Function[Evaluate[{t}], Evaluate[Piecewise[
         Transpose[{#, pieces}], Indeterminate]]] &, Transpose[history]];
   Thread[dvars → hfuns]
  ];
Listify[x_List] := x;
Listify[x_] := {x};
```

Find the solution for the DDE $x'(t) = x(t-1) - x(t)$ with $\phi(t) = \sin(t)$.

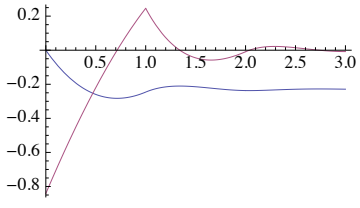
In[24]:=
```
sol = DDESteps[x[t - 1] - x[t], Sin[t], x, {t, 0, 3}]
```

Out[24]=
$$\left\{x \to \text{Function}\left[\{t\},\ \begin{cases} \text{Sin}[t] & t < 0 \\ -\frac{1}{2} e^{-t}\left(-\text{Cos}[1] + e^{t}\,\text{Cos}[1-t] - \text{Sin}[1] + e^{t}\,\text{Sin}[1-t]\right) & 0 \le t \\ -\frac{1}{2} e^{-t}\left(e - \text{Cos}[1] - e\,t\,\text{Cos}[1] + e^{t}\,\text{Cos}[2-t] - \text{Sin}[1] + e\,\text{Sin}[1] - e\,t\,\text{Sin}[1]\right) & 1 \le t \\ -\frac{1}{4} e^{-t}\left(2e - 2e^{2} + 2e^{2}t - 2\,\text{Cos}[1] - e^{2}\,\text{Cos}[1] - 2e\,t\,\text{Cos}[1] + \right. & 2 \le t \\ \quad 2e^{2}\,t\,\text{Cos}[1] - e^{2}t^{2}\,\text{Cos}[1] + e^{t}\,\text{Cos}[3-t] - 2\,\text{Sin}[1] + 2e\,\text{Sin}[1] - \\ \quad \left. 3e^{2}\,\text{Sin}[1] - 2e\,t\,\text{Sin}[1] + 4e^{2}\,t\,\text{Sin}[1] - e^{2}t^{2}\,\text{Sin}[1] - e^{t}\,\text{Sin}[3-t]\right) & \\ \text{Indeterminate} & \text{True} \end{cases}\right]\right\}$$

Plot the solution.

In[25]:=
```
Plot[Evaluate[{x[t], x'[t]} /. sol], {t, 0, 3}]
```

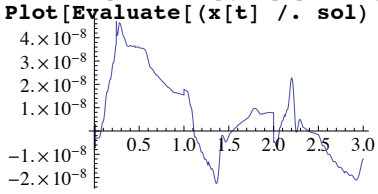
Out[25]=



Check the quality of the solution found by `NDSolve` by comparing to the exact solution.

In[26]:=
```
ndsol =
  First[NDSolve[{x'[t] == -x[t] + x[t - 1], x[t /; t ≤ 0] == Sin[t]}, x, {t, 0, 3}]];
Plot[Evaluate[(x[t] /. sol) - (x[t] /. ndsol)], {t, 0, 3}, PlotRange → All]
```

Out[27]=



The method will also work for neutral DDEs.

Find the solution for the neutral DDE $x'(t) = x'(t-1) - x(t)$ with $\phi(t) = \sin(t)$.

In[28]:=
```
sol = DDESteps[x'[t - 1] - x[t], Sin[t], x, {t, 0, 3}]
```

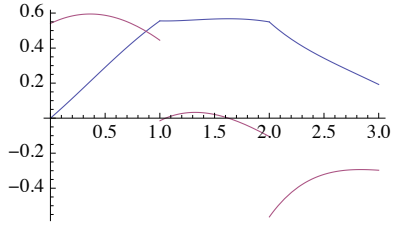
Out[28]=

$$\left\{x \to \text{Function}\left[\{t\},\ \begin{cases} \text{Sin}[t] & t < 0 \\ \frac{1}{2} e^{-t}\left(-\text{Cos}[1] + e^{t}\,\text{Cos}[1-t] + \text{Sin}[1] - e^{t}\,\text{Sin}[1-t]\right) & 0 \le t \\ \frac{1}{2} e^{-t}\left(e - \text{Cos}[1] - 2e\,\text{Cos}[1] + e\,t\,\text{Cos}[1] + e^{t}\,\text{Cos}[2-t] + \text{Sin}[1] + e\,\text{Sin}[1] - e\,t\,\text{Sin}[1]\right) & 1 \le t \\ \frac{1}{4} e^{-t}\left(2e + 6e^{2} - 2e^{2}t - 2\,\text{Cos}[1] - 4e\,\text{Cos}[1] - 13e^{2}\,\text{Cos}[1] + 2e\,t\,\text{Cos}[1] + 8e^{2}\,t\,\text{Cos}[1] - e^{2}t^{2}\,\text{Cos}[1] + e^{t}\,\text{Cos}[3-t] + 2\,\text{Sin}[1] + 2e\,\text{Sin}[1] + 7e^{2}\,\text{Sin}[1] - 2e\,t\,\text{Sin}[1] - 6e^{2}\,t\,\text{Sin}[1] + e^{2}t^{2}\,\text{Sin}[1] + e^{t}\,\text{Sin}[3-t]\right) & 2 \le t \\ \text{Indeterminate} & \text{True} \end{cases}\right]\right\}$$

Plot the solution.

In[29]:=
```
Plot[Evaluate[{x[t], x'[t]} /. sol], {t, 0, 3}]
```

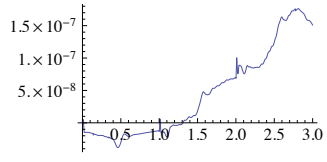
Out[29]=



Check the quality of the solution found by `NDSolve` by comparing to the exact solution.

In[30]:=
```
ndsol =
  First[NDSolve[{x'[t] == -x[t] + x'[t - 1], x[t /; t ≤ 0] == Sin[t]}, x, {t, 0, 3}]];
Plot[Evaluate[(x[t] /. sol) - (x[t] /. ndsol)], {t, 0, 3}, PlotRange → All]
```

Out[31]=



The symbolic method will also work with symbolic parameter values as long as `DSolve` is able to still able to find the solution.

Find the solution to a simple linear DDE with symbolic coefficients.

In[32]:=
```
sol = DDESteps[λ x[t] + μ x[t - 1], t, x, {t, 0, 2}]
```

Out[32]=
$$\left\{x \to \text{Function}\left[\{t\},\begin{cases} t & t < 0 \\ -\frac{\left(1-e^{t\lambda}-\lambda+e^{t\lambda}\lambda+t\lambda\right)\mu}{\lambda^2} & 0 \le t < 1 \\ -\frac{e^{-\lambda}\mu\left(e^{t\lambda}\lambda-e^{\lambda+t\lambda}\lambda+e^{\lambda+t\lambda}\lambda^2-\lambda e^{\lambda}\mu+\lambda e^{t\lambda}\mu+\lambda e^{\lambda}\lambda\mu-e^{\lambda}t\lambda\mu-e^{t\lambda}t\lambda\mu-e^{t\lambda}\lambda^2\mu+e^{t\lambda}t\lambda^2\mu\right)}{\lambda^3} & 1 \le t \le 2 \\ \text{Indeterminate} & \text{True} \end{cases}\right]\right\}$$

The reason the code was designed to take lists was so that it would work with systems

Solve a system of DDEs.

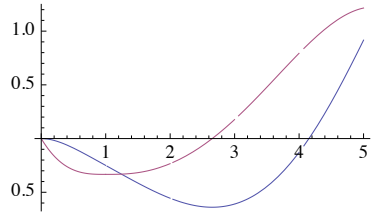
In[33]:=
```
ssol = DDESteps[{y[t], -x[t - 1]}, {t^2, 2 t}, {x, y}, {t, 0, 5}]
```

Out[33]=
$$\left\{x \to \text{Function}\left[\{t\},\begin{cases} t^2 & t < 0 \\ \frac{1}{12}\left(-6\,t^2+4\,t^3-t^4\right) & 0 \le t \\ \frac{1}{360}\left(52-216\,t+165\,t^2-140\,t^3+60\,t^4-12\,t^5+t^6\right) & 1 \le t \\ \frac{-3744+8640\,t-18\,088\,t^2+11\,872\,t^3-5040\,t^4+1456\,t^5-252\,t^6+24\,t^7-t^8}{20\,160} & 2 \le t \\ \frac{1}{1\,814\,400}\left(804\,654-2\,371\,680\,t+2\,210\,265\,t^2-1\,643\,400\,t^3+\right. & 3 \le t \\ \quad \left. 771\,120\,t^4-236\,376\,t^5+51\,030\,t^6-7560\,t^7+720\,t^8-40\,t^9+t^{10}\right) & \\ \frac{1}{239\,500\,800}\left(-168\,512\,584+394\,727\,040\,t-534\,391\,836\,t^2+359\,788\,000\,t^3-165\,844\,800\,t^4+\right. & 4 \le t \\ \quad \left. 55\,576\,224\,t^5-13\,370\,280\,t^6+2\,347\,488\,t^7-300\,960\,t^8+27\,280\,t^9-1650\,t^{10}+60\,t^{11}-t^{12}\right) & \\ \text{Indeterminate} & \text{True} \end{cases}\right], y \to \text{Function}\left[\{t\},\begin{cases} 2\,t & t < 0 \\ \frac{1}{3}\left(-3\,t+3\,t^2-t^3\right) & 0 \le t \\ \frac{1}{60}\left(-36+55\,t-70\,t^2+40\,t^3-10\,t^4+t^5\right) & 1 \le t \\ \frac{1080-4522\,t+4452\,t^2-2520\,t^3+910\,t^4-189\,t^5+21\,t^6-t^7}{2520} & 2 \le t \\ \frac{-237\,168+442\,053\,t-493\,020\,t^2+308\,448\,t^3-118\,188\,t^4+30\,618\,t^5-5292\,t^6+576\,t^7-36\,t^8+t^9}{181\,440} & 3 \le t \\ \frac{1}{19\,958\,400}\left(32\,893\,920-89\,065\,306\,t+89\,947\,000\,t^2-55\,281\,600\,t^3+23\,156\,760\,t^4-\right. & 4 \le t \\ \quad \left. 6\,685\,140\,t^5+1\,369\,368\,t^6-200\,640\,t^7+20\,460\,t^8-1375\,t^9+55\,t^{10}-t^{11}\right) & \\ \text{Indeterminate} & \text{True} \end{cases}\right]\right\}$$

Plot the solution.

In[34]:=
```
Plot[Evaluate[{x[t], y[t]} /. ssol], {t, 0, 5}]
```

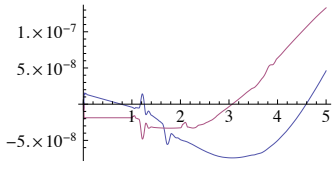
Out[34]=



Check the quality of the solution found by `NDSolve` by comparing to the exact solution.

In[35]:=
```
ndssol = First[NDSolve[{x'[t] == y[t], y'[t] == -x[t - 1],
      x[t /; t ≤ 0] == t^2, y[t /; t ≤ 0] == 2 t}, {x, y}, {t, 0, 5}]];
Plot[Evaluate[({x[t], y[t]} /. ssol) - ({x[t], y[t]} /. ndssol)], {t, 0, 5}]
```

Out[36]=



Since the method computes the discontinuity tree, it will also work for multiple constant delays. However, with multiple delays, the solution may become quite complicated quickly and `DSolve` can bog down with huge expressions.

Solve a nonlinear neutral DDE with two delays.

In[37]:=
```
sol = DDESteps[x[t] (x[t - Log[2]] - x'[t - 1]), 1, x, {t, 0, 2}]
```

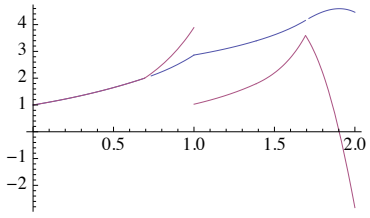
Out[37]=

$$\left\{x \to \text{Function}\left[\{t\},\ \begin{cases} 1 & t < 0 \\ e^t & 0 \le t < \text{Log}[2] \\ 2\, e^{-1+\frac{e^t}{2}} & \text{Log}[2] \le t < 1 \\ 2\, e^{\frac{1}{2}(-2+e)\, e^{-1+t}} & 1 \le t < 2\,\text{Log}[2] \\ 2\, e^{2-e^{-1+t}-\frac{2\,\text{ExpIntegralEi}[1]}{e}+\frac{2\,\text{ExpIntegralEi}\left[\frac{e^t}{4}\right]}{e}} & 2\,\text{Log}[2] \le t < 1+\text{Log}[2] \\ 2\, e^{2-2\,e^{\frac{-1+e^{-1+t}}{2}}-\frac{2\,\text{ExpIntegralEi}[1]}{e}-2\,\text{ExpIntegralEi}\left[\frac{1}{2}(-2+e)\right]+\frac{2\,\text{ExpIntegralEi}\left[\frac{e^t}{2}\right]}{e}+2\,\text{ExpIntegralEi}\left[\frac{1}{4}(-2+e)\, e^{-1+t}\right]} & 1+\text{Log}[2] \le t \le 2 \\ \text{Indeterminate} & \text{True} \end{cases}\right]\right\}$$

Plot the solution.

```
In[38]:= Plot[Evaluate[{x[t], x'[t]} /. sol], {t, 0, 2}]
```

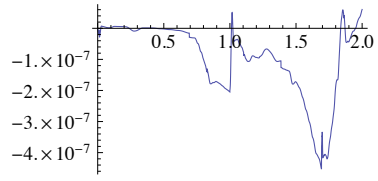
Out[38]=



Check the quality of the solution found by NDSolve by comparing to the exact solution.

```
In[39]:= ndsol = First[NDSolve[
           {x'[t] == x[t] (x[t - Log[2]] - x'[t - 1]), x[t /; t ≤ 0] == 1}, x, {t, 0, 2}]];
         Plot[Evaluate[(x[t] /. sol) - (x[t] /. ndsol)], {t, 0, 2}, PlotRange → All]
```

Out[40]=



Examples

Lotka-Volterra equations with delay

The Lotka-Volterra system models the growth and interaction of animal species assuming that the effect of one species on another is continuous and immediate. A delayed effect of one species on another can be modeled by introducing time lags in the interaction terms.

Consider the system

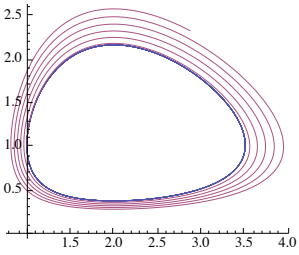
$$Y_1'(t) = Y_1(t)\,(Y_2(t-\tau_2) - 1),\ Y_2'(t) = Y_2(t)\,(2 - Y_1(t-\tau_1)). \tag{9}$$

With no delays, $\tau_1 = \tau_2 = 0$ the system (1) has an invariant $H(t) = 2\,ln\,Y_1 - Y_1 + ln\,Y_2 - Y_1$ that is constant for all t and there is a (neutrally) stable periodic solution.

Compare the solution with and without delays.

In[13]:=
```
lvsystem[τ1_, τ2_] := {
    Y₁'[t] == Y₁[t] (Y₂[t - τ1] - 1), Y₁[0] == 1,
    Y₂'[t] == Y₂[t] (2 - Y₁[t - τ2]), Y₂[0] == 1};
lv = First[NDSolve[lvsystem[0, 0], {Y₁, Y₂}, {t, 0, 25}]];
lvd = Quiet[First[NDSolve[lvsystem[.01, 0], {Y₁, Y₂}, {t, 0, 25}]]];
ParametricPlot[Evaluate[{Y₁[t], Y₂[t]} /. {lv, lvd}], {t, 0, 25}]
```

Out[16]=



In this example, the effect of even a small delay is to destabilize the periodic orbit. With different parameters in the delayed Lotka-Volterra system it has been shown that there are globally attractive equilibria.[TZ08]

Enzyme kinetics

Consider the system

$$\begin{aligned} y_1'(t) &= I_s - z\, y_1(t) \\ y_2'(t) &= z\, y_1(t) - y_2(t) \\ y_3'(t) &= y_2(t) - y_3(t) \\ y_2'(t) &= y_3(t) - \tfrac{1}{2}\, y_4(t) \end{aligned} \qquad z = \frac{k_1}{1+\alpha\,(y_4(t-\tau))^n} \tag{10}$$

modeling enzyme kinetics where I_s is a substrate supply maintained at a constant level and n molecules of the end product y_4 inhibits the reaction step $y_1 \to y_2$. [HNW93]

The system has an equilibrium when $\{y_1 = I_s/z,\ y_2 = y_3 = I_s,\ y_4 = 2\, I_s\}$.

Investigate solutions of (1) starting a small perturbation away from the equilibrium.

In[43]:=
```
Manipulate[
 Module[{t, y1, y2, y3, y4, z, sol},
  z = k1 / (1 + α y4[t - τ]^n);
  sol = First[NDSolve[{
       y1'[t] == Is - z y1[t], y1[t /; t ≤ 0] == Is * (1 + α (2 Is)^n) + ϵ,
       y2'[t] == z y1[t] - y2[t], y2[t /; t ≤ 0] == Is,
       y3'[t] == y2[t] - y3[t], y3[t /; t ≤ 0] == Is,
       y4'[t] == y3[t] - y4[t] / 2, y4[t /; t ≤ 0] == 2 Is},
      {y1, y2, y3, y4}, {t, 0, 200}]];
  Plot[Evaluate[{y1[t], y2[t], y3[t], y4[t]} /. sol], {t, 0, 200}]],
 {{Is, 10.5}, 1, 20}, {{α, 0.0005}, 0, .001}, {{k1, 1}, 0, 2},
 {{n, 3}, 1, 10, 1}, {{τ, 4}, 0, 10}, {{ϵ, 0.1}, 0, .25}]
```

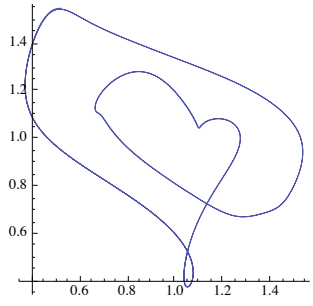
Mackey-Glass equation

The Mackey-Glass equation x'[t]=a x[t-τ]/(1 + x[t-τ]^n) - b x[t] was proposed to model the production of white blood cells. There are both periodic and chaotic solutions.

Here is a periodic solution of the Mackey-Glass equation. The plot is only shown after $t = 300$ to let transients die out.

In[31]:=
```
sol = First[ NDSolve[{x'[t] == (1 / 4) x[t - 15] / (1 + x[t - 15]^10) - x[t] / 10,
      x[t /; t ≤ 0] == 1 / 2}, x, {t, 0, 500}]];
ParametricPlot[Evaluate[{x[t], x[t - 15]} /. sol], {t, 300, 500}]
```

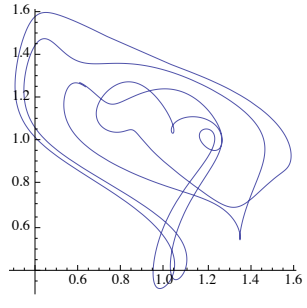
Out[32]=



Here is a chaotic solution of the Mackey-Glass equation.

In[44]:=
```
sol = First[ NDSolve[{x'[t] == (1 / 4) x[t - 17] / (1 + x[t - 17]^10) - x[t] / 10,
      x[t /; t ≤ 0] == 1 / 2}, x, {t, 0, 500}]];
ParametricPlot[Evaluate[{x[t], x[t - 17]} /. sol], {t, 300, 500}]
```

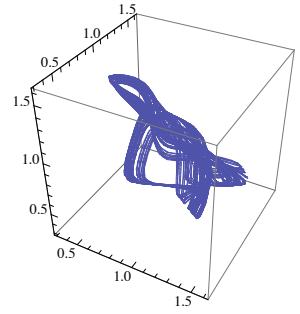
Out[45]=



This shows an embedding of the solution above in 3D $\{x(t), x(t-\tau), x(t-2\tau)\}$.

In[14]:=
```
sol = First[ NDSolve[{x'[t] == (1 / 4) x[t - 17] / (1 + x[t - 17]^10) - x[t] / 10,
      x[t /; t ≤ 0] == 1 / 2}, x, {t, 0, 5000}, MaxSteps → ∞]];
ParametricPlot3D[Evaluate[{x[t], x[t - 17], x[t - 34]} /. sol], {t, 500, 5000}]
```

Out[15]=

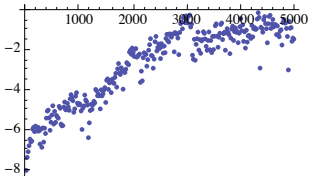


It is interesting to check the accuracy of the chaotic solution.

Compute the chaotic solution with another method and plot $\log_{10}|d|$ for the difference d between $x(t)$ computed by the different methods.

In[16]:=
```
solrk = First[ NDSolve[{x'[t] == (1 / 4) x[t - 17] / (1 + x[t - 17]^10) - x[t] / 10,
      x[t /; t ≤ 0] == 1 / 2}, x, {t, 0, 5000}, MaxSteps → ∞,
    Method → {"ExplicitRungeKutta", "DifferenceOrder" → 3}]];
ListPlot[Table[{t, RealExponent[(x[t] /. sol) - (x[t] /. solrk)]},
  {t, 17, 5000, 17}]]
```

Out[17]=



By the end of the interval, the differences between methods is order 1. Large deviation is typical in chaotic systems and in practice it is not possible or even necessary to get a very accurate solution for a large interval. However, if you do want a high quality solution, `NDSolve` allows you to use higher precision. For DDEs with higher precision, the "`StiffnessSwitching`" method is recommended.

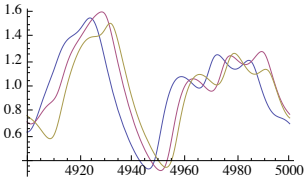
Compute the chaotic solution with higher precision and tolerances.

In[18]:=
```
hpsol = First[ NDSolve[{x'[t] == (1 / 4) x[t - 17] / (1 + x[t - 17]^10) - x[t] / 10,
      x[t /; t ≤ 0] == 1 / 2}, x, {t, 0, 5000}, MaxSteps → ∞,
    Method → "StiffnessSwitching", WorkingPrecision → 32 ]];
```

Plot the three solutions near the final time.

```
In[19]:= Plot[Evaluate[x[t] /. {hpsol, sol, solrk}], {t, 4900, 5000}, PlotRange → All]
```

Out[19]=



Norms in NDSolve

`NDSolve` uses norms of error estimates to determine when solutions satisfy error tolerances. In nearly all cases the norm has been weighted, or scaled, such that it is less than 1 if error tolerances have been satisfied and greater than one if error tolerances are not satisfied. One significant advantage of such a scaled norm is that a given method can be written without explicit reference to tolerances: the satisfaction of tolerances is found by comparing the scaled norm to 1, thus simplifying the code required for checking error estimates within methods.

Suppose that v is vector and u is a reference vector to compute weights with (typically u is an approximate solution vector). Then the scaled vector w to which the norm is applied has components:

$$w_i = \frac{|v_i|}{t_a + t_r |u_i|} \tag{11}$$

where absolute and relative tolerances t_a and t_r are derived respectively from the `AccuracyGoal -> ag` and `PrecisionGoal -> pg` options by $t_a = 10^{-ag}$ and $t_r = 10^{-pg}$.

The actual norm used is determined by the setting for the `NormFunction` option given to `NDSolve`.

option name	*default value*	
`NormFunction`	`Automatic`	a function to use to compute norms of error estimates in `NDSolve`

`NormFunction` option to `NDSolve`.

The setting for the `NormFunction` option can be any function that returns a scalar for a vector argument and satisfies the properties of a norm. If you specify a function that does not satisfy the required properties of a norm, `NDSolve` will almost surely run into problems and give an answer, if any, which is incorrect.

The default value of `Automatic` means that `NDSolve` may use different norms for different methods. Most methods use an infinity-norm, but the IDA method for DAEs uses a 2-norm because that helps maintain smoothness in the merit function for finding roots of the residual. It is strongly recommended that you use `Norm` with a particular value of p. For this reason, you can also use the shorthand `NormFunction -> p` in place of `NormFunction -> (Norm[#, p] / Length[#] ^ (1 / p) &)`. The most commonly used implementations for $p = 1$, $p = 2$, and $p = \infty$ have been specially optimized for speed.

This compares the overall error for computing the solution to the simple harmonic oscillator over 100 cycles with different norms specified.

```
In[1]:= Map[
          First[(1 - x[100 π]) /. NDSolve[{x''[t] + x[t] == 0, x[0] == 1, x'[0] == 0}, x,
              {t, 0, 100 π}, Method → ExplicitRungeKutta, NormFunction → #]] &, {1, 2, ∞}]
Out[1]= {8.62652×10^-8, 7.50564×10^-8, 5.81547×10^-8}
```

The reason that error decreases with increasing p is because the norms are normalized by multiplying with $1/n^{1/p}$, where n is the length of the vector. This is often important in `NDSolve` because in many cases, an attempt is being made to check the approximation to a function, where more points should give a better approximation, or less error.

Consider a finite difference approximation to the first derivative of a periodic function u given by $u_i' = \frac{u_{i+1} - u_i}{h}$ where $u_i = u(x_i)$ on a grid with uniform spacing $h = x_{i+1} - x_i$. In *Mathematica*, this can easily be computed using `ListCorrelate`.

This computes the error of the first derivative approximation for the cosine function on a grid with 16 points covering the interval $[0, 2\pi]$.

```
In[2]:= h = 2 π / 16.;
        grid = h Range[16];
        err16 = Sin[grid] - ListCorrelate[{1, -1} / h, Cos[grid], {1, 1}]
Out[2]= {-0.169324, -0.11903, -0.0506158, 0.0255046, 0.0977423, 0.1551, 0.188844, 0.193839,
         0.169324, 0.11903, 0.0506158, -0.0255046, -0.0977423, -0.1551, -0.188844, -0.193839}
```

This computes the error of the first derivative approximation for the cosine function on a grid with 32 points covering the interval $[0, 2\pi]$.

```
In[3]:= h = 2 π / 32.;
        grid = h Range[32];
        err32 = Sin[grid] - ListCorrelate[{1, -1} / h, Cos[grid], {1, 1}]
Out[3]= {-0.0947283, -0.0879564, -0.0778045, -0.0646625, -0.0490356, -0.0315243, -0.0128016, 0.00641315,
         0.0253814, 0.0433743, 0.0597003, 0.0737321, 0.0849304, 0.0928648, 0.0972306, 0.0978598,
         0.0947283, 0.0879564, 0.0778045, 0.0646625, 0.0490356, 0.0315243, 0.0128016, -0.00641315,
         -0.0253814, -0.0433743, -0.0597003, -0.0737321, -0.0849304, -0.0928648, -0.0972306, -0.0978598}
```

It is quite apparent that the pointwise error is significantly less with a larger number of points.

The 2 norms of the vectors are of the same order of magnitude.

```
In[4]:= {Norm[err16, 2], Norm[err32, 2]}
Out[4]= {0.552985, 0.392279}
```

The norms of the vectors are comparable because is because the number of components in the vector has increased, so the usual linear algebra norm does not properly reflect the convergence. Normalizing by multiplying by $1/n^{1/p}$ reflects the convergence in the function space properly.

The normalized 2 norms of the vectors reflect the convergence to the actual function. Since the approximation is first order, doubling the number of grid points should approximately halve the error.

```
In[5]:= {Norm[err16, 2] / Sqrt[16], Norm[err32, 2] / Sqrt[32]}
Out[5]= {0.138246, 0.0693457}
```

Note that if you specify a function an option value, and you intend to use it for PDE or function approximation solutions, you should be sure to include a proper normalization in the function.

ScaledVectorNorm

Methods that have error control need to determine whether a step satisfies local error tolerances or not. To simplify the process of checking this, utility function `ScaledVectorNorm` does the scaling (1) and computes the norm. The table includes the formulas for specific values of p for reference.

`ScaledVectorNorm[p,{tr,ta}][v,u]`	compute the normalized p-norm of the vector v scaling using scaling (1) with reference vector u and relative and absolute tolerances t_a and t_r
`ScaledVectorNorm[fun,{tr,ta}][v,u]`	compute the norm of the vector v using scaling (1) with reference vector u and relative and absolute tolerances t_a and t_r and the norm function fun
`ScaledVectorNorm[2,{tr,ta}][v,u]`	compute $\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{\lvert v_i\rvert}{t_a+t_r\lvert u_i\rvert}\right)^2}$ where n is the length of vectors v and u
`ScaledVectorNorm[∞,{tr,ta}][v,u]`	compute $\max\left(\frac{\lvert v_i\rvert}{t_a+t_r\lvert u_i\rvert}\right)$, $1 \le i \le n$ where n is the length of vectors v and u

`ScaledVectorNorm`.

This sets up a scaled vector norm object with the default machine-precision tolerances used in `NDSolve`.

```
In[10]:= svn = NDSolve`ScaledVectorNorm[2, {10.^-8, 10.^-8}]
Out[10]= NDSolve`ScaledVectorNorm[2, {1.×10^-8, 1.×10^-8}]
```

This applies the scaled norm object with a sample error and solution reference vector.

```
In[11]:= svn[{9.×10.^-9, 10.^-8}, {2., 1.}]
Out[11]= 0.412311
```

Because of the absolute tolerance term, the value comes out reasonably even if some of the components of the reference solution are zero.

```
In[12]:= svn[{9.×10.^-9, 10.^-8, 2×10^-8}, {1., 0., 0.}]
Out[12]= 1.31688
```

When setting up a method for `NDSolve`, you can get the appropriate `ScaledVectorNorm` object to use using the "`Norm`" method function of the `NDSolve`StateData` object.

Here is an `NDSolve`StateData` object.

```
In[13]:= state =
          First[NDSolve`ProcessEquations[{x''[t] + x[t] == 0, x[0] == 1, x'[0] == 0}, x, t]]
Out[13]= NDSolve`StateData[<0.>]
```

This gets the appropriate scaled norm to use from the state data.

```
In[14]:= svn = state["Norm"]
Out[14]= NDSolve`ScaledVectorNorm[∞, {1.05367×10^-8, 1.05367×10^-8}, NDSolve]
```

This applies it to a sample error vector using the initial condition as reference vector.

```
In[15]:= svn[{10.^-9, 10.^-8}, state@"SolutionVector"["Forward"]]
Out[15]= 0.949063
```

Stiffness Detection

Overview

Many differential equations exhibit some form of stiffness which restricts the step-size and hence effectiveness of explicit solution methods.

A number of implicit methods have been developed over the years to circumvent this problem.

For the same step size, implicit methods can be substantially less efficient than explicit methods, due to the overhead associated with the intrinsic linear algebra.

This cost can offset by the fact that, in certain regions, implicit methods can take substantially larger step sizes.

Several attempts have been made to provide user-friendly codes that automatically attempt to detect stiffness at runtime and switch between appropriate methods as necessary.

A number of strategies that have been proposed to automatically equip a code with a stiffness detection device are outlined here.

Particular attention is given to the problem of estimation of the dominant eigenvalue of a matrix in order to describe how stiffness detection is implemented in `NDSolve`.

Numerical examples illustrate the effectiveness of the strategy.

Initialization

Load some packages with predefined examples and utility functions.

```
In[1]:= Needs["DifferentialEquations`NDSolveProblems`"];
        Needs["DifferentialEquations`NDSolveUtilities`"];
        Needs["FunctionApproximations`"];
```

Introduction

Consider the numerical solution of initial value problems:

$$y'(t) = f(t,\ y(t)),\ y(0) = y_0,\ f : \mathbb{R} \times \mathbb{R}^n \mapsto \mathbb{R}^n \tag{12}$$

Stiffness is a combination of problem, solution method, initial condition and local error tolerances.

Stiffness limits the effectiveness of explicit solution methods due to restrictions on the size of steps that can be taken.

Stiffness arises in many practical systems as well as in the numerical solution of partial differential equations by the method of lines.

Example

The van der Pol oscillator is a non-conservative oscillator with nonlinear damping and is an example of a stiff system of ordinary differential equations:

$$y_1'(t) = y_2(t),$$

$$\varepsilon\, y_2'(t) = -y_1(t) + \left(1 - y_1(t)^2\right) y_2(t),$$

with $\varepsilon = 3/1000$.

Consider initial conditions.

$$y_1(0) = 2,\ y_2(0) = 0$$

and solve over the interval $t \in [0, 10]$.

The method "StiffnessSwitching" uses a pair of extrapolation methods by default:

- Explicit modified midpoint (Gragg smoothing), double-harmonic sequence 2, 4, 6,…
- Linearly implicit Euler, sub-harmonic sequence 2, 3, 4,…

Solution

This loads the problem from a package.

```
In[4]:= system = GetNDSolveProblem["VanderPol"];
```

Solve the system numerically using a nonstiff method.

```
In[5]:= solns = NDSolve[system, {T, 0, 10}, Method → "Extrapolation"];
```

```
NDSolve::ndstf :
  At T == 0.022920104414210326`, system appears to be stiff. Methods Automatic, BDF or
        StiffnessSwitching may be more appropriate. ≫
```

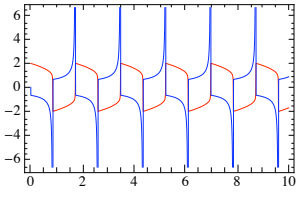
Solve the system using a method that switches when stiffness occurs.

```
In[6]:= sols = NDSolve[system, {T, 0, 10},
    Method → {"StiffnessSwitching", "NonstiffTest" -> False}];
```

Here is a plot of the two solution components. The sharp peaks (in blue) extend out to about 450 in magnitude and have been cropped.

```
In[7]:= Plot[Evaluate[Part[sols, 1, All, 2]], {T, 0, 10},
 PlotStyle -> {{Red}, {Blue}}, Axes -> False, Frame -> True]
```

Out[7]=

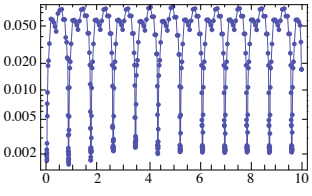


Stiffness can often occur in regions that follow rapid transients.

This plots the step sizes taken against time.

```
In[8]:= StepDataPlot[sols]
```

Out[8]=



The problem is that when the solution is changing rapidly, there is little point using a stiff solver, since local accuracy is the dominant issue.

For efficiency, it would be useful if the method could automatically detect regions where local accuracy (and not stability) is important.

Linear Stability

Linear stability theory arises from the study of Dahlquist's scalar linear test equation:

$$y'(t) = \lambda\, y(t), \quad \lambda \in \mathbb{C}, \quad \mathrm{Re}(\lambda) < 0 \tag{13}$$

as a simplified model for studying the initial value problem (12).

Stability is characterized by analyzing a method applied to (1) to obtain

$$y_{n+1} = R(z)\; y_n \tag{14}$$

where $z = h\,\lambda$ and $R(z)$ is the (rational) stability function.

The boundary of absolute stability is obtained by considering the region:

$$|R(z)| = 1$$

Explicit Euler Method

The explicit or forward Euler method:

$$y_{n+1} = y_n + h\, f(t_n, y_n)$$

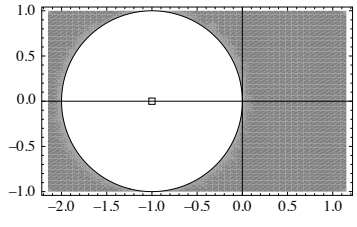
applied to (1) gives:

$$R(z) = 1 + z.$$

The shaded region represents instability, where $|R(z)| > 1$.

```
In[9]:= OrderStarPlot[1 + z, 1, z, FrameTicks -> True]
```

Out[9]=



The Linear Stability Boundary is often taken as the intersection with the negative real axis.

For the explicit Euler method LSB = -2.

For an eigenvalue of $\lambda = -1$, linear stability requirements mean that the step-size needs to satisfy $h < 2$, which is a very mild restriction.

However, for an eigenvalue of $\lambda = -10^6$, linear stability requirements mean that the step size needs to satisfy $h < 2\times10^{-6}$, which is a very severe restriction.

Example

This example shows the effect of stiffness on the step-size sequence when using an explicit Runge-Kutta method to solve a stiff system.

This system models a chemical reaction.

```
In[10]:= system = GetNDSolveProblem["Robertson"];
```

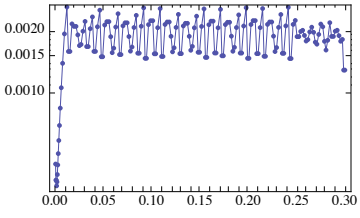
The system is solved by disabling the built-in stiffness detection.

```
In[11]:= sol = NDSolve[system, Method → {"ExplicitRungeKutta", "StiffnessTest" -> False}];
```

The step-size sequence starts to oscillate when the stability boundary is reached.

```
In[12]:= StepDataPlot[sol]
```

Out[12]=



- A large number of step rejections often has a negative impact on performance.
- The large number of steps taken adversely affects the accuracy of the computed solution.

The built-in detection does an excellent job of locating when stiffness occurs.

```
In[13]:= sol = NDSolve[system, Method → {"ExplicitRungeKutta", "StiffnessTest" -> True}];
```

```
NDSolve::ndstf:
  At T == 0.012555829610695773`, system appears to be stiff. Methods Automatic, BDF or
        StiffnessSwitching may be more appropriate. ≫
```

Implicit Euler Method

The implicit or backward Euler method:

$$y_{n+1} = y_n + h\, f(t_n,\ y_{n+1})$$

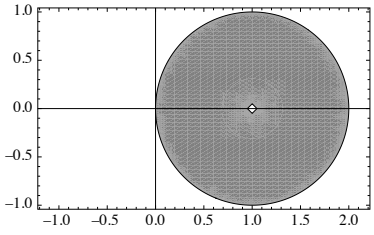
applied to (1) gives:

$$R(z) = \frac{1}{1 - z}$$

The method is unconditionally stable for the entire left half-plane.

```
In[14]:= OrderStarPlot[1 / (1 - z), 1, z, FrameTicks -> True]
```

Out[14]=



This means that to maintain stability there is no longer a restriction on the step size.

The drawback is that an implicit system of equations now has to be solved at each integration step.

Type Insensitivity

A type-insensitive solver recognizes and responds efficiently to stiffness at each step and so is insensitive to the (possibly changing) type of the problem.

One of the most established solvers of this class is LSODA [H83], [P83].

Later generations of LSODA such as CVODE no longer incorporate a stiffness detection device. The reason is because LSODA use norm bounds to estimate the dominant eigenvalue and these bounds, as will be seen later, can be quite inaccurate.

The low order of A(α)-stable BDF methods means that LSODA and CVODE are not very suitable for solving systems with high accuracy or systems where the dominant eigenvalue has a large

imaginary part. Alternative methods, such as those based on extrapolation of linearly implicit schemes, do not suffer from these issues.

Much of the work on stiffness detection was carried out in the 1980s and 1990s using standalone FORTRAN codes.

New linear algebra techniques and efficient software have since become available and these are readily accessible in *Mathematica*.

Stiffness can be a transient phenomenon, so detecting nonstiffness is equally important [S77], [B90].

"StiffnessTest" Method Option

There are several approaches that can be used to switch from a nonstiff to a stiff solver.

Direct Estimation

A convenient way of detecting stiffness is to directly estimate the dominant eigenvalue of the Jacobian J of the problem (see [S77], [P83], [S83], [S84a], [S84c], [R87] and [HW96]).

Such an estimate is often available as a by-product of the numerical integration and so it is reasonably inexpensive.

If v denotes an approximation to the eigenvector corresponding to dominant eigenvalue of the Jacobian, with $\|v\|$ sufficiently small, then by the mean value theorem a good approximation to the leading eigenvalue is:

$$\tilde{\lambda} = \frac{\|f(t, y+v) - f(t, y)\|}{\|v\|}.$$

Richardson's extrapolation provides a sequence of refinements that yields a quantity of this form, as do certain explicit Runge-Kutta methods.

Cost is at most two function evaluations, but often at least one of these is available as a by-product of the numerical integration, so it is reasonably inexpensive.

Let LSB denote the linear stability boundary—the intersection of the linear stability region with the negative real axis.

The product $h\tilde{\lambda}$ gives an estimate that can be compared to the linear stability boundary of a method in order to detect stiffness:

$$\left|h\tilde{\lambda}\right| \leq s\,|LSB| \tag{15}$$

where s is a safety factor.

Description

The methods "DoubleStep", "Extrapolation", and "ExplicitRungeKutta" have the option "StiffnessTest", which can be used to identify whether the method applied with the specified AccuracyGoal and PrecisionGoal tolerances to a given problem is stiff.

The method option "StiffnessTest" itself accepts a number of options that implement a weak form of (15) where the test is allowed to fail a specified number of times.

The reason for this is that some problems can be only mildly stiff in a certain region and an explicit integration method may still be efficient.

"NonstiffTest" Method Option

The "StiffnessSwitching" method has the option "NonstiffTest", which is used to switch back from a stiff method to a nonstiff method.

The following settings are allowed for the option "NonstiffTest"

- None or False (perform no test).
- "NormBound".
- "Direct".
- "SubspaceIteration".
- "KrylovIteration".
- "Automatic".

Switching to a Nonstiff Solver

An approach that is independent of the stiff method is used.

Given the Jacobian J (or an approximation) compute one of:

Norm Bound: $\| J \|$

Spectral Radius: $\rho(J) = \max |\lambda_i|$

Dominant Eigenvalue $\lambda_i : |\lambda_i| > |\lambda_j|$

Many linear algebra techniques focus on solving a single problem to high accuracy.

For stiffness detection, a succession of problems with solutions to one or two digits are adequate.

For a numerical discretization

$$0 = t_0 < t_1 < \cdots < t_n = T$$

consider a sequence k of matrices in some sub-interval(s)

$$J_{t_i},\ J_{t_{i+1}}, \ldots J_{t_{i+k-1}}$$

The spectra of the succession of matrices often changes very slowly from step to step.

The goal is to find a way of estimating (bounds on) dominant eigenvalues of a succession of matrices J_{t_i} that:

- Costs less than the work carried out in the linear algebra at each step in the stiff solver.
- Takes account of the step-to-step nature of the solver.

NormBound

A simple and efficient technique of obtaining a bound on the dominant eigenvalue is to use the norm of the Jacobian $\| J \|_p$ where typically $p = 1$ or $p = \infty$.

The method has complexity $O(n^2)$, which is less than the work carried out in the stiff solver.

This is the approach used by LSODA.

- Norm bounds for dense matrices overestimate and the bounds become worse as the dimension increases.
- Norm bounds can be tight for sparse or banded matrices of quite large dimension.

The setting "NormBound" of the option "NonstiffTest" computes $\|J\|_1$ and $\|J\|_\infty$ and returns the smaller of the two values.

Example

The following Jacobian matrix arises in the numerical solution of the van der Pol system using a stiff solver.

```
In[18]:= a = {{0., 1.}, {2623.532160943381, -69.56342161343568}};
```

Bounds based on norms overestimate the spectral radius by more than an order of magnitude.

```
In[19]:= {Abs[First[Eigenvalues[a]]], Norm[a, 1], Norm[a, Infinity]}
Out[19]= {96.6954, 2623.53, 2693.1}
```

Direct Eigenvalue Computation

For small problems ($n \leq 32$) it can be efficient just to compute the dominant eigenvalue directly.

- Hermitian matrices use the LAPACK function xgeev
- General matrices use the LAPACK function xsyevr

The setting "Direct" of the option "NonstiffTest" computes the dominant eigenvalue of J using the same LAPACK routines as Eigenvalues.

For larger problems the cost of direct eigenvalue computation is $O(n^3)$ which becomes prohibitive when compared to the cost of the linear algebra work in a stiff solver.

A number of iterative schemes have been implemented for this purpose. These effectively work by approximating the dominant eigenspace in a smaller subspace and using dense eigenvalue methods for the smaller problem.

The Power Method

Shampine has proposed the use of the power method for estimating the dominant eigenvalue of the Jacobian [S91].

The power method is perhaps not a very well-respected method, but has received a resurgence of interest due to its use in Google's page ranking.

The power method can be used when

- $A \in \mathbb{C}^{n \times n}$ has n linearly independent eigenvectors (diagonalizable)
- The eigenvalues can be ordered in magnitude as $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$
- λ_1 is the dominant eigenvalue of A.

Description

Given a starting vector $v_0 \in \mathbb{C}^n$ compute

$$v_k = A\, v_{k-1}$$

The Rayleigh quotient is used to compute an approximation to the dominant eigenvalue:

$$\lambda_1^{(k)} = \frac{v_{k-1}^* A\, v_{k-1}}{v_{k-1}^* v_{k-1}} = \frac{v_k^* v_{k-1}}{v_{k-1}^* v_{k-1}}$$

In practice, the approximate eigenvector is scaled at each step:

$$\hat{v}_k = \frac{v_k}{\| v_k \|}$$

Properties

The power method converges linearly with rate:

$$\left| \frac{\lambda_1}{\lambda_2} \right|$$

which can be slow.

In particular, the method does not converge when applied to a matrix with a dominant complex conjugate pair of eigenvalues.

Generalizations

The power method can be adapted to overcome the issue of equimodular eigenvalues (e.g. NAPACK)

However the modification does not generally address the issue of the slow rate of convergence for clustered eigenvalues.

There are two main approaches to generalizing the power method:

- Subspace iteration for small to medium dimensions
- Arnoldi iteration for large dimensions

Although the methods work quite differently, there are a number of core components that can be shared and optimized.

Subspace and Krylov iteration cost $O(n^2 m)$ operations.

They project an $n \times n$ matrix to an $m \times m$ matrix, where $m << n$.

The small matrix represents the dominant eigenspace and approximation uses dense eigenvalue routines.

Subspace Iteration

Subspace (or simultaneous) iteration generalizes the ideas in the power method by acting on m vectors at each step.

Start with an *orthonormal* set of vectors $V^{(0)} = \mathbb{C}^{n \times m}$, where usually $m << n$:

$$V^{(0)} = [v_1, \ldots, v_m]$$

Form the product with the matrix A:

$$Z^{(k)} = A V^{(k-1)}$$

In order to prevent all vectors from converging to multiples of the same dominant eigenvector v_1 of A, they are orthonormalized:

$$Q^{(k)} R^{(k)} = Z^{(k)} \quad \text{reduced QR factorization}$$

$$V^{(k)} = Q^{(k)}$$

The orthonormalization step is expensive compared to the matrix product.

Rayleigh-Ritz Projection

Input: matrix A and an orthonormal set of vectors V

- Compute the Rayleigh quotient $S = V^* A V$
- Compute the Schur decomposition $U^* S U = T$

The matrix S has small dimension $m \times m$.

Note that the Schur decomposition can be computed in real arithmetic when $S \in \mathbb{R}^{m \times m}$ using a quasi upper-triangular matrix T.

Convergence

Subspace (or simultaneous) iteration generalizes the ideas in the power method by acting on m vectors at each step.

SRRIT converges linearly with rate:

$$\left| \frac{\lambda_i}{\lambda_{m+1}} \right|, \; i = 1, \; \ldots, m$$

In particular the rate for the dominant eigenvalue is:

$$\left| \frac{\lambda_1}{\lambda_{m+1}} \right|$$

Therefore it can be beneficial to take e.g. $m = 3$ or more even if we are only interested in the dominant eigenvalue.

Error Control

A relative error test on successive approximations, dominant eigenvalue is:

$$\left| \frac{\lambda_1^{(k)} - \lambda_1^{(k-1)}}{\lambda_1^{(k)}} \right| \leq \text{tol}$$

This is not sufficient since it can be satisfied when convergence is slow.

If $|\lambda_i| = |\lambda_{i-1}|$ or $|\lambda_i| = |\lambda_{i+1}|$ then the i^{th} column of $Q^{(k)}$ is not uniquely determined.

The residual test used in SRRIT is:

$$r^{(k)} = A\,\hat{q}_i^{(k)} - \hat{Q}^{(k)}\,t_i^{(k)}, \qquad \left\| r^{(k)} \right\|_2 \leq \text{tol}$$

where $\hat{Q}^{(k)} = Q^{(k)}\,U^{(k)}$, $\hat{q}_i^{(k)}$ is the i^{th} column of $\hat{Q}^{(k)}$ and $t_i^{(k)}$ is the i^{th} column of $T^{(k)}$.

This is advantageous since it works for equimodular eigenvalues.

The first column position of the upper triangular matrix $T^{(k)}$ is tested because of the use of an ordered Schur decomposition.

Implementation

There are several implementations of subspace iteration.

- LOPSI [SJ81]
- Subspace iteration with Chebyshev acceleration [S84b], [DS93]
- Schur Rayleigh-Ritz iteration ([BS97] and [SLEPc05])

The implementation for use in "`NonstiffTest`" is based on:

- Schur Rayleigh-Ritz iteration [BS97]

"An attractive feature of SRRIT is that it displays monotonic consistency, that is, as the convergence tolerance decreases so does the size of the computed residuals" [LS96].

SRRIT makes use of an ordered Schur decomposition where eigenvalues of largest modulus appear in the upper-left entries.

Modified Gram-Schmidt with reorthonormalization is used to form $Q^{(k)}$, which is faster than Householder transformations.

The approximate dominant subspace $V_{t_i}^{(k)}$ at integration time t_i is used to start the iteration at the next integration step t_{i+1}:

$$V_{t_{i+1}}^{(0)} = V_{t_i}^{(k)}$$

KrylovIteration

Given an $n \times m$ matrix V whose columns v_i comprise an orthogonal basis of a given subspace $\mathcal{V}$:

$$V^T V = I \text{ and } \text{span}\{v_1, v_2, \ldots, v_m\} = \mathcal{V}$$

The Rayleigh-Ritz procedure consists of computing $H = V^T A V$ and solving the associated eigen-problem $H y_i = \theta_i y_i$.

The approximate eigenpairs of the original problem $\tilde{\lambda}_i$, $\tilde{x}_i$ satisfy $\tilde{\lambda} = \theta_i$ and $\tilde{x}_i = V y_i$, which are called Ritz values and Ritz vectors.

The process works best when the subspace $\mathcal{V}$ approximates an invariant subspace of A.

This process is effective when $\mathcal{V}$ is equal to the Krylov subspace associated with a matrix A and a given initial vector x as:

$$K_m(A, x) = \text{span}\{x, A x, A^2 x, \ldots, A^{m-1} x\}.$$

Description

The method of Arnoldi is a Krylov-based projection algorithm that computes an orthogonal basis of the Krylov subspace and produces a projected $m \times m$ matrix H with $\mathrm{m} << \mathrm{n}$.

Input: matrix A, the number of steps m, an initial vector v_1 of norm 1

Output: (V_m, H_m, f, β) with $\beta = \|f\|_2$

For $j = 1, 2, \ldots, m - 1$
    $w = A v_j$
    Orthogonalize w with respect to V_j to obtain $h_{i,j}$ for $i = 1, \ldots, j$
    $h_{j+1,j} = w$ (if $h_{j+1,j} = 0$ stop)
    $v_{j+1} = w / h_{j+1,j}$
end
$f = A v_m$
Orthogonalize f with respect to V_m to obtain $h_{i,m}$ for $i = 1, \ldots, m$
$\beta = \|f\|_2$

In the case of Arnoldi, H has an unreduced upper Hessenberg form (upper triangular with an additional nonzero subdiagonal).

Orthogonalization is usually carried out by means of a Gram-Schmidt procedure.

The quantities computed by the algorithm satisfy:

$$A\,V_m = V_m\,H_m + f\,e_m^*$$

The residual `f` gives an indication of proximity to an invariant subspace and the associated norm β indicates the accuracy of the computed Ritz pairs:

$$\left\|A\,\tilde{x}_i - \tilde{\lambda}_i\,\tilde{x}_i\right\|_2 = \left\|A\,V_m\,y_i - \theta_i\,V_m\,\tilde{x}_i\right\|_2 = \left\|\left(A\,V_m - V_m\,\tilde{x}_i\right)y_i\right\|_2 = \beta\left|e_m^*\,y_i\right|$$

Restarting

The Ritz pairs converge quickly if the initial vector x is rich in the direction of the desired eigenvalues.

When this is not the case then a restarting strategy is required in order to avoid excessive growth in both work and memory.

There are a several of strategies for restarting, in particular:

- Explicit restart — a new starting vector is a linear combination of a subset of the Ritz vectors.
- Implicit restart — a new starting vector is formed from the Arnoldi process combined with an implicitly shifted QR algorithm.

Explicit restart is relatively simple to implement, but implicit restart is more efficient since it retains the relevant eigeninformation of the larger problem. However implicit restart is difficult to implement in a numerically stable way.

An alternative which is much simpler to implement, but achieves the same effect as implicit restart, is a Krylov-Schur method [S01].

Implementation

A number of software implementations are available, in particular:

- ARPACK [ARPACK98]
- SLEPc [SLEPc05]

The implementation in "`NonstiffTest`" is based on Krylov-Schur Iteration.

Automatic Strategy

The "Automatic" setting uses an amalgamation of the methods as follows.

- For $n \leq 2*m$ direct eigenvalue computation is used. Either $m = \min(n, m_{\text{si}})$ or $m = \min(n, m_{\text{ki}})$ is used depending on which method is active.
- For $n > 2*m$ subspace iteration is used with a default basis size of $m_{\text{si}} = 8$. If the method succeeds then the resulting basis is used to start the method at the next integration step.
- If subspace iteration fails to converge after $\max_{\text{si}}$ iterations then the dominant vector is used to start the Krylov method with a default basis size of $m_{\text{ki}} = 16$. Subsequent integration steps use the Krylov method, starting with the resulting vector from the previous step.
- If Krylov iteration fails to converge after $\max_{\text{ki}}$ iterations then norm bounds are used for the current step. The next integration step will continue to try to use Krylov iteration.
- Since they are so inexpensive, norm bounds are always computed when subspace or Krylov iteration is used and the smaller of the absolute values is used.

Step Rejections

Caching of the time of evaluation ensures that the dominant eigenvalue estimate is not recomputed for rejected steps.

Stiffness detection is also performed for rejected steps since:

- Step rejections often occur for nonstiff solvers when working near the stability boundary
- Step rejections often occur for stiff solvers when resolving fast transients

Iterative Method Options

The iterative methods of "NonstiffTest" have options that can be modified:

```
In[20]:= Options[NDSolve`SubspaceIteration]
Out[20]= {BasisSize → Automatic, MaxIterations → Automatic, Tolerance → 1/10}

In[21]:= Options[NDSolve`KrylovIteration]
Out[21]= {BasisSize → Automatic, MaxIterations → Automatic, Tolerance → 1/10}
```

The default tolerance aims for just one correct digit, but often obtains substantially more accurate values—especially after a few successful iterations at successive steps.

The default values limiting the number of iterations are:

- For subspace iteration $\max_{\text{si}} = \max(25, n/(2\,m_{\text{si}}))$.
- For Krylov iteration $\max_{\text{ki}} = \max(50, n/m_{\text{ki}})$.

If these values are set too large then a convergence failure becomes too costly.

In difficult problems, it is better to share the work of convergence across steps. Since the methods effectively refine the basis vectors from the previous step, there is a reasonable chance of convergence in subsequent steps.

Latency and Switching

It is important to incorporate some form of latency in order to avoid a cycle where the "StiffnessSwitching" method continually tries to switch between stiff and nonstiff methods.

The options "MaxRepetitions" and "SafetyFactor" of "StiffnessTest" and "NonstiffTest" are used for this purpose.

The default settings allow switching to be quite reactive, which is appropriate for one-step integration methods.

- "StiffnessTest" is carried out at the end of a step with a nonstiff method. When either value of the option "MaxRepetitions" is reached, a step rejection occurs and the step is recomputed with a stiff method.
- "NonstiffTest" is preemptive. It is performed before a step is taken with a stiff solve using the Jacobian matrix from the previous step.

Examples

Van der Pol

Select an example system.

```
In[22]:= system = GetNDSolveProblem["VanderPol"];
```

StiffnessTest

The system is integrated successfully with the given method and the default option settings for "StiffnessTest".

```
In[23]:= NDSolve[system, Method → "ExplicitRungeKutta"]

Out[23]= {{Y1[T] → InterpolatingFunction[{{0., 2.5}}, <>][T],
   Y2[T] → InterpolatingFunction[{{0., 2.5}}, <>][T]}}
```

A longer integration is aborted and a message is issued when the stiffness test condition is not satisfied.

```
In[24]:= NDSolve[system, {T, 0, 10}, Method → "ExplicitRungeKutta"]

NDSolve::ndstf: At T == 4.353040548903924`, system appears to be stiff.
        Methods Automatic, BDF or StiffnessSwitching may be more appropriate.

Out[24]= {{Y1[T] → InterpolatingFunction[{{0., 4.35304}}, <>][T],
   Y2[T] → InterpolatingFunction[{{0., 4.35304}}, <>][T]}}
```

Using a unit safety factor and specifying that only one stiffness failure is allowed effectively gives a strict test. The specification uses the nested method option syntax.

```
In[25]:= NDSolve[system, Method → {"ExplicitRungeKutta",
    "StiffnessTest" → {True, "MaxRepetitions" → {1, 1}, "SafetyFactor" → 1} }]

NDSolve::ndstf:
  At T == 0.`, system appears to be stiff. Methods Automatic, BDF or StiffnessSwitching may
        be more appropriate.

Out[25]= {{Y1[T] → InterpolatingFunction[{{0., 0.}}, <>][T],
   Y2[T] → InterpolatingFunction[{{0., 0.}}, <>][T]}}
```

NonstiffTest

For such a small system, direct eigenvalue computation is used.

The example serves as a good test that the overall stiffness switching framework is behaving as expected.

Set up a function to monitor the switch between stiff and nonstiff methods and the step size taken. Data for the stiff and nonstiff solvers is put in separate lists by using a different tag for "Sow".

```
In[26]:= SetAttributes[SowSwitchingData, HoldFirst];

SowSwitchingData[told_, t_, method_NDSolve`StiffnessSwitching] :=
  (Sow[{t, t - told}, method["ActiveMethodPosition"]];
   told = t;);
```

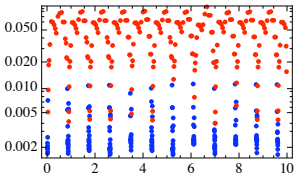
Solve the system and collect the data for the method switching.

```
In[28]:= T0 = 0;
         data =
           Last[
            Reap[
             sol = NDSolve[system, {T, 0, 10},
                 Method → StiffnessSwitching,
                 "MethodMonitor" :→ (SowSwitchingData[T0, T, NDSolve`Self];)
                ];
            ]
           ];
```

Plot the step sizes taken using an explicit solver (blue) and an implicit solver (red).

```
In[30]:= ListLogPlot[data, Axes → False, Frame → True, PlotStyle → {Blue, Red}]
```

Out[30]=



Compute the number of nonstiff and stiff steps taken (including rejected steps).

```
In[31]:= Map[Length, data]
Out[31]= {266, 272}
```

CUSP

The cusp catastrophe model for the nerve impulse mechanism [Z72]:

$$-\varepsilon\, y'(t) = y(t)^3 + a\, y(t) + b$$

Combining with the van der Pol oscillator gives rise to the CUSP system [HW96]:

$$\frac{\partial y}{\partial t} = -\frac{1}{\varepsilon}\left(y^3 + a\, y + b\right) + \sigma \frac{\partial^2 y}{\partial x^2}$$

$$\frac{\partial a}{\partial t} = -b + \frac{7}{100} v + \sigma \frac{\partial^2 a}{\partial x^2}$$

$$\frac{\partial b}{\partial t} = \left(1 - a^2\right) b - a - \frac{2}{5} y + \frac{7}{200} v + \sigma \frac{\partial^2 b}{\partial x^2}$$

where

$$v = \frac{u}{u - 1/10},\ u = (y - 7/10)(y - 13/10)$$

and $\sigma = 1/144$ and $\varepsilon = 10^{-4}$.

Discretization of the diffusion terms using the method of lines is used to obtain a system of ODEs of dimension $3n = 96$.

Unlike the van der Pol system, because of the size of the problem, iterative methods are used for eigenvalue estimation.

Step Size and Order Selection

Select the problem to solve.

```
In[32]:= system = GetNDSolveProblem["CUSP-Discretized"];
```

Set up a function to monitor the type of method used and step size. Additionally the order of the method is included as a Tooltip.

```
In[33]:= SetAttributes[SowOrderData, HoldFirst];

SowOrderData[told_, t_, method_NDSolve`StiffnessSwitching] :=
  (Sow[
    Tooltip[{t, t - told}, method["DifferenceOrder"]],
    method["ActiveMethodPosition"]
   ];
   told = t;);
```

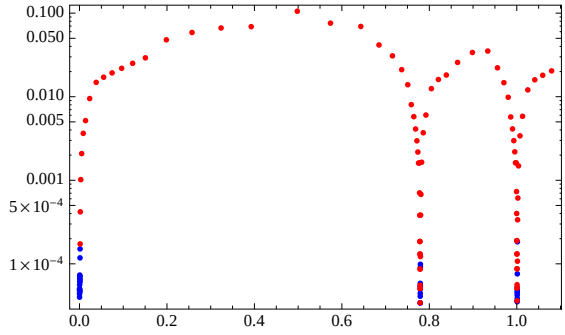
Collect the data for the order of the method as the integration proceeds.

```
In[35]:= T0 = 0;
data =
  Last[
   Reap[
    sol = NDSolve[system,
        Method → "StiffnessSwitching",
        "MethodMonitor" :→ (SowOrderData[T0, T, NDSolve`Self];)
       ];
   ]
  ];
```

Plot the step sizes taken using an explicit solver (blue) and an implicit solver (red). A Tooltip shows the order of the method at each step.

In[37]:=
```
ListLogPlot[data, Axes → False, Frame → True, PlotStyle → {Blue, Red}]
```

Out[37]=



Compute the total number of nonstiff and stiff steps taken (including rejected steps).

In[39]:=
```
Map[Length, data]
```

Out[39]=
```
{46, 120}
```

Jacobian Example

Define a function to collect the first few Jacobian matrices.

In[41]:=
```
SetAttributes[StiffnessJacobianMonitor, HoldFirst];

StiffnessJacobianMonitor[i_, method_NDSolve`StiffnessSwitching] :=
  If[SameQ[method["ActiveMethodPosition"], 2] && i < 5,
   If[MatrixQ[#],
       Sow[#];
       i = i + 1
     ] & @ method["Jacobian"]
  ];
```

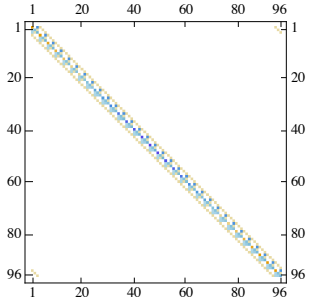
In[43]:=
```
i = 0;
jacdata = Reap[sol = NDSolve[system, Method → "StiffnessSwitching",
        "MethodMonitor" :→ (StiffnessJacobianMonitor[i, NDSolve`Self];)];
   ][[
   -1,
   1]];
```

A switch to a stiff method occurs near 0.00113425 and the first test for nonstiffness occurs at the next step $t_k \approx 0.00127887$.

Graphical illustration of the Jacobian J_{t_k}.

In[45]:=
```
MatrixPlot[First[jacdata]]
```

Out[45]=



Define a function to compute and display the first few eigenvalues of J_{t_k}, $J_{t_{k+1}}$,... and the norm bounds.

In[46]:=
```
DisplayJacobianData[jdata_] :=
  Module[{evdata, hlabels, vlabels},
   evdata =
    Map[
     Join[Eigenvalues[Normal[#], 4], {Norm[#, 1], Norm[#, Infinity]}] &, jdata];
   vlabels = {{""}, {"λ1"}, {"λ2"}, {"λ3"}, {"λ4"}, {"‖Jtk‖1"}, {"‖Jtk‖∞"}};
   hlabels = Table[Jtk, {k, Length[jdata]}];
   Grid[
    MapThread[Join, {vlabels, Join[{hlabels}, Transpose[evdata]]}], Frame → All]
  ];
```

In[47]:=
```
DisplayJacobianData[jacdata]
```

Out[47]=

	J_{t_1}	J_{t_2}	J_{t_3}	J_{t_4}	J_{t_5}
λ_1	-56 013.2	-56 009.7	-56 000.	-55 988.2	-55 959.6
λ_2	-56 007.9	-56 003.8	-55 992.2	-55 978.	-55 943.5
λ_3	-55 671.3	-55 670.7	-55 669.1	-55 667.1	-55 662.2
λ_4	-55 660.3	-55 658.3	-55 652.6	-55 645.7	-55 628.9
$\|J_{t_k}\|_1$	56 027.5	56 024.1	56 014.4	56 002.6	55 973.9
$\|J_{t_k}\|_\infty$	81 315.4	81 311.3	81 299.7	81 285.6	81 251.4

Norm bounds are quite sharp in this example.

Korteweg-deVries

The Korteweg-deVries partial differential equation is a mathematical model of waves on shallow water surfaces:

$$\frac{\partial U}{\partial t} + 6\,U\,\frac{\partial U}{\partial x} + \frac{\partial^3 U}{\partial x^3} = 0$$

We consider boundary conditions:

$$U(0,\ x) = e^{-x^2},\ U(t,\ -5) = U(t,\ 5)$$

and solve over the interval $t \in [0, 1]$.

Discretization using the method of lines is used to form a system of 192 ODEs.

Step Sizes

Select the problem to solve.

```
In[48]:= system = GetNDSolveProblem["Korteweg-deVries-PDE"];
```

The Backward Differentiation Formula methods used in LSODA run into difficulties solving this problem.

```
In[49]:= First[Timing[sollsoda = NDSolve[system, Method → LSODA];]]
```

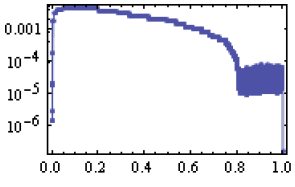
```
NDSolve::eerr :
  Warning: Scaled local spatial error estimate of 806.6079731642326` at T = 1.` in the direction of
      independent variable X is much greater than prescribed error tolerance. Grid
      spacing with 193 points may be too large to achieve the desired accuracy
      or precision.  A singularity may have formed or you may want to specify a
      smaller grid spacing using the MaxStepSize or MinPoints method options. ≫
```

```
Out[49]= 0.971852
```

A plot shows that the step sizes rapidly decrease.

```
In[50]:= StepDataPlot[sollsoda]
```

```
Out[50]=
```



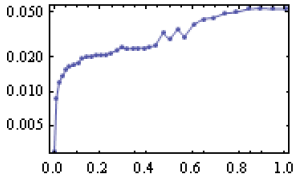
In contrast `StiffnessSwitching` immediately switches to using the linearly implicit Euler method which needs very few integration steps.

```
In[51]:= First[Timing[sol = NDSolve[system, Method -> "StiffnessSwitching"];]]
```

```
Out[51]= 0.165974
```

In[52]:=
```
StepDataPlot[sol]
```

Out[52]=



The extrapolation methods never switch back to a nonstiff solver once the stiff solver is chosen at the beginning of the integration.

Therefore this is a form of worst case example for the nonstiff detection.

Despite this, the cost of using subspace iteration is only a few percent of the total integration time.

Compute the time taken with switching to a nonstiff method disabled.

In[53]:=
```
First[Timing[sol = NDSolve[system,
      Method -> {"StiffnessSwitching", "NonstiffTest" -> False}];]]
```

Out[53]=
```
0.160974
```

Jacobian Example

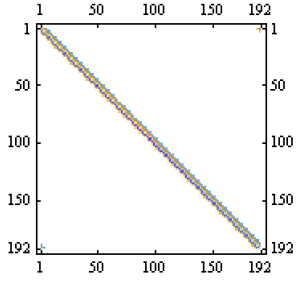
Collect data for the first few Jacobian matrices using the previously defined monitor function.

In[54]:=
```
i = 0;
jacdata = Reap[sol = NDSolve[system, Method → "StiffnessSwitching",
        "MethodMonitor" :→ (StiffnessJacobianMonitor[i, NDSolve`Self];)];
    ][[
    -1,
    1]];
```

Graphical illustration of the initial Jacobian J_{t_0}.

In[56]:=
```
MatrixPlot[First[jacdata]]
```

Out[56]=



Compute and display the first few eigenvalues of J_{t_k}, $J_{t_{k+1}}$,... and the norm bounds.

In[57]:=
```
DisplayJacobianData[jacdata]
```

Out[57]=

	J_{t_1}	J_{t_2}	J_{t_3}	J_{t_4}	J_{t_5}
λ_1	1.37916×10^{-8} + 32608. i	5.3745×10^{-6} + 32608. i	0.0000209094 + 32608. i	0.0000428279 + 32608. i	0.0000678117 + 32608.1 i
λ_2	1.37916×10^{-8} − 32608. i	5.3745×10^{-6} − 32608. i	0.0000209094 − 32608. i	0.0000428279 − 32608. i	0.0000678117 − 32608.1 i
λ_3	5.90398×10^{-8} + 32575.5 i	0.0000103621 + 32575.5 i	0.0000406475 + 32575.5 i	0.0000817789 + 32575.5 i	0.000125286 + 32575.6 i
λ_4	5.90398×10^{-8} − 32575.5 i	0.0000103621 − 32575.5 i	0.0000406475 − 32575.5 i	0.0000817789 − 32575.5 i	0.000125286 − 32575.6 i
$\|J_{t_k}\|_1$	38928.4	38928.4	38928.4	38930.	38932.9
$\|J_{t_k}\|_\infty$	38928.4	38928.4	38928.4	38930.1	38933.

Norm bounds overestimate slightly, but more importantly they give no indication of the relative size of real and imaginary parts.

Option Summary

StiffnessTest

option name	*default value*	
"MaxRepetitions"	{3,5}	specify the maximum number of successive and total times that the stiffness test (15) is allowed to fail
"SafetyFactor"	$\frac{4}{5}$	specify the safety factor to use in the right-hand side of the stiffness test (15)

Options of the method option "StiffnessTest".

NonstiffTest

option name	default value	
"MaxRepetitions"	{2, ∞}	specify the maximum number of successive and total times that the stiffness test (15) is allowed to fail
"SafetyFactor"	$\frac{4}{5}$	specify the safety factor to use in the right-hand side of the stiffness test (15)

Options of the method option "NonstiffTest".

Structured Systems

Numerical Methods for Solving the Lotka-Volterra Equations

Introduction

The Lotka-Volterra system arises in mathematical biology and models the growth of animal species. Consider two species where $Y_1(T)$ denotes the number of predators and $Y_2(T)$ denotes the number of prey. A particular case of the Lotka-Volterra differential system is:

$$\dot{Y}_1 = Y_1 (Y_2 - 1), \quad \dot{Y}_2 = Y_2 (2 - Y_1), \tag{1}$$

where the dot denotes differentiation with respect to time T.

The Lotka-Volterra system (9) has an invariant H, which is constant for all T:

$$H(Y_1, Y_2) = 2 \ln Y_1 - Y_1 + \ln Y_2 - Y_2. \tag{2}$$

The level curves of the invariant (2) are closed so that the solution is periodic. It is desirable that the numerical solution of (9) is also periodic, but this is not always the case. Note that (9) is a Poisson system:

$$\dot{Y} = B(Y)\nabla H(Y) = \begin{pmatrix} 0 & -Y_1 Y_2 \\ Y_1 Y_2 & 0 \end{pmatrix} \begin{pmatrix} \frac{2}{Y_1} - 1 \\ \frac{1}{Y_2} - 1 \end{pmatrix} \tag{3}$$

where $H(Y)$ is defined in (2).

Poisson systems and Poisson integrators are discussed in Chapter VII.2 of [HLW02] and [MQ02].

Load a package with some predefined problems and select the Lotka-Volterra system.

```
In[10]:= Needs["DifferentialEquations`NDSolveProblems`"];
         Needs["DifferentialEquations`NDSolveUtilities`"];
         Needs["DifferentialEquations`InterpolatingFunctionAnatomy`"];

         system = GetNDSolveProblem["LotkaVolterra"];
         invts = system["Invariants"];
         time = system["TimeData"];
         vars = system["DependentVariables"];
         step = 3 / 25;
```

Define a utility function for visualizing solutions.

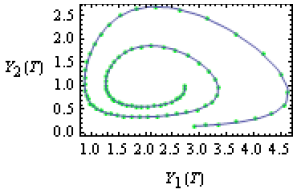
```
In[18]:= LotkaVolterraPlot[sol_, vars_, time_, opts___?OptionQ] :=
           Module[{data, data1, data2, ifuns, lplot, pplot},
            ifuns = First[vars /. sol];
            data1 = Part[ifuns, 1, 0]["ValuesOnGrid"];
            data2 = Part[ifuns, 2, 0]["ValuesOnGrid"];
            data = Transpose[{data1, data2}];
            commonopts = Sequence[Axes → False, Frame → True, FrameLabel →
                Join[Map[TraditionalForm, vars], {None, None}], RotateLabel → False];
            lplot = ListPlot[data, Evaluate[FilterRules[{opts}, Options[ListPlot]]],
              PlotStyle → {PointSize[0.02], RGBColor[0, 1, 0]}, Evaluate[commonopts]];
            pplot = ParametricPlot[Evaluate[ifuns], time, Evaluate[
                FilterRules[{opts}, Options[ParametricPlot]]], Evaluate[commonopts]];
            Show[lplot, pplot]
           ];
```

Explicit Euler

Use the explicit or forward Euler method to solve the system (9).

In[19]:=
```
fesol = NDSolve[system, Method → "ExplicitEuler", StartingStepSize → step];

LotkaVolterraPlot[fesol, vars, time]
```

Out[20]=

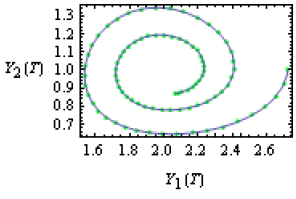


Backward Euler

Define the backward or implicit Euler method in terms of the RadauIIA implicit Runge-Kutta method and use it to solve (9). The resulting trajectory spirals from the initial conditions toward a fixed point at $(2, 1)$ in a clockwise direction.

In[21]:=
```
BackwardEuler = {"FixedStep", Method → {"ImplicitRungeKutta", "Coefficients" →
      "ImplicitRungeKuttaRadauIICoefficients", "DifferenceOrder" → 1,
    "ImplicitSolver" → {"FixedPoint", AccuracyGoal → MachinePrecision,
      PrecisionGoal → MachinePrecision, "IterationSafetyFactor" → 1}}};

besol = NDSolve[system, Method → BackwardEuler, StartingStepSize → step];

LotkaVolterraPlot[besol, vars, time]
```

Out[23]=

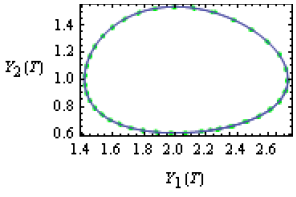


Projection

Projection of the forward Euler method using the invariant (2) of the Lotka-Volterra equations gives a periodic solution.

In[24]:=
```
pfesol = NDSolve[system,
    Method → {Projection, Method → "ExplicitEuler", Invariants → invts},
    StartingStepSize → step];

LotkaVolterraPlot[pfesol, vars, time]
```

Out[25]=



Splitting

Another approach for obtaining the correct qualitative behavior is to additively split (9) into two systems:

$$\begin{aligned} \dot{Y}_1 &= Y_1 (Y_2 - 1) & \dot{Y}_2 &= 0 \\ \dot{Y}_1 &= 0 & \dot{Y}_2 &= Y_2 (2 - Y_1). \end{aligned} \tag{4}$$

By appropriately solving (4) it is possible to construct Poisson integrators.

Define the equations for splitting of the Lotka-Volterra equations.

In[26]:=
```
eqs = system["System"];
Y1 = eqs;
Part[Y1, 2, 2] = 0;
Y2 = eqs;
Part[Y2, 1, 2] = 0;
```

Symplectic Euler

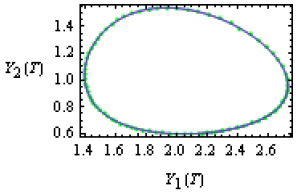
Define the symplectic Euler method in terms of a splitting method using the backward and forward Euler methods for each system in (4).

In[31]:=
```
SymplecticEuler = {"Splitting",
    "DifferenceOrder" → 1, "Equations" → {Y1, Y2},
   "Method" → {BackwardEuler, "ExplicitEuler"}};

sesol = NDSolve[system, Method → SymplecticEuler, StartingStepSize → step];
```

The numerical solution using the symplectic Euler method is periodic.

```
In[33]:= LotkaVolterraPlot[sesol, vars, time]
```

Out[33]=



Flows

Consider splitting the Lotka-Volterra equations and computing the flow (or exact solution) of each system in (4). The solutions can be found as follows, where the constants should be related to the initial conditions at each step.

```
In[34]:= DSolve[Y1, vars, T]
```

Out[34]= $\{\{Y_2[T] \to C[1],\ Y_1[T] \to e^{T(-1+C[1])}\, C[2]\}\}$

```
In[35]:= DSolve[Y2, vars, T]
```

Out[35]= $\{\{Y_1[T] \to C[1],\ Y_2[T] \to e^{T(2-C[1])}\, C[2]\}\}$

An advantage of locally computing the flow is that it yields an *explicit,* and hence very efficient, integration procedure. The "LocallyExact" method provides a general way of computing the flow of each splitting using DSolve only during the initialization phase.

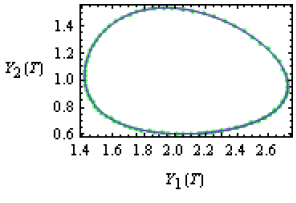
Set up a hybrid symbolic-numeric splitting method and use it to solve the Lotka-Volterra system.

```
In[36]:= SplittingLotkaVolterra = {"Splitting",
           "DifferenceOrder" → 1, "Equations" → {Y1, Y2},
           "Method" → {"LocallyExact", "LocallyExact"}};

         spsol = NDSolve[system, Method → SplittingLotkaVolterra, StartingStepSize → step];
```

The numerical solution using the splitting method is periodic.

```
In[38]:= LotkaVolterraPlot[spsol, vars, time]
```

Out[38]=



Rigid Body Solvers

Introduction

The equations of motion for a free rigid body whose center of mass is at the origin are given by the following Euler equations (see [MR99]).

$$\begin{pmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \end{pmatrix} = \begin{pmatrix} 0 & y_3/I_3 & -y_2/I_2 \\ -y_3/I_3 & 0 & y_1/I_1 \\ y_2/I_2 & -y_1/I_1 & 0 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Two quadratic first integrals of the system are:

$$\begin{aligned} I(y) &= y_1^2 + y_2^2 + y_3^2 \\ H(y) &= \frac{1}{2}\left(\frac{y_1^2}{I_1} + \frac{y_2^2}{I_2} + \frac{y_3^2}{I_3}\right). \end{aligned}$$

The first constraint effectively confines the motion from $\mathbb{R}^3$ to a sphere. The second constraint represents the kinetic energy of the system and, in conjunction with the first invariant, effectively confines the motion to ellipsoids on the sphere.

Numerical experiments for various methods are given in [HLW02] and a variety of `NDSolve` methods will now be compared.

Manifold Generation and Utility Functions

Load some useful packages.

In[6]:=
```
Needs["DifferentialEquations`NDSolveProblems`"];
Needs["DifferentialEquations`NDSolveUtilities`"];
```

Define Euler's equations for rigid body motion together with the invariants of the system.

In[8]:=
```
system = GetNDSolveProblem["RigidBody"];
eqs = system["System"];
vars = system["DependentVariables"];
time = system["TimeData"];
invariants = system["Invariants"];
```

The equations of motion evolve as closed curves on the unit sphere. This generates a three-dimensional graphics object to represent the unit sphere.

In[13]:=
```
UnitSphere = Graphics3D[{EdgeForm[], Sphere[]}, Boxed → False];
```

This function superimposes a solution from NDSolve on a given manifold.

```
In[14]:= PlotSolutionOnManifold[sol_, vars_, time_, manifold_, opts___?OptionQ] :=
          Module[{solplot},
           solplot = ParametricPlot3D[
             Evaluate[vars /. sol], time, opts, Boxed → False, Axes → False];
           Show[solplot, manifold, opts]
          ]
```

This function plots the various solution components.

```
In[15]:= PlotSolutionComponents[sols_, vars_, time_, opts___?OptionQ] :=
           Module[{ifuns, plotopts},
            ifuns = vars /. First[sols];
            Table[plotopts = Sequence[PlotLabel →
                 StringForm["`1` vs time", Part[vars, i]], Frame → True, Axes → False];
             Plot[Evaluate[Part[ifuns, i]], time, opts, Evaluate[plotopts]],
             {i, Length[vars]}]
           ];
```

Method Comparison

Various integration methods can be used to solve Euler's equations and they each have different associated costs and different dynamical properties.

Adams Multistep Method

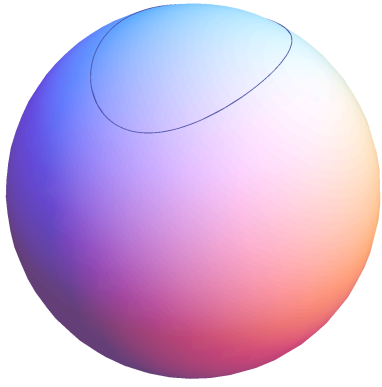
Here an Adams method is used to solve the equations of motion.

```
In[21]:= AdamsSolution = NDSolve[system, Method → "Adams"];
```

This shows the solution trajectory by superimposing it on the unit sphere.

In[22]:=
```
PlotSolutionOnManifold[AdamsSolution, vars, time, UnitSphere, PlotRange → All]
```

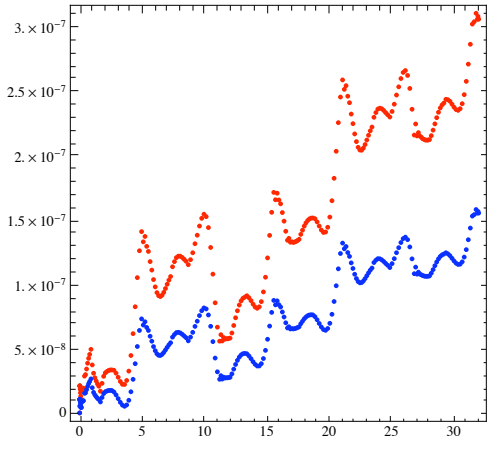
Out[22]=



The solution appears visually to give a closed curve on the sphere. However, a plot of the error reveals that neither constraint is conserved particularly well.

In[23]:=
```
InvariantErrorPlot[invariants, vars, T, AdamsSolution, PlotStyle → {Red, Blue}]
```

Out[23]=



Euler and Implicit Midpoint Methods

This solves the equations of motion using Euler's method with a specified fixed step size.

In[16]:=
```
EulerSolution = NDSolve[system,
   Method → {"FixedStep", Method → "ExplicitEuler"}, StartingStepSize → 1 / 20];
```

This solves the equations of motion using the implicit midpoint method with a specified fixed step size.

In[17]:=
```
ImplicitMidpoint = {"FixedStep", Method → {"ImplicitRungeKutta",
      "Coefficients" → "ImplicitRungeKuttaGaussCoefficients", DifferenceOrder → 2,
      "ImplicitSolver" → {FixedPoint, "AccuracyGoal" → MachinePrecision,
        "PrecisionGoal" → MachinePrecision, "IterationSafetyFactor" → 1}}};

IMPSolution =
  NDSolve[system, Method → ImplicitMidpoint, StartingStepSize → 3 / 10];
```

This shows the superimposition on the unit sphere of the numerical solution of the equations of motion for Euler's method (left) and the implicit midpoint rule (right).

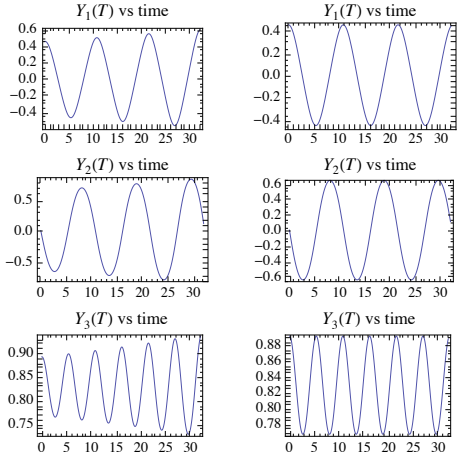
In[19]:=
```
EulerPlotOnSphere =
  PlotSolutionOnManifold[EulerSolution, vars, time, UnitSphere, PlotRange → All];

IMPPlotOnSphere =
  PlotSolutionOnManifold[IMPSolution, vars, time, UnitSphere, PlotRange → All];

GraphicsArray[{EulerPlotOnSphere, IMPPlotOnSphere}]
```

Out[21]=

This shows the components of the numerical solution using Euler's method (left) and the implicit midpoint rule (right).

In[30]:=
```
EulerSolutionPlots = PlotSolutionComponents[EulerSolution, vars, time];

IMPSolutionPlots = PlotSolutionComponents[IMPSolution, vars, time];

GraphicsArray[Transpose[{EulerSolutionPlots, IMPSolutionPlots}]]
```

Out[32]=



Orthogonal Projection Method

Here the "OrthogonalProjection" method is used to solve the equations.

```
In[33]:= OPSolution = NDSolve[system, Method → {"OrthogonalProjection",
           Dimensions → {3, 1}, Method → "ExplicitEuler"}, StartingStepSize → 1 / 20];
```

Only the orthogonal constraint is conserved so the curve is not closed.

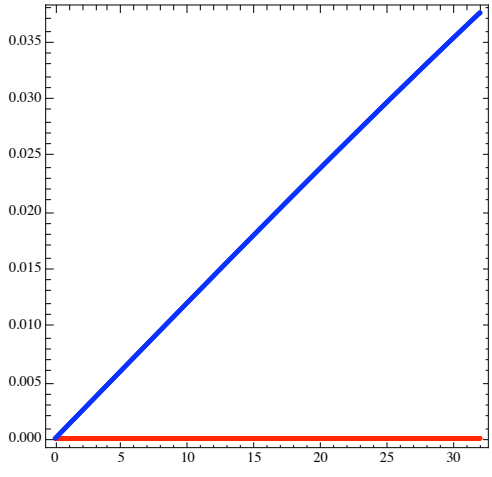
```
In[34]:= PlotSolutionOnManifold[OPSolution, vars, time, UnitSphere, PlotRange → All]
```

Out[34]=

Plotting the error in the invariants against time, it can be seen that the orthogonal projection method conserves only one of the two invariants.

```
In[35]:= InvariantErrorPlot[invariants, vars, T, OPSolution, PlotStyle → {Red, Blue}]
```

Out[35]=



Projection Method

The method "Projection" takes a set of constraints and projects the solution onto a manifold at the end of each integration step.

Generally all the invariants of the problem should be used in the projection; otherwise the numerical solution may actually be qualitatively worse than the unprojected solution.

The following specifies the integration method and defers determination of the constraints until the invocation of `NDSolve`.

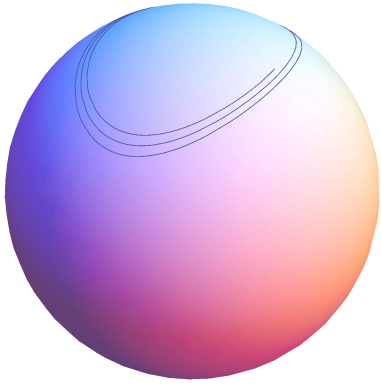
In[36]:=
```
ProjectionMethod = {Projection,
    Method → {"FixedStep", Method → "ExplicitEuler"}, "Invariants" :→ invts};
```

Projecting One Constraint

This projects the first constraint onto the manifold.

In[37]:=
```
invts = First[invariants];

projsol1 = NDSolve[system, Method → ProjectionMethod, StartingStepSize → 1 / 20];

PlotSolutionOnManifold[projsol1, vars, time, UnitSphere, PlotRange → All]
```

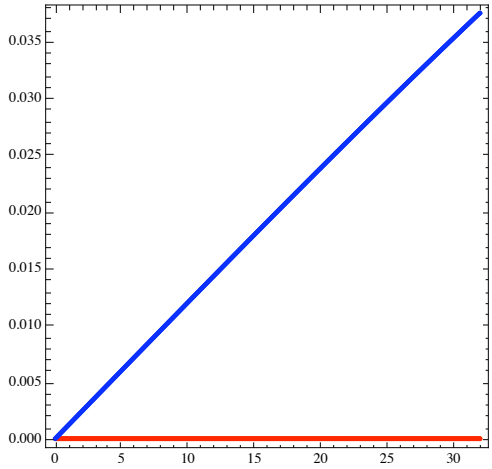
Out[39]=



Only the first invariant is conserved.

In[40]:=
```
InvariantErrorPlot[invariants, vars, T, projsol1, PlotStyle → {Red, Blue}]
```

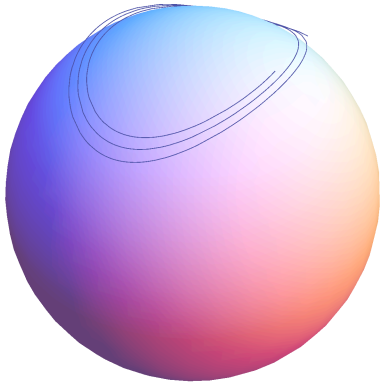
Out[40]=



This projects the second constraint onto the manifold.

In[41]:=
```
invts = Last[invariants];
projsol2 = NDSolve[system, Method → ProjectionMethod, StartingStepSize → 1 / 20];
PlotSolutionOnManifold[projsol2, vars, time, UnitSphere, PlotRange → All]
```

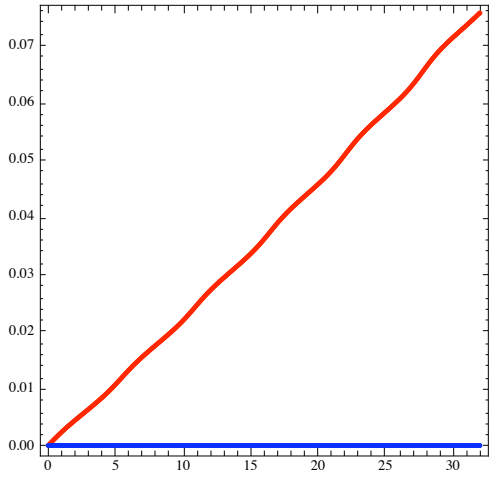
Out[43]=



Only the second invariant is conserved.

In[44]:=
```
InvariantErrorPlot[invariants, vars, T, projsol2, PlotStyle → {Red, Blue}]
```

Out[44]=



Projecting Multiple Constraints

This projects both constraints onto the manifold.

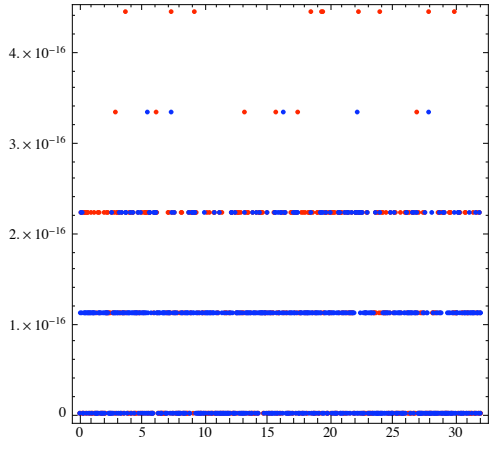
In[45]:=
```
invts = invariants;

projsol = NDSolve[system, Method → ProjectionMethod, StartingStepSize → 1 / 20];

PlotSolutionOnManifold[projsol, vars, time, UnitSphere, PlotRange → All]
```

Out[47]=

Now both invariants are conserved.

In[48]:=
```
InvariantErrorPlot[invariants, vars, T, projsol, PlotStyle → {Red, Blue}]
```

Out[48]=



"Splitting" Method

A splitting that yields an efficient explicit integration method was derived independently by McLachlan [M93] and Reich [R93].

Write the flow of an ODE $\dot{y} = Y$ as $y(t) = \exp(t\,Y)\,(y(0))$.

The differential system is split into three components, $YH1$, $YH2$, and $YH3$, each of which is Hamiltonian and can be solved exactly.

The Hamiltonian systems are solved and recombined at each integration step as:

$$\exp(t\,Y) \approx \exp(1/2\,t\,YH1)\exp(1/2\,t\,YH2)\exp(t\,YH3)\exp(1/2\,t\,YH2)\exp(1/2\,t\,YH1).$$

This defines an appropriate splitting into Hamiltonian vector fields.

```
In[49]:= Grad[H_, x_?VectorQ] := Map[D[H, #] &, x];
         isub = {I1 -> 2, I2 -> 1, I3 -> 2 / 3};
         H1 = Y1[T]^2 / (2 I1) /. isub;
         H2 = Y2[T]^2 / (2 I2) /. isub;
         H3 = Y3[T]^2 / (2 I3) /. isub;
         JX = {{0, -Y3[T], Y2[T]}, {Y3[T], 0, -Y1[T]}, {-Y2[T], Y1[T], 0}};
         YH1 = Thread[D[vars, T] == JX.Grad[H1, vars]];
         YH2 = Thread[D[vars, T] == JX.Grad[H2, vars]];
         YH3 = Thread[D[vars, T] == JX.Grad[H3, vars]];
```

Here is the differential system for Euler's equations.

```
In[58]:= eqs
```

$$\text{Out[58]}= \left\{Y_1'[T] == \frac{1}{2} Y_2[T]\, Y_3[T],\ Y_2'[T] == -Y_1[T]\, Y_3[T],\ Y_3'[T] == \frac{1}{2} Y_1[T]\, Y_2[T]\right\}$$

Here are the three split vector fields.

```
In[59]:= YH1
```

$$\text{Out[59]}= \left\{Y_1'[T] == 0,\ Y_2'[T] == \frac{1}{2} Y_1[T]\, Y_3[T],\ Y_3'[T] == -\frac{1}{2} Y_1[T]\, Y_2[T]\right\}$$

```
In[60]:= YH2
```

$$\text{Out[60]}= \{Y_1'[T] == -Y_2[T]\, Y_3[T],\ Y_2'[T] == 0,\ Y_3'[T] == Y_1[T]\, Y_2[T]\}$$

```
In[61]:= YH3
```

$$\text{Out[61]}= \left\{Y_1'[T] == \frac{3}{2} Y_2[T]\, Y_3[T],\ Y_2'[T] == -\frac{3}{2} Y_1[T]\, Y_3[T],\ Y_3'[T] == 0\right\}$$

Solution

This defines a symmetric second-order splitting method. The coefficients are automatically determined from the structure of the equations and are an extension of the Strang splitting.

```
In[62]:= SplittingMethod =
           {"Splitting",
            "DifferenceOrder" → 2,
            "Equations" → {YH1, YH2, YH3, YH2, YH1},
            "Method" → {"LocallyExact"}};
```

This solves the system and graphically displays the solution.

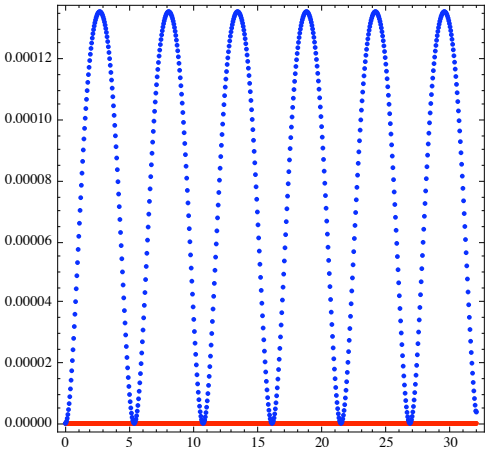
In[63]:=
```
splitsol = NDSolve[system, Method → SplittingMethod, StartingStepSize → 1 / 20];
PlotSolutionOnManifold[splitsol, vars, time, UnitSphere, PlotRange → All]
```

Out[64]=

One of the invariants is preserved up to roundoff while the error in the second invariant remains bounded.

In[65]:=
```
InvariantErrorPlot[invariants, vars, T, splitsol, PlotStyle → {Red, Blue}]
```

Out[65]=



Components and Data Structures in NDSolve

Introduction

NDSolve is broken up into several basic steps. For advanced usage, it can sometimes be advantageous to access components to carry out each of these steps separately.

- Equation processing and method selection
- Method initialization
- Numerical solution
- Solution processing

NDSolve performs each of these steps internally, hiding the details from a casual user. However, for advanced usage it can sometimes be advantageous to access components to carry out each of these steps separately.

Here are the low-level functions that are used to break up these steps.

- NDSolve`ProcessEquations
- NDSolve`Iterate
- NDSolve`ProcessSolutions

NDSolve`ProcessEquations classifies the differential system into initial value problem, boundary value problem, differential-algebraic problem, partial differential problem, etc. It also chooses appropriate default integration methods and constructs the main NDSolve`StateData data structure.

NDSolve`Iterate advances the numerical solution. The first invocation (there can be several) initializes the numerical integration methods.

NDSolve`ProcessSolutions converts numerical data into an InterpolatingFunction to represent each solution.

Note that `NDSolve`ProcessEquations` can take a significant portion of the overall time to solve a differential system. In such cases, it can be useful to perform this step only once and use `NDSolve`Reinitialize` to repeatedly solve for different options or initial conditions.

Example

Process equations and set up data structures for solving the differential system.

```
In[1]:= ndssdata =
          First[NDSolve`ProcessEquations[{y''[t] + y[t] == 0, y[0] == 1, y'[0] == 0},
            {y, y'}, t, Method → "ExplicitRungeKutta"]]
Out[1]= NDSolve`StateData[<0.>]
```

Initialize the method "ExplicitRungeKutta" and integrate the system up to time 10. The return value of `NDSolve`Iterate` is `Null` in order to avoid extra references, which would lead to undesirable copying.

```
In[2]:= NDSolve`Iterate[ndssdata, 10]
```

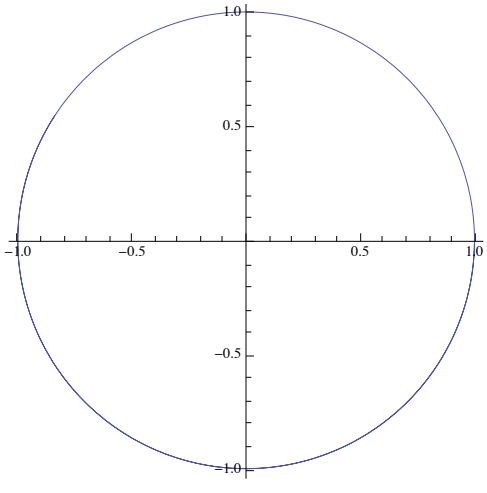
Convert each set of solution data into an `InterpolatingFunction`.

```
In[3]:= ndsol = NDSolve`ProcessSolutions[ndssdata]
Out[3]= {y → InterpolatingFunction[{{0., 10.}}, <>], y' → InterpolatingFunction[{{0., 10.}}, <>]}
```

Representing the solution as an `InterpolatingFunction` allows continuous output even for points that are not part of the numerical solution grid.

```
In[4]:= ParametricPlot[{y[t], y'[t]} /. ndsol, {t, 0, 10}]
```

Out[4]=



Creating NDSolve`StateData Objects

ProcessEquations

The first stage of any solution using NDSolve is processing the equations specified into a form that can be efficiently accessed by the actual integration algorithms. This stage minimally involves determining the differential order of each variable, making substitutions needed to get a first-order system, solving for the time derivatives of the functions in terms of the functions, and forming the result into a "NumericalFunction" object. If you want to save the time of repeating this process for the same set of equations or if you want more control over the numerical integration process, the processing stage can be executed separately with NDSolve`ProcessEquations.

NDSolve`ProcessEquations[{eqn_1, eqn_2, ...}, {u_1, u_2, ...}, t]

process the differential equations {eqn_1, eqn_2, ...} for the functions {u_1, u_2, ...} into a normal form; return a list of NDSolve`StateData objects containing the solution and data associated with each solution for the time derivatives of the functions in terms of the functions; t may be specified in a list with a range of values as in NDSolve

NDSolve`ProcessEquations[{eqn_1, eqn_2, ...}, {u_1, u_2, ...}, {x_1, $x_{1\,min}$, $x_{1\,max}$}, {x_2, $x_{2\,min}$, $x_{2\,max}$}, ...]

process the partial differential equations {eqn_1, eqn_2, ...} for the functions {u_1, u_2, ...} into a normal form; return a list of NDSolve`StateData objects containing the solution and data associated with each solution for the time derivatives of the functions in terms of the functions; if x_j is the temporal variable, it need not be specified with the boundaries $x_{j\,min}$, $x_{j\,max}$

Processing equations for NDSolve.

This creates a list of two NDSolve`StateData objects because there are two possible solutions for the y' in terms of y.

```
In[1]:= NDSolve`ProcessEquations[{y'[x]^2 == y[x] + x, y[0] == 1}, y, x]
Out[1]= {NDSolve`StateData[<0.>], NDSolve`StateData[<0.>]}
```

Reinitialize

It is not uncommon that the solution to a more sophisticated problem involves solving the same differential equation repeatedly, but with different initial conditions. In some cases, processing equations may be as time-consuming as numerically integrating the differential equations. In these situations, it is a significant advantage to be able to simply give new initial values.

NDSolve`Reinitialize[*state*, *conditions*]	assuming the equations and variables are the same as the ones used to create the NDSolve`StateData object *state*, form a list of new NDSolve`StateData objects, one for each of the possible solutions for the initial values of the functions of the equations *conditions*

Reusing processed equations.

This creates an NDSolve`StateData object for the harmonic oscillator.

```
In[2]:= state =
         First[NDSolve`ProcessEquations[{x''[t] + x[t] == 0, x[0] == 0, x'[0] == 1}, x, t]]
Out[2]= NDSolve`StateData[<0.>]
```

This creates three new NDSolve`StateData objects, each with a different initial condition.

```
In[3]:= newstate = NDSolve`Reinitialize[state, {x[1]^3 == 1, x'[1] == 0}]
Out[3]= {NDSolve`StateData[<1.>], NDSolve`StateData[<1.>], NDSolve`StateData[<1.>]}
```

Using NDSolve`Reinitialize may save computation time when you need to solve the same differential equation for many different initial conditions, as you might in a shooting method for boundary value problems.

A subset of NDSolve options can be specified as options to NDSolve`Reinitialize.

This creates a new NDSolve`StateData object, specifying a starting step size.

```
In[3]:= newstate =
         NDSolve`Reinitialize[state, {x[0] == 0, x'[0] == 1}, StartingStepSize → 1 / 10]
Out[3]= {NDSolve`StateData[<0.>]}
```

Iterating Solutions

One important use of NDSolve`StateData objects is to have more control of the integration. For some problems, it is appropriate to check the solution and start over or change parameters, depending on certain conditions.

NDSolve`Iterate[*state*,*t*]	compute the solution of the differential equation in an NDSolve`StateData object that has been assigned as the value of the variable *state* from the current time up to time *t*

Iterating solutions to differential equations.

This creates an NDSolve`StateData object that contains the information needed to solve the equation for an oscillator with a varying coefficient using an explicit Runge-Kutta method.

```
In[4]:= state =
         First[NDSolve`ProcessEquations[{x''[t] + (1 + 4 UnitStep[Sin[t]]) x[t] == 0,
             x[0] == 1, x'[0] == 0}, x, t, Method → "ExplicitRungeKutta"]]
Out[4]= NDSolve`StateData[<0.>]
```

Note that when you use NDSolve`ProcessEquations, you do not need to give the range of the t variable explicitly because that information is not needed to set up the equations in a form ready to solve. (For PDEs, you do have to give the ranges of all spatial variables, however, since that information is essential for determining an appropriate discretization.)

This computes the solution out to time $t = 1$.

```
In[5]:= NDSolve`Iterate[state, 1]
```

NDSolve`Iterate does not return a value because it modifies the NDSolve`StateData object assigned to the variable state. Thus, the command affects the value of the variable in a manner similar to setting parts of a list, as described in "Manipulating Lists by Their Indices". You can see that the value of *state* has changed since it now displays the current time to which it is integrated.

The output form of *state* shows the range of times over which the solution has been integrated.

```
In[6]:= state
Out[6]= NDSolve`StateData[<0.,1.>]
```

If you want to integrate further, you can call `NDSolve`Iterate` again, but with a larger value for time.

This computes the solution out to time $t = 3$.

```
In[7]:= NDSolve`Iterate[state, 3]
```

You can specify a time that is earlier than the first current time, in which case the integration proceeds backwards with respect to time.

This computes the solution from the initial condition backwards to $t = -\pi/2$.

```
In[8]:= NDSolve`Iterate[state, -Pi / 2]
```

`NDSolve`Iterate` allows you to specify intermediate times at which to stop. This can be useful, for example, to avoid discontinuities. Typically, this strategy is more effective with so-called one-step methods, such as the explicit Runge-Kutta method used in this example. However, it generally works with the default `NDSolve` method as well.

This computes the solution out to $t = 10\pi$, making sure that the solution does not have problems with the points of discontinuity in the coefficients at $t = \pi, 2\pi, \ldots$.

```
In[9]:= NDSolve`Iterate[state, π Range[10]]
```

Getting Solution Functions

Once you have integrated a system up to a certain time, typically you want to be able to look at the current solution values and to generate an approximate function representing the solution computed so far. The command `NDSolve`ProcessSolutions` allows you to do both.

`NDSolve`ProcessSolutions[`*state*`]`	give the solutions that have been computed in *state* as a list of rules with `InterpolatingFunction` objects

Getting solutions as `InterpolatingFunction` objects.

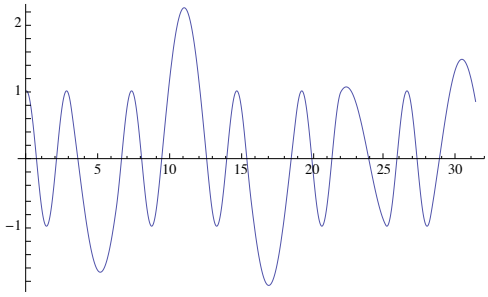
This extracts the solution computed in the previous section as an `InterpolatingFunction` object.

```
In[10]:= sol = NDSolve`ProcessSolutions[state]
Out[10]= {x → InterpolatingFunction[{{-1.5708, 31.4159}}, <>]}
```

This plots the solution.

```
In[11]:= Plot[Evaluate[x[t] /. sol], {t, 0, 10 Pi}]
```

Out[11]=



Just as when using `NDSolve` directly, there will be a rule for each function you specified in the second argument to `NDSolve`ProcessEquations`. Only the specified components of the solutions are saved in such a way that an `InterpolatingFunction` object can be created.

`NDSolve`ProcessSolutions[` *state*`,`*dir*`]`	give the solutions that have been most recently computed in direction *dir* in *state* as a list of rules with values for both the functions and their derivatives

Obtaining the current solution values.

This gives the current solution values and derivatives in the forward direction.

```
In[12]:= sol = NDSolve`ProcessSolutions[state, "Forward"]
Out[12]= {x[31.4159] → 0.843755, x'[31.4159] → -1.20016, x''[31.4159] → -0.843755}
```

The choices you can give for the direction *dir* are `"Forward"` and `"Backward"`, which refer to the integration forward and backward from the initial condition.

`"Forward"`	integration in the direction of increasing values of the temporal variable
`"Backward"`	integration in the direction of decreasing values of the temporal variables
`"Active"`	integration in the direction that is currently being integrated; typically, this value should only be called from method initialization that is used during an active integration

Integration direction specifications.

The output given by NDSolve`ProcessSolution is always given in terms of the dependent variables, either at a specific value of the independent variable, or interpolated over all of the saved values. This means that when a partial differential equation is being integrated, you will get results representing the dependent variables over the spatial variables.

This computes the solution to the heat equation from time $t = -1/4$ to $t = 2$.

```
In[13]:= state = First[NDSolve`ProcessEquations[{D[u[t, x], t] == D[u[t, x], x, x],
           u[0, x] == Cos[π / 2 x], u[t, 0] == 1 , u[t, 1] == 0}, u, t, {x, 0, 1}]];
         NDSolve`Iterate[state, {-1 / 4, 2}]
```

This gives the solution at $t = 2$.

```
In[15]:= NDSolve`ProcessSolutions[state, "Forward"]

Out[15]= {u[2., x] → InterpolatingFunction[{{0., 1.}}, <>][x],
          u^(1,0)[2., x] → InterpolatingFunction[{{0., 1.}}, <>][x]}
```

The solution is given as an InterpolatingFunction object that interpolates over the spatial variable x.

This gives the solution at $t = -1/4$.

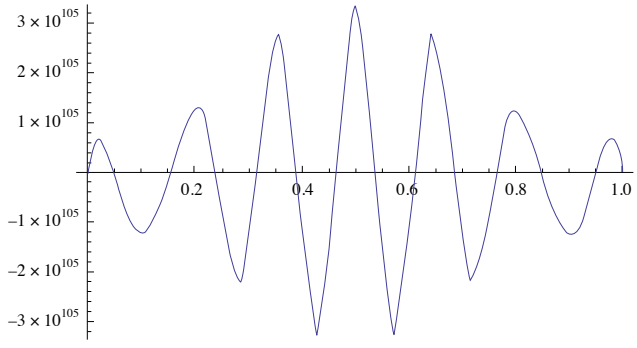
```
In[16]:= NDSolve`ProcessSolutions[state, "Backward"]

NDSolve::eerr: Warning: Scaled local spatial error estimate of 638.6378240455119`
    at t = −0.25 in the direction of independent variable x
    is much greater than prescribed error tolerance. Grid spacing with 15
    points may be too large to achieve the desired accuracy or precision.  A
      singularity may have formed or you may want to specify a smaller
      grid spacing using the MaxStepSize or MinPoints method options. ≫

Out[16]= {u[-0.25, x] → InterpolatingFunction[{{0., 1.}}, <>][x],
          u^(1,0)[-0.25, x] → InterpolatingFunction[{{0., 1.}}, <>][x]}
```

When you process the current solution for partial differential equations, the spatial error estimate is checked. (It is not generally checked except when solutions are produced because doing so would be quite time consuming.) Since it is excessive, the NDSolve::eerr message is issued. The typical association of the word "backward" with the heat equation as implying instability gives a clue to what is wrong in this example.

Here is a plot of the solution at $t = 1/4$.

```
In[17]:= Plot[Evaluate[u[-0.25, x] /. %], {x, 0, 1}]
```

Out[17]=



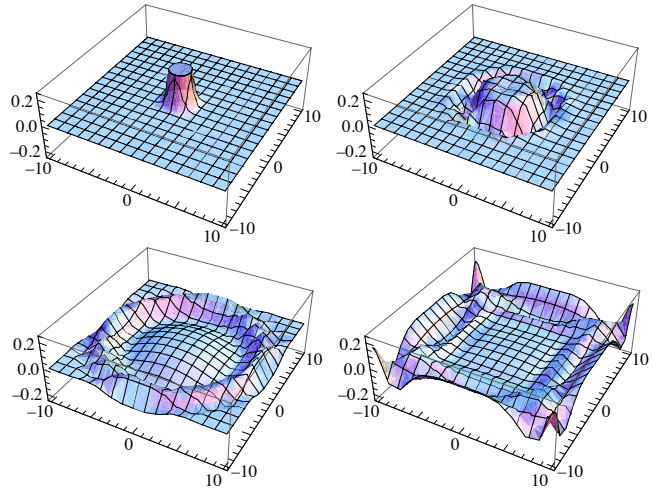
The plot of the solution shows that instability is indeed the problem.

Even though the heat equation example is simple enough to know that the solution backward in time is problematic, using `NDSolve`Iterate` and `NDSolve`ProcessSolutions` to monitor the solution of a PDE can be used to save computing a solution that turns out not to be as accurate as desired. Another simple form of monitoring follows.

Entering the following commands generates a sequence of plots showing the solution of a generalization of the sine-Gordon equation as it is being computed.

In[58]:=
```
L = -10;
state = First[NDSolve`ProcessEquations[{D[u[t, x, y], t, t] ==
      D[u[t, x, y], x, x] + D[u[t, x, y], y, y] - Sin[u[t, x, y]],
     u[0, x, y] == Exp[-(x^2 + y^2)], Derivative[1, 0, 0][u][0, x, y] == 0,
     u[t, -L, y] == u[t, L, y], u[t, x, -L] == u[t, x, L]}, u, t, {x, -L, L},
    {y, -L, L}, Method → {"MethodOfLines", "SpatialDiscretization" →
       {"TensorProductGrid", "DifferenceOrder" -> "Pseudospectral"}}]];
GraphicsGrid[Partition[Table[
   NDSolve`Iterate[state, τ];
   Plot3D[Evaluate[u[τ, x, y] /. NDSolve`ProcessSolutions[state, "Forward"]],
    {x, -L, L}, {y, -L, L}, PlotRange → {-1 / 4, 1 / 4}],
   {τ, 0., 20., 5.}], 2]]
```

Out[60]=



When you monitor a solution in this way, it is usually possible to interrupt the computation if you see that the solution found is sufficient. You can still use the `NDSolve`StateData` object to get the solutions that have been computed.

NDSolve`StateData Methods

An `NDSolve`StateData` object contains a lot of information, but it is arranged in a manner which makes it easy to iterate solutions, and not in a manner which makes it easy to understand where the information is kept. However, sometimes you will want to get information from the state data object: for this reason several method functions have been defined to make accessing the information easy.

state@"TemporalVariable"	give the independent variable that the dependent variables (functions) depend on
state@"DependentVariables"	give a list of the dependent variables (functions) to be solved for
state@"VariableDimensions"	give the dimensions of each of the dependent variables (functions)
state@"VariablePositions"	give the positions in the solution vector for each of the dependent variables
state@"VariableTransformation"	give the transformation of variables from the original problem variables to the working variables
state@"NumericalFunction"	give the "NumericalFunction" object used to evaluate the derivatives of the solution vector with respect to the temporal variable t
state@"ProcessExpression"[*args*,*expr*,*dims*]	process the expression *expr* using the same variable transformations that NDSolve used to generate state to give a "NumericalFunction" object for numerically evaluating *expr*; *args* are the arguments for the numerical function and should either be All or a list of arguments that are dependent variables of the system; *dims* should be Automatic or an explicit list giving the expected dimensions of the numerical function result
state@"SystemSize"	give the effective number of first-order ordinary differential equations being solved
state@"MaxSteps"	give the maximum number of steps allowed for iterating the differential equations
state@"WorkingPrecision"	give the working precision used to solve the equations
state@"Norm"	the scaled norm to use for gauging error

General method functions for an NDSolve`StateData object *state*.

Much of the available information depends on the current solution values. Each NDSolve`StateData object keeps solution information for solutions in both the forward and backward direction. At the initial condition these are the same, but once the problem has been iterated in either direction, these will be different.

state@"CurrentTime"[*dir*]	give the current value of the temporal variable in the integration direction *dir*
state@"SolutionVector"[*dir*]	give the current value of the solution vector in the integration direction *dir*
state@"SolutionDerivativeVector"[*dir*]	give the current value of the derivative with respect to the temporal variable of the solution vector in the integration direction *dir*
state@"TimeStep"[*dir*]	give the time step size for the next step in the integration direction *dir*
state@"TimeStepsUsed"[*dir*]	give the number of time steps used to get to the current time in the integration direction *dir*
state@"MethodData"[*dir*]	give the method data object used in the integration direction *dir*

Directional method functions for an NDSolve`StateData object *state*.

If the direction argument is omitted, the functions will return a list with the data for both directions (a list with a single element at the initial condition). Otherwise, the direction can be "Forward", "Backward", or "Active" as specified in the previous subsection.

Here is an NDSolve`StateData object for a solution of the nonlinear Schrodinger equation that has been computed up to $t = 1$.

```
In[24]:= state = First[NDSolve`ProcessEquations[
           {I D[u[t, x], t] == D[u[t, x], x, x] + Abs[u[t, x]]^2 u[t, x],
            u[0, x] == Sech[x] Exp[π I x], u[t, -15] == u[t, 15]},
           u, t, {x, -15, 15}, Method → StiffnessSwitching]];
         NDSolve`Iterate[state, 1];
         state
Out[24]= NDSolve`StateData[<0.,1.>]
```

"Current" refers to the most recent point reached in the integration.

This gives the current time in both the forward and backward directions.

```
In[27]:= state@"CurrentTime"
Out[27]= {0., 1.}
```

This gives the size of the system of ordinary differential equations being solved.

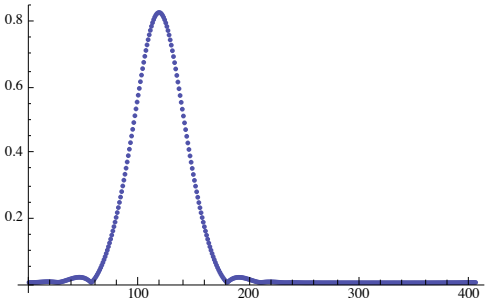
```
In[28]:= state@"SystemSize"
Out[28]= 400
```

The method functions are relatively low-level hooks into the data structure; they do little processing on the data returned to you. Thus, unlike `NDSolve`ProcessSolutions`, the solutions given are simply vectors of data points relating to the system of ordinary differential equations `NDSolve` is solving.

This makes a plot of the modulus of current solution in the forward direction.

```
In[29]:= ListPlot[Abs[state@SolutionVector["Forward"]]]
```

Out[29]=



This plot does not show the correspondence with the x-grid values correctly. To get the correspondence with the spatial grid correctly, you must use `NDSolve`ProcessSolutions`.

There is a tremendous amount of control provided by these methods, but an exhaustive set of examples is beyond the scope of this documentation.

One of the most important uses of the information from an `NDSolve`StateData` object is to initialize integration methods. Examples are shown in "The NDSolve Method Plug-in Framework".

Utility Packages for Numerical Differential Equation Solving

InterpolatingFunctionAnatomy

`NDSolve` returns solutions as `InterpolatingFunction` objects. Most of the time, simply using these as functions does what is needed, but occasionally it is useful to access the data inside, which includes the actual values and points `NDSolve` computed when taking steps. The exact structure of an `InterpolatingFunction` object is arranged to make the data storage efficient

and evaluation at a given point fast. This structure may change between *Mathematica* versions, so code that is written in terms of accessing parts of `InterpolatingFunction` objects may not work with new versions of *Mathematica*. The `DifferentialEquations`InterpolatingFunctionAnatomy`` package provides an interface to the data in an `InterpolatingFunction` object that will be maintained for future *Mathematica* versions.

`InterpolatingFunctionDomain[`*ifun*`]`	return a list with the domain of definition for each of the dimensions of the `InterpolatingFunction` object *ifun*
`InterpolatingFunctionCoordinates[`*ifun*`]`	return a list with the coordinates at which data is specified in each of the dimensions for the `InterpolatingFunction` object *ifun*
`InterpolatingFunctionGrid[`*ifun*`]`	return the grid of points at which data is specified for the `InterpolatingFunction` object *ifun*
`InterpolatingFunctionValuesOnGrid[`*ifun*`]`	return the values that would be returned by evaluating the `InterpolatingFunction` object *ifun* at each of its grid points
`InterpolatingFunctionInterpolationOrder[`*ifun*`]`	return the interpolation order used for each of the dimensions for the `InterpolatingFunction` object *ifun*
`InterpolatingFunctionDerivativeOrder[`*ifun*`]`	return the order of the derivative of the base function for which values are specified when evaluating the `InterpolatingFunction` object *ifun*

Anatomy of `InterpolatingFunction` objects.

This loads the package.

```
In[21]:= Needs["DifferentialEquations`InterpolatingFunctionAnatomy`"];
```

One common situation where the `InterpolatingFunctionAnatomy` package is useful is when `NDSolve` cannot compute a solution over the full range of values that you specified, and you want to plot all of the solution that was computed to try to understand better what might have gone wrong.

Here is an example of a differential equation which cannot be computed up to the specified endpoint.

```
In[2]:= ifun = First[x /. NDSolve[{x'[t] == Exp[x[t]] - x[t], x[0] == 1}, x, {t, 0, 10}]]

        NDSolve::ndsz:
          At t == 0.5160191740198964`, step size is effectively zero; singularity or stiff system suspected. >>

Out[2]= InterpolatingFunction[{{0., 0.516019}}, <>]
```

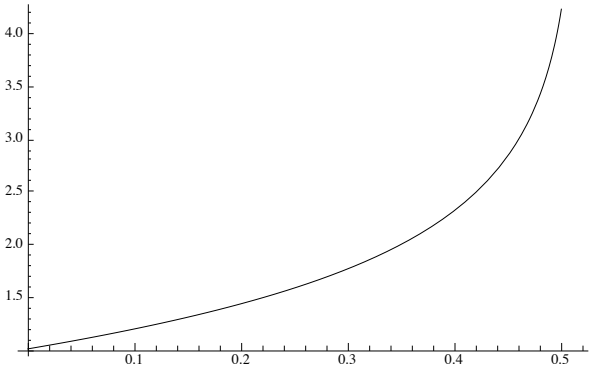
This gets the domain.

```
In[3]:= domain = InterpolatingFunctionDomain[ifun]
Out[3]= {{0., 0.516019}}
```

Once the domain has been returned in a list, it is easy to use `Part` to get the desired endpoints and make the plot.

```
In[4]:= {begin, end} = domain[[1]];
        Plot[ifun[t], {t, begin, end}]
```

Out[5]=



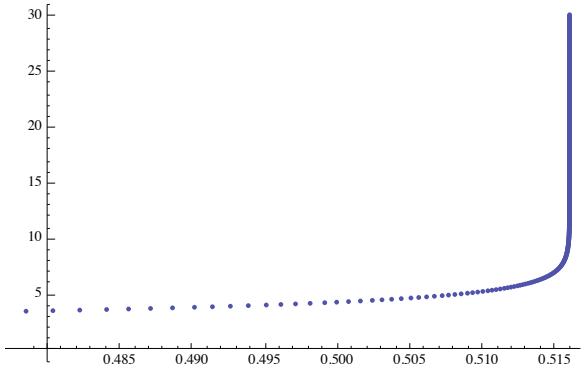
From the plot, it is quite apparent that a singularity has formed and it will not be possible to integrate the system any further.

Sometimes it is useful to see where `NDSolve` took steps. Getting the coordinates is useful for doing this.

This shows the values that `NDSolve` computed at each step it took. It is quite apparent from this that nearly all of the steps were used to try to resolve the singularity.

```
In[6]:= coords = First[InterpolatingFunctionCoordinates[ifun]];
        ListPlot[Transpose[{coords, ifun[coords]}]]
```

Out[7]=



The package is particularly useful for analyzing the computed solutions of PDEs.

With this initial condition, Burgers' equation forms a steep front.

In[8]:=
```
mdfun =
 First[u /. NDSolve[{D[u[x, t], t] == 0.01 D[u[x, t], x, x] - u[x, t] D[u[x, t], x],
     u[0, t] == u[1, t], u[x, 0] == Sin[2 Pi x]}, u, {x, 0, 1}, {t, 0, 0.5}]]
```

NDSolve::ndsz :
At t == 0.472151168326526`, step size is effectively zero; singularity or stiff system suspected. ≫

NDSolve::eerr : Warning: Scaled local spatial error estimate of 9.135898727911074`*^12
at t = 0.472151168326526` in the direction of independent variable x
is much greater than prescribed error tolerance. Grid spacing with 27
points may be too large to achieve the desired accuracy or precision. A
singularity may have formed or you may want to specify a smaller
grid spacing using the MaxStepSize or MinPoints method options. ≫

Out[8]=
```
InterpolatingFunction[{{..., 0., 1., ...}, {0., 0.472151}}, <>]
```

This shows the number of points used in each dimension.

In[9]:=
```
Map[Length, InterpolatingFunctionCoordinates[mdfun]]
```

Out[9]=
```
{27, 312}
```

This shows the interpolation order used in each dimension.

In[10]:=
```
InterpolatingFunctionInterpolationOrder[mdfun]
```

Out[10]=
```
{5, 3}
```

This shows that the inability to resolve the front has manifested itself as numerical instability.

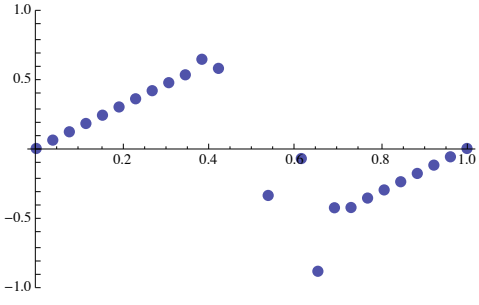
In[11]:=
```
Max[Abs[InterpolatingFunctionValuesOnGrid[mdfun]]]
```

Out[11]=
$$1.14928 \times 10^{12}$$

This shows the values computed at the spatial grid points at the endpoint of the temporal integration.

In[12]:=
```
end = InterpolatingFunctionDomain[mdfun][[2, -1]];
X = InterpolatingFunctionCoordinates[mdfun][[1]];
ListPlot[Transpose[{X, mdfun[X, end]}],
 PlotStyle → PointSize[.025], PlotRange → {-1, 1}]
```

Out[14]=

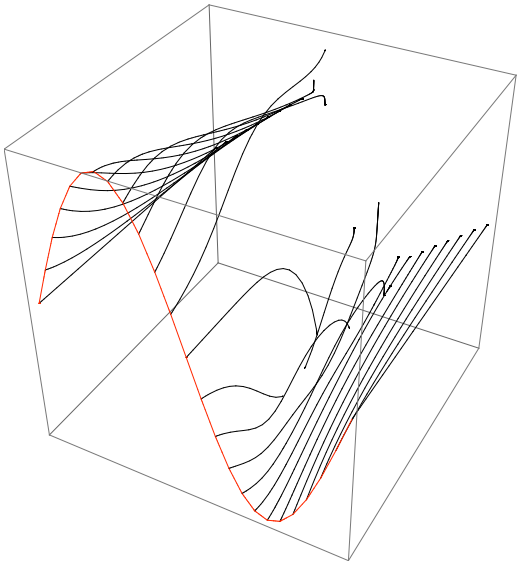


It is easily seen from the point plot that the front has not been resolved.

This makes a 3D plot showing the time evolution for each of the spatial grid points. The initial condition is shown in red.

```
In[15]:= Show[Graphics3D[{Map[Line, MapThread[Append, {InterpolatingFunctionGrid[mdfun],
           InterpolatingFunctionValuesOnGrid[mdfun]}, 2]],
       {RGBColor[1, 0, 0], Line[Transpose[{X, 0. X, mdfun[X, 0.]}]]}}],
  BoxRatios → {1, 1, 1}, PlotRange → {All, All, {-1, 1}}]
```

Out[15]=



When a derivative of an `InterpolatingFunction` object is taken, a new `InterpolatingFunction` object is returned that gives the requested derivative when evaluated at a point. The `InterpolatingFunctionDerivativeOrder` is a way of determining what derivative will be evaluated.

The derivative returns a new `InterpolatingFunction` object.

```
In[16]:= dmdfun = Derivative[0, 1][mdfun]
Out[16]= InterpolatingFunction[{{..., 0., 1., ...}, {0., 0.472151}}, <>]
```

This shows what derivative will be evaluated.

```
In[17]:= InterpolatingFunctionDerivativeOrder[dmdfun]
Out[17]= Derivative[0, 1]
```

NDSolveUtilities

A number of utility routines have been written to facilitate the investigation and comparison of various NDSolve methods. These functions have been collected in the package DifferentialEquations`NDSolveUtilities`.

CompareMethods[*sys*, *refsol*, *methods*, *opts*]	return statistics for various methods applied to the system *sys*
FinalSolutions[*sys*, *sols*]	return the solution values at the end of the numerical integration for various solutions *sols* corresponding to the system *sys*
InvariantErrorPlot[*invts*, *dvars*, *ivar*, *sol*, *opts*]	return a plot of the error in the invariants *invts* for the solution *sol*
RungeKuttaLinearStabilityFunction[*amat*, *bvec*, *var*]	return the linear stability function for the Runge-Kutta method with coefficient matrix *amat* and weight vector *bvec* using the variable *var*
StepDataPlot[*sols*, *opts*]	return plots of the step sizes taken for the solutions *sols* on a logarithmic scale

Functions provided in the NDSolveUtilities package.

This loads the package.

```
In[18]:= Needs["DifferentialEquations`NDSolveUtilities`"]
```

A useful means of analyzing Runge-Kutta methods is to study how they behave when applied to a scalar linear test problem (see the package FunctionApproximations.m).

This assigns the (exact or infinitely precise) coefficients for the 2-stage implicit Runge-Kutta Gauss method of order 4.

```
In[19]:= {amat, bvec, cvec} = NDSolve`ImplicitRungeKuttaGaussCoefficients[4, Infinity]
```

$$\text{Out[19]}= \left\{\left\{\left\{\frac{1}{4}, \frac{1}{12}\left(3-2\sqrt{3}\right)\right\}, \left\{\frac{1}{12}\left(3+2\sqrt{3}\right), \frac{1}{4}\right\}\right\}, \left\{\frac{1}{2}, \frac{1}{2}\right\}, \left\{\frac{1}{6}\left(3-\sqrt{3}\right), \frac{1}{6}\left(3+\sqrt{3}\right)\right\}\right\}$$

This computes the linear stability function, which corresponds to the (2,2) Padé approximation to the exponential at the origin.

```
In[20]:= RungeKuttaLinearStabilityFunction[amat, bvec, z]
```

$$\text{Out[20]}= \frac{1+\frac{z}{2}+\frac{z^2}{12}}{1-\frac{z}{2}+\frac{z^2}{12}}$$

Examples of the functions `CompareMethods`, `FinalSolutions`, `RungeKuttaLinearStabilityFunction`, and `StepDataPlot` can be found within "ExplicitRungeKutta Method for NDSolve".

Examples of the function `InvariantErrorPlot` can be found within "Projection Method for NDSolve".

InvariantErrorPlot Options

The function `InvariantErrorPlot` has a number of options that can be used to control the form of the result.

option name	*default value*	
`InvariantDimensions`	`Automatic`	specify the dimensions of the invariants
`InvariantErrorFunction`	`Abs[Subtract[#1,#2]]&`	specify the function to use for comparing errors
`InvariantErrorSampleRate`	`Automatic`	specify how often errors are sampled

Options of the function `InvariantErrorPlot`.

The default value for `InvariantDimensions` is to determine the dimensions from the structure of the input, `Dimensions[`*invts*`]`.

The default value for `InvariantErrorFunction` is a function to compute the absolute error.

The default value for `InvariantErrorSampleRate` is to sample all points if there are less than 1000 steps taken. Above this threshold a logarithmic sample rate is used.

Advanced Numerical Differential Equation Solving in Mathematica: References

[AP91] Ascher U. and L. Petzold. "Projected Implicit Runge-Kutta Methods for Differential Algebraic Equations" *SIAM J. Numer. Anal.* 28 (1991): 1097-1120

[AP98] Ascher U. and L. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations.* SIAM Press (1998)

[ARPACK98] Lehoucq, R. B., D. C. Sorensen, and C. Yang. ARPACK Users' Guide, Solution of Large-Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods, SIAM (1998)

[ATLAS00] Whaley R. C., A. Petitet, and J. J. Dongarra. "Automated Empirical Optimization of Software and the ATLAS Project" Available electronically from http://math-atlas.sourceforge.net/

[BD83] Bader G. and P. Deuflhard. "A Semi-Implicit Mid-Point Rule for Stiff Systems of Ordinary Differential Equations" *Numer. Math* 41 (1983): 373-398

[BS97] Bai Z. and G. W. Stewart. "SRRIT: a Fortran Subroutine to Calculate the Dominant Invariant Subspace of a Nonsymmetric Matrix" *ACM Trans. Math. Soft.* 23 4 (1997): 494-513

[BG94] Benettin G. and A. Giorgilli. "On the Hamiltonian Interpolation of Near to the Identity Symplectic Mappings with Application to Symplectic Integration Algorithms" *J. Stat. Phys.* 74 (1994): 1117-1143

[BZ65] Berezin I. S. and N. P. Zhidkov. *Computing Methods, Volume 2*. Pergamon (1965)

[BM02] Blanes S. and P. C. Moan. "Practical Symplectic Partitioned Runge-Kutta and Runge-Kutta-Nyström Methods" *J. Comput. Appl. Math.* 142 (2002): 313-330

[BCR99a] Blanes S., F. Casas, and J. Ros. "Symplectic Integration with Processing: A General Study" *SIAM J. Sci. Comput.* 21 (1999): 711-727

[BCR99b] Blanes S., F. Casas, and J. Ros. "Extrapolation of Symplectic Integrators" Report DAMTP NA09, Cambridge University (1999)

[BS89a] Bogacki P. and L. F. Shampine. "A 3(2) Pair of Runge-Kutta Formulas" *Appl. Math. Letters* 2 (1989): 1-9

[BS89b] Bogacki P. and L. F. Shampine. "An Efficient Runge-Kutta (4, 5) Pair" Report 89-20, Math. Dept. Southern Methodist University, Dallas, Texas (1989)

[BGS93] Brankin R. W., I. Gladwell and L. F. Shampine. "RKSUITE: A Suite of Explicit Runge-Kutta Codes" In *Contributions to Numerical Mathematics*, R. P. Agarwal, ed., 41-53 (1993)

[BCP89] Brenan K., S. Campbell, and L. Petzold. *Numerical Solutions of Initial-Value Problems in Differential-Algebraic Equations*. Elsevier Science Publishing (1989)

[BHP94] Brown P. N., A. C. Hindmarsh, and L. R. Petzold. "Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems" *SIAM J. Sci. Comput.* 15 (1994): 1467-1488

[BHP98] Brown P. N., A. C. Hindmarsh, and L. R. Petzold. "Consistent Initial Condition Calculation for Differential-Algebraic Systems" *SIAM J. Sci. Comput.* 19 (1998): 1495-1512

[B87] Butcher J. C. *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*. John Wiley (1987)

[B90] Butcher J. C. "Order, Stepsize and Stiffness Switching" *Computing.* 44 3, (1990): 209-220

[BS64] Bulirsch R. and J. Stoer. "Fehlerabschätzungen und Extrapolation mit Rationalen Funktionen bei Verfahren vom Richardson-Typus" *Numer. Math.* 6 (1964): 413-427

[CIZ97] Calvo M. P., A. Iserles, and A. Zanna. "Numerical Solution of Isospectral Flows" *Math. Comp.* 66, no. 220 (1997): 1461-1486

[CIZ99] Calvo M. P., A. Iserles, and A. Zanna. "Conservative Methods for the Toda Lattice Equations" *IMA J. Numer. Anal.* 19 (1999): 509-523

[CR91] Candy J. and R. Rozmus. "A Symplectic Integration Algorithm for Separable Hamiltonian Functions" *J. Comput. Phys.* 92 (1991): 230-256

[CH94] Cohen S. D. and A. C. Hindmarsh. *CVODE User Guide*. Lawrence Livermore National Laboratory report UCRL-MA-118618, September 1994

[CH96] Cohen S. D. and A. C. Hindmarsh. "CVODE, a Stiff/Nonstiff ODE Solver in C" *Computers in Physics* 10, no. 2 (1996): 138-143

[C87] Cooper G. J. "Stability of Runge-Kutta Methods for Trajectory Problems" *IMA J. Numer. Anal.* 7 (1987): 1-13

[DP80] Dormand J. R. and P. J. Prince. "A Family of Embedded Runge-Kutta Formulae" *J. Comp. Appl. Math.* 6 (1980): 19-26

[DL01] Del Buono N. and L. Lopez. "Runge-Kutta Type Methods Based on Geodesics for Systems of ODEs on the Stiefel Manifold" *BIT* 41 (5 (2001): 912-923

[D83] Deuflhard P. "Order and Step Size Control in Extrapolation Methods" *Numer. Math.* 41 (1983): 399-422

[D85] Deuflhard P. "Recent Progress in Extrapolation Methods for Ordinary Differential Equations" *SIAM Rev.* 27 (1985): 505-535

[DN87] Deuflhard P. and U. Nowak. "Extrapolation Integrators for Quasilinear Implicit ODEs" In *Large-scale scientific computing*, (P. Deuflhard and B. Engquist eds.) Birkhäuser, (1987)

[DS93] Duff I. S. and J. A. Scott. "Computing Selected Eigenvalues of Sparse Unsymmetric Matrices Using Subspace Iteration" *ACM Trans. Math. Soft.* 19 2, (1993): 137-159

[DHZ87] Deuflhard P., E. Hairer, and J. Zugck. "One-Step and Extrapolation Methods for Differential-Algebraic Systems" *Numer. Math.* 51 (1987): 501-516

[DRV94] Dieci L., R. D. Russel, and E. S. Van Vleck. "Unitary Integrators and Applications to Continuous Orthonormalization Techniques" *SIAM J. Num. Anal.* 31 (1994): 261-281

[DV99] Dieci L. and E. S. Van Vleck. "Computation of Orthonormal Factors for Fundamental Solution Matrices" *Numer. Math.* 83 (1999): 599-620

[DLP98a] Diele F., L. Lopez, and R. Peluso. "The Cayley Transform in the Numerical Solution of Unitary Differential Systems" *Adv. Comput. Math.* 8 (1998): 317-334

[DLP98b] Diele F., L. Lopez, and T. Politi. "One Step Semi-Explicit Methods Based on the Cayley Transform for Solving Isospectral Flows" *J. Comput. Appl. Math.* 89 (1998): 219-223

[ET92] Earn D. J. D. and S. Tremaine. "Exact Numerical Studies of Hamiltonian Maps: Iterating without Roundoff Error" *Physica D.* 56 (1992): 1-22

[F69] Fehlberg E. "Low-Order Classical Runge-Kutta Formulas with Step Size Control and Their Application to Heat Transfer Problems" NASA Technical Report 315, 1969 (extract published in *Computing* 6 (1970): 61-71)

[FR90] Forest E. and R. D. Ruth. "Fourth Order Symplectic Integration" *Physica D.* 43 (1990): 105-117

[F92] Fornberg B. "Fast Generation of Weights in Finite Difference Formulas" In *Recent Developments in Numerical Methods and Software for ODEs/DAEs/PDEs* (G. D. Byrne and W. E. Schiesser eds.). World Scientific (1992)

[F96a] Fornberg B. *A Practical Guide to Pseudospectral Methods*. Cambridge University Press (1996)

[F98] Fornberg B. "Calculation of Weights in Finite Difference Formulas" *SIAM Review* 40, no. 3 (1998): 685-691 (Available in PDF)

[F96b] Fukushima T. "Reduction of Round-off Errors in the Extrapolation Methods and its Application to the Integration of Orbital Motion" *Astron. J.* 112, no. 3 (1996): 1298-1301

[G51] Gill S. "A Process for the Step-by-Step Integration of Differential Equations in an Automatic Digital Computing Machine" *Proc. Cambridge Philos. Soc.* 47 (1951): 96-108

[G65] Gragg W. B. "On Extrapolation Algorithms for Ordinary Initial Value Problems" *SIAM J. Num. Anal.* 2 (1965): 384-403

[GØ84] Gear C. W. and O. Østerby. "Solving Ordinary Differential Equations with Discontinuities" *ACM Trans. Math. Soft.* 10 (1984): 23-44

[G91] Gustafsson K. "Control Theoretic Techniques for Stepsize Selection in Explicit Runge-Kutta Methods" *ACM Trans. Math. Soft.* 17, (1991): 533-554

[G94] Gustafsson K. "Control Theoretic Techniques for Stepsize Selection in Implicit Runge-Kutta Methods" *ACM Trans. Math. Soft.* 20, (1994): 496-517

[GMW81] Gill P., W. Murray, and M. Wright. *Practical Optimization*. Academic Press (1981)

[GDC91] Gladman B., M. Duncan, and J. Candy. "Symplectic Integrators for Long-Term Integrations in Celestial Mechanics" *Celest. Mech.* 52 (1991): 221-240

[GSB87] Gladwell I., L. F. Shampine and R. W. Brankin. "Automatic Selection of the Initial Step Size for an ODE Solver" *J. Comp. Appl. Math.* 18 (1987): 175-192

[GVL96] Golub G. H. and C. F. Van Loan. *Matrix Computations*, 3rd ed. Johns Hopkins University Press (1996)

[H83] Hindmarsh A. C. "ODEPACK, A Systematized Collection of ODE Solvers" In Scientific Computing (R. S. Stepleman et al. eds.) Vol. 1 of IMACS Transactions on Scientific Computation (1983): 55-64

[H94] Hairer E. "Backward Analysis of Numerical Integrators and Symplectic Methods" *Annals of Numerical Mathematics* 1 (1984): 107-132

[H97] Hairer E. "Variable Time Step integration with Symplectic Methods" *Appl. Numer. Math.* 25 (1997): 219-227

[H00] Hairer E. "Symmetric Projection Methods for Differential Equations on Manifolds" *BIT* 40, no. 4 (2000): 726-734

[HL97] Hairer E. and Ch. Lubich. "The Life-Span of Backward Error Analysis for Numerical Integrators" *Numer. Math.* 76 (1997): 441-462. Erratum: http://www.unige.ch/math/folks/hairer

[HL88a] Hairer E. and Ch. Lubich. "Extrapolation at Stiff Differential Equations" *Numer. Math.* 52 (1988): 377-400

[HL88b] Hairer E. and Ch. Lubich. "On Extrapolation Methods for Stiff and Differential-Algebraic Equations" *Teubner Texte zur Mathematik* 104 (1988): 64-73

[HO90] Hairer E. and A. Ostermann. "Dense Output for Extrapolation Methods" *Numer. Math.* 58 (1990): 419-439

[HW96] Hairer E. and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, 2nd ed. Springer-Verlag (1996)

[HW99] Hairer E. and G. Wanner. "Stiff Differential Equations Solved by Radau Methods" *J. Comp. Appl. Math.* 111 (1999): 93-111

[HLW02] Hairer E., Ch. Lubich, and G. Wanner. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. Springer-Verlag (2002)

[HNW93] Hairer E., S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd ed. Springer-Verlag (1993)

[H97] Higham D. ""Time-Stepping and Preserving Orthonormality" *BIT* 37, no. 1 (1997): 24-36

[H89] Higham N. J. "Matrix Nearness Problems and Applications" In *Applications of Matrix Theory* (M. J. C. Gover and S. Barnett eds.). Oxford University Press (1989): 1-27

[H96] Higham N. J. *Accuracy and Stability of Numerical Algorithms*. SIAM (1996)

[H83] Hindmarsh A. C. "ODEPACK, A Systematized Collection of ODE Solvers" In *Scientific Computing* (R. S. Stepleman et al. eds). North-Holland (1983): 55-64

[HT99] Hindmarsh A. and A. Taylor. *User Documentation for IDA: A Differential-Algebraic Equation Solver for Sequential and Parallel Computers*. Lawrence Livermore National Laboratory report, UCRL-MA-136910, December 1999.

[KL97] Kahan W. H. and R. C. Li. "Composition Constants for Raising the Order of Unconventional Schemes for Ordinary Differential Equations" *Math. Comp.* 66 (1997): 1089-1099

[K65] Kahan W. H. "Further Remarks on Reducing Truncation Errors" *Comm. ACM.* 8 (1965): 40

[K93] Koren I. *Computer Arithmetic Algorithms*. Prentice Hall (1993)

[L87] Lambert J. D. *Numerical Methods for Ordinary Differential Equations*. John Wiley (1987)

[LS96] Lehoucq, R. B and J. A. Scott. "An Evaluation of Software for Computing Eigenvalues of Sparse Nonsymmetric Matrices." Preprint MCS-P547-1195, Argonne National Laboratory, (1996)

[LAPACK99] Anderson E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorenson. *LAPACK Users' Guide*, 3rd ed. SIAM (1999)

[M68] Marchuk G. "Some Applications of Splitting-Up Methods to the Solution of Mathematical Physics Problems" *Aplikace Matematiky* 13 (1968): 103-132

[MR99] Marsden J. E. and T. Ratiu. *Introduction to Mechanics and Symmetry: Texts in Applied Mathematics, Vol. 17*, 2nd ed. Springer-Verlag (1999)

[M93] McLachlan R. I. "Explicit Lie-Poisson Integration and the Euler Equations" *Phys. Rev. Lett.* 71 (1993): 3043-3046

[M95a] McLachlan R. I. "On the Numerical Integration of Ordinary Differential Equations by Symmetric Composition Methods" *SIAM J. Sci. Comp.* 16 (1995): 151-168

[M95b] McLachlan R. I. "Composition Methods in the Presence of Small Parameters" *BIT* 35 (1995): 258-268

[M01] McLachlan R. I. "Families of High-Order Composition Methods" *Numerical Algorithms* 31 (2002): 233-246

[MA92] McLachlan R. I. and P. Atela. "The Accuracy of Symplectic Integrators" *Nonlinearity* 5 (1992): 541-562

[MQ02] McLachlan R. I. and G. R. W. Quispel. "Splitting Methods" *Acta Numerica* 11 (2002): 341-434

[MG80] Mitchell A. and D. Griffiths. *The Finite Difference Method in Partial Differential Equations*. John Wiley and Sons (1980)

[M65a] Møller O. "Quasi Double-Precision in Floating Point Addition" *BIT* 5 (1965): 37-50

[M65b] Møller O. "Note on Quasi Double-Precision" *BIT* 5 (1965): 251-255

[M97] Murua A. "On Order Conditions for Partitioned Symplectic Methods" *SIAM J. Numer. Anal.* 34, no. 6 (1997): 2204-2211

[MS99] Murua A. and J. M. Sanz-Serna. "Order Conditions for Numerical Integrators Obtained by Composing Simpler Integrators" *Phil. Trans. Royal Soc. A* 357 (1999): 1079-1100

[M04] Moler C. B. *Numerical Computing with MATLAB*. SIAM (2004)

[Na79] Na T. Y. *Computational Methods in Engineering: Boundary Value Problems*. Academic Press (1979)

[OS92] Okunbor D. I. and R. D. Skeel. "Explicit Canonical Methods for Hamiltonian Systems" *Math. Comp.* 59 (1992): 439-455

[O95] Olsson H. "Practical Implementation of Runge-Kutta Methods for Initial Value Problems" Licentiate thesis, Department of Computer Science, Lund University, 1995

[O98] Olsson H. "Runge-Kutta Solution of Initial Value Problems: Methods, Algorithms and Implementation" PhD Thesis, Department of Computer Science, Lund University, 1998

[OS00] Olsson H. and G. Söderlind. "The Approximate Runge-Kutta Computational Process" *BIT* 40 (2 (2000): 351-373

[P83] Petzold L. R. "Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations" *SIAM J. Sci. Stat. Comput.* 4 (1983): 136-148

[QSS00] Quarteroni A., R. Sacco, and F. Saleri. *Numerical Mathematics*. Springer-Verlag (2000)

[QV94] Quarteroni A. and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer-Verlag (1994)

[QT90] Quinn T. and S. Tremaine. "Roundoff Error in Long-Term Planetary Orbit Integrations" *Astron. J.* 99, no. 3 (1990): 1016-1023

[R93] Reich S. "Numerical Integration of the Generalized Euler Equations" *Tech. Rep.* 93-20, Dept. Comput. Sci. Univ. of British Columbia (1993)

[R99] Reich S. "Backward Error Analysis for Numerical Integrators" *SIAM J. Num. Anal.* 36 (1999): 1549-1570

[R98] Rubinstein B. "Numerical Solution of Linear Boundary Value Problems" *Mathematica MathSource* package, http://library.wolfram.com/database/MathSource/2127/

[RM57] Richtmeyer R. and K. Morton. *Difference Methods for Initial Value Problems*. Krieger Publishing Company (1994) (original edition 1957)

[R87] Robertson B. C. "Detecting Stiffness with Explicit Runge-Kutta Formulas" Report 193/87, Dept. Comp. Sci. University of Toronto (1987)

[S84b] Saad Y. "Chebyshev Acceleration Techniques for Solving Nonsymmetric Eigenvalue Problems" *Math. Comp.* 42, (1984): 567-588

[SC94] Sanz-Serna J. M. and M. P. Calvo. *Numerical Hamiltonian Problems: Applied Mathematics and Mathematical Computation*, no. 7. Chapman and Hall (1994)

[S91] Schiesser W. *The Numerical Method of Lines*. Academic Press (1991)

[S77] Shampine L. F." Stiffness and Non-Stiff Differential Equation Solvers II: Detecting Stiffness with Runge-Kutta Methods" *ACM Trans. Math. Soft.* 3 1 (1977): 44-53

[S83] Shampine L. F. "Type-Insensitive ODE Codes Based on Extrapolation Methods" *SIAM J. Sci. Stat. Comput.* 4 1 (1984): 635-644

[S84a] Shampine L. F. "Stiffness and the Automatic Selection of ODE Code" *J. Comp. Phys.* 54 1 (1984): 74-86

[S86] Shampine L. F. "Conservation Laws and the Numerical Solution of ODEs" *Comp. Maths. Appl.* 12B (1986): 1287-1296

[S87] Shampine L. F. "Control of Step Size and Order in Extrapolation Codes" *J. Comp. Appl. Math.* 18 (1987): 3-16

[S91] Shampine L. F. "Diagnosing Stiffness for Explicit Runge-Kutta Methods" *SIAM J. Sci. Stat. Comput.* 12 2 (1991): 260-272

[S94] Shampine L. F. *Numerical Solution of Ordinary Differential Equations*. Chapman and Hall (1994)

[SB83] Shampine L. F. and L. S. Baca. "Smoothing the Extrapolated Midpoint Rule" *Numer. Math.* 41 (1983): 165-175

[SG75] Shampine L. F. and M. Gordon. *Computer Solutions of Ordinary Differential Equations*. W. H. Freeman (1975)

[SR97] Shampine L. F. and M. W Reichelt. *Solving ODEs with MATLAB*. *SIAM J. Sci. Comp. 18-1* (1997): 1-22

[ST00] Shampine L. F. and S. Thompson. "Solving Delay Differential Equations with dde23" Available electronically from http://www.runet.edu/~thompson/webddes/tutorial.pdf

[ST01] Shampine L. F. and S. Thompson. "Solving DDEs in MATLAB" *Appl. Numer. Math.* 37 (2001): 441-458

[SGT03] Shampine L. F., I. Gladwell, and S. Thompson. *Solving ODEs with MATLAB*. Cambridge University Press (2003)

[SBB83] Shampine L. F., L. S. Baca, and H. J. Bauer. "Output in Extrapolation Codes" *Comp. and Maths. with Appl.* 9 (1983): 245-255

[SS03] Sofroniou M. and G. Spaletta. "Increment Formulations for Rounding Error Reduction in the Numerical Solution of Structured Differential Systems" *Future Generation Computer Systems* 19, no. 3 (2003): 375-383

[SS04] Sofroniou M. and G. Spaletta. "Construction of Explicit Runge-Kutta Pairs with Stiffness Detection" *Mathematical and Computer Modelling*, special issue on The Numerical Analysis of Ordinary Differential Equations, 40, no. 11-12 (2004): 1157-1169

[SS05] Sofroniou M. and G. Spaletta. "Derivation of Symmetric Composition Constants for Symmetric Integrators" *Optimization Methods and Software* 20, no. 4-5 (2005): 597-613

[SS06] Sofroniou M. and G. Spaletta. "Hybrid Solvers for Splitting and Composition Methods" *J. Comp. Appl. Math.*, special issue from the International Workshop on the Technological Aspects of Mathematics, 185, no. 2 (2006): 278-291

[S84c] Sottas G. "Dynamic Adaptive Selection Between Explicit and Implicit Methods When Solving ODEs" Report, Sect. de math, University of Genève, 1984

[S07] Sprott J.C. "A Simple Chaotic Delay Differential Equation", Phys. Lett. A. 366 (2007): 397-402

[S68] Strang G. "On the Construction of Difference Schemes" *SIAM J. Num. Anal.* 5 (1968): 506-517

[S70] Stetter H. J. "Symmetric Two-Step Algorithms for Ordinary Differential Equations" *Computing* 5 (1970): 267-280

[S01] Stewart G. W. "A Krylov-Schur Algorithm for Large Eigenproblems" SIAM J. Matrix Anal. Appl. 23 3, (2001): 601-614

[SJ81] Stewart W. J. and A. Jennings. "LOPSI: A Simultaneous Iteration Method for Real Matrices" *ACM Trans. Math. Soft.* 7 2, (1981): 184-198

[S90] Suzuki M. "Fractal Decomposition of Exponential Operators with Applications to Many-Body Theories and Monte Carlo Simulations" *Phys. Lett. A* 146 (1990): 319-323

[SLEPc05] Hernandez V., J. E. Roman and V. Vidal"SRRIT: a Fortran Subroutine to Calculate the Dominant Invariant Subspace of a Nonsymmetric Matrix" *ACM Trans. Math. Soft.* 31 3, (2005): 351-362

[T59] Trotter H. F. "On the Product of Semi-Group Operators" *Proc. Am. Math. Soc.* 10 (1959): 545-551

[TZ08] Tang, Z. H. and Zou, X. "Global attractivity in a predator-prey System with Pure Delays", Proc. Edinburgh Math. Soc. 51 (2008): 495-508

[V78] Verner J. H. "Explicit Runge-Kutta Methods with Estimates of The Local Truncation Error" *SIAM J. Num. Anal.* 15 (1978): 772-790.

[V79] Vitasek E. "A-Stability and Numerical Solution of Evolution Problems" *IAC 'Mauro Picone', Series III* 186 (1979): 42

[W76] Whitham G. B. *Linear and Nonlinear Waves*. John Wiley and Sons (1976)

[WH91] Wisdom J. and M. Holman. "Symplectic Maps for the N-Body Problem" *Astron. J.* 102 (1991): 1528-1538

[Y90] Yoshida H. "Construction of High Order Symplectic Integrators" *Phys. Lett. A.* 150 (1990): 262-268

[Z98] Zanna A. "On the Numerical Solution of Isospectral Flows" Ph.D. Thesis, Cambridge University, 1998

[Z72] Zeeman E. C. "Differential Equations for the Heartbeat and Nerve Impulse". In *Towards a Theoretical Biology* (C. H. Waddington, ed.). Edinburgh Univeristy Press, 4 (1972): 8-67

[Z06] Zennaro M. "The numerical solution of delay differential equations", Lecture notes, Dobbiaco Summer Chool on Delay Differential Equations and Applications (2006)